The Python Library Reference

Release 3.13.0rc1

Guido van Rossum and the Python development team

July 31, 2024

Python Software Foundation
Email: docs@python.org

CONTENTS

1 Introduction **3**
- 1.1 Notes on availability . . . 3
  - 1.1.1 WebAssembly platforms . . . 4
  - 1.1.2 iOS . . . 4

2 Built-in Functions **5**

3 Built-in Constants **33**
- 3.1 Constants added by the `site` module . . . 34

4 Built-in Types **35**
- 4.1 Truth Value Testing . . . 35
- 4.2 Boolean Operations — `and`, `or`, `not` . . . 35
- 4.3 Comparisons . . . 36
- 4.4 Numeric Types — `int`, `float`, `complex` . . . 36
  - 4.4.1 Bitwise Operations on Integer Types . . . 38
  - 4.4.2 Additional Methods on Integer Types . . . 38
  - 4.4.3 Additional Methods on Float . . . 41
  - 4.4.4 Hashing of numeric types . . . 42
- 4.5 Boolean Type - `bool` . . . 43
- 4.6 Iterator Types . . . 44
  - 4.6.1 Generator Types . . . 44
- 4.7 Sequence Types — `list`, `tuple`, `range` . . . 44
  - 4.7.1 Common Sequence Operations . . . 45
  - 4.7.2 Immutable Sequence Types . . . 47
  - 4.7.3 Mutable Sequence Types . . . 47
  - 4.7.4 Lists . . . 48
  - 4.7.5 Tuples . . . 49
  - 4.7.6 Ranges . . . 49
- 4.8 Text Sequence Type — `str` . . . 51
  - 4.8.1 String Methods . . . 52
  - 4.8.2 `printf`-style String Formatting . . . 60
- 4.9 Binary Sequence Types — `bytes`, `bytearray`, `memoryview` . . . 63
  - 4.9.1 Bytes Objects . . . 63
  - 4.9.2 Bytearray Objects . . . 64
  - 4.9.3 Bytes and Bytearray Operations . . . 65
  - 4.9.4 `printf`-style Bytes Formatting . . . 77
  - 4.9.5 Memory Views . . . 79
- 4.10 Set Types — `set`, `frozenset` . . . 86
- 4.11 Mapping Types — `dict` . . . 89

4.11.1 Dictionary view objects . . . 92
4.12 Context Manager Types . . . 94
4.13 Type Annotation Types — Generic Alias, Union . . . 95
4.13.1 Generic Alias Type . . . 95
4.13.2 Union Type . . . 99
4.14 Other Built-in Types . . . 101
4.14.1 Modules . . . 101
4.14.2 Classes and Class Instances . . . 101
4.14.3 Functions . . . 101
4.14.4 Methods . . . 102
4.14.5 Code Objects . . . 102
4.14.6 Type Objects . . . 102
4.14.7 The Null Object . . . 103
4.14.8 The Ellipsis Object . . . 103
4.14.9 The NotImplemented Object . . . 103
4.14.10 Internal Objects . . . 103
4.15 Special Attributes . . . 103
4.16 Integer string conversion length limitation . . . 104
4.16.1 Affected APIs . . . 105
4.16.2 Configuring the limit . . . 105
4.16.3 Recommended configuration . . . 106

5 Built-in Exceptions **107**
5.1 Exception context . . . 107
5.2 Inheriting from built-in exceptions . . . 108
5.3 Base classes . . . 108
5.4 Concrete exceptions . . . 109
5.4.1 OS exceptions . . . 115
5.5 Warnings . . . 116
5.6 Exception groups . . . 117
5.7 Exception hierarchy . . . 119

6 Text Processing Services **121**
6.1 `string` — Common string operations . . . 121
6.1.1 String constants . . . 121
6.1.2 Custom String Formatting . . . 122
6.1.3 Format String Syntax . . . 123
6.1.4 Template strings . . . 130
6.1.5 Helper functions . . . 132
6.2 `re` — Regular expression operations . . . 132
6.2.1 Regular Expression Syntax . . . 133
6.2.2 Module Contents . . . 140
6.2.3 Regular Expression Objects . . . 146
6.2.4 Match Objects . . . 148
6.2.5 Regular Expression Examples . . . 151
6.3 `difflib` — Helpers for computing deltas . . . 156
6.3.1 SequenceMatcher Objects . . . 161
6.3.2 SequenceMatcher Examples . . . 164
6.3.3 Differ Objects . . . 164
6.3.4 Differ Example . . . 165
6.3.5 A command-line interface to difflib . . . 166
6.3.6 ndiff example . . . 167
6.4 `textwrap` — Text wrapping and filling . . . 169
6.5 `unicodedata` — Unicode Database . . . 173

6.6 stringprep — Internet String Preparation . . . 175
6.7 readline — GNU readline interface . . . 177
6.7.1 Init file . . . 177
6.7.2 Line buffer . . . 178
6.7.3 History file . . . 178
6.7.4 History list . . . 178
6.7.5 Startup hooks . . . 179
6.7.6 Completion . . . 179
6.7.7 Example . . . 180
6.8 rlcompleter — Completion function for GNU readline . . . 181

7 Binary Data Services **183**
7.1 struct — Interpret bytes as packed binary data . . . 183
7.1.1 Functions and Exceptions . . . 184
7.1.2 Format Strings . . . 184
7.1.3 Applications . . . 189
7.1.4 Classes . . . 190
7.2 codecs — Codec registry and base classes . . . 191
7.2.1 Codec Base Classes . . . 194
7.2.2 Encodings and Unicode . . . 201
7.2.3 Standard Encodings . . . 202
7.2.4 Python Specific Encodings . . . 205
7.2.5 encodings.idna — Internationalized Domain Names in Applications . . . 207
7.2.6 encodings.mbcs — Windows ANSI codepage . . . 208
7.2.7 encodings.utf_8_sig — UTF-8 codec with BOM signature . . . 208

8 Data Types **209**
8.1 datetime — Basic date and time types . . . 209
8.1.1 Aware and Naive Objects . . . 210
8.1.2 Constants . . . 210
8.1.3 Available Types . . . 210
8.1.4 timedelta Objects . . . 212
8.1.5 date Objects . . . 215
8.1.6 datetime Objects . . . 220
8.1.7 time Objects . . . 232
8.1.8 tzinfo Objects . . . 236
8.1.9 timezone Objects . . . 243
8.1.10 strftime() and strptime() Behavior . . . 244
8.2 zoneinfo — IANA time zone support . . . 248
8.2.1 Using ZoneInfo . . . 248
8.2.2 Data sources . . . 249
8.2.3 The ZoneInfo class . . . 250
8.2.4 Functions . . . 253
8.2.5 Globals . . . 253
8.2.6 Exceptions and warnings . . . 254
8.3 calendar — General calendar-related functions . . . 254
8.3.1 Command-Line Usage . . . 260
8.4 collections — Container datatypes . . . 261
8.4.1 ChainMap objects . . . 262
8.4.2 Counter objects . . . 264
8.4.3 deque objects . . . 268
8.4.4 defaultdict objects . . . 271
8.4.5 namedtuple() Factory Function for Tuples with Named Fields . . . 273
8.4.6 OrderedDict objects . . . 276

8.4.7 UserDict objects . . . 279
8.4.8 UserList objects . . . 279
8.4.9 UserString objects . . . 280
8.5 collections.abc — Abstract Base Classes for Containers . . . 280
8.5.1 Collections Abstract Base Classes . . . 281
8.5.2 Collections Abstract Base Classes – Detailed Descriptions . . . 283
8.5.3 Examples and Recipes . . . 285
8.6 heapq — Heap queue algorithm . . . 286
8.6.1 Basic Examples . . . 287
8.6.2 Priority Queue Implementation Notes . . . 288
8.6.3 Theory . . . 289
8.7 bisect — Array bisection algorithm . . . 290
8.7.1 Performance Notes . . . 291
8.7.2 Searching Sorted Lists . . . 292
8.7.3 Examples . . . 292
8.8 array — Efficient arrays of numeric values . . . 293
8.9 weakref — Weak references . . . 297
8.9.1 Weak Reference Objects . . . 301
8.9.2 Example . . . 302
8.9.3 Finalizer Objects . . . 303
8.9.4 Comparing finalizers with __del__() methods . . . 304
8.10 types — Dynamic type creation and names for built-in types . . . 305
8.10.1 Dynamic Type Creation . . . 305
8.10.2 Standard Interpreter Types . . . 307
8.10.3 Additional Utility Classes and Functions . . . 311
8.10.4 Coroutine Utility Functions . . . 312
8.11 copy — Shallow and deep copy operations . . . 312
8.12 pprint — Data pretty printer . . . 314
8.12.1 Functions . . . 314
8.12.2 PrettyPrinter Objects . . . 315
8.12.3 Example . . . 317
8.13 reprlib — Alternate repr() implementation . . . 320
8.13.1 Repr Objects . . . 321
8.13.2 Subclassing Repr Objects . . . 322
8.14 enum — Support for enumerations . . . 323
8.14.1 Module Contents . . . 324
8.14.2 Data Types . . . 325
8.14.3 Utilities and Decorators . . . 337
8.14.4 Notes . . . 338
8.15 graphlib — Functionality to operate with graph-like structures . . . 339
8.15.1 Exceptions . . . 341

9 Numeric and Mathematical Modules 343
9.1 numbers — Numeric abstract base classes . . . 343
9.1.1 The numeric tower . . . 343
9.1.2 Notes for type implementers . . . 344
9.2 math — Mathematical functions . . . 346
9.2.1 Number-theoretic and representation functions . . . 346
9.2.2 Power and logarithmic functions . . . 350
9.2.3 Trigonometric functions . . . 352
9.2.4 Angular conversion . . . 353
9.2.5 Hyperbolic functions . . . 353
9.2.6 Special functions . . . 353
9.2.7 Constants . . . 354

9.3 cmath — Mathematical functions for complex numbers . . . 355
9.3.1 Conversions to and from polar coordinates . . . 355
9.3.2 Power and logarithmic functions . . . 356
9.3.3 Trigonometric functions . . . 356
9.3.4 Hyperbolic functions . . . 357
9.3.5 Classification functions . . . 357
9.3.6 Constants . . . 358
9.4 decimal — Decimal fixed-point and floating-point arithmetic . . . 359
9.4.1 Quick-start Tutorial . . . 360
9.4.2 Decimal objects . . . 363
9.4.3 Context objects . . . 371
9.4.4 Constants . . . 378
9.4.5 Rounding modes . . . 378
9.4.6 Signals . . . 379
9.4.7 Floating-Point Notes . . . 380
9.4.8 Working with threads . . . 382
9.4.9 Recipes . . . 382
9.4.10 Decimal FAQ . . . 385
9.5 fractions — Rational numbers . . . 388
9.6 random — Generate pseudo-random numbers . . . 392
9.6.1 Bookkeeping functions . . . 393
9.6.2 Functions for bytes . . . 393
9.6.3 Functions for integers . . . 393
9.6.4 Functions for sequences . . . 394
9.6.5 Discrete distributions . . . 395
9.6.6 Real-valued distributions . . . 395
9.6.7 Alternative Generator . . . 397
9.6.8 Notes on Reproducibility . . . 397
9.6.9 Examples . . . 398
9.6.10 Recipes . . . 400
9.6.11 Command-line usage . . . 401
9.6.12 Command-line example . . . 402
9.7 statistics — Mathematical statistics functions . . . 402
9.7.1 Averages and measures of central location . . . 403
9.7.2 Measures of spread . . . 404
9.7.3 Statistics for relations between two inputs . . . 404
9.7.4 Function details . . . 404
9.7.5 Exceptions . . . 414
9.7.6 NormalDist objects . . . 414
9.7.7 Examples and Recipes . . . 416

10 Functional Programming Modules **419**
10.1 itertools — Functions creating iterators for efficient looping . . . 419
10.1.1 Itertool Functions . . . 421
10.1.2 Itertools Recipes . . . 430
10.2 functools — Higher-order functions and operations on callable objects . . . 436
10.2.1 partial Objects . . . 446
10.3 operator — Standard operators as functions . . . 446
10.3.1 Mapping Operators to Functions . . . 451
10.3.2 In-place Operators . . . 452

11 File and Directory Access **455**
11.1 pathlib — Object-oriented filesystem paths . . . 455
11.1.1 Basic use . . . 456

11.1.2 Exceptions . . . 457
11.1.3 Pure paths . . . 457
11.1.4 Concrete paths . . . 467
11.1.5 Pattern language . . . 479
11.1.6 Comparison to the glob module . . . 480
11.1.7 Comparison to the os and os.path modules . . . 480
11.2 os.path — Common pathname manipulations . . . 482
11.3 fileinput — Iterate over lines from multiple input streams . . . 489
11.4 stat — Interpreting stat() results . . . 491
11.5 filecmp — File and Directory Comparisons . . . 498
11.5.1 The dircmp class . . . 498
11.6 tempfile — Generate temporary files and directories . . . 500
11.6.1 Examples . . . 504
11.6.2 Deprecated functions and variables . . . 505
11.7 glob — Unix style pathname pattern expansion . . . 506
11.7.1 Examples . . . 508
11.8 fnmatch — Unix filename pattern matching . . . 508
11.9 linecache — Random access to text lines . . . 510
11.10 shutil — High-level file operations . . . 510
11.10.1 Directory and files operations . . . 511
11.10.2 Archiving operations . . . 517
11.10.3 Querying the size of the output terminal . . . 521

12 Data Persistence **523**
12.1 pickle — Python object serialization . . . 523
12.1.1 Relationship to other Python modules . . . 524
12.1.2 Data stream format . . . 525
12.1.3 Module Interface . . . 525
12.1.4 What can be pickled and unpickled? . . . 529
12.1.5 Pickling Class Instances . . . 530
12.1.6 Custom Reduction for Types, Functions, and Other Objects . . . 536
12.1.7 Out-of-band Buffers . . . 537
12.1.8 Restricting Globals . . . 538
12.1.9 Performance . . . 540
12.1.10 Examples . . . 540
12.2 copyreg — Register pickle support functions . . . 541
12.2.1 Example . . . 541
12.3 shelve — Python object persistence . . . 541
12.3.1 Restrictions . . . 543
12.3.2 Example . . . 544
12.4 marshal — Internal Python object serialization . . . 544
12.5 dbm — Interfaces to Unix “databases” . . . 546
12.5.1 dbm.sqlite3 — SQLite backend for dbm . . . 548
12.5.2 dbm.gnu — GNU database manager . . . 549
12.5.3 dbm.ndbm — New Database Manager . . . 550
12.5.4 dbm.dumb — Portable DBM implementation . . . 551
12.6 sqlite3 — DB-API 2.0 interface for SQLite databases . . . 553
12.6.1 Tutorial . . . 553
12.6.2 Reference . . . 555
12.6.3 How-to guides . . . 577
12.6.4 Explanation . . . 585

13 Data Compression and Archiving **587**
13.1 zlib — Compression compatible with **gzip** . . . 587

13.2 gzip — Support for **gzip** files . . . 591
13.2.1 Examples of usage . . . 593
13.2.2 Command Line Interface . . . 594
13.3 bz2 — Support for **bzip2** compression . . . 595
13.3.1 (De)compression of files . . . 595
13.3.2 Incremental (de)compression . . . 597
13.3.3 One-shot (de)compression . . . 598
13.3.4 Examples of usage . . . 598
13.4 lzma — Compression using the LZMA algorithm . . . 600
13.4.1 Reading and writing compressed files . . . 600
13.4.2 Compressing and decompressing data in memory . . . 601
13.4.3 Miscellaneous . . . 604
13.4.4 Specifying custom filter chains . . . 604
13.4.5 Examples . . . 605
13.5 zipfile — Work with ZIP archives . . . 606
13.5.1 ZipFile Objects . . . 607
13.5.2 Path Objects . . . 612
13.5.3 PyZipFile Objects . . . 613
13.5.4 ZipInfo Objects . . . 614
13.5.5 Command-Line Interface . . . 616
13.5.6 Decompression pitfalls . . . 617
13.6 tarfile — Read and write tar archive files . . . 618
13.6.1 TarFile Objects . . . 622
13.6.2 TarInfo Objects . . . 626
13.6.3 Extraction filters . . . 628
13.6.4 Command-Line Interface . . . 632
13.6.5 Examples . . . 633
13.6.6 Supported tar formats . . . 634
13.6.7 Unicode issues . . . 635

14 File Formats **637**
14.1 csv — CSV File Reading and Writing . . . 637
14.1.1 Module Contents . . . 637
14.1.2 Dialects and Formatting Parameters . . . 642
14.1.3 Reader Objects . . . 643
14.1.4 Writer Objects . . . 643
14.1.5 Examples . . . 644
14.2 configparser — Configuration file parser . . . 645
14.2.1 Quick Start . . . 645
14.2.2 Supported Datatypes . . . 647
14.2.3 Fallback Values . . . 648
14.2.4 Supported INI File Structure . . . 648
14.2.5 Unnamed Sections . . . 649
14.2.6 Interpolation of values . . . 650
14.2.7 Mapping Protocol Access . . . 651
14.2.8 Customizing Parser Behaviour . . . 652
14.2.9 Legacy API Examples . . . 657
14.2.10 ConfigParser Objects . . . 658
14.2.11 RawConfigParser Objects . . . 662
14.2.12 Exceptions . . . 663
14.3 tomllib — Parse TOML files . . . 664
14.3.1 Examples . . . 665
14.3.2 Conversion Table . . . 665
14.4 netrc — netrc file processing . . . 665

14.4.1 netrc Objects . . . 666
14.5 plistlib — Generate and parse Apple .plist files . . . 667
14.5.1 Examples . . . 669

15 Cryptographic Services **671**
15.1 hashlib — Secure hashes and message digests . . . 671
15.1.1 Hash algorithms . . . 671
15.1.2 Usage . . . 672
15.1.3 Constructors . . . 672
15.1.4 Attributes . . . 673
15.1.5 Hash Objects . . . 673
15.1.6 SHAKE variable length digests . . . 674
15.1.7 File hashing . . . 674
15.1.8 Key derivation . . . 675
15.1.9 BLAKE2 . . . 676
15.2 hmac — Keyed-Hashing for Message Authentication . . . 683
15.3 secrets — Generate secure random numbers for managing secrets . . . 685
15.3.1 Random numbers . . . 685
15.3.2 Generating tokens . . . 686
15.3.3 Other functions . . . 687
15.3.4 Recipes and best practices . . . 687

16 Generic Operating System Services **689**
16.1 os — Miscellaneous operating system interfaces . . . 689
16.1.1 File Names, Command Line Arguments, and Environment Variables . . . 690
16.1.2 Python UTF-8 Mode . . . 690
16.1.3 Process Parameters . . . 691
16.1.4 File Object Creation . . . 699
16.1.5 File Descriptor Operations . . . 699
16.1.6 Files and Directories . . . 713
16.1.7 Process Management . . . 740
16.1.8 Interface to the scheduler . . . 755
16.1.9 Miscellaneous System Information . . . 756
16.1.10 Random numbers . . . 758
16.2 io — Core tools for working with streams . . . 760
16.2.1 Overview . . . 760
16.2.2 Text Encoding . . . 761
16.2.3 High-level Module Interface . . . 762
16.2.4 Class hierarchy . . . 763
16.2.5 Performance . . . 774
16.3 time — Time access and conversions . . . 774
16.3.1 Functions . . . 776
16.3.2 Clock ID Constants . . . 785
16.3.3 Timezone Constants . . . 786
16.4 argparse — Parser for command-line options, arguments and sub-commands . . . 787
16.4.1 Core Functionality . . . 788
16.4.2 Quick Links for add_argument() . . . 788
16.4.3 Example . . . 789
16.4.4 ArgumentParser objects . . . 790
16.4.5 The add_argument() method . . . 799
16.4.6 The parse_args() method . . . 810
16.4.7 Other utilities . . . 814
16.4.8 Upgrading optparse code . . . 822
16.4.9 Exceptions . . . 823

16.5 logging — Logging facility for Python . . . 823
16.5.1 Logger Objects . . . 825
16.5.2 Logging Levels . . . 829
16.5.3 Handler Objects . . . 830
16.5.4 Formatter Objects . . . 832
16.5.5 Filter Objects . . . 834
16.5.6 LogRecord Objects . . . 834
16.5.7 LogRecord attributes . . . 836
16.5.8 LoggerAdapter Objects . . . 838
16.5.9 Thread Safety . . . 838
16.5.10 Module-Level Functions . . . 839
16.5.11 Module-Level Attributes . . . 843
16.5.12 Integration with the warnings module . . . 844
16.6 logging.config — Logging configuration . . . 844
16.6.1 Configuration functions . . . 845
16.6.2 Security considerations . . . 847
16.6.3 Configuration dictionary schema . . . 847
16.6.4 Configuration file format . . . 854
16.7 logging.handlers — Logging handlers . . . 857
16.7.1 StreamHandler . . . 858
16.7.2 FileHandler . . . 858
16.7.3 NullHandler . . . 859
16.7.4 WatchedFileHandler . . . 859
16.7.5 BaseRotatingHandler . . . 860
16.7.6 RotatingFileHandler . . . 861
16.7.7 TimedRotatingFileHandler . . . 862
16.7.8 SocketHandler . . . 863
16.7.9 DatagramHandler . . . 864
16.7.10 SysLogHandler . . . 865
16.7.11 NTEventLogHandler . . . 867
16.7.12 SMTPHandler . . . 868
16.7.13 MemoryHandler . . . 868
16.7.14 HTTPHandler . . . 869
16.7.15 QueueHandler . . . 870
16.7.16 QueueListener . . . 871
16.8 getpass — Portable password input . . . 872
16.9 curses — Terminal handling for character-cell displays . . . 873
16.9.1 Functions . . . 874
16.9.2 Window Objects . . . 881
16.9.3 Constants . . . 888
16.10 curses.textpad — Text input widget for curses programs . . . 901
16.10.1 Textbox objects . . . 902
16.11 curses.ascii — Utilities for ASCII characters . . . 903
16.12 curses.panel — A panel stack extension for curses . . . 907
16.12.1 Functions . . . 907
16.12.2 Panel Objects . . . 907
16.13 platform — Access to underlying platform's identifying data . . . 908
16.13.1 Cross Platform . . . 908
16.13.2 Java Platform . . . 910
16.13.3 Windows Platform . . . 911
16.13.4 macOS Platform . . . 911
16.13.5 iOS Platform . . . 911
16.13.6 Unix Platforms . . . 912
16.13.7 Linux Platforms . . . 912

16.13.8 Android Platform . . . 912
16.14 errno — Standard errno system symbols . . . 913
16.15 ctypes — A foreign function library for Python . . . 920
16.15.1 ctypes tutorial . . . 920
16.15.2 ctypes reference . . . 939

17 Concurrent Execution **955**
17.1 threading — Thread-based parallelism . . . 955
17.1.1 Thread-Local Data . . . 958
17.1.2 Thread Objects . . . 958
17.1.3 Lock Objects . . . 961
17.1.4 RLock Objects . . . 962
17.1.5 Condition Objects . . . 963
17.1.6 Semaphore Objects . . . 965
17.1.7 Event Objects . . . 967
17.1.8 Timer Objects . . . 967
17.1.9 Barrier Objects . . . 968
17.1.10 Using locks, conditions, and semaphores in the with statement . . . 969
17.2 multiprocessing — Process-based parallelism . . . 970
17.2.1 Introduction . . . 970
17.2.2 Reference . . . 977
17.2.3 Programming guidelines . . . 1007
17.2.4 Examples . . . 1011
17.3 multiprocessing.shared_memory — Shared memory for direct access across processes . . . 1017
17.4 The concurrent package . . . 1023
17.5 concurrent.futures — Launching parallel tasks . . . 1023
17.5.1 Executor Objects . . . 1023
17.5.2 ThreadPoolExecutor . . . 1024
17.5.3 ProcessPoolExecutor . . . 1026
17.5.4 Future Objects . . . 1028
17.5.5 Module Functions . . . 1029
17.5.6 Exception classes . . . 1030
17.6 subprocess — Subprocess management . . . 1030
17.6.1 Using the subprocess Module . . . 1031
17.6.2 Security Considerations . . . 1040
17.6.3 Popen Objects . . . 1040
17.6.4 Windows Popen Helpers . . . 1043
17.6.5 Older high-level API . . . 1045
17.6.6 Replacing Older Functions with the subprocess Module . . . 1047
17.6.7 Legacy Shell Invocation Functions . . . 1050
17.6.8 Notes . . . 1051
17.7 sched — Event scheduler . . . 1052
17.7.1 Scheduler Objects . . . 1053
17.8 queue — A synchronized queue class . . . 1054
17.8.1 Queue Objects . . . 1055
17.8.2 SimpleQueue Objects . . . 1057
17.9 contextvars — Context Variables . . . 1058
17.9.1 Context Variables . . . 1058
17.9.2 Manual Context Management . . . 1059
17.9.3 asyncio support . . . 1061
17.10 _thread — Low-level threading API . . . 1062

18 Networking and Interprocess Communication **1065**
18.1 asyncio — Asynchronous I/O . . . 1065

18.1.1 Runners . . . 1066
18.1.2 Coroutines and Tasks . . . 1069
18.1.3 Streams . . . 1088
18.1.4 Synchronization Primitives . . . 1097
18.1.5 Subprocesses . . . 1103
18.1.6 Queues . . . 1108
18.1.7 Exceptions . . . 1111
18.1.8 Event Loop . . . 1112
18.1.9 Futures . . . 1138
18.1.10 Transports and Protocols . . . 1141
18.1.11 Policies . . . 1156
18.1.12 Platform Support . . . 1160
18.1.13 Extending . . . 1161
18.1.14 High-level API Index . . . 1163
18.1.15 Low-level API Index . . . 1165
18.1.16 Developing with asyncio . . . 1171
18.2 socket — Low-level networking interface . . . 1175
18.2.1 Socket families . . . 1175
18.2.2 Module contents . . . 1178
18.2.3 Socket Objects . . . 1192
18.2.4 Notes on socket timeouts . . . 1199
18.2.5 Example . . . 1200
18.3 ssl — TLS/SSL wrapper for socket objects . . . 1204
18.3.1 Functions, Constants, and Exceptions . . . 1205
18.3.2 SSL Sockets . . . 1217
18.3.3 SSL Contexts . . . 1222
18.3.4 Certificates . . . 1232
18.3.5 Examples . . . 1234
18.3.6 Notes on non-blocking sockets . . . 1236
18.3.7 Memory BIO Support . . . 1237
18.3.8 SSL session . . . 1239
18.3.9 Security considerations . . . 1240
18.3.10 TLS 1.3 . . . 1241
18.4 select — Waiting for I/O completion . . . 1242
18.4.1 /dev/poll Polling Objects . . . 1244
18.4.2 Edge and Level Trigger Polling (epoll) Objects . . . 1245
18.4.3 Polling Objects . . . 1246
18.4.4 Kqueue Objects . . . 1247
18.4.5 Kevent Objects . . . 1248
18.5 selectors — High-level I/O multiplexing . . . 1249
18.5.1 Introduction . . . 1250
18.5.2 Classes . . . 1250
18.5.3 Examples . . . 1253
18.6 signal — Set handlers for asynchronous events . . . 1253
18.6.1 General rules . . . 1253
18.6.2 Module contents . . . 1254
18.6.3 Examples . . . 1261
18.6.4 Note on SIGPIPE . . . 1262
18.6.5 Note on Signal Handlers and Exceptions . . . 1262
18.7 mmap — Memory-mapped file support . . . 1263
18.7.1 MADV_* Constants . . . 1267
18.7.2 MAP_* Constants . . . 1268

19 Internet Data Handling **1271**

19.1 email — An email and MIME handling package . . . 1271
19.1.1 email.message: Representing an email message . . . 1272
19.1.2 email.parser: Parsing email messages . . . 1280
19.1.3 email.generator: Generating MIME documents . . . 1284
19.1.4 email.policy: Policy Objects . . . 1287
19.1.5 email.errors: Exception and Defect classes . . . 1294
19.1.6 email.headerregistry: Custom Header Objects . . . 1295
19.1.7 email.contentmanager: Managing MIME Content . . . 1301
19.1.8 email: Examples . . . 1304
19.1.9 email.message.Message: Representing an email message using the compat32 API . . . 1310
19.1.10 email.mime: Creating email and MIME objects from scratch . . . 1319
19.1.11 email.header: Internationalized headers . . . 1321
19.1.12 email.charset: Representing character sets . . . 1324
19.1.13 email.encoders: Encoders . . . 1326
19.1.14 email.utils: Miscellaneous utilities . . . 1327
19.1.15 email.iterators: Iterators . . . 1329
19.2 json — JSON encoder and decoder . . . 1331
19.2.1 Basic Usage . . . 1333
19.2.2 Encoders and Decoders . . . 1335
19.2.3 Exceptions . . . 1338
19.2.4 Standard Compliance and Interoperability . . . 1338
19.2.5 Command Line Interface . . . 1340
19.3 mailbox — Manipulate mailboxes in various formats . . . 1341
19.3.1 Mailbox objects . . . 1341
19.3.2 Message objects . . . 1352
19.3.3 Exceptions . . . 1360
19.3.4 Examples . . . 1361
19.4 mimetypes — Map filenames to MIME types . . . 1362
19.4.1 MimeTypes Objects . . . 1364
19.5 base64 — Base16, Base32, Base64, Base85 Data Encodings . . . 1365
19.5.1 Security Considerations . . . 1368
19.6 binascii — Convert between binary and ASCII . . . 1369
19.7 quopri — Encode and decode MIME quoted-printable data . . . 1371

20 Structured Markup Processing Tools 1373
20.1 html — HyperText Markup Language support . . . 1373
20.2 html.parser — Simple HTML and XHTML parser . . . 1373
20.2.1 Example HTML Parser Application . . . 1374
20.2.2 HTMLParser Methods . . . 1375
20.2.3 Examples . . . 1376
20.3 html.entities — Definitions of HTML general entities . . . 1378
20.4 XML Processing Modules . . . 1379
20.4.1 XML vulnerabilities . . . 1379
20.4.2 The defusedxml Package . . . 1380
20.5 xml.etree.ElementTree — The ElementTree XML API . . . 1381
20.5.1 Tutorial . . . 1381
20.5.2 XPath support . . . 1386
20.5.3 Reference . . . 1389
20.5.4 XInclude support . . . 1392
20.5.5 Reference . . . 1393
20.6 xml.dom — The Document Object Model API . . . 1401
20.6.1 Module Contents . . . 1402
20.6.2 Objects in the DOM . . . 1403
20.6.3 Conformance . . . 1411

20.7 xml.dom.minidom — Minimal DOM implementation . . . 1412
20.7.1 DOM Objects . . . 1414
20.7.2 DOM Example . . . 1415
20.7.3 minidom and the DOM standard . . . 1416
20.8 xml.dom.pulldom — Support for building partial DOM trees . . . 1417
20.8.1 DOMEventStream Objects . . . 1418
20.9 xml.sax — Support for SAX2 parsers . . . 1419
20.9.1 SAXException Objects . . . 1421
20.10 xml.sax.handler — Base classes for SAX handlers . . . 1421
20.10.1 ContentHandler Objects . . . 1423
20.10.2 DTDHandler Objects . . . 1425
20.10.3 EntityResolver Objects . . . 1425
20.10.4 ErrorHandler Objects . . . 1425
20.10.5 LexicalHandler Objects . . . 1426
20.11 xml.sax.saxutils — SAX Utilities . . . 1426
20.12 xml.sax.xmlreader — Interface for XML parsers . . . 1428
20.12.1 XMLReader Objects . . . 1429
20.12.2 IncrementalParser Objects . . . 1430
20.12.3 Locator Objects . . . 1430
20.12.4 InputSource Objects . . . 1430
20.12.5 The Attributes Interface . . . 1431
20.12.6 The AttributesNS Interface . . . 1432
20.13 xml.parsers.expat — Fast XML parsing using Expat . . . 1432
20.13.1 XMLParser Objects . . . 1433
20.13.2 ExpatError Exceptions . . . 1438
20.13.3 Example . . . 1438
20.13.4 Content Model Descriptions . . . 1439
20.13.5 Expat error constants . . . 1440

21 Internet Protocols and Support **1443**
21.1 webbrowser — Convenient web-browser controller . . . 1443
21.1.1 Browser Controller Objects . . . 1446
21.2 wsgiref — WSGI Utilities and Reference Implementation . . . 1446
21.2.1 wsgiref.util – WSGI environment utilities . . . 1446
21.2.2 wsgiref.headers – WSGI response header tools . . . 1448
21.2.3 wsgiref.simple_server – a simple WSGI HTTP server . . . 1449
21.2.4 wsgiref.validate — WSGI conformance checker . . . 1450
21.2.5 wsgiref.handlers – server/gateway base classes . . . 1451
21.2.6 wsgiref.types – WSGI types for static type checking . . . 1455
21.2.7 Examples . . . 1455
21.3 urllib — URL handling modules . . . 1457
21.4 urllib.request — Extensible library for opening URLs . . . 1457
21.4.1 Request Objects . . . 1462
21.4.2 OpenerDirector Objects . . . 1464
21.4.3 BaseHandler Objects . . . 1465
21.4.4 HTTPRedirectHandler Objects . . . 1466
21.4.5 HTTPCookieProcessor Objects . . . 1467
21.4.6 ProxyHandler Objects . . . 1467
21.4.7 HTTPPasswordMgr Objects . . . 1468
21.4.8 HTTPPasswordMgrWithPriorAuth Objects . . . 1468
21.4.9 AbstractBasicAuthHandler Objects . . . 1468
21.4.10 HTTPBasicAuthHandler Objects . . . 1469
21.4.11 ProxyBasicAuthHandler Objects . . . 1469
21.4.12 AbstractDigestAuthHandler Objects . . . 1469

21.4.13 HTTPDigestAuthHandler Objects . . . 1469
21.4.14 ProxyDigestAuthHandler Objects . . . 1469
21.4.15 HTTPHandler Objects . . . 1469
21.4.16 HTTPSHandler Objects . . . 1469
21.4.17 FileHandler Objects . . . 1470
21.4.18 DataHandler Objects . . . 1470
21.4.19 FTPHandler Objects . . . 1470
21.4.20 CacheFTPHandler Objects . . . 1470
21.4.21 UnknownHandler Objects . . . 1470
21.4.22 HTTPErrorProcessor Objects . . . 1470
21.4.23 Examples . . . 1471
21.4.24 Legacy interface . . . 1473
21.4.25 `urllib.request` Restrictions . . . 1476
21.5 `urllib.response` — Response classes used by urllib . . . 1476
21.6 `urllib.parse` — Parse URLs into components . . . 1477
21.6.1 URL Parsing . . . 1477
21.6.2 URL parsing security . . . 1482
21.6.3 Parsing ASCII Encoded Bytes . . . 1482
21.6.4 Structured Parse Results . . . 1483
21.6.5 URL Quoting . . . 1484
21.7 `urllib.error` — Exception classes raised by urllib.request . . . 1486
21.8 `urllib.robotparser` — Parser for robots.txt . . . 1487
21.9 `http` — HTTP modules . . . 1488
21.9.1 HTTP status codes . . . 1489
21.9.2 HTTP status category . . . 1490
21.9.3 HTTP methods . . . 1491
21.10 `http.client` — HTTP protocol client . . . 1492
21.10.1 HTTPConnection Objects . . . 1494
21.10.2 HTTPResponse Objects . . . 1497
21.10.3 Examples . . . 1498
21.10.4 HTTPMessage Objects . . . 1500
21.11 `ftplib` — FTP protocol client . . . 1500
21.11.1 Reference . . . 1500
21.12 `poplib` — POP3 protocol client . . . 1507
21.12.1 POP3 Objects . . . 1508
21.12.2 POP3 Example . . . 1510
21.13 `imaplib` — IMAP4 protocol client . . . 1510
21.13.1 IMAP4 Objects . . . 1512
21.13.2 IMAP4 Example . . . 1517
21.14 `smtplib` — SMTP protocol client . . . 1517
21.14.1 SMTP Objects . . . 1520
21.14.2 SMTP Example . . . 1524
21.15 `uuid` — UUID objects according to **RFC 4122** . . . 1524
21.15.1 Command-Line Usage . . . 1528
21.15.2 Example . . . 1528
21.15.3 Command-Line Example . . . 1529
21.16 `socketserver` — A framework for network servers . . . 1529
21.16.1 Server Creation Notes . . . 1530
21.16.2 Server Objects . . . 1531
21.16.3 Request Handler Objects . . . 1533
21.16.4 Examples . . . 1534
21.17 `http.server` — HTTP servers . . . 1538
21.17.1 Security Considerations . . . 1544
21.18 `http.cookies` — HTTP state management . . . 1545

21.18.1 Cookie Objects . . . 1546
21.18.2 Morsel Objects . . . 1546
21.18.3 Example . . . 1548
21.19 http.cookiejar — Cookie handling for HTTP clients . . . 1549
21.19.1 CookieJar and FileCookieJar Objects . . . 1551
21.19.2 FileCookieJar subclasses and co-operation with web browsers . . . 1553
21.19.3 CookiePolicy Objects . . . 1553
21.19.4 DefaultCookiePolicy Objects . . . 1554
21.19.5 Cookie Objects . . . 1556
21.19.6 Examples . . . 1557
21.20 xmlrpc — XMLRPC server and client modules . . . 1558
21.21 xmlrpc.client — XML-RPC client access . . . 1558
21.21.1 ServerProxy Objects . . . 1560
21.21.2 DateTime Objects . . . 1561
21.21.3 Binary Objects . . . 1562
21.21.4 Fault Objects . . . 1563
21.21.5 ProtocolError Objects . . . 1563
21.21.6 MultiCall Objects . . . 1564
21.21.7 Convenience Functions . . . 1565
21.21.8 Example of Client Usage . . . 1565
21.21.9 Example of Client and Server Usage . . . 1566
21.22 xmlrpc.server — Basic XML-RPC servers . . . 1566
21.22.1 SimpleXMLRPCServer Objects . . . 1567
21.22.2 CGIXMLRPCRequestHandler . . . 1570
21.22.3 Documenting XMLRPC server . . . 1571
21.22.4 DocXMLRPCServer Objects . . . 1572
21.22.5 DocCGIXMLRPCRequestHandler . . . 1572
21.23 ipaddress — IPv4/IPv6 manipulation library . . . 1572
21.23.1 Convenience factory functions . . . 1573
21.23.2 IP Addresses . . . 1573
21.23.3 IP Network definitions . . . 1578
21.23.4 Interface objects . . . 1585
21.23.5 Other Module Level Functions . . . 1586
21.23.6 Custom Exceptions . . . 1587

22 Multimedia Services 1589
22.1 wave — Read and write WAV files . . . 1589
22.1.1 Wave_read Objects . . . 1590
22.1.2 Wave_write Objects . . . 1591
22.2 colorsys — Conversions between color systems . . . 1592

23 Internationalization 1593
23.1 gettext — Multilingual internationalization services . . . 1593
23.1.1 GNU **gettext** API . . . 1593
23.1.2 Class-based API . . . 1594
23.1.3 Internationalizing your programs and modules . . . 1598
23.1.4 Acknowledgements . . . 1601
23.2 locale — Internationalization services . . . 1601
23.2.1 Background, details, hints, tips and caveats . . . 1609
23.2.2 For extension writers and programs that embed Python . . . 1609
23.2.3 Access to message catalogs . . . 1610

24 Program Frameworks 1611
24.1 turtle — Turtle graphics . . . 1611

24.1.1 Introduction . . . 1611
24.1.2 Get started . . . 1611
24.1.3 Tutorial . . . 1612
24.1.4 How to… . . . 1614
24.1.5 Turtle graphics reference . . . 1616
24.1.6 Methods of RawTurtle/Turtle and corresponding functions . . . 1619
24.1.7 Methods of TurtleScreen/Screen and corresponding functions . . . 1637
24.1.8 Public classes . . . 1644
24.1.9 Explanation . . . 1646
24.1.10 Help and configuration . . . 1646
24.1.11 turtledemo — Demo scripts . . . 1649
24.1.12 Changes since Python 2.6 . . . 1650
24.1.13 Changes since Python 3.0 . . . 1651
24.2 cmd — Support for line-oriented command interpreters . . . 1651
24.2.1 Cmd Objects . . . 1651
24.2.2 Cmd Example . . . 1654
24.3 shlex — Simple lexical analysis . . . 1656
24.3.1 shlex Objects . . . 1658
24.3.2 Parsing Rules . . . 1660
24.3.3 Improved Compatibility with Shells . . . 1661

25 Graphical User Interfaces with Tk 1663
25.1 tkinter — Python interface to Tcl/Tk . . . 1663
25.1.1 Architecture . . . 1664
25.1.2 Tkinter Modules . . . 1665
25.1.3 Tkinter Life Preserver . . . 1667
25.1.4 Threading model . . . 1670
25.1.5 Handy Reference . . . 1670
25.1.6 File Handlers . . . 1676
25.2 tkinter.colorchooser — Color choosing dialog . . . 1677
25.3 tkinter.font — Tkinter font wrapper . . . 1677
25.4 Tkinter Dialogs . . . 1679
25.4.1 tkinter.simpledialog — Standard Tkinter input dialogs . . . 1679
25.4.2 tkinter.filedialog — File selection dialogs . . . 1679
25.4.3 tkinter.commondialog — Dialog window templates . . . 1682
25.5 tkinter.messagebox — Tkinter message prompts . . . 1682
25.6 tkinter.scrolledtext — Scrolled Text Widget . . . 1684
25.7 tkinter.dnd — Drag and drop support . . . 1685
25.8 tkinter.ttk — Tk themed widgets . . . 1686
25.8.1 Using Ttk . . . 1686
25.8.2 Ttk Widgets . . . 1687
25.8.3 Widget . . . 1687
25.8.4 Combobox . . . 1689
25.8.5 Spinbox . . . 1690
25.8.6 Notebook . . . 1691
25.8.7 Progressbar . . . 1694
25.8.8 Separator . . . 1695
25.8.9 Sizegrip . . . 1695
25.8.10 Treeview . . . 1695
25.8.11 Ttk Styling . . . 1701
25.9 IDLE . . . 1706
25.9.1 Menus . . . 1706
25.9.2 Editing and Navigation . . . 1711
25.9.3 Startup and Code Execution . . . 1714

25.9.4 Help and Preferences . . . 1717
25.9.5 idlelib . . . 1718

26 Development Tools **1719**
26.1 `typing` — Support for type hints . . . 1719
26.1.1 Specification for the Python Type System . . . 1720
26.1.2 Type aliases . . . 1720
26.1.3 NewType . . . 1721
26.1.4 Annotating callable objects . . . 1722
26.1.5 Generics . . . 1723
26.1.6 Annotating tuples . . . 1724
26.1.7 The type of class objects . . . 1725
26.1.8 User-defined generic types . . . 1726
26.1.9 The `Any` type . . . 1729
26.1.10 Nominal vs structural subtyping . . . 1730
26.1.11 Module contents . . . 1730
26.1.12 Deprecation Timeline of Major Features . . . 1774
26.2 `pydoc` — Documentation generator and online help system . . . 1774
26.3 Python Development Mode . . . 1775
26.3.1 Effects of the Python Development Mode . . . 1776
26.3.2 ResourceWarning Example . . . 1777
26.3.3 Bad file descriptor error example . . . 1778
26.4 `doctest` — Test interactive Python examples . . . 1779
26.4.1 Simple Usage: Checking Examples in Docstrings . . . 1781
26.4.2 Simple Usage: Checking Examples in a Text File . . . 1781
26.4.3 How It Works . . . 1782
26.4.4 Basic API . . . 1790
26.4.5 Unittest API . . . 1791
26.4.6 Advanced API . . . 1794
26.4.7 Debugging . . . 1799
26.4.8 Soapbox . . . 1802
26.5 `unittest` — Unit testing framework . . . 1803
26.5.1 Basic example . . . 1804
26.5.2 Command-Line Interface . . . 1805
26.5.3 Test Discovery . . . 1807
26.5.4 Organizing test code . . . 1808
26.5.5 Re-using old test code . . . 1810
26.5.6 Skipping tests and expected failures . . . 1810
26.5.7 Distinguishing test iterations using subtests . . . 1812
26.5.8 Classes and functions . . . 1813
26.5.9 Class and Module Fixtures . . . 1833
26.5.10 Signal Handling . . . 1835
26.6 `unittest.mock` — mock object library . . . 1835
26.6.1 Quick Guide . . . 1836
26.6.2 The Mock Class . . . 1838
26.6.3 The patchers . . . 1856
26.6.4 MagicMock and magic method support . . . 1866
26.6.5 Helpers . . . 1869
26.6.6 Order of precedence of `side_effect`, `return_value` and *wraps* . . . 1877
26.7 `unittest.mock` — getting started . . . 1879
26.7.1 Using Mock . . . 1879
26.7.2 Patch Decorators . . . 1885
26.7.3 Further Examples . . . 1887
26.8 `test` — Regression tests package for Python . . . 1900

26.8.1 Writing Unit Tests for the test package . . . 1901
26.8.2 Running tests using the command-line interface . . . 1902
26.9 test.support — Utilities for the Python test suite . . . 1903
26.10 test.support.socket_helper — Utilities for socket tests . . . 1913
26.11 test.support.script_helper — Utilities for the Python execution tests . . . 1913
26.12 test.support.bytecode_helper — Support tools for testing correct bytecode generation . . . 1915
26.13 test.support.threading_helper — Utilities for threading tests . . . 1915
26.14 test.support.os_helper — Utilities for os tests . . . 1916
26.15 test.support.import_helper — Utilities for import tests . . . 1918
26.16 test.support.warnings_helper — Utilities for warnings tests . . . 1919

27 Debugging and Profiling 1923
27.1 Audit events table . . . 1923
27.2 bdb — Debugger framework . . . 1927
27.3 faulthandler — Dump the Python traceback . . . 1933
27.3.1 Dumping the traceback . . . 1933
27.3.2 Fault handler state . . . 1934
27.3.3 Dumping the tracebacks after a timeout . . . 1934
27.3.4 Dumping the traceback on a user signal . . . 1934
27.3.5 Issue with file descriptors . . . 1935
27.3.6 Example . . . 1935
27.4 pdb — The Python Debugger . . . 1935
27.4.1 Debugger Commands . . . 1938
27.5 The Python Profilers . . . 1944
27.5.1 Introduction to the profilers . . . 1944
27.5.2 Instant User's Manual . . . 1945
27.5.3 profile and cProfile Module Reference . . . 1947
27.5.4 The Stats Class . . . 1949
27.5.5 What Is Deterministic Profiling? . . . 1951
27.5.6 Limitations . . . 1952
27.5.7 Calibration . . . 1952
27.5.8 Using a custom timer . . . 1953
27.6 timeit — Measure execution time of small code snippets . . . 1953
27.6.1 Basic Examples . . . 1954
27.6.2 Python Interface . . . 1954
27.6.3 Command-Line Interface . . . 1956
27.6.4 Examples . . . 1957
27.7 trace — Trace or track Python statement execution . . . 1959
27.7.1 Command-Line Usage . . . 1959
27.7.2 Programmatic Interface . . . 1960
27.8 tracemalloc — Trace memory allocations . . . 1961
27.8.1 Examples . . . 1962
27.8.2 API . . . 1967

28 Software Packaging and Distribution 1973
28.1 ensurepip — Bootstrapping the pip installer . . . 1973
28.1.1 Command line interface . . . 1974
28.1.2 Module API . . . 1974
28.2 venv — Creation of virtual environments . . . 1975
28.2.1 Creating virtual environments . . . 1976
28.2.2 How venvs work . . . 1977
28.2.3 API . . . 1978
28.2.4 An example of extending EnvBuilder . . . 1981
28.3 zipapp — Manage executable Python zip archives . . . 1985

28.3.1 Basic Example . . . 1985
28.3.2 Command-Line Interface . . . 1986
28.3.3 Python API . . . 1986
28.3.4 Examples . . . 1987
28.3.5 Specifying the Interpreter . . . 1988
28.3.6 Creating Standalone Applications with zipapp . . . 1988
28.3.7 The Python Zip Application Archive Format . . . 1989

29 Python Runtime Services **1991**
29.1 sys — System-specific parameters and functions . . . 1991
29.2 sys.monitoring — Execution event monitoring . . . 2017
29.2.1 Tool identifiers . . . 2018
29.2.2 Events . . . 2019
29.2.3 Turning events on and off . . . 2021
29.2.4 Registering callback functions . . . 2022
29.3 sysconfig — Provide access to Python's configuration information . . . 2023
29.3.1 Configuration variables . . . 2023
29.3.2 Installation paths . . . 2024
29.3.3 User scheme . . . 2024
29.3.4 Home scheme . . . 2025
29.3.5 Prefix scheme . . . 2026
29.3.6 Installation path functions . . . 2027
29.3.7 Other functions . . . 2028
29.3.8 Using sysconfig as a script . . . 2029
29.4 builtins — Built-in objects . . . 2030
29.5 __main__ — Top-level code environment . . . 2030
29.5.1 __name__ == '__main__' . . . 2031
29.5.2 __main__.py in Python Packages . . . 2033
29.5.3 import __main__ . . . 2034
29.6 warnings — Warning control . . . 2036
29.6.1 Warning Categories . . . 2036
29.6.2 The Warnings Filter . . . 2037
29.6.3 Temporarily Suppressing Warnings . . . 2039
29.6.4 Testing Warnings . . . 2040
29.6.5 Updating Code For New Versions of Dependencies . . . 2040
29.6.6 Available Functions . . . 2041
29.6.7 Available Context Managers . . . 2043
29.7 dataclasses — Data Classes . . . 2043
29.7.1 Module contents . . . 2044
29.7.2 Post-init processing . . . 2050
29.7.3 Class variables . . . 2051
29.7.4 Init-only variables . . . 2051
29.7.5 Frozen instances . . . 2051
29.7.6 Inheritance . . . 2051
29.7.7 Re-ordering of keyword-only parameters in __init__() . . . 2052
29.7.8 Default factory functions . . . 2052
29.7.9 Mutable default values . . . 2053
29.7.10 Descriptor-typed fields . . . 2054
29.8 contextlib — Utilities for with-statement contexts . . . 2054
29.8.1 Utilities . . . 2055
29.8.2 Examples and Recipes . . . 2064
29.8.3 Single use, reusable and reentrant context managers . . . 2067
29.9 abc — Abstract Base Classes . . . 2070
29.10 atexit — Exit handlers . . . 2075

29.10.1 atexit Example . . . 2076
29.11 traceback — Print or retrieve a stack traceback . . . 2076
29.11.1 TracebackException Objects . . . 2079
29.11.2 StackSummary Objects . . . 2081
29.11.3 FrameSummary Objects . . . 2082
29.11.4 Traceback Examples . . . 2082
29.12 __future__ — Future statement definitions . . . 2085
29.12.1 Module Contents . . . 2085
29.13 gc — Garbage Collector interface . . . 2086
29.14 inspect — Inspect live objects . . . 2090
29.14.1 Types and members . . . 2090
29.14.2 Retrieving source code . . . 2096
29.14.3 Introspecting callables with the Signature object . . . 2097
29.14.4 Classes and functions . . . 2102
29.14.5 The interpreter stack . . . 2105
29.14.6 Fetching attributes statically . . . 2107
29.14.7 Current State of Generators, Coroutines, and Asynchronous Generators . . . 2108
29.14.8 Code Objects Bit Flags . . . 2109
29.14.9 Buffer flags . . . 2110
29.14.10Command Line Interface . . . 2111
29.15 site — Site-specific configuration hook . . . 2111
29.15.1 sitecustomize . . . 2112
29.15.2 usercustomize . . . 2113
29.15.3 Readline configuration . . . 2113
29.15.4 Module contents . . . 2113
29.15.5 Command Line Interface . . . 2114

30 Custom Python Interpreters **2115**
30.1 code — Interpreter base classes . . . 2115
30.1.1 Interactive Interpreter Objects . . . 2116
30.1.2 Interactive Console Objects . . . 2117
30.2 codeop — Compile Python code . . . 2117

31 Importing Modules **2119**
31.1 zipimport — Import modules from Zip archives . . . 2119
31.1.1 zipimporter Objects . . . 2120
31.1.2 Examples . . . 2121
31.2 pkgutil — Package extension utility . . . 2121
31.3 modulefinder — Find modules used by a script . . . 2125
31.3.1 Example usage of ModuleFinder . . . 2125
31.4 runpy — Locating and executing Python modules . . . 2126
31.5 importlib — The implementation of import . . . 2128
31.5.1 Introduction . . . 2129
31.5.2 Functions . . . 2130
31.5.3 importlib.abc – Abstract base classes related to import . . . 2131
31.5.4 importlib.machinery – Importers and path hooks . . . 2138
31.5.5 importlib.util – Utility code for importers . . . 2144
31.5.6 Examples . . . 2147
31.6 importlib.resources – Package resource reading, opening and access . . . 2150
31.6.1 Functional API . . . 2151
31.7 importlib.resources.abc – Abstract base classes for resources . . . 2153
31.8 importlib.metadata – Accessing package metadata . . . 2155
31.8.1 Overview . . . 2156
31.8.2 Functional API . . . 2156

31.8.3 Distributions . . . 2159
31.8.4 Distribution Discovery . . . 2160
31.8.5 Extending the search algorithm . . . 2160
31.9 The initialization of the sys.path module search path . . . 2162
31.9.1 Virtual environments . . . 2163
31.9.2 _pth files . . . 2163
31.9.3 Embedded Python . . . 2163

32 Python Language Services 2165
32.1 ast — Abstract Syntax Trees . . . 2165
32.1.1 Abstract Grammar . . . 2165
32.1.2 Node classes . . . 2168
32.1.3 ast Helpers . . . 2199
32.1.4 Compiler Flags . . . 2203
32.1.5 Command-Line Usage . . . 2203
32.2 symtable — Access to the compiler's symbol tables . . . 2204
32.2.1 Generating Symbol Tables . . . 2204
32.2.2 Examining Symbol Tables . . . 2204
32.2.3 Command-Line Usage . . . 2208
32.3 token — Constants used with Python parse trees . . . 2208
32.4 keyword — Testing for Python keywords . . . 2212
32.5 tokenize — Tokenizer for Python source . . . 2212
32.5.1 Tokenizing Input . . . 2213
32.5.2 Command-Line Usage . . . 2214
32.5.3 Examples . . . 2214
32.6 tabnanny — Detection of ambiguous indentation . . . 2216
32.7 pyclbr — Python module browser support . . . 2217
32.7.1 Function Objects . . . 2218
32.7.2 Class Objects . . . 2218
32.8 py_compile — Compile Python source files . . . 2219
32.8.1 Command-Line Interface . . . 2220
32.9 compileall — Byte-compile Python libraries . . . 2221
32.9.1 Command-line use . . . 2221
32.9.2 Public functions . . . 2223
32.10 dis — Disassembler for Python bytecode . . . 2225
32.10.1 Command-line interface . . . 2226
32.10.2 Bytecode analysis . . . 2226
32.10.3 Analysis functions . . . 2227
32.10.4 Python Bytecode Instructions . . . 2230
32.10.5 Opcode collections . . . 2247
32.11 pickletools — Tools for pickle developers . . . 2248
32.11.1 Command line usage . . . 2249
32.11.2 Programmatic Interface . . . 2249

33 MS Windows Specific Services 2251
33.1 msvcrt — Useful routines from the MS VC++ runtime . . . 2251
33.1.1 File Operations . . . 2251
33.1.2 Console I/O . . . 2252
33.1.3 Other Functions . . . 2253
33.2 winreg — Windows registry access . . . 2254
33.2.1 Functions . . . 2254
33.2.2 Constants . . . 2260
33.2.3 Registry Handle Objects . . . 2262
33.3 winsound — Sound-playing interface for Windows . . . 2263

34 Unix Specific Services **2267**
34.1 posix — The most common POSIX system calls . . . 2267
34.1.1 Large File Support . . . 2267
34.1.2 Notable Module Contents . . . 2268
34.2 pwd — The password database . . . 2268
34.3 grp — The group database . . . 2269
34.4 termios — POSIX style tty control . . . 2270
34.4.1 Example . . . 2271
34.5 tty — Terminal control functions . . . 2271
34.6 pty — Pseudo-terminal utilities . . . 2272
34.6.1 Example . . . 2273
34.7 fcntl — The fcntl and ioctl system calls . . . 2274
34.8 resource — Resource usage information . . . 2277
34.8.1 Resource Limits . . . 2277
34.8.2 Resource Usage . . . 2280
34.9 syslog — Unix syslog library routines . . . 2281
34.9.1 Examples . . . 2284

35 Modules command-line interface (CLI) **2285**

36 Superseded Modules **2287**
36.1 getopt — C-style parser for command line options . . . 2287
36.2 optparse — Parser for command line options . . . 2289
36.2.1 Background . . . 2291
36.2.2 Tutorial . . . 2293
36.2.3 Reference Guide . . . 2300
36.2.4 Option Callbacks . . . 2311
36.2.5 Extending optparse . . . 2315
36.2.6 Exceptions . . . 2318

37 Security Considerations **2319**

A Glossary **2321**

B About these documents **2339**
B.1 Contributors to the Python Documentation . . . 2339

C History and License **2341**
C.1 History of the software . . . 2341
C.2 Terms and conditions for accessing or otherwise using Python . . . 2342
C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.13.0rc1 . . . 2342
C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 . . . 2343
C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 . . . 2344
C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 . . . 2345
C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0rc1 DOCUMENTATION . . . 2345
C.3 Licenses and Acknowledgements for Incorporated Software . . . 2346
C.3.1 Mersenne Twister . . . 2346
C.3.2 Sockets . . . 2347
C.3.3 Asynchronous socket services . . . 2347
C.3.4 Cookie management . . . 2348
C.3.5 Execution tracing . . . 2348
C.3.6 UUencode and UUdecode functions . . . 2349
C.3.7 XML Remote Procedure Calls . . . 2349
C.3.8 test_epoll . . . 2350

C.3.9 Select kqueue . . . 2350
C.3.10 SipHash24 . . . 2351
C.3.11 strtod and dtoa . . . 2351
C.3.12 OpenSSL . . . 2352
C.3.13 expat . . . 2355
C.3.14 libffi . . . 2356
C.3.15 zlib . . . 2356
C.3.16 cfuhash . . . 2357
C.3.17 libmpdec . . . 2357
C.3.18 W3C C14N test suite . . . 2358
C.3.19 mimalloc . . . 2359
C.3.20 asyncio . . . 2359
C.3.21 Global Unbounded Sequences (GUS) . . . 2360

D Copyright **2361**

Bibliography **2363**

Python Module Index **2365**

Index **2369**

While reference-index describes the exact syntax and semantics of the Python language, this library reference manual describes the standard library that is distributed with Python. It also describes some of the optional components that are commonly included in Python distributions.

Python's standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed below. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs.

The Python installers for the Windows platform usually include the entire standard library and often also include many additional components. For Unix-like operating systems Python is normally provided as a collection of packages, so it may be necessary to use the packaging tools provided with the operating system to obtain some or all of the optional components.

In addition to the standard library, there is an active collection of hundreds of thousands of components (from individual programs and modules to packages and entire application development frameworks), available from the Python Package Index.

CHAPTER

ONE

INTRODUCTION

The "Python library" contains several different kinds of components.

It contains data types that would normally be considered part of the "core" of a language, such as numbers and lists. For these types, the Python language core defines the form of literals and places some constraints on their semantics, but does not fully define the semantics. (On the other hand, the language core does define syntactic properties like the spelling and priorities of operators.)

The library also contains built-in functions and exceptions — objects that can be used by all Python code without the need of an `import` statement. Some of these are defined by the core language, but many are not essential for the core semantics and are only described here.

The bulk of the library, however, consists of a collection of modules. There are many ways to dissect this collection. Some modules are written in C and built in to the Python interpreter; others are written in Python and imported in source form. Some modules provide interfaces that are highly specific to Python, like printing a stack trace; some provide interfaces that are specific to particular operating systems, such as access to specific hardware; others provide interfaces that are specific to a particular application domain, like the World Wide Web. Some modules are available in all versions and ports of Python; others are only available when the underlying system supports or requires them; yet others are available only when a particular configuration option was chosen at the time when Python was compiled and installed.

This manual is organized "from the inside out:" it first describes the built-in functions, data types and exceptions, and finally the modules, grouped in chapters of related modules.

This means that if you start reading this manual from the start, and skip to the next chapter when you get bored, you will get a reasonable overview of the available modules and application areas that are supported by the Python library. Of course, you don't *have* to read it like a novel — you can also browse the table of contents (in front of the manual), or look for a specific function, module or term in the index (in the back). And finally, if you enjoy learning about random subjects, you choose a random page number (see module `random`) and read a section or two. Regardless of the order in which you read the sections of this manual, it helps to start with chapter *Built-in Functions*, as the remainder of the manual assumes familiarity with this material.

Let the show begin!

1.1 Notes on availability

- An "Availability: Unix" note means that this function is commonly found on Unix systems. It does not make any claims about its existence on a specific operating system.
- If not separately noted, all functions that claim "Availability: Unix" are supported on macOS and iOS, both of which build on a Unix core.
- If an availability note contains both a minimum Kernel version and a minimum libc version, then both conditions must hold. For example a feature with note *Availability: Linux >= 3.17 with glibc >= 2.27* requires both Linux 3.17 or newer and glibc 2.27 or newer.

1.1.1 WebAssembly platforms

The WebAssembly platforms `wasm32-emscripten` (Emscripten) and `wasm32-wasi` (WASI) provide a subset of POSIX APIs. WebAssembly runtimes and browsers are sandboxed and have limited access to the host and external resources. Any Python standard library module that uses processes, threading, networking, signals, or other forms of inter-process communication (IPC), is either not available or may not work as on other Unix-like systems. File I/O, file system, and Unix permission-related functions are restricted, too. Emscripten does not permit blocking I/O. Other blocking operations like *sleep()* block the browser event loop.

The properties and behavior of Python on WebAssembly platforms depend on the Emscripten-SDK or WASI-SDK version, WASM runtimes (browser, NodeJS, wasmtime), and Python build time flags. WebAssembly, Emscripten, and WASI are evolving standards; some features like networking may be supported in the future.

For Python in the browser, users should consider Pyodide or PyScript. PyScript is built on top of Pyodide, which itself is built on top of CPython and Emscripten. Pyodide provides access to browsers' JavaScript and DOM APIs as well as limited networking capabilities with JavaScript's `XMLHttpRequest` and `Fetch` APIs.

- Process-related APIs are not available or always fail with an error. That includes APIs that spawn new processes (*fork()*, *execve()*), wait for processes (*waitpid()*), send signals (*kill()*), or otherwise interact with processes. The *subprocess* is importable but does not work.
- The *socket* module is available, but is limited and behaves differently from other platforms. On Emscripten, sockets are always non-blocking and require additional JavaScript code and helpers on the server to proxy TCP through WebSockets; see Emscripten Networking for more information. WASI snapshot preview 1 only permits sockets from an existing file descriptor.
- Some functions are stubs that either don't do anything and always return hardcoded values.
- Functions related to file descriptors, file permissions, file ownership, and links are limited and don't support some operations. For example, WASI does not permit symlinks with absolute file names.

1.1.2 iOS

iOS is, in most respects, a POSIX operating system. File I/O, socket handling, and threading all behave as they would on any POSIX operating system. However, there are several major differences between iOS and other POSIX systems.

- iOS can only use Python in "embedded" mode. There is no Python REPL, and no ability to execute binaries that are part of the normal Python developer experience, such as **pip**. To add Python code to your iOS app, you must use the Python embedding API to add a Python interpreter to an iOS app created with Xcode. See the iOS usage guide for more details.
- An iOS app cannot use any form of subprocessing, background processing, or inter-process communication. If an iOS app attempts to create a subprocess, the process creating the subprocess will either lock up, or crash. An iOS app has no visibility of other applications that are running, nor any ability to communicate with other running applications, outside of the iOS-specific APIs that exist for this purpose.
- iOS apps have limited access to modify system resources (such as the system clock). These resources will often be *readable*, but attempts to modify those resources will usually fail.
- iOS apps have a limited concept of console input and output. `stdout` and `stderr` *exist*, and content written to `stdout` and `stderr` will be visible in logs when running in Xcode, but this content *won't* be recorded in the system log. If a user who has installed your app provides their app logs as a diagnostic aid, they will not include any detail written to `stdout` or `stderr`.

  iOS apps have no concept of `stdin` at all. While iOS apps can have a keyboard, this is a software feature, not something that is attached to `stdin`.

  As a result, Python library that involve console manipulation (such as *curses* and *readline*) are not available on iOS.

CHAPTER

TWO

BUILT-IN FUNCTIONS

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

Built-in Functions

A
abs()
aiter()
all()
anext()
any()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

abs(*x*)

Return the absolute value of a number. The argument may be an integer, a floating-point number, or an object implementing `__abs__()`. If the argument is a complex number, its magnitude is returned.

aiter(*async_iterable*)

Return an *asynchronous iterator* for an *asynchronous iterable*. Equivalent to calling `x.__aiter__()`.

Note: Unlike *iter()*, *aiter()* has no 2-argument variant.

Added in version 3.10.

all(*iterable*)

Return `True` if all elements of the *iterable* are true (or if the iterable is empty). Equivalent to:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

awaitable anext(*async_iterator*)

awaitable anext(*async_iterator*, *default*)

When awaited, return the next item from the given *asynchronous iterator*, or *default* if given and the iterator is exhausted.

This is the async variant of the *next()* builtin, and behaves similarly.

This calls the `__anext__()` method of *async_iterator*, returning an *awaitable*. Awaiting this returns the next value of the iterator. If *default* is given, it is returned if the iterator is exhausted, otherwise *StopAsyncIteration* is raised.

Added in version 3.10.

any(*iterable*)

Return `True` if any element of the *iterable* is true. If the iterable is empty, return `False`. Equivalent to:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

ascii(*object*)

As *repr()*, return a string containing a printable representation of an object, but escape the non-ASCII characters in the string returned by *repr()* using `\x`, `\u`, or `\U` escapes. This generates a string similar to that returned by *repr()* in Python 2.

bin(*x*)

Convert an integer number to a binary string prefixed with "0b". The result is a valid Python expression. If *x* is not a Python *int* object, it has to define an `__index__()` method that returns an integer. Some examples:

```
>>> bin(3)
'0b11'
>>> bin(-10)
'-0b1010'
```

If the prefix "0b" is desired or not, you can use either of the following ways.

```
>>> format(14, '#b'), format(14, 'b')
('0b1110', '1110')
>>> f'{14:#b}', f'{14:b}'
('0b1110', '1110')
```

See also *format()* for more information.

class bool(*object=False*, */*)

Return a Boolean value, i.e. one of `True` or `False`. The argument is converted using the standard *truth testing procedure*. If the argument is false or omitted, this returns `False`; otherwise, it returns `True`. The *bool* class is a subclass of *int* (see *Numeric Types — int, float, complex*). It cannot be subclassed further. Its only instances are `False` and `True` (see *Boolean Type - bool*).

Changed in version 3.7: The parameter is now positional-only.

breakpoint(**args*, ***kws*)

This function drops you into the debugger at the call site. Specifically, it calls *sys.breakpointhook()*, passing `args` and `kws` straight through. By default, `sys.breakpointhook()` calls *pdb.set_trace()* expecting no arguments. In this case, it is purely a convenience function so you don't have to explicitly import *pdb* or type as much code to enter the debugger. However, *sys.breakpointhook()* can be set to some other function and *breakpoint()* will automatically call that, allowing you to drop into the debugger of choice. If *sys.breakpointhook()* is not accessible, this function will raise *RuntimeError*.

By default, the behavior of *breakpoint()* can be changed with the `PYTHONBREAKPOINT` environment variable. See *sys.breakpointhook()* for usage details.

Note that this is not guaranteed if *sys.breakpointhook()* has been replaced.

Raises an *auditing event* `builtins.breakpoint` with argument `breakpointhook`.

Added in version 3.7.

class bytearray(*source=b''*)

class bytearray(*source*, *encoding*)

class bytearray(*source*, *encoding*, *errors*)

Return a new array of bytes. The *bytearray* class is a mutable sequence of integers in the range 0 <= x < 256. It has most of the usual methods of mutable sequences, described in *Mutable Sequence Types*, as well as most methods that the *bytes* type has, see *Bytes and Bytearray Operations*.

The optional *source* parameter can be used to initialize the array in a few different ways:

- If it is a *string*, you must also give the *encoding* (and optionally, *errors*) parameters; *bytearray()* then converts the string to bytes using *str.encode()*.
- If it is an *integer*, the array will have that size and will be initialized with null bytes.
- If it is an object conforming to the buffer interface, a read-only buffer of the object will be used to initialize the bytes array.
- If it is an *iterable*, it must be an iterable of integers in the range `0 <= x < 256`, which are used as the initial contents of the array.

Without an argument, an array of size 0 is created.

See also *Binary Sequence Types — bytes, bytearray, memoryview* and *Bytearray Objects*.

class bytes(*source=b''*)

class bytes(*source*, *encoding*)

class bytes(*source*, *encoding*, *errors*)

Return a new "bytes" object which is an immutable sequence of integers in the range `0 <= x < 256`. *bytes* is an immutable version of *bytearray* – it has the same non-mutating methods and the same indexing and slicing behavior.

Accordingly, constructor arguments are interpreted as for *bytearray()*.

Bytes objects can also be created with literals, see strings.

See also *Binary Sequence Types — bytes, bytearray, memoryview*, *Bytes Objects*, and *Bytes and Bytearray Operations*.

callable(*object*)

Return *True* if the *object* argument appears callable, *False* if not. If this returns `True`, it is still possible that a call fails, but if it is `False`, calling *object* will never succeed. Note that classes are callable (calling a class returns a new instance); instances are callable if their class has a `__call__()` method.

Added in version 3.2: This function was first removed in Python 3.0 and then brought back in Python 3.2.

chr(*i*)

Return the string representing a character whose Unicode code point is the integer *i*. For example, `chr(97)` returns the string `'a'`, while `chr(8364)` returns the string `'€'`. This is the inverse of *ord()*.

The valid range for the argument is from 0 through 1,114,111 (0x10FFFF in base 16). *ValueError* will be raised if *i* is outside that range.

@classmethod

Transform a method into a class method.

A class method receives the class as an implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```
class C:
    @classmethod
    def f(cls, arg1, arg2): ...
```

The `@classmethod` form is a function *decorator* – see function for details.

A class method can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see *staticmethod()* in this section. For more information on class methods, see types.

Changed in version 3.9: Class methods can now wrap other *descriptors* such as *property()*.

Changed in version 3.10: Class methods now inherit the method attributes (`__module__`, `__name__`, `__qualname__`, `__doc__` and `__annotations__`) and have a new `__wrapped__` attribute.

Deprecated since version 3.11, removed in version 3.13: Class methods can no longer wrap other *descriptors* such as *property()*.

compile(*source*, *filename*, *mode*, *flags=0*, *dont_inherit=False*, *optimize=-1*)

Compile the *source* into a code or AST object. Code objects can be executed by *exec()* or *eval()*. *source* can either be a normal string, a byte string, or an AST object. Refer to the *ast* module documentation for information on how to work with AST objects.

The *filename* argument should give the file from which the code was read; pass some recognizable value if it wasn't read from a file (`'<string>'` is commonly used).

The *mode* argument specifies what kind of code must be compiled; it can be `'exec'` if *source* consists of a sequence of statements, `'eval'` if it consists of a single expression, or `'single'` if it consists of a single

interactive statement (in the latter case, expression statements that evaluate to something other than `None` will be printed).

The optional arguments *flags* and *dont_inherit* control which *compiler options* should be activated and which future features should be allowed. If neither is present (or both are zero) the code is compiled with the same flags that affect the code that is calling `compile()`. If the *flags* argument is given and *dont_inherit* is not (or is zero) then the compiler options and the future statements specified by the *flags* argument are used in addition to those that would be used anyway. If *dont_inherit* is a non-zero integer then the *flags* argument is it – the flags (future features and compiler options) in the surrounding code are ignored.

Compiler options and future statements are specified by bits which can be bitwise ORed together to specify multiple options. The bitfield required to specify a given future feature can be found as the `compiler_flag` attribute on the `_Feature` instance in the `__future__` module. *Compiler flags* can be found in `ast` module, with `PyCF_` prefix.

The argument *optimize* specifies the optimization level of the compiler; the default value of `-1` selects the optimization level of the interpreter as given by `-O` options. Explicit levels are `0` (no optimization; `__debug__` is true), `1` (asserts are removed, `__debug__` is false) or `2` (docstrings are removed too).

This function raises `SyntaxError` if the compiled source is invalid, and `ValueError` if the source contains null bytes.

If you want to parse Python code into its AST representation, see `ast.parse()`.

Raises an *auditing event* `compile` with arguments `source` and `filename`. This event may also be raised by implicit compilation.

Note

When compiling a string with multi-line code in `'single'` or `'eval'` mode, input must be terminated by at least one newline character. This is to facilitate detection of incomplete and complete statements in the `code` module.

Warning

It is possible to crash the Python interpreter with a sufficiently large/complex string when compiling to an AST object due to stack depth limitations in Python's AST compiler.

Changed in version 3.2: Allowed use of Windows and Mac newlines. Also, input in `'exec'` mode does not have to end in a newline anymore. Added the *optimize* parameter.

Changed in version 3.5: Previously, `TypeError` was raised when null bytes were encountered in *source*.

Added in version 3.8: `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` can now be passed in flags to enable support for top-level `await`, `async for`, and `async with`.

class complex(*number=0*, */*)

class complex(*string*, */*)

class complex(*real=0*, *imag=0*)

Convert a single string or number to a complex number, or create a complex number from real and imaginary parts.

Examples:

```
>>> complex('+1.23')
(1.23+0j)
>>> complex('-4.5j')
-4.5j
>>> complex('-1.23+4.5j')
(-1.23+4.5j)
>>> complex('\t( -1.23+4.5J )\n')
(-1.23+4.5j)
>>> complex('-Infinity+NaNj')
(-inf+nanj)
>>> complex(1.23)
(1.23+0j)
>>> complex(imag=-4.5)
-4.5j
>>> complex(-1.23, 4.5)
(-1.23+4.5j)
```

If the argument is a string, it must contain either a real part (in the same format as for *float()*) or an imaginary part (in the same format but with a `'j'` or `'J'` suffix), or both real and imaginary parts (the sign of the imaginary part is mandatory in this case). The string can optionally be surrounded by whitespaces and the round parentheses `'('` and `')'`, which are ignored. The string must not contain whitespace between `'+'`, `'-'`, the `'j'` or `'J'` suffix, and the decimal number. For example, `complex('1+2j')` is fine, but `complex('1 + 2j')` raises *ValueError*. More precisely, the input must conform to the *complexvalue* production rule in the following grammar, after parentheses and leading and trailing whitespace characters are removed:

```
complexvalue   ::=   floatvalue |
                     floatvalue ("j" | "J") |
                     floatvalue sign absfloatvalue ("j" | "J")
```

If the argument is a number, the constructor serves as a numeric conversion like *int* and *float*. For a general Python object `x`, `complex(x)` delegates to `x.__complex__()`. If `__complex__()` is not defined then it falls back to `__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

If two arguments are provided or keyword arguments are used, each argument may be any numeric type (including complex). If both arguments are real numbers, return a complex number with the real component *real* and the imaginary component *imag*. If both arguments are complex numbers, return a complex number with the real component `real.real-imag.imag` and the imaginary component `real.imag+imag.real`. If one of arguments is a real number, only its real component is used in the above expressions.

If all arguments are omitted, returns `0j`.

The complex type is described in *Numeric Types — int, float, complex*.

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.8: Falls back to `__index__()` if `__complex__()` and `__float__()` are not defined.

delattr(*object*, *name*)

This is a relative of *setattr()*. The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, `delattr(x, 'foobar')` is equivalent to `del x.foobar`. *name* need not be a Python identifier (see *setattr()*).

class dict(***kwarg*)

class dict(*mapping*, ***kwarg*)

class dict(*iterable*, ***kwarg*)

Create a new dictionary. The *dict* object is the dictionary class. See *dict* and *Mapping Types — dict* for documentation about this class.

For other containers see the built-in *list*, *set*, and *tuple* classes, as well as the *collections* module.

dir()

dir(*object*)

Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object.

If the object has a method named `__dir__()`, this method will be called and must return the list of attributes. This allows objects that implement a custom `__getattr__()` or `__getattribute__()` function to customize the way *dir()* reports their attributes.

If the object does not provide `__dir__()`, the function tries its best to gather information from the object's *__dict__* attribute, if defined, and from its type object. The resulting list is not necessarily complete and may be inaccurate when the object has a custom `__getattr__()`.

The default *dir()* mechanism behaves differently with different types of objects, as it attempts to produce the most relevant, rather than complete, information:

- If the object is a module object, the list contains the names of the module's attributes.
- If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases.
- Otherwise, the list contains the object's attributes' names, the names of its class's attributes, and recursively of the attributes of its class's base classes.

The resulting list is sorted alphabetically. For example:

```
>>> import struct
>>> dir()   # show the names in the module namespace
['__builtins__', '__name__', 'struct']
>>> dir(struct)   # show the names in the struct module
['Struct', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__initializing__', '__loader__', '__name__', '__package__',
 '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']
...
>>> s = Shape()
>>> dir(s)
['area', 'location', 'perimeter']
```

Note

Because *dir()* is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

divmod(*a*, *b*)

Take two (non-complex) numbers as arguments and return a pair of numbers consisting of their quotient and

remainder when using integer division. With mixed operand types, the rules for binary arithmetic operators apply. For integers, the result is the same as `(a // b, a % b)`. For floating-point numbers the result is `(q, a % b)`, where *q* is usually `math.floor(a / b)` but may be 1 less than that. In any case `q * b + a % b` is very close to *a*, if `a % b` is non-zero it has the same sign as *b*, and `0 <= abs(a % b) < abs(b)`.

enumerate(*iterable*, *start=0*)

Return an enumerate object. *iterable* must be a sequence, an *iterator*, or some other object which supports iteration. The *__next__()* method of the iterator returned by *enumerate()* returns a tuple containing a count (from *start* which defaults to 0) and the values obtained from iterating over *iterable*.

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

Equivalent to:

```
def enumerate(iterable, start=0):
    n = start
    for elem in iterable:
        yield n, elem
        n += 1
```

eval(*source*, */*, *globals=None*, *locals=None*)

Parameters

- **source** (*str* | code object) – A Python expression.
- **globals** (*dict* | None) – The global namespace (default: None).
- **locals** (*mapping* | None) – The local namespace (default: None).

Returns

The result of the evaluated expression.

Raises

Syntax errors are reported as exceptions.

The *expression* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* mappings as global and local namespace. If the *globals* dictionary is present and does not contain a value for the key __builtins__, a reference to the dictionary of the built-in module *builtins* is inserted under that key before *expression* is parsed. That way you can control what builtins are available to the executed code by inserting your own __builtins__ dictionary into *globals* before passing it to *eval()*. If the *locals* mapping is omitted it defaults to the *globals* dictionary. If both mappings are omitted, the expression is executed with the *globals* and *locals* in the environment where *eval()* is called. Note, *eval()* will only have access to the *nested scopes* (non-locals) in the enclosing environment if they are already referenced in the scope that is calling *eval()* (e.g. via a `nonlocal` statement).

Example:

```
>>> x = 1
>>> eval('x+1')
2
```

This function can also be used to execute arbitrary code objects (such as those created by *compile()*). In this case, pass a code object instead of a string. If the code object has been compiled with `'exec'` as the *mode* argument, *eval()*'s return value will be None.

Hints: dynamic execution of statements is supported by the *exec()* function. The *globals()* and *locals()* functions return the current global and local dictionary, respectively, which may be useful to pass around for use by *eval()* or *exec()*.

If the given source is a string, then leading and trailing spaces and tabs are stripped.

See *ast.literal_eval()* for a function that can safely evaluate strings with expressions containing only literals.

Raises an *auditing event* `exec` with the code object as the argument. Code compilation events may also be raised.

Changed in version 3.13: The *globals* and *locals* arguments can now be passed as keywords.

Changed in version 3.13: The semantics of the default *locals* namespace have been adjusted as described for the *locals()* builtin.

exec(*source*, */*, *globals=None*, *locals=None*, ***, *closure=None*)

This function supports dynamic execution of Python code. *source* must be either a string or a code object. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs).[1] If it is a code object, it is simply executed. In all cases, the code that's executed is expected to be valid as file input (see the section file-input in the Reference Manual). Be aware that the `nonlocal`, `yield`, and `return` statements may not be used outside of function definitions even within the context of code passed to the *exec()* function. The return value is `None`.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only *globals* is provided, it must be a dictionary (and not a subclass of dictionary), which will be used for both the global and the local variables. If *globals* and *locals* are given, they are used for the global and local variables, respectively. If provided, *locals* can be any mapping object. Remember that at the module level, globals and locals are the same dictionary.

> **Note**
>
> When `exec` gets two separate objects as *globals* and *locals*, the code will be executed as if it were embedded in a class definition. This means functions and classes defined in the executed code will not be able to access variables assigned at the top level (as the "top level" variables are treated as class variables in a class definition).

If the *globals* dictionary does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module *builtins* is inserted under that key. That way you can control what builtins are available to the executed code by inserting your own `__builtins__` dictionary into *globals* before passing it to *exec()*.

The *closure* argument specifies a closure–a tuple of cellvars. It's only valid when the *object* is a code object containing free variables. The length of the tuple must exactly match the number of free variables referenced by the code object.

Raises an *auditing event* `exec` with the code object as the argument. Code compilation events may also be raised.

> **Note**
>
> The built-in functions *globals()* and *locals()* return the current global and local namespace, respectively, which may be useful to pass around for use as the second and third argument to *exec()*.

[1] Note that the parser only accepts the Unix-style end of line convention. If you are reading the code from a file, make sure to use newline conversion mode to convert Windows or Mac-style newlines.

> **Note**
>
> The default *locals* act as described for function `locals()` below. Pass an explicit *locals* dictionary if you need to see effects of the code on *locals* after function `exec()` returns.

Changed in version 3.11: Added the *closure* parameter.

Changed in version 3.13: The *globals* and *locals* arguments can now be passed as keywords.

Changed in version 3.13: The semantics of the default *locals* namespace have been adjusted as described for the `locals()` builtin.

filter(*function*, *iterable*)

Construct an iterator from those elements of *iterable* for which *function* is true. *iterable* may be either a sequence, a container which supports iteration, or an iterator. If *function* is `None`, the identity function is assumed, that is, all elements of *iterable* that are false are removed.

Note that `filter(function, iterable)` is equivalent to the generator expression `(item for item in iterable if function(item))` if function is not `None` and `(item for item in iterable if item)` if function is `None`.

See `itertools.filterfalse()` for the complementary function that returns elements of *iterable* for which *function* is false.

class float(*number=0.0*, */*)

class float(*string*, */*)

Return a floating-point number constructed from a number or a string.

Examples:

```
>>> float('+1.23')
1.23
>>> float('   -12345\n')
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf
```

If the argument is a string, it should contain a decimal number, optionally preceded by a sign, and optionally embedded in whitespace. The optional sign may be `'+'` or `'-'`; a `'+'` sign has no effect on the value produced. The argument may also be a string representing a NaN (not-a-number), or positive or negative infinity. More precisely, the input must conform to the *floatvalue* production rule in the following grammar, after leading and trailing whitespace characters are removed:

```
sign           ::=  "+" | "-"
infinity       ::=  "Infinity" | "inf"
nan            ::=  "nan"
digit          ::=  <a Unicode decimal digit, i.e. characters in Unicode general category
digitpart      ::=  digit (["_"] digit)*
number         ::=  [digitpart] "." digitpart | digitpart ["."]
exponent       ::=  ("e" | "E") [sign] digitpart
floatnumber    ::=  number [exponent]
absfloatvalue  ::=  floatnumber | infinity | nan
```

```
floatvalue      ::=   [sign] absfloatvalue
```

Case is not significant, so, for example, "inf", "Inf", "INFINITY", and "iNfINity" are all acceptable spellings for positive infinity.

Otherwise, if the argument is an integer or a floating-point number, a floating-point number with the same value (within Python's floating-point precision) is returned. If the argument is outside the range of a Python float, an `OverflowError` will be raised.

For a general Python object `x`, `float(x)` delegates to `x.__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

If no argument is given, `0.0` is returned.

The float type is described in *Numeric Types — int, float, complex*.

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.7: The parameter is now positional-only.

Changed in version 3.8: Falls back to `__index__()` if `__float__()` is not defined.

format(*value*, *format_spec=''*)

Convert a *value* to a "formatted" representation, as controlled by *format_spec*. The interpretation of *format_spec* will depend on the type of the *value* argument; however, there is a standard formatting syntax that is used by most built-in types: *Format Specification Mini-Language*.

The default *format_spec* is an empty string which usually gives the same effect as calling `str(value)`.

A call to `format(value, format_spec)` is translated to `type(value).__format__(value, format_spec)` which bypasses the instance dictionary when searching for the value's `__format__()` method. A `TypeError` exception is raised if the method search reaches `object` and the *format_spec* is non-empty, or if either the *format_spec* or the return value are not strings.

Changed in version 3.4: `object().__format__(format_spec)` raises `TypeError` if *format_spec* is not an empty string.

class frozenset(*iterable=set()*)

Return a new `frozenset` object, optionally with elements taken from *iterable*. `frozenset` is a built-in class. See `frozenset` and *Set Types — set, frozenset* for documentation about this class.

For other containers see the built-in `set`, `list`, `tuple`, and `dict` classes, as well as the `collections` module.

getattr(*object*, *name*)

getattr(*object*, *name*, *default*)

Return the value of the named attribute of *object*. *name* must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, `getattr(x, 'foobar')` is equivalent to `x.foobar`. If the named attribute does not exist, *default* is returned if provided, otherwise `AttributeError` is raised. *name* need not be a Python identifier (see `setattr()`).

> **Note**
>
> Since private name mangling happens at compilation time, one must manually mangle a private attribute's (attributes with two leading underscores) name in order to retrieve it with `getattr()`.

globals()

Return the dictionary implementing the current module namespace. For code within functions, this is set when the function is defined and remains the same regardless of where the function is called.

hasattr(*object*, *name*)

The arguments are an object and a string. The result is `True` if the string is the name of one of the object's attributes, `False` if not. (This is implemented by calling `getattr(object, name)` and seeing whether it raises an *AttributeError* or not.)

hash(*object*)

Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

> **Note**
>
> For objects with custom `__hash__()` methods, note that *hash()* truncates the return value based on the bit width of the host machine.

help()

help(*request*)

Invoke the built-in help system. (This function is intended for interactive use.) If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is generated.

Note that if a slash(/) appears in the parameter list of a function when invoking *help()*, it means that the parameters prior to the slash are positional-only. For more info, see the FAQ entry on positional-only parameters.

This function is added to the built-in namespace by the *site* module.

Changed in version 3.4: Changes to *pydoc* and *inspect* mean that the reported signatures for callables are now more comprehensive and consistent.

hex(*x*)

Convert an integer number to a lowercase hexadecimal string prefixed with "0x". If *x* is not a Python *int* object, it has to define an `__index__()` method that returns an integer. Some examples:

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
```

If you want to convert an integer number to an uppercase or lower hexadecimal string with prefix or not, you can use either of the following ways:

```
>>> '%#x' % 255, '%x' % 255, '%X' % 255
('0xff', 'ff', 'FF')
>>> format(255, '#x'), format(255, 'x'), format(255, 'X')
('0xff', 'ff', 'FF')
>>> f'{255:#x}', f'{255:x}', f'{255:X}'
('0xff', 'ff', 'FF')
```

See also *format()* for more information.

See also *int()* for converting a hexadecimal string to an integer using a base of 16.

> **Note**

To obtain a hexadecimal string representation for a float, use the *float.hex()* method.

id(*object*)

Return the "identity" of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same *id()* value.

CPython implementation detail: This is the address of the object in memory.

Raises an *auditing event* `builtins.id` with argument `id`.

input()

input(*prompt*)

If the *prompt* argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, *EOFError* is raised. Example:

```
>>> s = input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

If the *readline* module was loaded, then *input()* will use it to provide elaborate line editing and history features.

Raises an *auditing event* `builtins.input` with argument `prompt` before reading input

Raises an *auditing event* `builtins.input/result` with the result after successfully reading input.

class int(*number=0*, */*)

class int(*string*, */*, *base=10*)

Return an integer object constructed from a number or a string, or return `0` if no arguments are given.

Examples:

```
>>> int(123.45)
123
>>> int('123')
123
>>> int('   -12_345\n')
-12345
>>> int('FACE', 16)
64206
>>> int('0xface', 0)
64206
>>> int('01110011', base=2)
115
```

If the argument defines `__int__()`, `int(x)` returns `x.__int__()`. If the argument defines `__index__()`, it returns `x.__index__()`. If the argument defines `__trunc__()`, it returns `x.__trunc__()`. For floating-point numbers, this truncates towards zero.

If the argument is not a number or if *base* is given, then it must be a string, *bytes*, or *bytearray* instance representing an integer in radix *base*. Optionally, the string can be preceded by + or − (with no space in between), have leading zeros, be surrounded by whitespace, and have single underscores interspersed between digits.

A base-n integer string contains digits, each representing a value from 0 to n-1. The values 0–9 can be represented by any Unicode decimal digit. The values 10–35 can be represented by `a` to `z` (or `A` to `Z`). The default *base* is 10. The allowed bases are 0 and 2–36. Base-2, -8, and -16 strings can be optionally prefixed with `0b`/`0B`, `0o`/`0O`, or `0x`/`0X`, as with integer literals in code. For base 0, the string is interpreted in a similar way to an integer literal in code, in that the actual base is 2, 8, 10, or 16 as determined by the prefix. Base 0 also disallows leading zeros: `int('010', 0)` is not legal, while `int('010')` and `int('010', 8)` are.

The integer type is described in *Numeric Types — int, float, complex*.

Changed in version 3.4: If *base* is not an instance of *int* and the *base* object has a `base.__index__` method, that method is called to obtain an integer for the base. Previous versions used `base.__int__` instead of `base.__index__`.

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.7: The first parameter is now positional-only.

Changed in version 3.8: Falls back to `__index__()` if `__int__()` is not defined.

Changed in version 3.11: The delegation to `__trunc__()` is deprecated.

Changed in version 3.11: *int* string inputs and string representations can be limited to help avoid denial of service attacks. A *ValueError* is raised when the limit is exceeded while converting a string to an *int* or when converting an *int* into a string would exceed the limit. See the *integer string conversion length limitation* documentation.

isinstance(*object*, *classinfo*)

Return `True` if the *object* argument is an instance of the *classinfo* argument, or of a (direct, indirect, or *virtual*) subclass thereof. If *object* is not an object of the given type, the function always returns `False`. If *classinfo* is a tuple of type objects (or recursively, other such tuples) or a *Union Type* of multiple types, return `True` if *object* is an instance of any of the types. If *classinfo* is not a type or tuple of types and such tuples, a *TypeError* exception is raised. *TypeError* may not be raised for an invalid type if an earlier check succeeds.

Changed in version 3.10: *classinfo* can be a *Union Type*.

issubclass(*class*, *classinfo*)

Return `True` if *class* is a subclass (direct, indirect, or *virtual*) of *classinfo*. A class is considered a subclass of itself. *classinfo* may be a tuple of class objects (or recursively, other such tuples) or a *Union Type*, in which case return `True` if *class* is a subclass of any entry in *classinfo*. In any other case, a *TypeError* exception is raised.

Changed in version 3.10: *classinfo* can be a *Union Type*.

iter(*object*)

iter(*object*, *sentinel*)

Return an *iterator* object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, *object* must be a collection object which supports the *iterable* protocol (the `__iter__()` method), or it must support the sequence protocol (the `__getitem__()` method with integer arguments starting at `0`). If it does not support either of those protocols, *TypeError* is raised. If the second argument, *sentinel*, is given, then *object* must be a callable object. The iterator created in this case will call *object* with no arguments for each call to its *__next__()* method; if the value returned is equal to *sentinel*, *StopIteration* will be raised, otherwise the value will be returned.

See also *Iterator Types*.

One useful application of the second form of *iter()* is to build a block-reader. For example, reading fixed-width blocks from a binary database file until the end of file is reached:

```
from functools import partial
with open('mydata.db', 'rb') as f:
    for block in iter(partial(f.read, 64), b''):
        process_block(block)
```

len(*s*)

Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

CPython implementation detail: `len` raises *`OverflowError`* on lengths larger than *`sys.maxsize`*, such as *`range(2 ** 100)`*.

class list

class list(*iterable*)

Rather than being a function, *`list`* is actually a mutable sequence type, as documented in *Lists* and *Sequence Types — list, tuple, range*.

locals()

Return a mapping object representing the current local symbol table, with variable names as the keys, and their currently bound references as the values.

At module scope, as well as when using *`exec()`* or *`eval()`* with a single namespace, this function returns the same namespace as *`globals()`*.

At class scope, it returns the namespace that will be passed to the metaclass constructor.

When using `exec()` or `eval()` with separate local and global arguments, it returns the local namespace passed in to the function call.

In all of the above cases, each call to `locals()` in a given frame of execution will return the *same* mapping object. Changes made through the mapping object returned from `locals()` will be visible as assigned, reassigned, or deleted local variables, and assigning, reassigning, or deleting local variables will immediately affect the contents of the returned mapping object.

In an *optimized scope* (including functions, generators, and coroutines), each call to `locals()` instead returns a fresh dictionary containing the current bindings of the function's local variables and any nonlocal cell references. In this case, name binding changes made via the returned dict are *not* written back to the corresponding local variables or nonlocal cell references, and assigning, reassigning, or deleting local variables and nonlocal cell references does *not* affect the contents of previously returned dictionaries.

Calling `locals()` as part of a comprehension in a function, generator, or coroutine is equivalent to calling it in the containing scope, except that the comprehension's initialised iteration variables will be included. In other scopes, it behaves as if the comprehension were running as a nested function.

Calling `locals()` as part of a generator expression is equivalent to calling it in a nested generator function.

Changed in version 3.12: The behaviour of `locals()` in a comprehension has been updated as described in **PEP 709**.

Changed in version 3.13: As part of **PEP 667**, the semantics of mutating the mapping objects returned from this function are now defined. The behavior in *optimized scopes* is now as described above. Aside from being defined, the behaviour in other scopes remains unchanged from previous versions.

map(*function*, *iterable*, **iterables*)

Return an iterator that applies *function* to every item of *iterable*, yielding the results. If additional *iterables* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases where the function inputs are already arranged into argument tuples, see *`itertools.starmap()`*.

max(*iterable*, *, *key=None*)

max(*iterable*, *, *default*, *key=None*)

max(*arg1*, *arg2*, **args*, *key=None*)

Return the largest item in an iterable or the largest of two or more arguments.

If one positional argument is provided, it should be an *iterable*. The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned.

There are two optional keyword-only arguments. The *key* argument specifies a one-argument ordering function like that used for `list.sort()`. The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a `ValueError` is raised.

If multiple items are maximal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as `sorted(iterable, key=keyfunc, reverse=True)[0]` and `heapq.nlargest(1, iterable, key=keyfunc)`.

Changed in version 3.4: Added the *default* keyword-only parameter.

Changed in version 3.8: The *key* can be `None`.

class memoryview(*object*)

Return a "memory view" object created from the given argument. See *Memory Views* for more information.

min(*iterable*, *, *key=None*)

min(*iterable*, *, *default*, *key=None*)

min(*arg1*, *arg2*, **args*, *key=None*)

Return the smallest item in an iterable or the smallest of two or more arguments.

If one positional argument is provided, it should be an *iterable*. The smallest item in the iterable is returned. If two or more positional arguments are provided, the smallest of the positional arguments is returned.

There are two optional keyword-only arguments. The *key* argument specifies a one-argument ordering function like that used for `list.sort()`. The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a `ValueError` is raised.

If multiple items are minimal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as `sorted(iterable, key=keyfunc)[0]` and `heapq.nsmallest(1, iterable, key=keyfunc)`.

Changed in version 3.4: Added the *default* keyword-only parameter.

Changed in version 3.8: The *key* can be `None`.

next(*iterator*)

next(*iterator*, *default*)

Retrieve the next item from the *iterator* by calling its `__next__()` method. If *default* is given, it is returned if the iterator is exhausted, otherwise `StopIteration` is raised.

class object

Return a new featureless object. `object` is a base for all classes. It has methods that are common to all instances of Python classes. This function does not accept any arguments.

> **Note**
>
> `object` does *not* have a `__dict__`, so you can't assign arbitrary attributes to an instance of the `object` class.

oct(*x*)

Convert an integer number to an octal string prefixed with "0o". The result is a valid Python expression. If *x* is not a Python `int` object, it has to define an `__index__()` method that returns an integer. For example:

```
>>> oct(8)
'0o10'
>>> oct(-56)
'-0o70'
```

If you want to convert an integer number to an octal string either with the prefix "0o" or not, you can use either of the following ways.

```
>>> '%#o' % 10, '%o' % 10
('0o12', '12')
>>> format(10, '#o'), format(10, 'o')
('0o12', '12')
>>> f'{10:#o}', f'{10:o}'
('0o12', '12')
```

See also *format()* for more information.

open(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)

Open *file* and return a corresponding *file object*. If the file cannot be opened, an *OSError* is raised. See tut-files for more examples of how to use this function.

file is a *path-like object* giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed unless *closefd* is set to `False`.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to `'r'` which means open for reading in text mode. Other common values are `'w'` for writing (truncating the file if it already exists), `'x'` for exclusive creation, and `'a'` for appending (which on *some* Unix systems, means that *all* writes append to the end of the file regardless of the current seek position). In text mode, if *encoding* is not specified the encoding used is platform-dependent: *locale.getencoding()* is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave *encoding* unspecified.) The available modes are:

Character	Meaning
`'r'`	open for reading (default)
`'w'`	open for writing, truncating the file first
`'x'`	open for exclusive creation, failing if the file already exists
`'a'`	open for writing, appending to the end of file if it exists
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	open for updating (reading and writing)

The default mode is `'r'` (open for reading text, a synonym of `'rt'`). Modes `'w+'` and `'w+b'` open and truncate the file. Modes `'r+'` and `'r+b'` open the file with no truncation.

As mentioned in the *Overview*, Python distinguishes between binary and text I/O. Files opened in binary mode (including `'b'` in the *mode* argument) return contents as *bytes* objects without any decoding. In text mode (the default, or when `'t'` is included in the *mode* argument), the contents of the file are returned as *str*, the bytes having been first decoded using a platform-dependent encoding or using the specified *encoding* if given.

Note

Python doesn't depend on the underlying operating system's notion of text files; all the processing is done by Python itself, and is therefore platform-independent.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable when writing in text mode), and an integer > 1 to indicate the size in bytes of a fixed-size chunk buffer. Note that specifying a buffer size this way applies for binary buffered I/O, but `TextIOWrapper` (i.e., files opened with `mode='r+'`) would have another buffering. To disable buffering in `TextIOWrapper`, consider using the `write_through` flag for *io.TextIOWrapper.reconfigure()*. When no *buffering* argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device's "block size" and falling back on *io.DEFAULT_BUFFER_SIZE*. On many systems, the buffer will typically be 4096 or 8192 bytes long.
- "Interactive" text files (files for which *isatty()* returns `True`) use line buffering. Other text files use the policy described above for binary files.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent (whatever *locale.getencoding()* returns), but any *text encoding* supported by Python can be used. See the *codecs* module for the list of supported encodings.

errors is an optional string that specifies how encoding and decoding errors are to be handled—this cannot be used in binary mode. A variety of standard error handlers are available (listed under *Error Handlers*), though any error handling name that has been registered with *codecs.register_error()* is also valid. The standard names include:

- `'strict'` to raise a *ValueError* exception if there is an encoding error. The default value of `None` has the same effect.
- `'ignore'` ignores errors. Note that ignoring encoding errors can lead to data loss.
- `'replace'` causes a replacement marker (such as `'?'`) to be inserted where there is malformed data.
- `'surrogateescape'` will represent any incorrect bytes as low surrogate code units ranging from U+DC80 to U+DCFF. These surrogate code units will then be turned back into the same bytes when the `surrogateescape` error handler is used when writing data. This is useful for processing files in an unknown encoding.
- `'xmlcharrefreplace'` is only supported when writing to a file. Characters not supported by the encoding are replaced with the appropriate XML character reference `&#`*nnn*`;`.
- `'backslashreplace'` replaces malformed data by Python's backslashed escape sequences.
- `'namereplace'` (also only supported when writing) replaces unsupported characters with `\N{...}` escape sequences.

newline determines how to parse newline characters from the stream. It can be `None`, `''`, `'\n'`, `'\r'`, and `'\r\n'`. It works as follows:

- When reading input from the stream, if *newline* is `None`, universal newlines mode is enabled. Lines in the input can end in `'\n'`, `'\r'`, or `'\r\n'`, and these are translated into `'\n'` before being returned to the caller. If it is `''`, universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- When writing output to the stream, if *newline* is `None`, any `'\n'` characters written are translated to the system default line separator, *os.linesep*. If *newline* is `''` or `'\n'`, no translation takes place. If *newline* is any of the other legal values, any `'\n'` characters written are translated to the given string.

If *closefd* is `False` and a file descriptor rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given *closefd* must be `True` (the default); otherwise, an error will be raised.

A custom opener can be used by passing a callable as *opener*. The underlying file descriptor for the file object is then obtained by calling *opener* with (*file*, *flags*). *opener* must return an open file descriptor (passing *os.open* as

opener results in functionality similar to passing `None`).

The newly created file is *non-inheritable*.

The following example uses the *dir_fd* parameter of the *os.open()* function to open a file relative to a given directory:

```
>>> import os
>>> dir_fd = os.open('somedir', os.O_RDONLY)
>>> def opener(path, flags):
...     return os.open(path, flags, dir_fd=dir_fd)
...
>>> with open('spamspam.txt', 'w', opener=opener) as f:
...     print('This will be written to somedir/spamspam.txt', file=f)
...
>>> os.close(dir_fd)  # don't leak a file descriptor
```

The type of *file object* returned by the *open()* function depends on the mode. When *open()* is used to open a file in a text mode (`'w'`, `'r'`, `'wt'`, `'rt'`, etc.), it returns a subclass of *io.TextIOBase* (specifically *io.TextIOWrapper*). When used to open a file in a binary mode with buffering, the returned class is a subclass of *io.BufferedIOBase*. The exact class varies: in read binary mode, it returns an *io.BufferedReader*; in write binary and append binary modes, it returns an *io.BufferedWriter*, and in read/write mode, it returns an *io.BufferedRandom*. When buffering is disabled, the raw stream, a subclass of *io.RawIOBase*, *io.FileIO*, is returned.

See also the file handling modules, such as *fileinput*, *io* (where *open()* is declared), *os*, *os.path*, *tempfile*, and *shutil*.

Raises an *auditing event* `open` with arguments `file`, `mode`, `flags`.

The `mode` and `flags` arguments may have been modified or inferred from the original call.

Changed in version 3.3:

- The *opener* parameter was added.
- The `'x'` mode was added.
- *IOError* used to be raised, it is now an alias of *OSError*.
- *FileExistsError* is now raised if the file opened in exclusive creation mode (`'x'`) already exists.

Changed in version 3.4:

- The file is now non-inheritable.

Changed in version 3.5:

- If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).
- The `'namereplace'` error handler was added.

Changed in version 3.6:

- Support added to accept objects implementing *os.PathLike*.
- On Windows, opening a console buffer may return a subclass of *io.RawIOBase* other than *io.FileIO*.

Changed in version 3.11: The `'U'` mode has been removed.

ord(*c*)

Given a string representing one Unicode character, return an integer representing the Unicode code point of that character. For example, `ord('a')` returns the integer `97` and `ord('€')` (Euro sign) returns `8364`. This is the inverse of *chr()*.

pow(*base*, *exp*, *mod=None*)

Return *base* to the power *exp*; if *mod* is present, return *base* to the power *exp*, modulo *mod* (computed more efficiently than `pow(base, exp) % mod`). The two-argument form `pow(base, exp)` is equivalent to using the power operator: `base**exp`.

The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For *int* operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `pow(10, 2)` returns `100`, but `pow(10, -2)` returns `0.01`. For a negative base of type *int* or *float* and a non-integral exponent, a complex result is delivered. For example, `pow(-9, 0.5)` returns a value close to `3j`. Whereas, for a negative base of type *int* or *float* with an integral exponent, a float result is delivered. For example, `pow(-9, 2.0)` returns `81.0`.

For *int* operands *base* and *exp*, if *mod* is present, *mod* must also be of integer type and *mod* must be nonzero. If *mod* is present and *exp* is negative, *base* must be relatively prime to *mod*. In that case, `pow(inv_base, -exp, mod)` is returned, where *inv_base* is an inverse to *base* modulo *mod*.

Here's an example of computing an inverse for `38` modulo `97`:

```
>>> pow(38, -1, mod=97)
23
>>> 23 * 38 % 97 == 1
True
```

Changed in version 3.8: For *int* operands, the three-argument form of `pow` now allows the second argument to be negative, permitting computation of modular inverses.

Changed in version 3.8: Allow keyword arguments. Formerly, only positional arguments were supported.

print(**objects*, *sep=' '*, *end='\n'*, *file=None*, *flush=False*)

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file*, and *flush*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like *str()* does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. If no *objects* are given, *print()* will just write *end*.

The *file* argument must be an object with a `write(string)` method; if it is not present or `None`, *sys.stdout* will be used. Since printed arguments are converted to text strings, *print()* cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

Output buffering is usually determined by *file*. However, if *flush* is true, the stream is forcibly flushed.

Changed in version 3.3: Added the *flush* keyword argument.

class property(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

Return a property attribute.

fget is a function for getting an attribute value. *fset* is a function for setting an attribute value. *fdel* is a function for deleting an attribute value. And *doc* creates a docstring for the attribute.

A typical use is to define a managed attribute `x`:

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x
```

(continues on next page)

(continued from previous page)

```
    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

If *c* is an instance of *C*, `c.x` will invoke the getter, `c.x = value` will invoke the setter, and `del c.x` the deleter.

If given, *doc* will be the docstring of the property attribute. Otherwise, the property will copy *fget*'s docstring (if it exists). This makes it possible to create read-only properties easily using *`property()`* as a *decorator*:

```
class Parrot:
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```

The `@property` decorator turns the `voltage()` method into a "getter" for a read-only attribute with the same name, and it sets the docstring for *voltage* to "Get the current voltage."

`@getter`

`@setter`

`@deleter`

A property object has `getter`, `setter`, and `deleter` methods usable as decorators that create a copy of the property with the corresponding accessor function set to the decorated function. This is best explained with an example:

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

This code is exactly equivalent to the first example. Be sure to give the additional functions the same name as the original property (`x` in this case.)

The returned property object also has the attributes `fget`, `fset`, and `fdel` corresponding to the constructor arguments.

Changed in version 3.5: The docstrings of property objects are now writeable.

class range(*stop*)

class range(*start*, *stop*, *step=1*)

Rather than being a function, `range` is actually an immutable sequence type, as documented in *Ranges* and *Sequence Types — list, tuple, range*.

repr(*object*)

Return a string containing a printable representation of an object. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `eval()`; otherwise, the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a `__repr__()` method. If `sys.displayhook()` is not accessible, this function will raise `RuntimeError`.

This class has a custom representation that can be evaluated:

```
class Person:
   def __init__(self, name, age):
      self.name = name
      self.age = age

   def __repr__(self):
      return f"Person('{self.name}', {self.age})"
```

reversed(*seq*)

Return a reverse *iterator*. *seq* must be an object which has a `__reversed__()` method or supports the sequence protocol (the `__len__()` method and the `__getitem__()` method with integer arguments starting at `0`).

round(*number*, *ndigits=None*)

Return *number* rounded to *ndigits* precision after the decimal point. If *ndigits* is omitted or is `None`, it returns the nearest integer to its input.

For the built-in types supporting `round()`, values are rounded to the closest multiple of 10 to the power minus *ndigits*; if two multiples are equally close, rounding is done toward the even choice (so, for example, both `round(0.5)` and `round(-0.5)` are `0`, and `round(1.5)` is `2`). Any integer value is valid for *ndigits* (positive, zero, or negative). The return value is an integer if *ndigits* is omitted or `None`. Otherwise, the return value has the same type as *number*.

For a general Python object `number`, `round` delegates to `number.__round__`.

Note

The behavior of `round()` for floats can be surprising: for example, `round(2.675, 2)` gives `2.67` instead of the expected `2.68`. This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float. See tut-fp-issues for more information.

class set

class set(*iterable*)

Return a new `set` object, optionally with elements taken from *iterable*. `set` is a built-in class. See `set` and *Set Types — set, frozenset* for documentation about this class.

For other containers see the built-in `frozenset`, `list`, `tuple`, and `dict` classes, as well as the `collections` module.

setattr(*object*, *name*, *value*)

This is the counterpart of *getattr()*. The arguments are an object, a string, and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is equivalent to `x.foobar = 123`.

name need not be a Python identifier as defined in identifiers unless the object chooses to enforce that, for example in a custom `__getattribute__()` or via `__slots__`. An attribute whose name is not an identifier will not be accessible using the dot notation, but is accessible through *getattr()* etc..

> **Note**
>
> Since private name mangling happens at compilation time, one must manually mangle a private attribute's (attributes with two leading underscores) name in order to set it with *setattr()*.

class slice(*stop*)

class slice(*start*, *stop*, *step=None*)

Return a *slice* object representing the set of indices specified by `range(start, stop, step)`. The *start* and *step* arguments default to `None`.

start

stop

step

Slice objects have read-only data attributes `start`, `stop`, and `step` which merely return the argument values (or their default). They have no other explicit functionality; however, they are used by NumPy and other third-party packages.

Slice objects are also generated when extended indexing syntax is used. For example: `a[start:stop:step]` or `a[start:stop, i]`. See *itertools.islice()* for an alternate version that returns an *iterator*.

Changed in version 3.12: Slice objects are now *hashable* (provided *start*, *stop*, and *step* are hashable).

sorted(*iterable*, */*, ***, *key=None*, *reverse=False*)

Return a new sorted list from the items in *iterable*.

Has two optional arguments which must be specified as keyword arguments.

key specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). The default value is `None` (compare the elements directly).

reverse is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

Use *functools.cmp_to_key()* to convert an old-style *cmp* function to a *key* function.

The built-in *sorted()* function is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

The sort algorithm uses only < comparisons between items. While defining an `__lt__()` method will suffice for sorting, **PEP 8** recommends that all six rich comparisons be implemented. This will help avoid bugs when using the same data with other ordering tools such as *max()* that rely on a different underlying method. Implementing all six comparisons also helps avoid confusion for mixed type comparisons which can call reflected the `__gt__()` method.

For sorting examples and a brief sorting tutorial, see sortinghowto.

@staticmethod

Transform a method into a static method.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```
class C:
    @staticmethod
    def f(arg1, arg2, argN): ...
```

The `@staticmethod` form is a function *decorator* – see function for details.

A static method can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). Moreover, the static method *descriptor* is also callable, so it can be used in the class definition (such as `f()`).

Static methods in Python are similar to those found in Java or C++. Also, see *classmethod()* for a variant that is useful for creating alternate class constructors.

Like all decorators, it is also possible to call `staticmethod` as a regular function and do something with its result. This is needed in some cases where you need a reference to a function from a class body and you want to avoid the automatic transformation to instance method. For these cases, use this idiom:

```
def regular_function():
    ...

class C:
    method = staticmethod(regular_function)
```

For more information on static methods, see types.

Changed in version 3.10: Static methods now inherit the method attributes (`__module__`, `__name__`, `__qualname__`, `__doc__` and `__annotations__`), have a new `__wrapped__` attribute, and are now callable as regular functions.

class str(*object=''*)

class str(*object=b''*, *encoding='utf-8'*, *errors='strict'*)

Return a *str* version of *object*. See *str()* for details.

`str` is the built-in string *class*. For general information about strings, see *Text Sequence Type — str*.

sum(*iterable*, */*, *start=0*)

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

For some use cases, there are good alternatives to *sum()*. The preferred, fast way to concatenate a sequence of strings is by calling `''.join(sequence)`. To add floating-point values with extended precision, see *math.fsum()*. To concatenate a series of iterables, consider using *itertools.chain()*.

Changed in version 3.8: The *start* parameter can be specified as a keyword argument.

Changed in version 3.12: Summation of floats switched to an algorithm that gives higher accuracy and better commutativity on most builds.

class super

class super(*type*, *object_or_type=None*)

Return a proxy object that delegates method calls to a parent or sibling class of *type*. This is useful for accessing inherited methods that have been overridden in a class.

The *object_or_type* determines the *method resolution order* to be searched. The search starts from the class right after the *type*.

For example, if *__mro__* of *object_or_type* is `D -> B -> C -> A -> object` and the value of *type* is `B`, then *super()* searches `C -> A -> object`.

The *__mro__* attribute of the *object_or_type* lists the method resolution search order used by both *getattr()* and *super()*. The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

If the second argument is omitted, the super object returned is unbound. If the second argument is an object, `isinstance(obj, type)` must be true. If the second argument is a type, `issubclass(type2, type)` must be true (this is useful for classmethods).

When called directly within an ordinary method of a class, both arguments may be omitted ("zero-argument `super()`"). In this case, *type* will be the enclosing class, and *obj* will be the first argument of the immediately enclosing function (typically `self`). (This means that zero-argument `super()` will not work as expected within nested functions, including generator expressions, which implicitly create nested functions.)

There are two typical use cases for *super*. In a class hierarchy with single inheritance, *super* can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable. This use closely parallels the use of *super* in other programming languages.

The second use case is to support cooperative multiple inheritance in a dynamic execution environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement "diamond diagrams" where multiple base classes implement the same method. Good design dictates that such implementations have the same calling signature in every case (because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime).

For both use cases, a typical superclass call looks like this:

```
class C(B):
    def method(self, arg):
        super().method(arg)     # This does the same thing as:
                                # super(C, self).method(arg)
```

In addition to method lookups, *super()* also works for attribute lookups. One possible use case for this is calling *descriptors* in a parent or sibling class.

Note that *super()* is implemented as part of the binding process for explicit dotted attribute lookups such as `super().__getitem__(name)`. It does so by implementing its own `__getattribute__()` method for searching classes in a predictable order that supports cooperative multiple inheritance. Accordingly, *super()* is undefined for implicit lookups using statements or operators such as `super()[name]`.

Also note that, aside from the zero argument form, *super()* is not limited to use inside methods. The two argument form specifies the arguments exactly and makes the appropriate references. The zero argument form only works inside a class definition, as the compiler fills in the necessary details to correctly retrieve the class being defined, as well as accessing the current instance for ordinary methods.

For practical suggestions on how to design cooperative classes using *super()*, see guide to using super().

class tuple

class tuple(*iterable*)

Rather than being a function, *tuple* is actually an immutable sequence type, as documented in *Tuples* and *Sequence Types — list, tuple, range*.

class type(*object*)

class type(*name*, *bases*, *dict*, ***kwds*)

With one argument, return the type of an *object*. The return value is a type object and generally the same object as returned by *object.__class__*.

The *isinstance()* built-in function is recommended for testing the type of an object, because it takes subclasses into account.

With three arguments, return a new type object. This is essentially a dynamic form of the `class` statement. The *name* string is the class name and becomes the `__name__` attribute. The *bases* tuple contains the base classes and becomes the `__bases__` attribute; if empty, `object`, the ultimate base of all classes, is added. The *dict* dictionary contains attribute and method definitions for the class body; it may be copied or wrapped before becoming the `__dict__` attribute. The following two statements create identical `type` objects:

```
>>> class X:
...     a = 1
...
>>> X = type('X', (), dict(a=1))
```

See also *Type Objects*.

Keyword arguments provided to the three argument form are passed to the appropriate metaclass machinery (usually `__init_subclass__()`) in the same way that keywords in a class definition (besides *metaclass*) would.

See also class-customization.

Changed in version 3.6: Subclasses of `type` which don't override `type.__new__` may no longer use the one-argument form to get the type of an object.

vars()

vars(*object*)

Return the `__dict__` attribute for a module, class, instance, or any other object with a `__dict__` attribute.

Objects such as modules and instances have an updateable `__dict__` attribute; however, other objects may have write restrictions on their `__dict__` attributes (for example, classes use a `types.MappingProxyType` to prevent direct dictionary updates).

Without an argument, `vars()` acts like `locals()`.

A `TypeError` exception is raised if an object is specified but it doesn't have a `__dict__` attribute (for example, if its class defines the `__slots__` attribute).

Changed in version 3.13: The result of calling this function without an argument has been updated as described for the `locals()` builtin.

zip(**iterables*, *strict=False*)

Iterate over several iterables in parallel, producing tuples with an item from each one.

Example:

```
>>> for item in zip([1, 2, 3], ['sugar', 'spice', 'everything nice']):
...     print(item)
...
(1, 'sugar')
(2, 'spice')
(3, 'everything nice')
```

More formally: `zip()` returns an iterator of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument iterables.

Another way to think of `zip()` is that it turns rows into columns, and columns into rows. This is similar to transposing a matrix.

`zip()` is lazy: The elements won't be processed until the iterable is iterated on, e.g. by a `for` loop or by wrapping in a `list`.

One thing to consider is that the iterables passed to `zip()` could have different lengths; sometimes by design, and sometimes because of a bug in the code that prepared these iterables. Python offers three different approaches to dealing with this issue:

- By default, *zip()* stops when the shortest iterable is exhausted. It will ignore the remaining items in the longer iterables, cutting off the result to the length of the shortest iterable:

```
>>> list(zip(range(3), ['fee', 'fi', 'fo', 'fum']))
[(0, 'fee'), (1, 'fi'), (2, 'fo')]
```

- *zip()* is often used in cases where the iterables are assumed to be of equal length. In such cases, it's recommended to use the `strict=True` option. Its output is the same as regular *zip()*:

```
>>> list(zip(('a', 'b', 'c'), (1, 2, 3), strict=True))
[('a', 1), ('b', 2), ('c', 3)]
```

  Unlike the default behavior, it raises a *ValueError* if one iterable is exhausted before the others:

```
>>> for item in zip(range(3), ['fee', 'fi', 'fo', 'fum'], strict=True):
...     print(item)
...
(0, 'fee')
(1, 'fi')
(2, 'fo')
Traceback (most recent call last):
  ...
ValueError: zip() argument 2 is longer than argument 1
```

  Without the `strict=True` argument, any bug that results in iterables of different lengths will be silenced, possibly manifesting as a hard-to-find bug in another part of the program.

- Shorter iterables can be padded with a constant value to make all the iterables have the same length. This is done by *itertools.zip_longest()*.

Edge cases: With a single iterable argument, *zip()* returns an iterator of 1-tuples. With no arguments, it returns an empty iterator.

Tips and tricks:

- The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using `zip(*[iter(s)]*n, strict=True)`. This repeats the *same* iterator `n` times so that each output tuple has the result of `n` calls to the iterator. This has the effect of dividing the input into n-length chunks.

- *zip()* in conjunction with the `*` operator can be used to unzip a list:

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> list(zip(x, y))
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zip(x, y))
>>> x == list(x2) and y == list(y2)
True
```

Changed in version 3.10: Added the `strict` argument.

__import__(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)

> **Note**
>
> This is an advanced function that is not needed in everyday Python programming, unlike *importlib.import_module()*.

This function is invoked by the `import` statement. It can be replaced (by importing the *builtins* module and assigning to `builtins.__import__`) in order to change semantics of the `import` statement, but doing so is **strongly** discouraged as it is usually simpler to use import hooks (see **PEP 302**) to attain the same goals and does not cause issues with code which assumes the default import implementation is in use. Direct use of *__import__()* is also discouraged in favor of *importlib.import_module()*.

The function imports the module *name*, potentially using the given *globals* and *locals* to determine how to interpret the name in a package context. The *fromlist* gives the names of objects or submodules that should be imported from the module given by *name*. The standard implementation does not use its *locals* argument at all and uses its *globals* only to determine the package context of the `import` statement.

level specifies whether to use absolute or relative imports. `0` (the default) means only perform absolute imports. Positive values for *level* indicate the number of parent directories to search relative to the directory of the module calling *__import__()* (see **PEP 328** for the details).

When the *name* variable is of the form `package.module`, normally, the top-level package (the name up till the first dot) is returned, *not* the module named by *name*. However, when a non-empty *fromlist* argument is given, the module named by *name* is returned.

For example, the statement `import spam` results in bytecode resembling the following code:

```
spam = __import__('spam', globals(), locals(), [], 0)
```

The statement `import spam.ham` results in this call:

```
spam = __import__('spam.ham', globals(), locals(), [], 0)
```

Note how *__import__()* returns the toplevel module here because this is the object that is bound to a name by the `import` statement.

On the other hand, the statement `from spam.ham import eggs, sausage as saus` results in

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], 0)
eggs = _temp.eggs
saus = _temp.sausage
```

Here, the `spam.ham` module is returned from *__import__()*. From this object, the names to import are retrieved and assigned to their respective names.

If you simply want to import a module (potentially within a package) by name, use *importlib.import_module()*.

Changed in version 3.3: Negative values for *level* are no longer supported (which also changes the default value to 0).

Changed in version 3.9: When the command line options `-E` or `-I` are being used, the environment variable `PYTHONCASEOK` is now ignored.

CHAPTER

THREE

BUILT-IN CONSTANTS

A small number of constants live in the built-in namespace. They are:

False

The false value of the *bool* type. Assignments to `False` are illegal and raise a *SyntaxError*.

True

The true value of the *bool* type. Assignments to `True` are illegal and raise a *SyntaxError*.

None

An object frequently used to represent the absence of a value, as when default arguments are not passed to a function. Assignments to `None` are illegal and raise a *SyntaxError*. `None` is the sole instance of the *NoneType* type.

NotImplemented

A special value which should be returned by the binary special methods (e.g. `__eq__()`, `__lt__()`, `__add__()`, `__rsub__()`, etc.) to indicate that the operation is not implemented with respect to the other type; may be returned by the in-place binary special methods (e.g. `__imul__()`, `__iand__()`, etc.) for the same purpose. It should not be evaluated in a boolean context. `NotImplemented` is the sole instance of the *types.NotImplementedType* type.

> **Note**
>
> When a binary (or in-place) method returns `NotImplemented` the interpreter will try the reflected operation on the other type (or some other fallback, depending on the operator). If all attempts return `NotImplemented`, the interpreter will raise an appropriate exception. Incorrectly returning `NotImplemented` will result in a misleading error message or the `NotImplemented` value being returned to Python code.
>
> See *Implementing the arithmetic operations* for examples.

> **Note**
>
> `NotImplementedError` and `NotImplemented` are not interchangeable, even though they have similar names and purposes. See *NotImplementedError* for details on when to use it.

Changed in version 3.9: Evaluating `NotImplemented` in a boolean context is deprecated. While it currently evaluates as true, it will emit a *DeprecationWarning*. It will raise a *TypeError* in a future version of Python.

Ellipsis

The same as the ellipsis literal "`...`". Special value used mostly in conjunction with extended slicing syntax for user-defined container data types. `Ellipsis` is the sole instance of the *types.EllipsisType* type.

__debug__

This constant is true if Python was not started with an `-O` option. See also the `assert` statement.

> **Note**
>
> The names *None*, *False*, *True* and *__debug__* cannot be reassigned (assignments to them, even as an attribute name, raise *SyntaxError*), so they can be considered "true" constants.

3.1 Constants added by the `site` module

The *site* module (which is imported automatically during startup, except if the `-S` command-line option is given) adds several constants to the built-in namespace. They are useful for the interactive interpreter shell and should not be used in programs.

quit(*code=None*)

exit(*code=None*)

Objects that when printed, print a message like "Use quit() or Ctrl-D (i.e. EOF) to exit", and when called, raise *SystemExit* with the specified exit code.

copyright

credits

Objects that when printed or called, print the text of copyright or credits, respectively.

license

Object that when printed, prints the message "Type license() to see the full license text", and when called, displays the full license text in a pager-like fashion (one screen at a time).

CHAPTER

FOUR

BUILT-IN TYPES

The following sections describe the standard types that are built into the interpreter.

The principal built-in types are numerics, sequences, mappings, classes, instances and exceptions.

Some collection classes are mutable. The methods that add, subtract, or rearrange their members in place, and don't return a specific item, never return the collection instance itself but `None`.

Some operations are supported by several object types; in particular, practically all objects can be compared for equality, tested for truth value, and converted to a string (with the *`repr()`* function or the slightly different *`str()`* function). The latter function is implicitly used when an object is written by the *`print()`* function.

4.1 Truth Value Testing

Any object can be tested for truth value, for use in an `if` or `while` condition or as operand of the Boolean operations below.

By default, an object is considered true unless its class defines either a `__bool__()` method that returns `False` or a `__len__()` method that returns zero, when called with the object.[1] Here are most of the built-in objects considered false:

- constants defined to be false: `None` and `False`
- zero of any numeric type: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- empty sequences and collections: `''`, `()`, `[]`, `{}`, `set()`, `range(0)`

Operations and built-in functions that have a Boolean result always return `0` or `False` for false and `1` or `True` for true, unless otherwise stated. (Important exception: the Boolean operations `or` and `and` always return one of their operands.)

4.2 Boolean Operations — `and`, `or`, `not`

These are the Boolean operations, ordered by ascending priority:

Operation	Result	Notes
`x or y`	if *x* is true, then *x*, else *y*	(1)
`x and y`	if *x* is false, then *x*, else *y*	(2)
`not x`	if *x* is false, then `True`, else `False`	(3)

Notes:

[1] Additional information on these special methods may be found in the Python Reference Manual (customization).

(1) This is a short-circuit operator, so it only evaluates the second argument if the first one is false.

(2) This is a short-circuit operator, so it only evaluates the second argument if the first one is true.

(3) `not` has a lower priority than non-Boolean operators, so `not a == b` is interpreted as `not (a == b)`, and `a == not b` is a syntax error.

4.3 Comparisons

There are eight comparison operations in Python. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily; for example, `x < y <= z` is equivalent to `x < y and y <= z`, except that *y* is evaluated only once (but in both cases *z* is not evaluated at all when `x < y` is found to be false).

This table summarizes the comparison operations:

Operation	Meaning
`<`	strictly less than
`<=`	less than or equal
`>`	strictly greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal
`is`	object identity
`is not`	negated object identity

Objects of different types, except different numeric types, never compare equal. The == operator is always defined but for some object types (for example, class objects) is equivalent to `is`. The <, <=, > and >= operators are only defined where they make sense; for example, they raise a *`TypeError`* exception when one of the arguments is a complex number.

Non-identical instances of a class normally compare as non-equal unless the class defines the `__eq__()` method.

Instances of a class cannot be ordered with respect to other instances of the same class, or other types of object, unless the class defines enough of the methods `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` (in general, `__lt__()` and `__eq__()` are sufficient, if you want the conventional meanings of the comparison operators).

The behavior of the `is` and `is not` operators cannot be customized; also they can be applied to any two objects and never raise an exception.

Two more operations with the same syntactic priority, `in` and `not in`, are supported by types that are *iterable* or implement the `__contains__()` method.

4.4 Numeric Types — `int`, `float`, `complex`

There are three distinct numeric types: *integers*, *floating-point numbers*, and *complex numbers*. In addition, Booleans are a subtype of integers. Integers have unlimited precision. Floating-point numbers are usually implemented using `double` in C; information about the precision and internal representation of floating-point numbers for the machine on which your program is running is available in *`sys.float_info`*. Complex numbers have a real and imaginary part, which are each a floating-point number. To extract these parts from a complex number *z*, use `z.real` and `z.imag`. (The standard library includes the additional numeric types *`fractions.Fraction`*, for rationals, and *`decimal.Decimal`*, for floating-point numbers with user-definable precision.)

Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals (including hex, octal and binary numbers) yield integers. Numeric literals containing a decimal point or an exponent sign

yield floating-point numbers. Appending `'j'` or `'J'` to a numeric literal yields an imaginary number (a complex number with a zero real part) which you can add to an integer or float to get a complex number with real and imaginary parts.

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the "narrower" type is widened to that of the other, where integer is narrower than floating point, which is narrower than complex. A comparison between numbers of different types behaves as though the exact values of those numbers were being compared.[2]

The constructors *int()*, *float()*, and *complex()* can be used to produce numbers of a specific type.

All numeric types (except complex) support the following operations (for priorities of the operations, see operator-summary):

Operation	Result	Notes	Full documentation
`x + y`	sum of *x* and *y*		
`x - y`	difference of *x* and *y*		
`x * y`	product of *x* and *y*		
`x / y`	quotient of *x* and *y*		
`x // y`	floored quotient of *x* and *y*	(1)(2)	
`x % y`	remainder of `x / y`	(2)	
`-x`	*x* negated		
`+x`	*x* unchanged		
`abs(x)`	absolute value or magnitude of *x*		*abs()*
`int(x)`	*x* converted to integer	(3)(6)	*int()*
`float(x)`	*x* converted to floating point	(4)(6)	*float()*
`complex(re, im)`	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.	(6)	*complex()*
`c.conjugate()`	conjugate of the complex number *c*		
`divmod(x, y)`	the pair `(x // y, x % y)`	(2)	*divmod()*
`pow(x, y)`	*x* to the power *y*	(5)	*pow()*
`x ** y`	*x* to the power *y*	(5)	

Notes:

(1) Also referred to as integer division. For operands of type *int*, the result has type *int*. For operands of type *float*, the result has type *float*. In general, the result is a whole integer, though the result's type is not necessarily *int*. The result is always rounded towards minus infinity: `1//2` is `0`, `(-1)//2` is `-1`, `1//(-2)` is `-1`, and `(-1)//(-2)` is `0`.

(2) Not for complex numbers. Instead convert to floats using *abs()* if appropriate.

(3) Conversion from *float* to *int* truncates, discarding the fractional part. See functions *math.floor()* and *math.ceil()* for alternative conversions.

(4) float also accepts the strings "nan" and "inf" with an optional prefix "+" or "-" for Not a Number (NaN) and positive or negative infinity.

(5) Python defines `pow(0, 0)` and `0 ** 0` to be `1`, as is common for programming languages.

(6) The numeric literals accepted include the digits `0` to `9` or any Unicode equivalent (code points with the `Nd` property).

See the Unicode Standard for a complete list of code points with the `Nd` property.

All *numbers.Real* types (*int* and *float*) also include the following operations:

[2] As a consequence, the list `[1, 2]` is considered equal to `[1.0, 2.0]`, and similarly for tuples.

Operation	Result
`math.trunc(x)`	*x* truncated to `Integral`
`round(x[, n])`	*x* rounded to *n* digits, rounding half to even. If *n* is omitted, it defaults to 0.
`math.floor(x)`	the greatest `Integral` <= *x*
`math.ceil(x)`	the least `Integral` >= *x*

For additional numeric operations see the `math` and `cmath` modules.

4.4.1 Bitwise Operations on Integer Types

Bitwise operations only make sense for integers. The result of bitwise operations is calculated as though carried out in two's complement with an infinite number of sign bits.

The priorities of the binary bitwise operations are all lower than the numeric operations and higher than the comparisons; the unary operation ~ has the same priority as the other unary numeric operations (+ and −).

This table lists the bitwise operations sorted in ascending priority:

Operation	Result	Notes
`x | y`	bitwise *or* of *x* and *y*	(4)
`x ^ y`	bitwise *exclusive or* of *x* and *y*	(4)
`x & y`	bitwise *and* of *x* and *y*	(4)
`x << n`	*x* shifted left by *n* bits	(1)(2)
`x >> n`	*x* shifted right by *n* bits	(1)(3)
`~x`	the bits of *x* inverted	

Notes:

(1) Negative shift counts are illegal and cause a `ValueError` to be raised.

(2) A left shift by *n* bits is equivalent to multiplication by `pow(2, n)`.

(3) A right shift by *n* bits is equivalent to floor division by `pow(2, n)`.

(4) Performing these calculations with at least one extra sign extension bit in a finite two's complement representation (a working bit-width of `1 + max(x.bit_length(), y.bit_length())` or more) is sufficient to get the same result as if there were an infinite number of sign bits.

4.4.2 Additional Methods on Integer Types

The int type implements the `numbers.Integral` *abstract base class*. In addition, it provides a few more methods:

`int.`**`bit_length`**`()`

Return the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros:

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

More precisely, if `x` is nonzero, then `x.bit_length()` is the unique positive integer `k` such that `2**(k-1) <= abs(x) < 2**k`. Equivalently, when `abs(x)` is small enough to have a correctly rounded logarithm, then `k = 1 + int(log(abs(x), 2))`. If `x` is zero, then `x.bit_length()` returns `0`.

Equivalent to:

```
def bit_length(self):
    s = bin(self)       # binary representation:  bin(-37) --> '-0b100101'
    s = s.lstrip('-0b') # remove leading zeros and minus sign
    return len(s)       # len('100101') --> 6
```

Added in version 3.1.

`int.`**`bit_count`**`()`

Return the number of ones in the binary representation of the absolute value of the integer. This is also known as the population count. Example:

```
>>> n = 19
>>> bin(n)
'0b10011'
>>> n.bit_count()
3
>>> (-n).bit_count()
3
```

Equivalent to:

```
def bit_count(self):
    return bin(self).count("1")
```

Added in version 3.10.

`int.`**`to_bytes`**`(`*length=1*, *byteorder='big'*, ***, *signed=False*`)`

Return an array of bytes representing an integer.

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() + 7) // 8, byteorder='little')
b'\xe8\x03'
```

The integer is represented using *length* bytes, and defaults to 1. An *OverflowError* is raised if the integer is not representable with the given number of bytes.

The *byteorder* argument determines the byte order used to represent the integer, and defaults to `"big"`. If *byteorder* is `"big"`, the most significant byte is at the beginning of the byte array. If *byteorder* is `"little"`, the most significant byte is at the end of the byte array.

The *signed* argument determines whether two's complement is used to represent the integer. If *signed* is `False` and a negative integer is given, an *OverflowError* is raised. The default value for *signed* is `False`.

The default values can be used to conveniently turn an integer into a single byte object:

```
>>> (65).to_bytes()
b'A'
```

However, when using the default arguments, don't try to convert a value greater than 255 or you'll get an *OverflowError*.

Equivalent to:

```
def to_bytes(n, length=1, byteorder='big', signed=False):
    if byteorder == 'little':
        order = range(length)
    elif byteorder == 'big':
        order = reversed(range(length))
    else:
        raise ValueError("byteorder must be either 'little' or 'big'")

    return bytes((n >> i*8) & 0xff for i in order)
```

Added in version 3.2.

Changed in version 3.11: Added default argument values for `length` and `byteorder`.

classmethod `int.`**from_bytes**(*bytes*, *byteorder='big'*, ***, *signed=False*)

Return the integer represented by the given array of bytes.

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

The argument *bytes* must either be a *bytes-like object* or an iterable producing bytes.

The *byteorder* argument determines the byte order used to represent the integer, and defaults to `"big"`. If *byteorder* is `"big"`, the most significant byte is at the beginning of the byte array. If *byteorder* is `"little"`, the most significant byte is at the end of the byte array. To request the native byte order of the host system, use *sys.byteorder* as the byte order value.

The *signed* argument indicates whether two's complement is used to represent the integer.

Equivalent to:

```
def from_bytes(bytes, byteorder='big', signed=False):
    if byteorder == 'little':
        little_ordered = list(bytes)
    elif byteorder == 'big':
        little_ordered = list(reversed(bytes))
    else:
        raise ValueError("byteorder must be either 'little' or 'big'")

    n = sum(b << i*8 for i, b in enumerate(little_ordered))
    if signed and little_ordered and (little_ordered[-1] & 0x80):
        n -= 1 << 8*len(little_ordered)

    return n
```

Added in version 3.2.

Changed in version 3.11: Added default argument value for `byteorder`.

int.**as_integer_ratio**()

Return a pair of integers whose ratio is equal to the original integer and has a positive denominator. The integer ratio of integers (whole numbers) is always the integer as the numerator and `1` as the denominator.

Added in version 3.8.

int.**is_integer**()

Returns `True`. Exists for duck type compatibility with *float.is_integer()*.

Added in version 3.12.

4.4.3 Additional Methods on Float

The float type implements the *numbers.Real abstract base class*. float also has the following additional methods.

float.**as_integer_ratio**()

Return a pair of integers whose ratio is exactly equal to the original float. The ratio is in lowest terms and has a positive denominator. Raises *OverflowError* on infinities and a *ValueError* on NaNs.

float.**is_integer**()

Return `True` if the float instance is finite with integral value, and `False` otherwise:

```
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

Two methods support conversion to and from hexadecimal strings. Since Python's floats are stored internally as binary numbers, converting a float to or from a *decimal* string usually involves a small rounding error. In contrast, hexadecimal strings allow exact representation and specification of floating-point numbers. This can be useful when debugging, and in numerical work.

float.**hex**()

Return a representation of a floating-point number as a hexadecimal string. For finite floating-point numbers, this representation will always include a leading `0x` and a trailing `p` and exponent.

classmethod float.**fromhex**(*s*)

Class method to return the float represented by a hexadecimal string *s*. The string *s* may have leading and trailing whitespace.

Note that *float.hex()* is an instance method, while *float.fromhex()* is a class method.

A hexadecimal string takes the form:

```
[sign] ['0x'] integer ['.' fraction] ['p' exponent]
```

where the optional `sign` may by either `+` or `-`, `integer` and `fraction` are strings of hexadecimal digits, and `exponent` is a decimal integer with an optional leading sign. Case is not significant, and there must be at least one hexadecimal digit in either the integer or the fraction. This syntax is similar to the syntax specified in section 6.4.4.2 of the C99 standard, and also to the syntax used in Java 1.5 onwards. In particular, the output of *float.hex()* is usable as a hexadecimal floating-point literal in C or Java code, and hexadecimal strings produced by C's `%a` format character or Java's `Double.toHexString` are accepted by *float.fromhex()*.

Note that the exponent is written in decimal rather than hexadecimal, and that it gives the power of 2 by which to multiply the coefficient. For example, the hexadecimal string `0x3.a7p10` represents the floating-point number `(3 + 10./16 + 7./16**2) * 2.0**10`, or `3740.0`:

```
>>> float.fromhex('0x3.a7p10')
3740.0
```

Applying the reverse conversion to `3740.0` gives a different hexadecimal string representing the same number:

```
>>> float.hex(3740.0)
'0x1.d380000000000p+11'
```

4.4.4 Hashing of numeric types

For numbers `x` and `y`, possibly of different types, it's a requirement that `hash(x) == hash(y)` whenever `x == y` (see the `__hash__()` method documentation for more details). For ease of implementation and efficiency across a variety of numeric types (including *int*, *float*, *decimal.Decimal* and *fractions.Fraction*) Python's hash for numeric types is based on a single mathematical function that's defined for any rational number, and hence applies to all instances of *int* and *fractions.Fraction*, and all finite instances of *float* and *decimal.Decimal*. Essentially, this function is given by reduction modulo `P` for a fixed prime `P`. The value of `P` is made available to Python as the *modulus* attribute of *sys.hash_info*.

CPython implementation detail: Currently, the prime used is `P = 2**31 - 1` on machines with 32-bit C longs and `P = 2**61 - 1` on machines with 64-bit C longs.

Here are the rules in detail:

- If `x = m / n` is a nonnegative rational number and `n` is not divisible by `P`, define `hash(x)` as `m * invmod(n, P) % P`, where `invmod(n, P)` gives the inverse of `n` modulo `P`.
- If `x = m / n` is a nonnegative rational number and `n` is divisible by `P` (but `m` is not) then `n` has no inverse modulo `P` and the rule above doesn't apply; in this case define `hash(x)` to be the constant value `sys.hash_info.inf`.
- If `x = m / n` is a negative rational number define `hash(x)` as `-hash(-x)`. If the resulting hash is `-1`, replace it with `-2`.
- The particular values `sys.hash_info.inf` and `-sys.hash_info.inf` are used as hash values for positive infinity or negative infinity (respectively).
- For a *complex* number `z`, the hash values of the real and imaginary parts are combined by computing `hash(z.real) + sys.hash_info.imag * hash(z.imag)`, reduced modulo `2**sys.hash_info.width` so that it lies in `range(-2**(sys.hash_info.width - 1), 2**(sys.hash_info.width - 1))`. Again, if the result is `-1`, it's replaced with `-2`.

To clarify the above rules, here's some example Python code, equivalent to the built-in hash, for computing the hash of a rational number, *float*, or *complex*:

```
import sys, math

def hash_fraction(m, n):
    """Compute the hash of a rational number m / n.

    Assumes m and n are integers, with n positive.
    Equivalent to hash(fractions.Fraction(m, n)).

    """
    P = sys.hash_info.modulus
    # Remove common factors of P.  (Unnecessary if m and n already coprime.)
    while m % P == n % P == 0:
        m, n = m // P, n // P
```

(continues on next page)

(continued from previous page)

```
    if n % P == 0:
        hash_value = sys.hash_info.inf
    else:
        # Fermat's Little Theorem: pow(n, P-1, P) is 1, so
        # pow(n, P-2, P) gives the inverse of n modulo P.
        hash_value = (abs(m) % P) * pow(n, P - 2, P) % P
    if m < 0:
        hash_value = -hash_value
    if hash_value == -1:
        hash_value = -2
    return hash_value

def hash_float(x):
    """Compute the hash of a float x."""

    if math.isnan(x):
        return object.__hash__(x)
    elif math.isinf(x):
        return sys.hash_info.inf if x > 0 else -sys.hash_info.inf
    else:
        return hash_fraction(*x.as_integer_ratio())

def hash_complex(z):
    """Compute the hash of a complex number z."""

    hash_value = hash_float(z.real) + sys.hash_info.imag * hash_float(z.imag)
    # do a signed reduction modulo 2**sys.hash_info.width
    M = 2**(sys.hash_info.width - 1)
    hash_value = (hash_value & (M - 1)) - (hash_value & M)
    if hash_value == -1:
        hash_value = -2
    return hash_value
```

4.5 Boolean Type - `bool`

Booleans represent truth values. The *bool* type has exactly two constant instances: `True` and `False`.

The built-in function *bool()* converts any value to a boolean, if the value can be interpreted as a truth value (see section *Truth Value Testing* above).

For logical operations, use the *boolean operators* `and`, `or` and `not`. When applying the bitwise operators `&`, `|`, `^` to two booleans, they return a bool equivalent to the logical operations "and", "or", "xor". However, the logical operators `and`, `or` and `!=` should be preferred over `&`, `|` and `^`.

Deprecated since version 3.12: The use of the bitwise inversion operator ~ is deprecated and will raise an error in Python 3.14.

bool is a subclass of *int* (see *Numeric Types — int, float, complex*). In many numeric contexts, `False` and `True` behave like the integers 0 and 1, respectively. However, relying on this is discouraged; explicitly convert using *int()* instead.

4.6 Iterator Types

Python supports a concept of iteration over containers. This is implemented using two distinct methods; these are used to allow user-defined classes to support iteration. Sequences, described below in more detail, always support the iteration methods.

One method needs to be defined for container objects to provide *iterable* support:

container.**__iter__**()

Return an *iterator* object. The object is required to support the iterator protocol described below. If a container supports different types of iteration, additional methods can be provided to specifically request iterators for those iteration types. (An example of an object supporting multiple forms of iteration would be a tree structure which supports both breadth-first and depth-first traversal.) This method corresponds to the `tp_iter` slot of the type structure for Python objects in the Python/C API.

The iterator objects themselves are required to support the following two methods, which together form the *iterator protocol*:

iterator.**__iter__**()

Return the *iterator* object itself. This is required to allow both containers and iterators to be used with the `for` and `in` statements. This method corresponds to the `tp_iter` slot of the type structure for Python objects in the Python/C API.

iterator.**__next__**()

Return the next item from the *iterator*. If there are no further items, raise the *StopIteration* exception. This method corresponds to the `tp_iternext` slot of the type structure for Python objects in the Python/C API.

Python defines several iterator objects to support iteration over general and specific sequence types, dictionaries, and other more specialized forms. The specific types are not important beyond their implementation of the iterator protocol.

Once an iterator's *__next__()* method raises *StopIteration*, it must continue to do so on subsequent calls. Implementations that do not obey this property are deemed broken.

4.6.1 Generator Types

Python's *generator*s provide a convenient way to implement the iterator protocol. If a container object's *__iter__()* method is implemented as a generator, it will automatically return an iterator object (technically, a generator object) supplying the `__iter__()` and `__next__()` methods. More information about generators can be found in the documentation for the yield expression.

4.7 Sequence Types — `list`, `tuple`, `range`

There are three basic sequence types: lists, tuples, and range objects. Additional sequence types tailored for processing of *binary data* and *text strings* are described in dedicated sections.

4.7.1 Common Sequence Operations

The operations in the following table are supported by most sequence types, both mutable and immutable. The *collections.abc.Sequence* ABC is provided to make it easier to correctly implement these operations on custom sequence types.

This table lists the sequence operations sorted in ascending priority. In the table, *s* and *t* are sequences of the same type, *n*, *i*, *j* and *k* are integers and *x* is an arbitrary object that meets any type and value restrictions imposed by *s*.

The `in` and `not in` operations have the same priorities as the comparison operations. The `+` (concatenation) and `*` (repetition) operations have the same priority as the corresponding numeric operations.[3]

Operation	Result	Notes
`x in s`	`True` if an item of *s* is equal to *x*, else `False`	(1)
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`	(1)
`s + t`	the concatenation of *s* and *t*	(6)(7)
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times	(2)(7)
`s[i]`	*i*th item of *s*, origin 0	(3)
`s[i:j]`	slice of *s* from *i* to *j*	(3)(4)
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*	(3)(5)
`len(s)`	length of *s*	
`min(s)`	smallest item of *s*	
`max(s)`	largest item of *s*	
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)	(8)
`s.count(x)`	total number of occurrences of *x* in *s*	

Sequences of the same type also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. This means that to compare equal, every element must compare equal and the two sequences must be of the same type and have the same length. (For full details see comparisons in the language reference.)

Forward and reversed iterators over mutable sequences access values using an index. That index will continue to march forward (or backward) even if the underlying sequence is mutated. The iterator terminates only when an *IndexError* or a *StopIteration* is encountered (or when the index drops below zero).

Notes:

(1) While the `in` and `not in` operations are used only for simple containment testing in the general case, some specialised sequences (such as *str*, *bytes* and *bytearray*) also use them for subsequence testing:

```
>>> "gg" in "eggs"
True
```

(2) Values of *n* less than `0` are treated as `0` (which yields an empty sequence of the same type as *s*). Note that items in the sequence *s* are not copied; they are referenced multiple times. This often haunts new Python programmers; consider:

```
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

[3] They must have since the parser can't tell the type of the operands.

What has happened is that `[[]]` is a one-element list containing an empty list, so all three elements of `[[]] * 3` are references to this single empty list. Modifying any of the elements of `lists` modifies this single list. You can create a list of different lists this way:

```
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

Further explanation is available in the FAQ entry faq-multidimensional-list.

(3) If *i* or *j* is negative, the index is relative to the end of sequence *s*: `len(s) + i` or `len(s) + j` is substituted. But note that `-0` is still `0`.

(4) The slice of *s* from *i* to *j* is defined as the sequence of items with index *k* such that `i <= k < j`. If *i* or *j* is greater than `len(s)`, use `len(s)`. If *i* is omitted or `None`, use `0`. If *j* is omitted or `None`, use `len(s)`. If *i* is greater than or equal to *j*, the slice is empty.

(5) The slice of *s* from *i* to *j* with step *k* is defined as the sequence of items with index `x = i + n*k` such that `0 <= n < (j-i)/k`. In other words, the indices are `i`, `i+k`, `i+2*k`, `i+3*k` and so on, stopping when *j* is reached (but never including *j*). When *k* is positive, *i* and *j* are reduced to `len(s)` if they are greater. When *k* is negative, *i* and *j* are reduced to `len(s) - 1` if they are greater. If *i* or *j* are omitted or `None`, they become "end" values (which end depends on the sign of *k*). Note, *k* cannot be zero. If *k* is `None`, it is treated like `1`.

(6) Concatenating immutable sequences always results in a new object. This means that building up a sequence by repeated concatenation will have a quadratic runtime cost in the total sequence length. To get a linear runtime cost, you must switch to one of the alternatives below:

- if concatenating *str* objects, you can build a list and use *str.join()* at the end or else write to an *io.StringIO* instance and retrieve its value when complete
- if concatenating *bytes* objects, you can similarly use *bytes.join()* or *io.BytesIO*, or you can do in-place concatenation with a *bytearray* object. *bytearray* objects are mutable and have an efficient overallocation mechanism
- if concatenating *tuple* objects, extend a *list* instead
- for other types, investigate the relevant class documentation

(7) Some sequence types (such as *range*) only support item sequences that follow specific patterns, and hence don't support sequence concatenation or repetition.

(8) `index` raises *ValueError* when *x* is not found in *s*. Not all implementations support passing the additional arguments *i* and *j*. These arguments allow efficient searching of subsections of the sequence. Passing the extra arguments is roughly equivalent to using `s[i:j].index(x)`, only without copying any data and with the returned index being relative to the start of the sequence rather than the start of the slice.

4.7.2 Immutable Sequence Types

The only operation that immutable sequence types generally implement that is not also implemented by mutable sequence types is support for the *hash()* built-in.

This support allows immutable sequences, such as *tuple* instances, to be used as *dict* keys and stored in *set* and *frozenset* instances.

Attempting to hash an immutable sequence that contains unhashable values will result in *TypeError*.

4.7.3 Mutable Sequence Types

The operations in the following table are defined on mutable sequence types. The *collections.abc.MutableSequence* ABC is provided to make it easier to correctly implement these operations on custom sequence types.

In the table *s* is an instance of a mutable sequence type, *t* is any iterable object and *x* is an arbitrary object that meets any type and value restrictions imposed by *s* (for example, *bytearray* only accepts integers that meet the value restriction `0 <= x <= 255`).

Operation	Result	Notes
`s[i] = x`	item *i* of *s* is replaced by *x*	
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*	
`del s[i:j]`	same as `s[i:j] = []`	
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*	(1)
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list	
`s.append(x)`	appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`)	
`s.clear()`	removes all items from *s* (same as `del s[:]`)	(5)
`s.copy()`	creates a shallow copy of *s* (same as `s[:]`)	(5)
`s.extend(t)` or `s += t`	extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`)	
`s *= n`	updates *s* with its contents repeated *n* times	(6)
`s.insert(i, x)`	inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`)	
`s.pop()` or `s.pop(i)`	retrieves the item at *i* and also removes it from *s*	(2)
`s.remove(x)`	remove the first item from *s* where `s[i]` is equal to *x*	(3)
`s.reverse()`	reverses the items of *s* in place	(4)

Notes:

(1) If *k* is not equal to `1`, *t* must have the same length as the slice it is replacing.

(2) The optional argument *i* defaults to `-1`, so that by default the last item is removed and returned.

(3) `remove()` raises *ValueError* when *x* is not found in *s*.

(4) The `reverse()` method modifies the sequence in place for economy of space when reversing a large sequence. To remind users that it operates by side effect, it does not return the reversed sequence.

(5) `clear()` and `copy()` are included for consistency with the interfaces of mutable containers that don't support slicing operations (such as *dict* and *set*). `copy()` is not part of the *collections.abc.MutableSequence* ABC, but most concrete mutable sequence classes provide it.

Added in version 3.3: `clear()` and `copy()` methods.

(6) The value *n* is an integer, or an object implementing `__index__()`. Zero and negative values of *n* clear the sequence. Items in the sequence are not copied; they are referenced multiple times, as explained for `s * n` under *Common Sequence Operations*.

4.7.4 Lists

Lists are mutable sequences, typically used to store collections of homogeneous items (where the precise degree of similarity will vary by application).

class list([*iterable*])

Lists may be constructed in several ways:

- Using a pair of square brackets to denote the empty list: `[]`
- Using square brackets, separating items with commas: `[a]`, `[a, b, c]`
- Using a list comprehension: `[x for x in iterable]`
- Using the type constructor: `list()` or `list(iterable)`

The constructor builds a list whose items are the same and in the same order as *iterable*'s items. *iterable* may be either a sequence, a container that supports iteration, or an iterator object. If *iterable* is already a list, a copy is made and returned, similar to `iterable[:]`. For example, `list('abc')` returns `['a', 'b', 'c']` and `list( (1, 2, 3) )` returns `[1, 2, 3]`. If no argument is given, the constructor creates a new empty list, `[]`.

Many other operations also produce lists, including the *sorted()* built-in.

Lists implement all of the *common* and *mutable* sequence operations. Lists also provide the following additional method:

sort(*, *key=None*, *reverse=False*)

This method sorts the list in place, using only < comparisons between items. Exceptions are not suppressed - if any comparison operations fail, the entire sort operation will fail (and the list will likely be left in a partially modified state).

sort() accepts two arguments that can only be passed by keyword (*keyword-only arguments*):

key specifies a function of one argument that is used to extract a comparison key from each list element (for example, `key=str.lower`). The key corresponding to each item in the list is calculated once and then used for the entire sorting process. The default value of `None` means that list items are sorted directly without calculating a separate key value.

The *functools.cmp_to_key()* utility is available to convert a 2.x style *cmp* function to a *key* function.

reverse is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

This method modifies the sequence in place for economy of space when sorting a large sequence. To remind users that it operates by side effect, it does not return the sorted sequence (use *sorted()* to explicitly request a new sorted list instance).

The *sort()* method is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

For sorting examples and a brief sorting tutorial, see sortinghowto.

CPython implementation detail: While a list is being sorted, the effect of attempting to mutate, or even inspect, the list is undefined. The C implementation of Python makes the list appear empty for the duration, and raises *ValueError* if it can detect that the list has been mutated during a sort.

4.7.5 Tuples

Tuples are immutable sequences, typically used to store collections of heterogeneous data (such as the 2-tuples produced by the *enumerate()* built-in). Tuples are also used for cases where an immutable sequence of homogeneous data is needed (such as allowing storage in a *set* or *dict* instance).

class tuple([*iterable*])

Tuples may be constructed in a number of ways:

- Using a pair of parentheses to denote the empty tuple: `()`
- Using a trailing comma for a singleton tuple: `a,` or `(a,)`
- Separating items with commas: `a, b, c` or `(a, b, c)`
- Using the *tuple()* built-in: `tuple()` or `tuple(iterable)`

The constructor builds a tuple whose items are the same and in the same order as *iterable*'s items. *iterable* may be either a sequence, a container that supports iteration, or an iterator object. If *iterable* is already a tuple, it is returned unchanged. For example, `tuple('abc')` returns `('a', 'b', 'c')` and `tuple( [1, 2, 3] )` returns `(1, 2, 3)`. If no argument is given, the constructor creates a new empty tuple, `()`.

Note that it is actually the comma which makes a tuple, not the parentheses. The parentheses are optional, except in the empty tuple case, or when they are needed to avoid syntactic ambiguity. For example, `f(a, b, c)` is a function call with three arguments, while `f((a, b, c))` is a function call with a 3-tuple as the sole argument.

Tuples implement all of the *common* sequence operations.

For heterogeneous collections of data where access by name is clearer than access by index, *collections.namedtuple()* may be a more appropriate choice than a simple tuple object.

4.7.6 Ranges

The *range* type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in `for` loops.

class range(*stop*)

class range(*start*, *stop*[, *step*])

The arguments to the range constructor must be integers (either built-in *int* or any object that implements the `__index__()` special method). If the *step* argument is omitted, it defaults to `1`. If the *start* argument is omitted, it defaults to `0`. If *step* is zero, *ValueError* is raised.

For a positive *step*, the contents of a range `r` are determined by the formula `r[i] = start + step*i` where `i >= 0` and `r[i] < stop`.

For a negative *step*, the contents of the range are still determined by the formula `r[i] = start + step*i`, but the constraints are `i >= 0` and `r[i] > stop`.

A range object will be empty if `r[0]` does not meet the value constraint. Ranges do support negative indices, but these are interpreted as indexing from the end of the sequence determined by the positive indices.

Ranges containing absolute values larger than *sys.maxsize* are permitted but some features (such as *len()*) may raise *OverflowError*.

Range examples:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
```

(continues on next page)

(continued from previous page)

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

Ranges implement all of the *common* sequence operations except concatenation and repetition (due to the fact that range objects can only represent sequences that follow a strict pattern and repetition and concatenation will usually violate that pattern).

start

The value of the *start* parameter (or `0` if the parameter was not supplied)

stop

The value of the *stop* parameter

step

The value of the *step* parameter (or `1` if the parameter was not supplied)

The advantage of the *range* type over a regular *list* or *tuple* is that a *range* object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the `start`, `stop` and `step` values, calculating individual items and subranges as needed).

Range objects implement the *collections.abc.Sequence* ABC, and provide features such as containment tests, element index lookup, slicing and support for negative indices (see *Sequence Types — list, tuple, range*):

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

Testing range objects for equality with == and != compares them as sequences. That is, two range objects are considered equal if they represent the same sequence of values. (Note that two range objects that compare equal might have different *start*, *stop* and *step* attributes, for example `range(0) == range(2, 1, 3)` or `range(0, 3, 2) == range(0, 4, 2)`.)

Changed in version 3.2: Implement the Sequence ABC. Support slicing and negative indices. Test *int* objects for membership in constant time instead of iterating through all items.

Changed in version 3.3: Define '==' and '!=' to compare range objects based on the sequence of values they define (instead of comparing based on object identity).

Added the *start*, *stop* and *step* attributes.

See also

- The linspace recipe shows how to implement a lazy version of range suitable for floating-point applications.

4.8 Text Sequence Type — str

Textual data in Python is handled with *str* objects, or *strings*. Strings are immutable *sequences* of Unicode code points. String literals are written in a variety of ways:

- Single quotes: `'allows embedded "double" quotes'`
- Double quotes: `"allows embedded 'single' quotes"`
- Triple quoted: `'''Three single quotes'''`, `"""Three double quotes"""`

Triple quoted strings may span multiple lines - all associated whitespace will be included in the string literal.

String literals that are part of a single expression and have only whitespace between them will be implicitly converted to a single string literal. That is, `("spam " "eggs") == "spam eggs"`.

See strings for more about the various forms of string literal, including supported escape sequences, and the `r` ("raw") prefix that disables most escape sequence processing.

Strings may also be created from other objects using the *str* constructor.

Since there is no separate "character" type, indexing a string produces strings of length 1. That is, for a non-empty string *s*, `s[0] == s[0:1]`.

There is also no mutable string type, but *str.join()* or *io.StringIO* can be used to efficiently construct strings from multiple fragments.

Changed in version 3.3: For backwards compatibility with the Python 2 series, the `u` prefix is once again permitted on string literals. It has no effect on the meaning of string literals and cannot be combined with the `r` prefix.

class str(*object=''*)

class str(*object=b''*, *encoding='utf-8'*, *errors='strict'*)

Return a *string* version of *object*. If *object* is not provided, returns the empty string. Otherwise, the behavior of `str()` depends on whether *encoding* or *errors* is given, as follows.

If neither *encoding* nor *errors* is given, `str(object)` returns `type(object).__str__(object)`, which is the "informal" or nicely printable string representation of *object*. For string objects, this is the string itself. If *object* does not have a `__str__()` method, then *str()* falls back to returning *repr(object)*.

If at least one of *encoding* or *errors* is given, *object* should be a *bytes-like object* (e.g. *bytes* or *bytearray*). In this case, if *object* is a *bytes* (or *bytearray*) object, then `str(bytes, encoding, errors)` is equivalent to *bytes.decode(encoding, errors)*. Otherwise, the bytes object underlying the buffer object is obtained before calling *bytes.decode()*. See *Binary Sequence Types — bytes, bytearray, memoryview* and bufferobjects for information on buffer objects.

Passing a *bytes* object to *str()* without the *encoding* or *errors* arguments falls under the first case of returning the informal string representation (see also the `-b` command-line option to Python). For example:

```
>>> str(b'Zoot!')
"b'Zoot!'"
```

For more information on the `str` class and its methods, see *Text Sequence Type — str* and the *String Methods* section below. To output formatted strings, see the f-strings and *Format String Syntax* sections. In addition, see the *Text Processing Services* section.

4.8.1 String Methods

Strings implement all of the *common* sequence operations, along with the additional methods described below.

Strings also support two styles of string formatting, one providing a large degree of flexibility and customization (see `str.format()`, *Format String Syntax* and *Custom String Formatting*) and the other based on C `printf` style formatting that handles a narrower range of types and is slightly harder to use correctly, but is often faster for the cases it can handle (*printf-style String Formatting*).

The *Text Processing Services* section of the standard library covers a number of other modules that provide various text related utilities (including regular expression support in the `re` module).

`str.`**`capitalize`**`()`

Return a copy of the string with its first character capitalized and the rest lowercased.

Changed in version 3.8: The first character is now put into titlecase rather than uppercase. This means that characters like digraphs will only have their first letter capitalized, instead of the full character.

`str.`**`casefold`**`()`

Return a casefolded copy of the string. Casefolded strings may be used for caseless matching.

Casefolding is similar to lowercasing but more aggressive because it is intended to remove all case distinctions in a string. For example, the German lowercase letter `'ß'` is equivalent to `"ss"`. Since it is already lowercase, `lower()` would do nothing to `'ß'`; `casefold()` converts it to `"ss"`.

The casefolding algorithm is described in section 3.13 'Default Case Folding' of the Unicode Standard.

Added in version 3.3.

`str.`**`center`**`(`*width*`[`, *fillchar*`])`

Return centered in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

`str.`**`count`**`(`*sub*`[`, *start*`[`, *end*`]])`

Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

If *sub* is empty, returns the number of empty strings between characters which is the length of the string plus one.

`str.`**`encode`**`(`*encoding='utf-8'*, *errors='strict'*`)`

Return the string encoded to `bytes`.

encoding defaults to `'utf-8'`; see *Standard Encodings* for possible values.

errors controls how encoding errors are handled. If `'strict'` (the default), a `UnicodeError` exception is raised. Other possible values are `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` and any other name registered via `codecs.register_error()`. See *Error Handlers* for details.

For performance reasons, the value of *errors* is not checked for validity unless an encoding error actually occurs, *Python Development Mode* is enabled or a debug build is used.

Changed in version 3.1: Added support for keyword arguments.

Changed in version 3.9: The value of the *errors* argument is now checked in *Python Development Mode* and in debug mode.

str.**endswith**(*suffix*[, *start*[, *end*]])

Return `True` if the string ends with the specified *suffix*, otherwise return `False`. *suffix* can also be a tuple of suffixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

str.**expandtabs**(*tabsize=8*)

Return a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. Tab positions occur every *tabsize* characters (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the string, the current column is set to zero and the string is examined character by character. If the character is a tab (`\t`), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the character is a newline (`\n`) or return (`\r`), it is copied and the current column is reset to zero. Any other character is copied unchanged and the current column is incremented by one regardless of how the character is represented when printed.

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012     0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012 0123    01234'
```

str.**find**(*sub*[, *start*[, *end*]])

Return the lowest index in the string where substring *sub* is found within the slice `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` if *sub* is not found.

> **Note**
>
> The *find()* method should be used only if you need to know the position of *sub*. To check if *sub* is a substring or not, use the `in` operator:
>
> ```
> >>> 'Py' in 'Python'
> True
> ```

str.**format**(**args*, ***kwargs*)

Perform a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces `{}`. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

See *Format String Syntax* for a description of the various formatting options that can be specified in format strings.

> **Note**
>
> When formatting a number (*int*, *float*, *complex*, *decimal.Decimal* and subclasses) with the `n` type (ex: `'{:n}'.format(1234)`), the function temporarily sets the `LC_CTYPE` locale to the `LC_NUMERIC` locale to decode `decimal_point` and `thousands_sep` fields of `localeconv()` if they are non-ASCII or longer than 1 byte, and the `LC_NUMERIC` locale is different than the `LC_CTYPE` locale. This temporary change affects other threads.

Changed in version 3.7: When formatting a number with the `n` type, the function sets temporarily the `LC_CTYPE` locale to the `LC_NUMERIC` locale in some cases.

str.**format_map**(*mapping*, /)

Similar to `str.format(**mapping)`, except that `mapping` is used directly and not copied to a *dict*. This is useful if for example `mapping` is a dict subclass:

```
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...
>>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'
```

Added in version 3.2.

str.**index**(*sub*[, *start*[, *end*]])

Like *find()*, but raise *ValueError* when the substring is not found.

str.**isalnum**()

Return `True` if all characters in the string are alphanumeric and there is at least one character, `False` otherwise. A character `c` is alphanumeric if one of the following returns `True`: `c.isalpha()`, `c.isdecimal()`, `c.isdigit()`, or `c.isnumeric()`.

str.**isalpha**()

Return `True` if all characters in the string are alphabetic and there is at least one character, `False` otherwise. Alphabetic characters are those characters defined in the Unicode character database as "Letter", i.e., those with general category property being one of "Lm", "Lt", "Lu", "Ll", or "Lo". Note that this is different from the Alphabetic property defined in the section 4.10 'Letters, Alphabetic, and Ideographic' of the Unicode Standard.

str.**isascii**()

Return `True` if the string is empty or all characters in the string are ASCII, `False` otherwise. ASCII characters have code points in the range U+0000-U+007F.

Added in version 3.7.

str.**isdecimal**()

Return `True` if all characters in the string are decimal characters and there is at least one character, `False` otherwise. Decimal characters are those that can be used to form numbers in base 10, e.g. U+0660, ARABIC-INDIC DIGIT ZERO. Formally a decimal character is a character in the Unicode General Category "Nd".

str.**isdigit**()

Return `True` if all characters in the string are digits and there is at least one character, `False` otherwise. Digits include decimal characters and digits that need special handling, such as the compatibility superscript digits. This covers digits which cannot be used to form numbers in base 10, like the Kharosthi numbers. Formally, a digit is a character that has the property value Numeric_Type=Digit or Numeric_Type=Decimal.

str.**isidentifier**()

Return `True` if the string is a valid identifier according to the language definition, section identifiers.

keyword.iskeyword() can be used to test whether string `s` is a reserved identifier, such as `def` and `class`.

Example:

```
>>> from keyword import iskeyword

>>> 'hello'.isidentifier(), iskeyword('hello')
(True, False)
>>> 'def'.isidentifier(), iskeyword('def')
(True, True)
```

str.**islower**()

Return `True` if all cased characters[4] in the string are lowercase and there is at least one cased character, `False` otherwise.

str.**isnumeric**()

Return `True` if all characters in the string are numeric characters, and there is at least one character, `False` otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH. Formally, numeric characters are those with the property value Numeric_Type=Digit, Numeric_Type=Decimal or Numeric_Type=Numeric.

str.**isprintable**()

Return `True` if all characters in the string are printable or the string is empty, `False` otherwise. Nonprintable characters are those characters defined in the Unicode character database as "Other" or "Separator", excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when *repr()* is invoked on a string. It has no bearing on the handling of strings written to *sys.stdout* or *sys.stderr*.)

str.**isspace**()

Return `True` if there are only whitespace characters in the string and there is at least one character, `False` otherwise.

A character is *whitespace* if in the Unicode character database (see *unicodedata*), either its general category is `Zs` ("Separator, space"), or its bidirectional class is one of `WS`, `B`, or `S`.

str.**istitle**()

Return `True` if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return `False` otherwise.

str.**isupper**()

Return `True` if all cased characters[4] in the string are uppercase and there is at least one cased character, `False` otherwise.

```
>>> 'BANANA'.isupper()
True
>>> 'banana'.isupper()
False
>>> 'baNana'.isupper()
False
>>> ' '.isupper()
False
```

str.**join**(*iterable*)

Return a string which is the concatenation of the strings in *iterable*. A *TypeError* will be raised if there are any non-string values in *iterable*, including *bytes* objects. The separator between elements is the string providing this method.

str.**ljust**(*width*[, *fillchar*])

Return the string left justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

str.**lower**()

Return a copy of the string with all the cased characters[4] converted to lowercase.

The lowercasing algorithm used is described in section 3.13 'Default Case Folding' of the Unicode Standard.

[4] Cased characters are those with general category property being one of "Lu" (Letter, uppercase), "Ll" (Letter, lowercase), or "Lt" (Letter, titlecase).

str.**lstrip**([*chars*])

Return a copy of the string with leading characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.lstrip()
'spacious   '
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

See *str.removeprefix()* for a method that will remove a single prefix string rather than all of a set of characters. For example:

```
>>> 'Arthur: three!'.lstrip('Arthur: ')
'ee!'
>>> 'Arthur: three!'.removeprefix('Arthur: ')
'three!'
```

static str.**maketrans**(*x*[, *y*[, *z*]])

This static method returns a translation table usable for *str.translate()*.

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters (strings of length 1) to Unicode ordinals, strings (of arbitrary lengths) or `None`. Character keys will then be converted to ordinals.

If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to `None` in the result.

str.**partition**(*sep*)

Split the string at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

str.**removeprefix**(*prefix*, /)

If the string starts with the *prefix* string, return `string[len(prefix):]`. Otherwise, return a copy of the original string:

```
>>> 'TestHook'.removeprefix('Test')
'Hook'
>>> 'BaseTestCase'.removeprefix('Test')
'BaseTestCase'
```

Added in version 3.9.

str.**removesuffix**(*suffix*, /)

If the string ends with the *suffix* string and that *suffix* is not empty, return `string[:-len(suffix)]`. Otherwise, return a copy of the original string:

```
>>> 'MiscTests'.removesuffix('Tests')
'Misc'
>>> 'TmpDirMixin'.removesuffix('Tests')
'TmpDirMixin'
```

Added in version 3.9.

str.**replace**(*old*, *new*, *count=-1*)

Return a copy of the string with all occurrences of substring *old* replaced by *new*. If *count* is given, only the first *count* occurrences are replaced. If *count* is not specified or `-1`, then all occurrences are replaced.

Changed in version 3.13: *count* is now supported as a keyword argument.

str.**rfind**(*sub*[, *start*[, *end*]])

Return the highest index in the string where substring *sub* is found, such that *sub* is contained within `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` on failure.

str.**rindex**(*sub*[, *start*[, *end*]])

Like *rfind()* but raises *ValueError* when the substring *sub* is not found.

str.**rjust**(*width*[, *fillchar*])

Return the string right justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

str.**rpartition**(*sep*)

Split the string at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

str.**rsplit**(*sep=None*, *maxsplit=-1*)

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done, the *rightmost* ones. If *sep* is not specified or `None`, any whitespace string is a separator. Except for splitting from the right, *rsplit()* behaves like *split()* which is described in detail below.

str.**rstrip**([*chars*])

Return a copy of the string with trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

See *str.removesuffix()* for a method that will remove a single suffix string rather than all of a set of characters. For example:

```
>>> 'Monty Python'.rstrip(' Python')
'M'
>>> 'Monty Python'.removesuffix(' Python')
'Monty'
```

str.**split**(*sep=None*, *maxsplit=-1*)

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters as a single delimiter (to split with multiple delimiters, use *re.split()*). Splitting an empty string with a specified separator returns `['']`.

For example:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
>>> '1<>2<>3<4'.split('<>')
['1', '2', '3<4']
```

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

For example:

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> '   1   2   3   '.split()
['1', '2', '3']
```

`str.`**`splitlines`**(*keepends=False*)

Return a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless *keepends* is given and true.

This method splits on the following line boundaries. In particular, the boundaries are a superset of *universal newlines*.

Representation	Description
`\n`	Line Feed
`\r`	Carriage Return
`\r\n`	Carriage Return + Line Feed
`\v` or `\x0b`	Line Tabulation
`\f` or `\x0c`	Form Feed
`\x1c`	File Separator
`\x1d`	Group Separator
`\x1e`	Record Separator
`\x85`	Next Line (C1 Control Code)
`\u2028`	Line Separator
`\u2029`	Paragraph Separator

Changed in version 3.2: `\v` and `\f` added to list of line boundaries.

For example:

```
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

Unlike *`split()`* when a delimiter string *sep* is given, this method returns an empty list for the empty string, and a terminal line break does not result in an extra line:

```
>>> "".splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

For comparison, `split('\n')` gives:

```
>>> ''.split('\n')
['']
>>> 'Two lines\n'.split('\n')
['Two lines', '']
```

`str.`**`startswith`**`(`*prefix*`[`, *start*`[`, *end*`]]`)

Return `True` if string starts with the *prefix*, otherwise return `False`. *prefix* can also be a tuple of prefixes to look for. With optional *start*, test string beginning at that position. With optional *end*, stop comparing string at that position.

`str.`**`strip`**`(`*chars*`])`

Return a copy of the string with the leading and trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix or suffix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

The outermost leading and trailing *chars* argument values are stripped from the string. Characters are removed from the leading end until reaching a string character that is not contained in the set of characters in *chars*. A similar action takes place on the trailing end. For example:

```
>>> comment_string = '#....... Section 3.2.1 Issue #32 .......'
>>> comment_string.strip('.#! ')
'Section 3.2.1 Issue #32'
```

`str.`**`swapcase`**`()`

Return a copy of the string with uppercase characters converted to lowercase and vice versa. Note that it is not necessarily true that `s.swapcase().swapcase() == s`.

`str.`**`title`**`()`

Return a titlecased version of the string where words start with an uppercase character and the remaining characters are lowercase.

For example:

```
>>> 'Hello world'.title()
'Hello World'
```

The algorithm uses a simple language-independent definition of a word as groups of consecutive letters. The definition works in many contexts but it means that apostrophes in contractions and possessives form word boundaries, which may not be the desired result:

```
>>> "they're bill's friends from the UK".title()
"They'Re Bill'S Friends From The Uk"
```

The *`string.capwords()`* function does not have this problem, as it splits words on spaces only.

Alternatively, a workaround for apostrophes can be constructed using regular expressions:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0).capitalize(),
...                   s)
...
>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

`str.`**`translate`**`(`*table*`)`

Return a copy of the string in which each character has been mapped through the given translation table. The table must be an object that implements indexing via `__getitem__()`, typically a *mapping* or *sequence*. When indexed by a Unicode ordinal (an integer), the table object can do any of the following: return a Unicode ordinal or a string, to map the character to one or more other characters; return `None`, to delete the character from the return string; or raise a *`LookupError`* exception, to map the character to itself.

You can use *`str.maketrans()`* to create a translation map from character-to-character mappings in different formats.

See also the *`codecs`* module for a more flexible approach to custom character mappings.

`str.`**`upper`**`()`

Return a copy of the string with all the cased characters[Page 55, 4] converted to uppercase. Note that `s.upper().isupper()` might be `False` if `s` contains uncased characters or if the Unicode category of the resulting character(s) is not "Lu" (Letter, uppercase), but e.g. "Lt" (Letter, titlecase).

The uppercasing algorithm used is described in section 3.13 'Default Case Folding' of the Unicode Standard.

`str.`**`zfill`**`(`*width*`)`

Return a copy of the string left filled with ASCII `'0'` digits to make a string of length *width*. A leading sign prefix (`'+'`/`'-'`) is handled by inserting the padding *after* the sign character rather than before. The original string is returned if *width* is less than or equal to `len(s)`.

For example:

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

4.8.2 `printf`-style String Formatting

> **Note**
>
> The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). Using the newer formatted string literals, the *`str.format()`* interface, or *template strings* may help avoid these errors. Each of these alternatives provides their own trade-offs and benefits of simplicity, flexibility, and/or extensibility.

String objects have one unique built-in operation: the `%` operator (modulo). This is also known as the string *formatting* or *interpolation* operator. Given `format % values` (where *format* is a string), `%` conversion specifications in *format* are replaced with zero or more elements of *values*. The effect is similar to using the `sprintf()` function in the C language. For example:

```
>>> print('%s has %d quote types.' % ('Python', 2))
Python has 2 quote types.
```

If *format* requires a single argument, *values* may be a single non-tuple object.[5] Otherwise, *values* must be a tuple with exactly the number of items specified by the format string, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The `'%'` character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence of characters (for example, `(somename)`).
3. Conversion flags (optional), which affect the result of some conversion types.
4. Minimum field width (optional). If specified as an `'*'` (asterisk), the actual width is read from the next element of the tuple in *values*, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a `'.'` (dot) followed by the precision. If specified as `'*'` (an asterisk), the actual precision is read from the next element of the tuple in *values*, and the value to convert comes after the precision.
6. Length modifier (optional).
7. Conversion type.

When the right argument is a dictionary (or other mapping type), then the formats in the string *must* include a parenthesised mapping key into that dictionary inserted immediately after the `'%'` character. The mapping key selects the value to be formatted from the mapping. For example:

```
>>> print('%(language)s has %(number)03d quote types.' %
...       {'language': "Python", "number": 2})
Python has 002 quote types.
```

In this case no `*` specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

Flag	Meaning
`'#'`	The value conversion will use the "alternate form" (where defined below).
`'0'`	The conversion will be zero padded for numeric values.
`'-'`	The converted value is left adjusted (overrides the `'0'` conversion if both are given).
`' '`	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
`'+'`	A sign character (`'+'` or `'-'`) will precede the conversion (overrides a "space" flag).

A length modifier (`h`, `l`, or `L`) may be present, but is ignored as it is not necessary for Python – so e.g. `%ld` is identical to `%d`.

The conversion types are:

[5] To format only a tuple you should therefore provide a singleton tuple whose only element is the tuple to be formatted.

Conversion	Meaning	Notes
`'d'`	Signed integer decimal.	
`'i'`	Signed integer decimal.	
`'o'`	Signed octal value.	(1)
`'u'`	Obsolete type – it is identical to `'d'`.	(6)
`'x'`	Signed hexadecimal (lowercase).	(2)
`'X'`	Signed hexadecimal (uppercase).	(2)
`'e'`	Floating-point exponential format (lowercase).	(3)
`'E'`	Floating-point exponential format (uppercase).	(3)
`'f'`	Floating-point decimal format.	(3)
`'F'`	Floating-point decimal format.	(3)
`'g'`	Floating-point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'G'`	Floating-point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'c'`	Single character (accepts integer or single character string).	
`'r'`	String (converts any Python object using *repr()*).	(5)
`'s'`	String (converts any Python object using *str()*).	(5)
`'a'`	String (converts any Python object using *ascii()*).	(5)
`'%'`	No argument is converted, results in a `'%'` character in the result.	

Notes:

(1) The alternate form causes a leading octal specifier (`'0o'`) to be inserted before the first digit.

(2) The alternate form causes a leading `'0x'` or `'0X'` (depending on whether the `'x'` or `'X'` format was used) to be inserted before the first digit.

(3) The alternate form causes the result to always contain a decimal point, even if no digits follow it.

The precision determines the number of digits after the decimal point and defaults to 6.

(4) The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be.

The precision determines the number of significant digits before and after the decimal point and defaults to 6.

(5) If precision is `N`, the output is truncated to `N` characters.

(6) See **PEP 237**.

Since Python strings have an explicit length, `%s` conversions do not assume that `'\0'` is the end of the string.

Changed in version 3.1: `%f` conversions for numbers whose absolute value is over 1e50 are no longer replaced by `%g` conversions.

4.9 Binary Sequence Types — bytes, bytearray, memoryview

The core built-in types for manipulating binary data are *bytes* and *bytearray*. They are supported by *memoryview* which uses the buffer protocol to access the memory of other binary objects without needing to make a copy.

The *array* module supports efficient storage of basic data types like 32-bit integers and IEEE754 double-precision floating values.

4.9.1 Bytes Objects

Bytes objects are immutable sequences of single bytes. Since many major binary protocols are based on the ASCII text encoding, bytes objects offer several methods that are only valid when working with ASCII compatible data and are closely related to string objects in a variety of other ways.

class bytes([*source*[, *encoding*[, *errors*]]])

Firstly, the syntax for bytes literals is largely the same as that for string literals, except that a `b` prefix is added:

- Single quotes: `b'still allows embedded "double" quotes'`
- Double quotes: `b"still allows embedded 'single' quotes"`
- Triple quoted: `b'''3 single quotes'''`, `b"""3 double quotes"""`

Only ASCII characters are permitted in bytes literals (regardless of the declared source code encoding). Any binary values over 127 must be entered into bytes literals using the appropriate escape sequence.

As with string literals, bytes literals may also use a `r` prefix to disable processing of escape sequences. See strings for more about the various forms of bytes literal, including supported escape sequences.

While bytes literals and representations are based on ASCII text, bytes objects actually behave like immutable sequences of integers, with each value in the sequence restricted such that `0 <= x < 256` (attempts to violate this restriction will trigger *ValueError*). This is done deliberately to emphasise that while many binary formats include ASCII based elements and can be usefully manipulated with some text-oriented algorithms, this is not generally the case for arbitrary binary data (blindly applying text processing algorithms to binary data formats that are not ASCII compatible will usually lead to data corruption).

In addition to the literal forms, bytes objects can be created in a number of other ways:

- A zero-filled bytes object of a specified length: `bytes(10)`
- From an iterable of integers: `bytes(range(20))`
- Copying existing binary data via the buffer protocol: `bytes(obj)`

Also see the *bytes* built-in.

Since 2 hexadecimal digits correspond precisely to a single byte, hexadecimal numbers are a commonly used format for describing binary data. Accordingly, the bytes type has an additional class method to read data in that format:

classmethod fromhex(*string*)

This *bytes* class method returns a bytes object, decoding the given string object. The string must contain two hexadecimal digits per byte, with ASCII whitespace being ignored.

```
>>> bytes.fromhex('2Ef0 F1f2  ')
b'.\xf0\xf1\xf2'
```

Changed in version 3.7: *bytes.fromhex()* now skips all ASCII whitespace in the string, not just spaces.

A reverse conversion function exists to transform a bytes object into its hexadecimal representation.

hex([*sep*[, *bytes_per_sep*]])

Return a string object containing two hexadecimal digits for each byte in the instance.

```
>>> b'\xf0\xf1\xf2'.hex()
'f0f1f2'
```

If you want to make the hex string easier to read, you can specify a single character separator *sep* parameter to include in the output. By default, this separator will be included between each byte. A second optional *bytes_per_sep* parameter controls the spacing. Positive values calculate the separator position from the right, negative values from the left.

```
>>> value = b'\xf0\xf1\xf2'
>>> value.hex('-')
'f0-f1-f2'
>>> value.hex('_', 2)
'f0_f1f2'
>>> b'UUDDLRLRAB'.hex(' ', -4)
'55554444 4c524c52 4142'
```

Added in version 3.5.

Changed in version 3.8: *bytes.hex()* now supports optional *sep* and *bytes_per_sep* parameters to insert separators between bytes in the hex output.

Since bytes objects are sequences of integers (akin to a tuple), for a bytes object *b*, `b[0]` will be an integer, while `b[0:1]` will be a bytes object of length 1. (This contrasts with text strings, where both indexing and slicing will produce a string of length 1)

The representation of bytes objects uses the literal format (`b'...'`) since it is often more useful than e.g. `bytes([46, 46, 46])`. You can always convert a bytes object into a list of integers using `list(b)`.

4.9.2 Bytearray Objects

bytearray objects are a mutable counterpart to *bytes* objects.

class bytearray([*source*[, *encoding*[, *errors*]]])

There is no dedicated literal syntax for bytearray objects, instead they are always created by calling the constructor:

- Creating an empty instance: `bytearray()`
- Creating a zero-filled instance with a given length: `bytearray(10)`
- From an iterable of integers: `bytearray(range(20))`
- Copying existing binary data via the buffer protocol: `bytearray(b'Hi!')`

As bytearray objects are mutable, they support the *mutable* sequence operations in addition to the common bytes and bytearray operations described in *Bytes and Bytearray Operations*.

Also see the *bytearray* built-in.

Since 2 hexadecimal digits correspond precisely to a single byte, hexadecimal numbers are a commonly used format for describing binary data. Accordingly, the bytearray type has an additional class method to read data in that format:

classmethod fromhex(*string*)

This *bytearray* class method returns bytearray object, decoding the given string object. The string must contain two hexadecimal digits per byte, with ASCII whitespace being ignored.

```
>>> bytearray.fromhex('2Ef0 F1f2  ')
bytearray(b'.\xf0\xf1\xf2')
```

Changed in version 3.7: *bytearray.fromhex()* now skips all ASCII whitespace in the string, not just spaces.

A reverse conversion function exists to transform a bytearray object into its hexadecimal representation.

hex([*sep*[, *bytes_per_sep*]])

Return a string object containing two hexadecimal digits for each byte in the instance.

```
>>> bytearray(b'\xf0\xf1\xf2').hex()
'f0f1f2'
```

Added in version 3.5.

Changed in version 3.8: Similar to *bytes.hex()*, *bytearray.hex()* now supports optional *sep* and *bytes_per_sep* parameters to insert separators between bytes in the hex output.

Since bytearray objects are sequences of integers (akin to a list), for a bytearray object *b*, `b[0]` will be an integer, while `b[0:1]` will be a bytearray object of length 1. (This contrasts with text strings, where both indexing and slicing will produce a string of length 1)

The representation of bytearray objects uses the bytes literal format (`bytearray(b'...')`) since it is often more useful than e.g. `bytearray([46, 46, 46])`. You can always convert a bytearray object into a list of integers using `list(b)`.

4.9.3 Bytes and Bytearray Operations

Both bytes and bytearray objects support the *common* sequence operations. They interoperate not just with operands of the same type, but with any *bytes-like object*. Due to this flexibility, they can be freely mixed in operations without causing errors. However, the return type of the result may depend on the order of operands.

Note

The methods on bytes and bytearray objects don't accept strings as their arguments, just as the methods on strings don't accept bytes as their arguments. For example, you have to write:

```
a = "abc"
b = a.replace("a", "f")
```

and:

```
a = b"abc"
b = a.replace(b"a", b"f")
```

Some bytes and bytearray operations assume the use of ASCII compatible binary formats, and hence should be avoided when working with arbitrary binary data. These restrictions are covered below.

Note

Using these ASCII based operations to manipulate binary data that is not stored in an ASCII based format may lead to data corruption.

The following methods on bytes and bytearray objects can be used with arbitrary binary data.

bytes.**count**(*sub*[, *start*[, *end*]])

bytearray.**count**(*sub*[, *start*[, *end*]])

Return the number of non-overlapping occurrences of subsequence *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

If *sub* is empty, returns the number of empty slices between characters which is the length of the bytes object plus one.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

bytes.**removeprefix**(*prefix*, /)

bytearray.**removeprefix**(*prefix*, /)

If the binary data starts with the *prefix* string, return `bytes[len(prefix):]`. Otherwise, return a copy of the original binary data:

```
>>> b'TestHook'.removeprefix(b'Test')
b'Hook'
>>> b'BaseTestCase'.removeprefix(b'Test')
b'BaseTestCase'
```

The *prefix* may be any *bytes-like object*.

> **Note**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

Added in version 3.9.

bytes.**removesuffix**(*suffix*, /)

bytearray.**removesuffix**(*suffix*, /)

If the binary data ends with the *suffix* string and that *suffix* is not empty, return `bytes[:-len(suffix)]`. Otherwise, return a copy of the original binary data:

```
>>> b'MiscTests'.removesuffix(b'Tests')
b'Misc'
>>> b'TmpDirMixin'.removesuffix(b'Tests')
b'TmpDirMixin'
```

The *suffix* may be any *bytes-like object*.

> **Note**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

Added in version 3.9.

bytes.**decode**(*encoding='utf-8'*, *errors='strict'*)

bytearray.**decode**(*encoding='utf-8'*, *errors='strict'*)

Return the bytes decoded to a *str*.

encoding defaults to `'utf-8'`; see *Standard Encodings* for possible values.

errors controls how decoding errors are handled. If `'strict'` (the default), a *UnicodeError* exception is raised. Other possible values are `'ignore'`, `'replace'`, and any other name registered via *codecs.register_error()*. See *Error Handlers* for details.

For performance reasons, the value of *errors* is not checked for validity unless a decoding error actually occurs, *Python Development Mode* is enabled or a debug build is used.

> **Note**
>
> Passing the *encoding* argument to *str* allows decoding any *bytes-like object* directly, without needing to make a temporary `bytes` or `bytearray` object.

Changed in version 3.1: Added support for keyword arguments.

Changed in version 3.9: The value of the *errors* argument is now checked in *Python Development Mode* and in debug mode.

`bytes.`**`endswith`**(*suffix*[, *start*[, *end*]])

`bytearray.`**`endswith`**(*suffix*[, *start*[, *end*]])

Return `True` if the binary data ends with the specified *suffix*, otherwise return `False`. *suffix* can also be a tuple of suffixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

The suffix(es) to search for may be any *bytes-like object*.

`bytes.`**`find`**(*sub*[, *start*[, *end*]])

`bytearray.`**`find`**(*sub*[, *start*[, *end*]])

Return the lowest index in the data where the subsequence *sub* is found, such that *sub* is contained in the slice `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` if *sub* is not found.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

> **Note**
>
> The *find()* method should be used only if you need to know the position of *sub*. To check if *sub* is a substring or not, use the `in` operator:
>
> ```
> >>> b'Py' in b'Python'
> True
> ```

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

`bytes.`**`index`**(*sub*[, *start*[, *end*]])

`bytearray.`**`index`**(*sub*[, *start*[, *end*]])

Like *find()*, but raise *ValueError* when the subsequence is not found.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

`bytes.`**`join`**(*iterable*)

bytearray.**join**(*iterable*)

Return a bytes or bytearray object which is the concatenation of the binary data sequences in *iterable*. A `TypeError` will be raised if there are any values in *iterable* that are not *bytes-like objects*, including `str` objects. The separator between elements is the contents of the bytes or bytearray object providing this method.

static bytes.**maketrans**(*from*, *to*)

static bytearray.**maketrans**(*from*, *to*)

This static method returns a translation table usable for `bytes.translate()` that will map each character in *from* into the character at the same position in *to*; *from* and *to* must both be *bytes-like objects* and have the same length.

Added in version 3.1.

bytes.**partition**(*sep*)

bytearray.**partition**(*sep*)

Split the sequence at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself or its bytearray copy, and the part after the separator. If the separator is not found, return a 3-tuple containing a copy of the original sequence, followed by two empty bytes or bytearray objects.

The separator to search for may be any *bytes-like object*.

bytes.**replace**(*old*, *new*[, *count*])

bytearray.**replace**(*old*, *new*[, *count*])

Return a copy of the sequence with all occurrences of subsequence *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

The subsequence to search for and its replacement may be any *bytes-like object*.

> **Note**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**rfind**(*sub*[, *start*[, *end*]])

bytearray.**rfind**(*sub*[, *start*[, *end*]])

Return the highest index in the sequence where the subsequence *sub* is found, such that *sub* is contained within `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` on failure.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

bytes.**rindex**(*sub*[, *start*[, *end*]])

bytearray.**rindex**(*sub*[, *start*[, *end*]])

Like `rfind()` but raises `ValueError` when the subsequence *sub* is not found.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

bytes.**rpartition**(*sep*)

bytearray.**rpartition**(*sep*)

Split the sequence at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself or its bytearray copy, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty bytes or bytearray objects, followed by a copy of the original sequence.

The separator to search for may be any *bytes-like object*.

`bytes.`**`startswith`**(*prefix*[, *start*[, *end*]])

`bytearray.`**`startswith`**(*prefix*[, *start*[, *end*]])

Return `True` if the binary data starts with the specified *prefix*, otherwise return `False`. *prefix* can also be a tuple of prefixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

The prefix(es) to search for may be any *bytes-like object*.

`bytes.`**`translate`**(*table*, /, *delete=b''*)

`bytearray.`**`translate`**(*table*, /, *delete=b''*)

Return a copy of the bytes or bytearray object where all bytes occurring in the optional argument *delete* are removed, and the remaining bytes have been mapped through the given translation table, which must be a bytes object of length 256.

You can use the `bytes.maketrans()` method to create a translation table.

Set the *table* argument to `None` for translations that only delete characters:

```
>>> b'read this short text'.translate(None, b'aeiou')
b'rd ths shrt txt'
```

Changed in version 3.6: *delete* is now supported as a keyword argument.

The following methods on bytes and bytearray objects have default behaviours that assume the use of ASCII compatible binary formats, but can still be used with arbitrary binary data by passing appropriate arguments. Note that all of the bytearray methods in this section do *not* operate in place, and instead produce new objects.

`bytes.`**`center`**(*width*[, *fillbyte*])

`bytearray.`**`center`**(*width*[, *fillbyte*])

Return a copy of the object centered in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For `bytes` objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

> **Note**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.`**`ljust`**(*width*[, *fillbyte*])

`bytearray.`**`ljust`**(*width*[, *fillbyte*])

Return a copy of the object left justified in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For `bytes` objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

> **Note**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.`**`lstrip`**([*chars*])

bytearray.**lstrip**([*chars*])

Return a copy of the sequence with specified leading bytes removed. The *chars* argument is a binary sequence specifying the set of byte values to be removed - the name refers to the fact this method is usually used with ASCII characters. If omitted or `None`, the *chars* argument defaults to removing ASCII whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped:

```
>>> b'   spacious   '.lstrip()
b'spacious   '
>>> b'www.example.com'.lstrip(b'cmowz.')
b'example.com'
```

The binary sequence of byte values to remove may be any *bytes-like object*. See `removeprefix()` for a method that will remove a single prefix string rather than all of a set of characters. For example:

```
>>> b'Arthur: three!'.lstrip(b'Arthur: ')
b'ee!'
>>> b'Arthur: three!'.removeprefix(b'Arthur: ')
b'three!'
```

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**rjust**(*width*[, *fillbyte*])

bytearray.**rjust**(*width*[, *fillbyte*])

Return a copy of the object right justified in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For `bytes` objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**rsplit**(*sep=None*, *maxsplit=-1*)

bytearray.**rsplit**(*sep=None*, *maxsplit=-1*)

Split the binary sequence into subsequences of the same type, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done, the *rightmost* ones. If *sep* is not specified or `None`, any subsequence consisting solely of ASCII whitespace is a separator. Except for splitting from the right, `rsplit()` behaves like `split()` which is described in detail below.

bytes.**rstrip**([*chars*])

bytearray.**rstrip**([*chars*])

Return a copy of the sequence with specified trailing bytes removed. The *chars* argument is a binary sequence specifying the set of byte values to be removed - the name refers to the fact this method is usually used with ASCII characters. If omitted or `None`, the *chars* argument defaults to removing ASCII whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped:

```
>>> b'   spacious   '.rstrip()
b'   spacious'
```

(continues on next page)

(continued from previous page)

```
>>> b'mississippi'.rstrip(b'ipz')
b'mississ'
```

The binary sequence of byte values to remove may be any *bytes-like object*. See `removesuffix()` for a method that will remove a single suffix string rather than all of a set of characters. For example:

```
>>> b'Monty Python'.rstrip(b' Python')
b'M'
>>> b'Monty Python'.removesuffix(b' Python')
b'Monty'
```

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.`**`split`**(*sep=None*, *maxsplit=-1*)

`bytearray.`**`split`**(*sep=None*, *maxsplit=-1*)

Split the binary sequence into subsequences of the same type, using *sep* as the delimiter string. If *maxsplit* is given and non-negative, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or is `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty subsequences (for example, `b'1,,2'.split(b',')` returns `[b'1', b'', b'2']`). The *sep* argument may consist of a multibyte sequence as a single delimiter. Splitting an empty sequence with a specified separator returns `[b'']` or `[bytearray(b'')]` depending on the type of object being split. The *sep* argument may be any *bytes-like object*.

For example:

```
>>> b'1,2,3'.split(b',')
[b'1', b'2', b'3']
>>> b'1,2,3'.split(b',', maxsplit=1)
[b'1', b'2,3']
>>> b'1,2,,3,'.split(b',')
[b'1', b'2', b'', b'3', b'']
>>> b'1<>2<>3<4'.split(b'<>')
[b'1', b'2', b'3<4']
```

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive ASCII whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the sequence has leading or trailing whitespace. Consequently, splitting an empty sequence or a sequence consisting solely of ASCII whitespace without a specified separator returns `[]`.

For example:

```
>>> b'1 2 3'.split()
[b'1', b'2', b'3']
>>> b'1 2 3'.split(maxsplit=1)
[b'1', b'2 3']
>>> b'   1   2   3   '.split()
[b'1', b'2', b'3']
```

`bytes.`**`strip`**([*chars*])

bytearray.**strip**([*chars*])

Return a copy of the sequence with specified leading and trailing bytes removed. The *chars* argument is a binary sequence specifying the set of byte values to be removed - the name refers to the fact this method is usually used with ASCII characters. If omitted or `None`, the *chars* argument defaults to removing ASCII whitespace. The *chars* argument is not a prefix or suffix; rather, all combinations of its values are stripped:

```
>>> b'   spacious   '.strip()
b'spacious'
>>> b'www.example.com'.strip(b'cmowz.')
b'example'
```

The binary sequence of byte values to remove may be any *bytes-like object*.

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

The following methods on bytes and bytearray objects assume the use of ASCII compatible binary formats and should not be applied to arbitrary binary data. Note that all of the bytearray methods in this section do *not* operate in place, and instead produce new objects.

bytes.**capitalize**()

bytearray.**capitalize**()

Return a copy of the sequence with each byte interpreted as an ASCII character, and the first byte capitalized and the rest lowercased. Non-ASCII byte values are passed through unchanged.

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**expandtabs**(*tabsize=8*)

bytearray.**expandtabs**(*tabsize=8*)

Return a copy of the sequence where all ASCII tab characters are replaced by one or more ASCII spaces, depending on the current column and the given tab size. Tab positions occur every *tabsize* bytes (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the sequence, the current column is set to zero and the sequence is examined byte by byte. If the byte is an ASCII tab character (`b'\t'`), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the current byte is an ASCII newline (`b'\n'`) or carriage return (`b'\r'`), it is copied and the current column is reset to zero. Any other byte value is copied unchanged and the current column is incremented by one regardless of how the byte value is represented when printed:

```
>>> b'01\t012\t0123\t01234'.expandtabs()
b'01      012     0123    01234'
>>> b'01\t012\t0123\t01234'.expandtabs(4)
b'01  012 0123    01234'
```

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**isalnum**()

bytearray.**isalnum**()

Return `True` if all bytes in the sequence are alphabetical ASCII characters or ASCII decimal digits and the sequence is not empty, `False` otherwise. Alphabetic ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`. ASCII decimal digits are those byte values in the sequence `b'0123456789'`.

For example:

```
>>> b'ABCabc1'.isalnum()
True
>>> b'ABC abc1'.isalnum()
False
```

bytes.**isalpha**()

bytearray.**isalpha**()

Return `True` if all bytes in the sequence are alphabetic ASCII characters and the sequence is not empty, `False` otherwise. Alphabetic ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

For example:

```
>>> b'ABCabc'.isalpha()
True
>>> b'ABCabc1'.isalpha()
False
```

bytes.**isascii**()

bytearray.**isascii**()

Return `True` if the sequence is empty or all bytes in the sequence are ASCII, `False` otherwise. ASCII bytes are in the range 0-0x7F.

Added in version 3.7.

bytes.**isdigit**()

bytearray.**isdigit**()

Return `True` if all bytes in the sequence are ASCII decimal digits and the sequence is not empty, `False` otherwise. ASCII decimal digits are those byte values in the sequence `b'0123456789'`.

For example:

```
>>> b'1234'.isdigit()
True
>>> b'1.23'.isdigit()
False
```

bytes.**islower**()

bytearray.**islower**()

Return `True` if there is at least one lowercase ASCII character in the sequence and no uppercase ASCII characters, `False` otherwise.

For example:

```
>>> b'hello world'.islower()
True
>>> b'Hello world'.islower()
False
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

`bytes.`**`isspace`**`()`

`bytearray.`**`isspace`**`()`

Return `True` if all bytes in the sequence are ASCII whitespace and the sequence is not empty, `False` otherwise. ASCII whitespace characters are those byte values in the sequence `b' \t\n\r\x0b\f'` (space, tab, newline, carriage return, vertical tab, form feed).

`bytes.`**`istitle`**`()`

`bytearray.`**`istitle`**`()`

Return `True` if the sequence is ASCII titlecase and the sequence is not empty, `False` otherwise. See *`bytes.title()`* for more details on the definition of "titlecase".

For example:

```
>>> b'Hello World'.istitle()
True
>>> b'Hello world'.istitle()
False
```

`bytes.`**`isupper`**`()`

`bytearray.`**`isupper`**`()`

Return `True` if there is at least one uppercase alphabetic ASCII character in the sequence and no lowercase ASCII characters, `False` otherwise.

For example:

```
>>> b'HELLO WORLD'.isupper()
True
>>> b'Hello world'.isupper()
False
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

`bytes.`**`lower`**`()`

`bytearray.`**`lower`**`()`

Return a copy of the sequence with all the uppercase ASCII characters converted to their corresponding lowercase counterpart.

For example:

```
>>> b'Hello World'.lower()
b'hello world'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

> **Note**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**splitlines**(*keepends=False*)

bytearray.**splitlines**(*keepends=False*)

Return a list of the lines in the binary sequence, breaking at ASCII line boundaries. This method uses the *universal newlines* approach to splitting lines. Line breaks are not included in the resulting list unless *keepends* is given and true.

For example:

```
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines()
[b'ab c', b'', b'de fg', b'kl']
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
[b'ab c\n', b'\n', b'de fg\r', b'kl\r\n']
```

Unlike *split()* when a delimiter string *sep* is given, this method returns an empty list for the empty string, and a terminal line break does not result in an extra line:

```
>>> b"".split(b'\n'), b"Two lines\n".split(b'\n')
([b''], [b'Two lines', b''])
>>> b"".splitlines(), b"One line\n".splitlines()
([], [b'One line'])
```

bytes.**swapcase**()

bytearray.**swapcase**()

Return a copy of the sequence with all the lowercase ASCII characters converted to their corresponding uppercase counterpart and vice-versa.

For example:

```
>>> b'Hello World'.swapcase()
b'hELLO wORLD'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Unlike *str.swapcase()*, it is always the case that `bin.swapcase().swapcase() == bin` for the binary versions. Case conversions are symmetrical in ASCII, even though that is not generally true for arbitrary Unicode code points.

> **Note**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**title**()

bytearray.**title**()

Return a titlecased version of the binary sequence where words start with an uppercase ASCII character and the remaining characters are lowercase. Uncased byte values are left unmodified.

For example:

```
>>> b'Hello world'.title()
b'Hello World'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. All other byte values are uncased.

The algorithm uses a simple language-independent definition of a word as groups of consecutive letters. The definition works in many contexts but it means that apostrophes in contractions and possessives form word boundaries, which may not be the desired result:

```
>>> b"they're bill's friends from the UK".title()
b"They'Re Bill'S Friends From The Uk"
```

A workaround for apostrophes can be constructed using regular expressions:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(rb"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0:1].upper() +
...                              mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase(b"they're bill's friends.")
b"They're Bill's Friends."
```

> **Note**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.`**`upper`**`()`

`bytearray.`**`upper`**`()`

Return a copy of the sequence with all the lowercase ASCII characters converted to their corresponding uppercase counterpart.

For example:

```
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

> **Note**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.`**`zfill`**`(`*width*`)`

`bytearray.`**`zfill`**`(`*width*`)`

Return a copy of the sequence left filled with ASCII `b'0'` digits to make a sequence of length *width*. A leading sign

prefix (`b'+'`/`b'-'`) is handled by inserting the padding *after* the sign character rather than before. For *bytes* objects, the original sequence is returned if *width* is less than or equal to `len(seq)`.

For example:

```
>>> b"42".zfill(5)
b'00042'
>>> b"-42".zfill(5)
b'-0042'
```

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

4.9.4 `printf`-style Bytes Formatting

Note

The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). If the value being printed may be a tuple or dictionary, wrap it in a tuple.

Bytes objects (`bytes`/`bytearray`) have one unique built-in operation: the `%` operator (modulo). This is also known as the bytes *formatting* or *interpolation* operator. Given `format % values` (where *format* is a bytes object), `%` conversion specifications in *format* are replaced with zero or more elements of *values*. The effect is similar to using the `sprintf()` in the C language.

If *format* requires a single argument, *values* may be a single non-tuple object.[Page 61, 5] Otherwise, *values* must be a tuple with exactly the number of items specified by the format bytes object, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The `'%'` character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence of characters (for example, `(somename)`).
3. Conversion flags (optional), which affect the result of some conversion types.
4. Minimum field width (optional). If specified as an `'*'` (asterisk), the actual width is read from the next element of the tuple in *values*, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a `'.'` (dot) followed by the precision. If specified as `'*'` (an asterisk), the actual precision is read from the next element of the tuple in *values*, and the value to convert comes after the precision.
6. Length modifier (optional).
7. Conversion type.

When the right argument is a dictionary (or other mapping type), then the formats in the bytes object *must* include a parenthesised mapping key into that dictionary inserted immediately after the `'%'` character. The mapping key selects the value to be formatted from the mapping. For example:

```
>>> print(b'%(language)s has %(number)03d quote types.' %
...       {b'language': b"Python", b"number": 2})
b'Python has 002 quote types.'
```

In this case no * specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

Flag	Meaning
'#'	The value conversion will use the "alternate form" (where defined below).
'0'	The conversion will be zero padded for numeric values.
'-'	The converted value is left adjusted (overrides the '0' conversion if both are given).
' '	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
'+'	A sign character ('+' or '-') will precede the conversion (overrides a "space" flag).

A length modifier (h, l, or L) may be present, but is ignored as it is not necessary for Python – so e.g. %ld is identical to %d.

The conversion types are:

Conversion	Meaning	Notes
'd'	Signed integer decimal.	
'i'	Signed integer decimal.	
'o'	Signed octal value.	(1)
'u'	Obsolete type – it is identical to 'd'.	(8)
'x'	Signed hexadecimal (lowercase).	(2)
'X'	Signed hexadecimal (uppercase).	(2)
'e'	Floating-point exponential format (lowercase).	(3)
'E'	Floating-point exponential format (uppercase).	(3)
'f'	Floating-point decimal format.	(3)
'F'	Floating-point decimal format.	(3)
'g'	Floating-point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
'G'	Floating-point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
'c'	Single byte (accepts integer or single byte objects).	
'b'	Bytes (any object that follows the buffer protocol or has __bytes__()).	(5)
's'	's' is an alias for 'b' and should only be used for Python2/3 code bases.	(6)
'a'	Bytes (converts any Python object using repr(obj).encode('ascii', 'backslashreplace')).	(5)
'r'	'r' is an alias for 'a' and should only be used for Python2/3 code bases.	(7)
'%'	No argument is converted, results in a '%' character in the result.	

Notes:

(1) The alternate form causes a leading octal specifier ('0o') to be inserted before the first digit.

(2) The alternate form causes a leading '0x' or '0X' (depending on whether the 'x' or 'X' format was used) to be inserted before the first digit.

(3) The alternate form causes the result to always contain a decimal point, even if no digits follow it.

The precision determines the number of digits after the decimal point and defaults to 6.

(4) The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be.

The precision determines the number of significant digits before and after the decimal point and defaults to 6.

(5) If precision is `N`, the output is truncated to `N` characters.

(6) `b'%s'` is deprecated, but will not be removed during the 3.x series.

(7) `b'%r'` is deprecated, but will not be removed during the 3.x series.

(8) See **PEP 237**.

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

See also

PEP 461 - Adding % formatting to bytes and bytearray

Added in version 3.5.

4.9.5 Memory Views

memoryview objects allow Python code to access the internal data of an object that supports the buffer protocol without copying.

class memoryview(*object*)

Create a *memoryview* that references *object*. *object* must support the buffer protocol. Built-in objects that support the buffer protocol include *bytes* and *bytearray*.

A *memoryview* has the notion of an *element*, which is the atomic memory unit handled by the originating *object*. For many simple types such as *bytes* and *bytearray*, an element is a single byte, but other types such as *array.array* may have bigger elements.

`len(view)` is equal to the length of *tolist*, which is the nested list representation of the view. If `view.ndim = 1`, this is equal to the number of elements in the view.

Changed in version 3.12: If `view.ndim == 0`, `len(view)` now raises *TypeError* instead of returning 1.

The *itemsize* attribute will give you the number of bytes in a single element.

A *memoryview* supports slicing and indexing to expose its data. One-dimensional slicing will result in a subview:

```
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
>>> bytes(v[1:4])
b'bce'
```

If *format* is one of the native format specifiers from the *struct* module, indexing with an integer or a tuple of integers is also supported and returns a single *element* with the correct type. One-dimensional memoryviews can be indexed with an integer or a one-integer tuple. Multi-dimensional memoryviews can be indexed with tuples of exactly *ndim* integers where *ndim* is the number of dimensions. Zero-dimensional memoryviews can be indexed with the empty tuple.

Here is an example with a non-byte format:

```
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -33333333, 44444444])
>>> m = memoryview(a)
>>> m[0]
-11111111
>>> m[-1]
44444444
>>> m[::2].tolist()
[-11111111, -33333333]
```

If the underlying object is writable, the memoryview supports one-dimensional slice assignment. Resizing is not allowed:

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
>>> data
bytearray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have different structures
>>> v[2:6] = b'spam'
>>> data
bytearray(b'z1spam')
```

One-dimensional memoryviews of *hashable* (read-only) types with formats 'B', 'b' or 'c' are also hashable. The hash is defined as `hash(m) == hash(m.tobytes())`:

```
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::-2]) == hash(b'abcefg'[::-2])
True
```

Changed in version 3.3: One-dimensional memoryviews can now be sliced. One-dimensional memoryviews with formats 'B', 'b' or 'c' are now *hashable*.

Changed in version 3.4: memoryview is now registered automatically with *collections.abc.Sequence*

Changed in version 3.5: memoryviews can now be indexed with tuple of integers.

memoryview has several methods:

__eq__(*exporter*)

A memoryview and a **PEP 3118** exporter are equal if their shapes are equivalent and if all corresponding values are equal when the operands' respective format codes are interpreted using *struct* syntax.

For the subset of *struct* format strings currently supported by *tolist()*, `v` and `w` are equal if `v.tolist() == w.tolist()`:

```
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

If either format string is not supported by the *struct* module, then the objects will always compare as unequal (even if the format strings and buffer contents are identical):

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

Note that, as with floating-point numbers, `v is w` does *not* imply `v == w` for memoryview objects.

Changed in version 3.3: Previous versions compared the raw memory disregarding the item format and the logical array structure.

tobytes(*order='C'*)

Return the data in the buffer as a bytestring. This is equivalent to calling the *bytes* constructor on the memoryview.

```
>>> m = memoryview(b"abc")
>>> m.tobytes()
b'abc'
>>> bytes(m)
b'abc'
```

For non-contiguous arrays the result is equal to the flattened list representation with all elements converted to bytes. *tobytes()* supports all format strings, including those that are not in *struct* module syntax.

Added in version 3.8: *order* can be {'C', 'F', 'A'}. When *order* is 'C' or 'F', the data of the original array is converted to C or Fortran order. For contiguous views, 'A' returns an exact copy of the physical memory. In

particular, in-memory Fortran order is preserved. For non-contiguous views, the data is converted to C first. *order=None* is the same as *order='C'*.

hex([*sep*[, *bytes_per_sep*]])

Return a string object containing two hexadecimal digits for each byte in the buffer.

```
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

Added in version 3.5.

Changed in version 3.8: Similar to *bytes.hex()*, *memoryview.hex()* now supports optional *sep* and *bytes_per_sep* parameters to insert separators between bytes in the hex output.

tolist()

Return the data in the buffer as a list of elements.

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

Changed in version 3.3: *tolist()* now supports all single character native formats in *struct* module syntax as well as multi-dimensional representations.

toreadonly()

Return a readonly version of the memoryview object. The original memoryview object is unchanged.

```
>>> m = memoryview(bytearray(b'abc'))
>>> mm = m.toreadonly()
>>> mm.tolist()
[97, 98, 99]
>>> mm[0] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot modify read-only memory
>>> m[0] = 43
>>> mm.tolist()
[43, 98, 99]
```

Added in version 3.8.

release()

Release the underlying buffer exposed by the memoryview object. Many objects take special actions when a view is held on them (for example, a *bytearray* would temporarily forbid resizing); therefore, calling release() is handy to remove these restrictions (and free any dangling resources) as soon as possible.

After this method has been called, any further operation on the view raises a *ValueError* (except *release()* itself which can be called multiple times):

```
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

The context management protocol can be used for a similar effect, using the `with` statement:

```
>>> with memoryview(b'abc') as m:
...     m[0]
...
97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

Added in version 3.2.

cast (*format*[, *shape*])

Cast a memoryview to a new format or shape. *shape* defaults to `[byte_length//new_itemsize]`, which means that the result view will be one-dimensional. The return value is a new memoryview, but the buffer itself is not copied. Supported casts are 1D -> C-*contiguous* and C-contiguous -> 1D.

The destination format is restricted to a single element native format in *struct* syntax. One of the formats must be a byte format ('B', 'b' or 'c'). The byte length of the result must be the same as the original length. Note that all byte lengths may depend on the operating system.

Cast 1D/long to 1D/unsigned bytes:

```
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24
```

Cast 1D/unsigned bytes to 1D/char:

```
>>> b = bytearray(b'zyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  ...
TypeError: memoryview: invalid type for format 'B'
>>> y = x.cast('c')
```

(continues on next page)

(continued from previous page)

```
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')
```

Cast 1D/bytes to 3D/ints to 1D/signed char:

```
>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
>>> len(y)
2
>>> y.nbytes
48
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
>>> z.nbytes
48
```

Cast 1D/unsigned long to 2D/unsigned long:

```
>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

Added in version 3.3.

Changed in version 3.5: The source format is no longer restricted when casting to a byte view.

There are also several readonly attributes available:

obj

The underlying object of the memoryview:

```
>>> b  = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True
```

Added in version 3.3.

nbytes

`nbytes == product(shape) * itemsize == len(m.tobytes())`. This is the amount of space in bytes that the array would use in a contiguous representation. It is not necessarily equal to `len(m)`:

```
>>> import array
>>> a = array.array('i', [1,2,3,4,5])
>>> m = memoryview(a)
>>> len(m)
5
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12
```

Multi-dimensional arrays:

```
>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3,4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0, 16.5]]
>>> len(y)
3
>>> y.nbytes
96
```

Added in version 3.3.

readonly

A bool indicating whether the memory is read only.

format

A string containing the format (in *struct* module style) for each element in the view. A memoryview can be created from exporters with arbitrary format strings, but some methods (e.g. *tolist()*) are restricted to native single element formats.

Changed in version 3.3: format `'B'` is now handled according to the struct module syntax. This means that `memoryview(b'abc')[0] == b'abc'[0] == 97`.

itemsize

The size in bytes of each element of the memoryview:

```
>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True
```

ndim

An integer indicating how many dimensions of a multi-dimensional array the memory represents.

shape

A tuple of integers the length of *ndim* giving the shape of the memory as an N-dimensional array.

Changed in version 3.3: An empty tuple instead of `None` when ndim = 0.

strides

A tuple of integers the length of *ndim* giving the size in bytes to access each element for each dimension of the array.

Changed in version 3.3: An empty tuple instead of `None` when ndim = 0.

suboffsets

Used internally for PIL-style arrays. The value is informational only.

c_contiguous

A bool indicating whether the memory is C-*contiguous*.

Added in version 3.3.

f_contiguous

A bool indicating whether the memory is Fortran *contiguous*.

Added in version 3.3.

contiguous

A bool indicating whether the memory is *contiguous*.

Added in version 3.3.

4.10 Set Types — `set`, `frozenset`

A *set* object is an unordered collection of distinct *hashable* objects. Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference. (For other containers see the built-in *dict*, *list*, and *tuple* classes, and the *collections* module.)

Like other collections, sets support `x in set`, `len(set)`, and `for x in set`. Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

There are currently two built-in set types, *set* and *frozenset*. The *set* type is mutable — the contents can be changed using methods like `add()` and `remove()`. Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set. The *frozenset* type is immutable and *hashable* — its contents cannot be altered after it is created; it can therefore be used as a dictionary key or as an element of another set.

Non-empty sets (not frozensets) can be created by placing a comma-separated list of elements within braces, for example: `{'jack', 'sjoerd'}`, in addition to the *set* constructor.

The constructors for both classes work the same:

class set([*iterable*])

class frozenset([*iterable*])

Return a new set or frozenset object whose elements are taken from *iterable*. The elements of a set must be *hashable*. To represent sets of sets, the inner sets must be *frozenset* objects. If *iterable* is not specified, a new empty set is returned.

Sets can be created by several means:

- Use a comma-separated list of elements within braces: `{'jack', 'sjoerd'}`
- Use a set comprehension: `{c for c in 'abracadabra' if c not in 'abc'}`
- Use the type constructor: `set()`, `set('foobar')`, `set(['a', 'b', 'foo'])`

Instances of *set* and *frozenset* provide the following operations:

len(s)

Return the number of elements in set *s* (cardinality of *s*).

x in s

Test *x* for membership in *s*.

x not in s

Test *x* for non-membership in *s*.

isdisjoint(*other*)

Return `True` if the set has no elements in common with *other*. Sets are disjoint if and only if their intersection is the empty set.

issubset(*other*)

set <= other

Test whether every element in the set is in *other*.

set < other

Test whether the set is a proper subset of *other*, that is, `set <= other and set != other`.

issuperset(*other*)

set >= other

Test whether every element in *other* is in the set.

set > other

Test whether the set is a proper superset of *other*, that is, `set >= other and set != other`.

union(**others*)

set | other | ...

Return a new set with elements from the set and all others.

intersection(**others*)

set & other & ...

Return a new set with elements common to the set and all others.

difference(**others*)

set - other - ...

Return a new set with elements in the set that are not in the others.

symmetric_difference(*other*)

set ^ other

Return a new set with elements in either the set or *other* but not both.

copy()

Return a shallow copy of the set.

Note, the non-operator versions of *union()*, *intersection()*, *difference()*, *symmetric_difference()*, *issubset()*, and *issuperset()* methods will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets. This precludes

error-prone constructions like `set('abc') & 'cbs'` in favor of the more readable `set('abc').intersection('cbs')`.

Both *set* and *frozenset* support set to set comparisons. Two sets are equal if and only if every element of each set is contained in the other (each is a subset of the other). A set is less than another set if and only if the first set is a proper subset of the second set (is a subset, but is not equal). A set is greater than another set if and only if the first set is a proper superset of the second set (is a superset, but is not equal).

Instances of *set* are compared to instances of *frozenset* based on their members. For example, `set('abc') == frozenset('abc')` returns `True` and so does `set('abc') in set([frozenset('abc')])`.

The subset and equality comparisons do not generalize to a total ordering function. For example, any two nonempty disjoint sets are not equal and are not subsets of each other, so *all* of the following return `False`: `a<b`, `a==b`, or `a>b`.

Since sets only define partial ordering (subset relationships), the output of the *list.sort()* method is undefined for lists of sets.

Set elements, like dictionary keys, must be *hashable*.

Binary operations that mix *set* instances with *frozenset* return the type of the first operand. For example: `frozenset('ab') | set('bc')` returns an instance of *frozenset*.

The following table lists operations available for *set* that do not apply to immutable instances of *frozenset*:

update(**others*)

set |= other | ...

Update the set, adding elements from all others.

intersection_update(**others*)

set &= other & ...

Update the set, keeping only elements found in it and all others.

difference_update(**others*)

set -= other | ...

Update the set, removing elements found in others.

symmetric_difference_update(*other*)

set ^= other

Update the set, keeping only elements found in either set, but not in both.

add(*elem*)

Add element *elem* to the set.

remove(*elem*)

Remove element *elem* from the set. Raises *KeyError* if *elem* is not contained in the set.

discard(*elem*)

Remove element *elem* from the set if it is present.

pop()

Remove and return an arbitrary element from the set. Raises *KeyError* if the set is empty.

clear()

Remove all elements from the set.

Note, the non-operator versions of the *update()*, *intersection_update()*, *difference_update()*, and *symmetric_difference_update()* methods will accept any iterable as an argument.

Note, the *elem* argument to the `__contains__()`, `remove()`, and `discard()` methods may be a set. To support searching for an equivalent frozenset, a temporary one is created from *elem*.

4.11 Mapping Types — `dict`

A *mapping* object maps *hashable* values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the *dictionary*. (For other containers see the built-in `list`, `set`, and `tuple` classes, and the `collections` module.)

A dictionary's keys are *almost* arbitrary values. Values that are not *hashable*, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Values that compare equal (such as `1`, `1.0`, and `True`) can be used interchangeably to index the same dictionary entry.

class dict(***kwargs*)

class dict(*mapping*, ***kwargs*)

class dict(*iterable*, ***kwargs*)

Return a new dictionary initialized from an optional positional argument and a possibly empty set of keyword arguments.

Dictionaries can be created by several means:

- Use a comma-separated list of `key: value` pairs within braces: `{'jack': 4098, 'sjoerd': 4127}` or `{4098: 'jack', 4127: 'sjoerd'}`
- Use a dict comprehension: `{}`, `{x: x ** 2 for x in range(10)}`
- Use the type constructor: `dict()`, `dict([('foo', 100), ('bar', 200)])`, `dict(foo=100, bar=200)`

If no positional argument is given, an empty dictionary is created. If a positional argument is given and it is a mapping object, a dictionary is created with the same key-value pairs as the mapping object. Otherwise, the positional argument must be an *iterable* object. Each item in the iterable must itself be an iterable with exactly two objects. The first object of each item becomes a key in the new dictionary, and the second object the corresponding value. If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument. If a key being added is already present, the value from the keyword argument replaces the value from the positional argument.

To illustrate, the following examples all return a dictionary equal to `{"one": 1, "two": 2, "three": 3}`:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> f = dict({'one': 1, 'three': 3}, two=2)
>>> a == b == c == d == e == f
True
```

Providing keyword arguments as in the first example only works for keys that are valid Python identifiers. Otherwise, any valid keys can be used.

These are the operations that dictionaries support (and therefore, custom mapping types should support too):

list(d)

Return a list of all the keys used in the dictionary *d*.

len(d)

Return the number of items in the dictionary *d*.

d[key]

Return the item of *d* with key *key*. Raises a *KeyError* if *key* is not in the map.

If a subclass of dict defines a method __missing__() and *key* is not present, the d[key] operation calls that method with the key *key* as argument. The d[key] operation then returns or raises whatever is returned or raised by the __missing__(key) call. No other operations or methods invoke __missing__(). If __missing__() is not defined, *KeyError* is raised. __missing__() must be a method; it cannot be an instance variable:

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
...
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

The example above shows part of the implementation of *collections.Counter*. A different __missing__ method is used by *collections.defaultdict*.

d[key] = value

Set d[key] to *value*.

del d[key]

Remove d[key] from *d*. Raises a *KeyError* if *key* is not in the map.

key in d

Return True if *d* has a key *key*, else False.

key not in d

Equivalent to not key in d.

iter(d)

Return an iterator over the keys of the dictionary. This is a shortcut for iter(d.keys()).

clear()

Remove all items from the dictionary.

copy()

Return a shallow copy of the dictionary.

classmethod fromkeys(*iterable*, *value=None*, /)

Create a new dictionary with keys from *iterable* and values set to *value*.

fromkeys() is a class method that returns a new dictionary. *value* defaults to None. All of the values refer to just a single instance, so it generally doesn't make sense for *value* to be a mutable object such as an empty list. To get distinct values, use a dict comprehension instead.

get(*key*, *default=None*)

Return the value for *key* if *key* is in the dictionary, else *default*. If *default* is not given, it defaults to `None`, so that this method never raises a `KeyError`.

items()

Return a new view of the dictionary's items (`(key, value)` pairs). See the *documentation of view objects*.

keys()

Return a new view of the dictionary's keys. See the *documentation of view objects*.

pop(*key*[, *default*])

If *key* is in the dictionary, remove it and return its value, else return *default*. If *default* is not given and *key* is not in the dictionary, a `KeyError` is raised.

popitem()

Remove and return a `(key, value)` pair from the dictionary. Pairs are returned in LIFO (last-in, first-out) order.

`popitem()` is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling `popitem()` raises a `KeyError`.

Changed in version 3.7: LIFO order is now guaranteed. In prior versions, `popitem()` would return an arbitrary key/value pair.

reversed(d)

Return a reverse iterator over the keys of the dictionary. This is a shortcut for `reversed(d.keys())`.

Added in version 3.8.

setdefault(*key*, *default=None*)

If *key* is in the dictionary, return its value. If not, insert *key* with a value of *default* and return *default*. *default* defaults to `None`.

update([*other*])

Update the dictionary with the key/value pairs from *other*, overwriting existing keys. Return `None`.

`update()` accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: `d.update(red=1, blue=2)`.

values()

Return a new view of the dictionary's values. See the *documentation of view objects*.

An equality comparison between one `dict.values()` view and another will always return `False`. This also applies when comparing `dict.values()` to itself:

```
>>> d = {'a': 1}
>>> d.values() == d.values()
False
```

d | other

Create a new dictionary with the merged keys and values of *d* and *other*, which must both be dictionaries. The values of *other* take priority when *d* and *other* share keys.

Added in version 3.9.

d |= other

Update the dictionary *d* with keys and values from *other*, which may be either a *mapping* or an *iterable* of key/value pairs. The values of *other* take priority when *d* and *other* share keys.

Added in version 3.9.

Dictionaries compare equal if and only if they have the same `(key, value)` pairs (regardless of ordering). Order comparisons ('<', '<=', '>=', '>') raise *TypeError*.

Dictionaries preserve insertion order. Note that updating a key does not affect the order. Keys added after deletion are inserted at the end.

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
>>> list(d.values())
[1, 2, 3, 4]
>>> d["one"] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d["two"]
>>> d["two"] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

Changed in version 3.7: Dictionary order is guaranteed to be insertion order. This behavior was an implementation detail of CPython from 3.6.

Dictionaries and dictionary views are reversible.

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(reversed(d))
['four', 'three', 'two', 'one']
>>> list(reversed(d.values()))
[4, 3, 2, 1]
>>> list(reversed(d.items()))
[('four', 4), ('three', 3), ('two', 2), ('one', 1)]
```

Changed in version 3.8: Dictionaries are now reversible.

See also

types.MappingProxyType can be used to create a read-only view of a *dict*.

4.11.1 Dictionary view objects

The objects returned by *dict.keys()*, *dict.values()* and *dict.items()* are *view objects*. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

Dictionary views can be iterated over to yield their respective data, and support membership tests:

len(dictview)

Return the number of entries in the dictionary.

iter(dictview)

Return an iterator over the keys, values or items (represented as tuples of `(key, value)`) in the dictionary.

Keys and values are iterated over in insertion order. This allows the creation of `(value, key)` pairs using *zip()*: `pairs = zip(d.values(), d.keys())`. Another way to create the same list is `pairs = [(v, k) for (k, v) in d.items()]`.

Iterating views while adding or deleting entries in the dictionary may raise a *RuntimeError* or fail to iterate over all entries.

Changed in version 3.7: Dictionary order is guaranteed to be insertion order.

x in dictview

Return `True` if *x* is in the underlying dictionary's keys, values or items (in the latter case, *x* should be a `(key, value)` tuple).

reversed(dictview)

Return a reverse iterator over the keys, values or items of the dictionary. The view will be iterated in reverse order of the insertion.

Changed in version 3.8: Dictionary views are now reversible.

dictview.mapping

Return a *types.MappingProxyType* that wraps the original dictionary to which the view refers.

Added in version 3.10.

Keys views are set-like since their entries are unique and *hashable*. Items views also have set-like operations since the (key, value) pairs are unique and the keys are hashable. If all values in an items view are hashable as well, then the items view can interoperate with other sets. (Values views are not treated as set-like since the entries are generally not unique.) For set-like views, all of the operations defined for the abstract base class *collections.abc.Set* are available (for example, ==, <, or ^). While using set operators, set-like views accept any iterable as the other operand, unlike sets which only accept sets as the input.

An example of dictionary view usage:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
...
>>> print(n)
504

>>> # keys and values are iterated over in the same order (insertion order)
>>> list(keys)
['eggs', 'sausage', 'bacon', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['bacon', 'spam']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
```

(continues on next page)

(continued from previous page)

```
{'bacon'}
>>> keys ^ {'sausage', 'juice'} == {'juice', 'sausage', 'bacon', 'spam'}
True
>>> keys | ['juice', 'juice', 'juice'] == {'bacon', 'spam', 'juice'}
True

>>> # get back a read-only proxy for the original dictionary
>>> values.mapping
mappingproxy({'bacon': 1, 'spam': 500})
>>> values.mapping['spam']
500
```

4.12 Context Manager Types

Python's `with` statement supports the concept of a runtime context defined by a context manager. This is implemented using a pair of methods that allow user-defined classes to define a runtime context that is entered before the statement body is executed and exited when the statement ends:

contextmanager.**__enter__**()

Enter the runtime context and return either this object or another object related to the runtime context. The value returned by this method is bound to the identifier in the `as` clause of `with` statements using this context manager.

An example of a context manager that returns itself is a *file object*. File objects return themselves from __enter__() to allow *open()* to be used as the context expression in a `with` statement.

An example of a context manager that returns a related object is the one returned by *decimal.localcontext()*. These managers set the active decimal context to a copy of the original decimal context and then return the copy. This allows changes to be made to the current decimal context in the body of the `with` statement without affecting code outside the `with` statement.

contextmanager.**__exit__**(*exc_type*, *exc_val*, *exc_tb*)

Exit the runtime context and return a Boolean flag indicating if any exception that occurred should be suppressed. If an exception occurred while executing the body of the `with` statement, the arguments contain the exception type, value and traceback information. Otherwise, all three arguments are `None`.

Returning a true value from this method will cause the `with` statement to suppress the exception and continue execution with the statement immediately following the `with` statement. Otherwise the exception continues propagating after this method has finished executing. Exceptions that occur during execution of this method will replace any exception that occurred in the body of the `with` statement.

The exception passed in should never be reraised explicitly - instead, this method should return a false value to indicate that the method completed successfully and does not want to suppress the raised exception. This allows context management code to easily detect whether or not an `__exit__()` method has actually failed.

Python defines several context managers to support easy thread synchronisation, prompt closure of files or other objects, and simpler manipulation of the active decimal arithmetic context. The specific types are not treated specially beyond their implementation of the context management protocol. See the *contextlib* module for some examples.

Python's *generator*s and the *contextlib.contextmanager* decorator provide a convenient way to implement these protocols. If a generator function is decorated with the *contextlib.contextmanager* decorator, it will return a context manager implementing the necessary *__enter__()* and *__exit__()* methods, rather than the iterator produced by an undecorated generator function.

Note that there is no specific slot for any of these methods in the type structure for Python objects in the Python/C API. Extension types wanting to define these methods must provide them as a normal Python accessible method. Compared to the overhead of setting up the runtime context, the overhead of a single class dictionary lookup is negligible.

4.13 Type Annotation Types — Generic Alias, Union

The core built-in types for *type annotations* are *Generic Alias* and *Union*.

4.13.1 Generic Alias Type

`GenericAlias` objects are generally created by subscripting a class. They are most often used with container classes, such as *list* or *dict*. For example, `list[int]` is a `GenericAlias` object created by subscripting the `list` class with the argument *int*. `GenericAlias` objects are intended primarily for use with *type annotations*.

> **Note**
>
> It is generally only possible to subscript a class if the class implements the special method `__class_getitem__()`.

A `GenericAlias` object acts as a proxy for a *generic type*, implementing *parameterized generics*.

For a container class, the argument(s) supplied to a subscription of the class may indicate the type(s) of the elements an object contains. For example, `set[bytes]` can be used in type annotations to signify a *set* in which all the elements are of type *bytes*.

For a class which defines `__class_getitem__()` but is not a container, the argument(s) supplied to a subscription of the class will often indicate the return type(s) of one or more methods defined on an object. For example, *regular expressions* can be used on both the *str* data type and the *bytes* data type:

- If `x = re.search('foo', 'foo')`, `x` will be a *re.Match* object where the return values of `x.group(0)` and `x[0]` will both be of type *str*. We can represent this kind of object in type annotations with the `GenericAlias` `re.Match[str]`.
- If `y = re.search(b'bar', b'bar')`, (note the `b` for *bytes*), `y` will also be an instance of `re.Match`, but the return values of `y.group(0)` and `y[0]` will both be of type *bytes*. In type annotations, we would represent this variety of *re.Match* objects with `re.Match[bytes]`.

`GenericAlias` objects are instances of the class *types.GenericAlias*, which can also be used to create `GenericAlias` objects directly.

`T[X, Y, ...]`

Creates a `GenericAlias` representing a type `T` parameterized by types *X*, *Y*, and more depending on the `T` used. For example, a function expecting a *list* containing *float* elements:

```
def average(values: list[float]) -> float:
    return sum(values) / len(values)
```

Another example for *mapping* objects, using a *dict*, which is a generic type expecting two type parameters representing the key type and the value type. In this example, the function expects a `dict` with keys of type *str* and values of type *int*:

```
def send_post_request(url: str, body: dict[str, int]) -> None:
    ...
```

The builtin functions *isinstance()* and *issubclass()* do not accept `GenericAlias` types for their second argument:

```
>>> isinstance([1, 2], list[str])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: isinstance() argument 2 cannot be a parameterized generic
```

The Python runtime does not enforce *type annotations*. This extends to generic types and their type parameters. When creating a container object from a `GenericAlias`, the elements in the container are not checked against their type. For example, the following code is discouraged, but will run without errors:

```
>>> t = list[str]
>>> t([1, 2, 3])
[1, 2, 3]
```

Furthermore, parameterized generics erase type parameters during object creation:

```
>>> t = list[str]
>>> type(t)
<class 'types.GenericAlias'>

>>> l = t()
>>> type(l)
<class 'list'>
```

Calling *repr()* or *str()* on a generic shows the parameterized type:

```
>>> repr(list[int])
'list[int]'

>>> str(list[int])
'list[int]'
```

The `__getitem__()` method of generic containers will raise an exception to disallow mistakes like `dict[str][str]`:

```
>>> dict[str][str]
Traceback (most recent call last):
  ...
TypeError: dict[str] is not a generic class
```

However, such expressions are valid when *type variables* are used. The index must have as many elements as there are type variable items in the `GenericAlias` object's *__args__*.

```
>>> from typing import TypeVar
>>> Y = TypeVar('Y')
>>> dict[str, Y][int]
dict[str, int]
```

Standard Generic Classes

The following standard library classes support parameterized generics. This list is non-exhaustive.

- *tuple*
- *list*
- *dict*
- *set*
- *frozenset*
- *type*
- *collections.deque*
- *collections.defaultdict*
- *collections.OrderedDict*
- *collections.Counter*
- *collections.ChainMap*
- *collections.abc.Awaitable*
- *collections.abc.Coroutine*
- *collections.abc.AsyncIterable*
- *collections.abc.AsyncIterator*
- *collections.abc.AsyncGenerator*
- *collections.abc.Iterable*
- *collections.abc.Iterator*
- *collections.abc.Generator*
- *collections.abc.Reversible*
- *collections.abc.Container*
- *collections.abc.Collection*
- *collections.abc.Callable*
- *collections.abc.Set*
- *collections.abc.MutableSet*
- *collections.abc.Mapping*
- *collections.abc.MutableMapping*
- *collections.abc.Sequence*
- *collections.abc.MutableSequence*
- *collections.abc.ByteString*
- *collections.abc.MappingView*
- *collections.abc.KeysView*
- *collections.abc.ItemsView*
- *collections.abc.ValuesView*

- *contextlib.AbstractContextManager*
- *contextlib.AbstractAsyncContextManager*
- *dataclasses.Field*
- *functools.cached_property*
- *functools.partialmethod*
- *os.PathLike*
- *queue.LifoQueue*
- *queue.Queue*
- *queue.PriorityQueue*
- *queue.SimpleQueue*
- *re.Pattern*
- *re.Match*
- *shelve.BsdDbShelf*
- *shelve.DbfilenameShelf*
- *shelve.Shelf*
- *types.MappingProxyType*
- *weakref.WeakKeyDictionary*
- *weakref.WeakMethod*
- *weakref.WeakSet*
- *weakref.WeakValueDictionary*

Special Attributes of `GenericAlias` objects

All parameterized generics implement special read-only attributes.

`genericalias.__origin__`

This attribute points at the non-parameterized generic class:

```
>>> list[int].__origin__
<class 'list'>
```

`genericalias.__args__`

This attribute is a *tuple* (possibly of length 1) of generic types passed to the original `__class_getitem__()` of the generic class:

```
>>> dict[str, list[int]].__args__
(<class 'str'>, list[int])
```

`genericalias.__parameters__`

This attribute is a lazily computed tuple (possibly empty) of unique type variables found in `__args__`:

```
>>> from typing import TypeVar

>>> T = TypeVar('T')
>>> list[T].__parameters__
(~T,)
```

Note

A `GenericAlias` object with *typing.ParamSpec* parameters may not have correct `__parameters__` after substitution because *typing.ParamSpec* is intended primarily for static type checking.

`genericalias.__unpacked__`

A boolean that is true if the alias has been unpacked using the `*` operator (see *TypeVarTuple*).

Added in version 3.11.

See also

PEP 484 - **Type Hints**
Introducing Python's framework for type annotations.

PEP 585 - **Type Hinting Generics In Standard Collections**
Introducing the ability to natively parameterize standard-library classes, provided they implement the special class method `__class_getitem__()`.

Generics, *user-defined generics* and *typing.Generic*
Documentation on how to implement generic classes that can be parameterized at runtime and understood by static type-checkers.

Added in version 3.9.

4.13.2 Union Type

A union object holds the value of the `|` (bitwise or) operation on multiple *type objects*. These types are intended primarily for *type annotations*. The union type expression enables cleaner type hinting syntax compared to *typing.Union*.

`X | Y | ...`

Defines a union object which holds types *X*, *Y*, and so forth. `X | Y` means either X or Y. It is equivalent to `typing.Union[X, Y]`. For example, the following function expects an argument of type *int* or *float*:

```
def square(number: int | float) -> int | float:
    return number ** 2
```

Note

The `|` operand cannot be used at runtime to define unions where one or more members is a forward reference. For example, `int | "Foo"`, where `"Foo"` is a reference to a class not yet defined, will fail at runtime. For unions which include forward references, present the whole expression as a string, e.g. `"int | Foo"`.

`union_object == other`

Union objects can be tested for equality with other union objects. Details:

- Unions of unions are flattened:

```
(int | str) | float == int | str | float
```

- Redundant types are removed:

```
int | str | int == int | str
```

- When comparing unions, the order is ignored:

```
int | str == str | int
```

- It is compatible with *typing.Union*:

```
int | str == typing.Union[int, str]
```

- Optional types can be spelled as a union with `None`:

```
str | None == typing.Optional[str]
```

isinstance(obj, union_object)

issubclass(obj, union_object)

Calls to *isinstance()* and *issubclass()* are also supported with a union object:

```
>>> isinstance("", int | str)
True
```

However, *parameterized generics* in union objects cannot be checked:

```
>>> isinstance(1, int | list[int])  # short-circuit evaluation
True
>>> isinstance([1], int | list[int])
Traceback (most recent call last):
  ...
TypeError: isinstance() argument 2 cannot be a parameterized generic
```

The user-exposed type for the union object can be accessed from *types.UnionType* and used for *isinstance()* checks. An object cannot be instantiated from the type:

```
>>> import types
>>> isinstance(int | str, types.UnionType)
True
>>> types.UnionType()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot create 'types.UnionType' instances
```

Note

The `__or__()` method for type objects was added to support the syntax `X | Y`. If a metaclass implements `__or__()`, the Union may override it:

```
>>> class M(type):
...     def __or__(self, other):
...         return "Hello"
...
```

```
>>> class C(metaclass=M):
...     pass
...
>>> C | int
'Hello'
>>> int | C
int | C
```

See also

PEP 604 – PEP proposing the `X | Y` syntax and the Union type.

Added in version 3.10.

4.14 Other Built-in Types

The interpreter supports several other kinds of objects. Most of these support only one or two operations.

4.14.1 Modules

The only special operation on a module is attribute access: `m.name`, where *m* is a module and *name* accesses a name defined in *m*'s symbol table. Module attributes can be assigned to. (Note that the `import` statement is not, strictly speaking, an operation on a module object; `import foo` does not require a module object named *foo* to exist, rather it requires an (external) *definition* for a module named *foo* somewhere.)

A special attribute of every module is `__dict__`. This is the dictionary containing the module's symbol table. Modifying this dictionary will actually change the module's symbol table, but direct assignment to the `__dict__` attribute is not possible (you can write `m.__dict__['a'] = 1`, which defines `m.a` to be `1`, but you can't write `m.__dict__ = {}`). Modifying `__dict__` directly is not recommended.

Modules built into the interpreter are written like this: `<module 'sys' (built-in)>`. If loaded from a file, they are written as `<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>`.

4.14.2 Classes and Class Instances

See objects and class for these.

4.14.3 Functions

Function objects are created by function definitions. The only operation on a function object is to call it: `func(argument-list)`.

There are really two flavors of function objects: built-in functions and user-defined functions. Both support the same operation (to call the function), but the implementation is different, hence the different object types.

See function for more information.

4.14.4 Methods

Methods are functions that are called using the attribute notation. There are two flavors: built-in methods (such as `append()` on lists) and class instance method. Built-in methods are described with the types that support them.

If you access a method (a function defined in a class namespace) through an instance, you get a special object: a *bound method* (also called instance method) object. When called, it will add the `self` argument to the argument list. Bound methods have two special read-only attributes: `m.__self__` is the object on which the method operates, and `m.__func__` is the function implementing the method. Calling `m(arg-1, arg-2, ..., arg-n)` is completely equivalent to calling `m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)`.

Like function objects, bound method objects support getting arbitrary attributes. However, since method attributes are actually stored on the underlying function object (`method.__func__`), setting method attributes on bound methods is disallowed. Attempting to set an attribute on a method results in an *AttributeError* being raised. In order to set a method attribute, you need to explicitly set it on the underlying function object:

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method'  # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

See instance-methods for more information.

4.14.5 Code Objects

Code objects are used by the implementation to represent "pseudo-compiled" executable Python code such as a function body. They differ from function objects because they don't contain a reference to their global execution environment. Code objects are returned by the built-in *compile()* function and can be extracted from function objects through their `__code__` attribute. See also the *code* module.

Accessing `__code__` raises an *auditing event* `object.__getattr__` with arguments `obj` and `"__code__"`.

A code object can be executed or evaluated by passing it (instead of a source string) to the *exec()* or *eval()* built-in functions.

See types for more information.

4.14.6 Type Objects

Type objects represent the various object types. An object's type is accessed by the built-in function *type()*. There are no special operations on types. The standard module *types* defines names for all standard built-in types.

Types are written like this: `<class 'int'>`.

4.14.7 The Null Object

This object is returned by functions that don't explicitly return a value. It supports no special operations. There is exactly one null object, named `None` (a built-in name). `type(None)()` produces the same singleton.

It is written as `None`.

4.14.8 The Ellipsis Object

This object is commonly used by slicing (see slicings). It supports no special operations. There is exactly one ellipsis object, named *`Ellipsis`* (a built-in name). `type(Ellipsis)()` produces the *`Ellipsis`* singleton.

It is written as `Ellipsis` or `...`.

4.14.9 The NotImplemented Object

This object is returned from comparisons and binary operations when they are asked to operate on types they don't support. See comparisons for more information. There is exactly one *`NotImplemented`* object. `type(NotImplemented)()` produces the singleton instance.

It is written as `NotImplemented`.

4.14.10 Internal Objects

See types for this information. It describes stack frame objects, traceback objects, and slice objects.

4.15 Special Attributes

The implementation adds a few special read-only attributes to several object types, where they are relevant. Some of these are not reported by the *`dir()`* built-in function.

`object.`**`__dict__`**

A dictionary or other mapping object used to store an object's (writable) attributes.

`instance.`**`__class__`**

The class to which a class instance belongs.

`class.`**`__bases__`**

The tuple of base classes of a class object.

`definition.`**`__name__`**

The name of the class, function, method, descriptor, or generator instance.

`definition.`**`__qualname__`**

The *qualified name* of the class, function, method, descriptor, or generator instance.

Added in version 3.3.

`definition.`**`__type_params__`**

The type parameters of generic classes, functions, and *type aliases*.

Added in version 3.12.

class.**__mro__**

This attribute is a tuple of classes that are considered when looking for base classes during method resolution.

class.**mro**()

This method can be overridden by a metaclass to customize the method resolution order for its instances. It is called at class instantiation, and its result is stored in *__mro__*.

class.**__subclasses__**()

Each class keeps a list of weak references to its immediate subclasses. This method returns a list of all those references still alive. The list is in definition order. Example:

```
>>> int.__subclasses__()
[<class 'bool'>, <enum 'IntEnum'>, <flag 'IntFlag'>, <class 're._constants._
↪NamedIntConstant'>]
```

class.**__static_attributes__**

A tuple containing names of attributes of this class which are accessed through `self.X` from any function in its body.

Added in version 3.13.

4.16 Integer string conversion length limitation

CPython has a global limit for converting between *int* and *str* to mitigate denial of service attacks. This limit *only* applies to decimal or other non-power-of-two number bases. Hexadecimal, octal, and binary conversions are unlimited. The limit can be configured.

The *int* type in CPython is an arbitrary length number stored in binary form (commonly known as a "bignum"). There exists no algorithm that can convert a string to a binary integer or a binary integer to a string in linear time, *unless* the base is a power of 2. Even the best known algorithms for base 10 have sub-quadratic complexity. Converting a large value such as `int('1' * 500_000)` can take over a second on a fast CPU.

Limiting conversion size offers a practical way to avoid CVE-2020-10735.

The limit is applied to the number of digit characters in the input or output string when a non-linear conversion algorithm would be involved. Underscores and the sign are not counted towards the limit.

When an operation would exceed the limit, a *ValueError* is raised:

```
>>> import sys
>>> sys.set_int_max_str_digits(4300)  # Illustrative, this is the default.
>>> _ = int('2' * 5432)
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300 digits) for integer string conversion: value has␣
↪5432 digits; use sys.set_int_max_str_digits() to increase the limit
>>> i = int('2' * 4300)
>>> len(str(i))
4300
>>> i_squared = i*i
>>> len(str(i_squared))
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300 digits) for integer string conversion; use sys.
↪set_int_max_str_digits() to increase the limit
>>> len(hex(i_squared))
```

(continues on next page)

(continued from previous page)

```
7144
>>> assert int(hex(i_squared), base=16) == i*i  # Hexadecimal is unlimited.
```

The default limit is 4300 digits as provided in *sys.int_info.default_max_str_digits*. The lowest limit that can be configured is 640 digits as provided in *sys.int_info.str_digits_check_threshold*.

Verification:

```
>>> import sys
>>> assert sys.int_info.default_max_str_digits == 4300, sys.int_info
>>> assert sys.int_info.str_digits_check_threshold == 640, sys.int_info
>>> msg = int('578966293710682886880994035146873798396722250538762761564'
...           '9252925514383915483333812743580549779436104706260696366600'
...           '571186405732').to_bytes(53, 'big')
...
```

Added in version 3.11.

4.16.1 Affected APIs

The limitation only applies to potentially slow conversions between *int* and *str* or *bytes*:

- `int(string)` with default base 10.
- `int(string, base)` for all bases that are not a power of 2.
- `str(integer)`.
- `repr(integer)`.
- any other string conversion to base 10, for example `f"{integer}"`, `"{}".format(integer)`, or `b"%d" % integer`.

The limitations do not apply to functions with a linear algorithm:

- `int(string, base)` with base 2, 4, 8, 16, or 32.
- *int.from_bytes()* and *int.to_bytes()*.
- *hex()*, *oct()*, *bin()*.
- *Format Specification Mini-Language* for hex, octal, and binary numbers.
- *str* to *float*.
- *str* to *decimal.Decimal*.

4.16.2 Configuring the limit

Before Python starts up you can use an environment variable or an interpreter command line flag to configure the limit:

- `PYTHONINTMAXSTRDIGITS`, e.g. `PYTHONINTMAXSTRDIGITS=640 python3` to set the limit to 640 or `PYTHONINTMAXSTRDIGITS=0 python3` to disable the limitation.
- `-X int_max_str_digits`, e.g. `python3 -X int_max_str_digits=640`
- *sys.flags.int_max_str_digits* contains the value of `PYTHONINTMAXSTRDIGITS` or `-X int_max_str_digits`. If both the env var and the `-X` option are set, the `-X` option takes precedence. A value of *-1* indicates that both were unset, thus a value of *sys.int_info.default_max_str_digits* was used during initialization.

From code, you can inspect the current limit and set a new one using these *sys* APIs:

- *sys.get_int_max_str_digits()* and *sys.set_int_max_str_digits()* are a getter and setter for the interpreter-wide limit. Subinterpreters have their own limit.

Information about the default and minimum can be found in *sys.int_info*:

- *sys.int_info.default_max_str_digits* is the compiled-in default limit.
- *sys.int_info.str_digits_check_threshold* is the lowest accepted value for the limit (other than 0 which disables it).

Added in version 3.11.

Caution

Setting a low limit *can* lead to problems. While rare, code exists that contains integer constants in decimal in their source that exceed the minimum threshold. A consequence of setting the limit is that Python source code containing decimal integer literals longer than the limit will encounter an error during parsing, usually at startup time or import time or even at installation time - anytime an up to date `.pyc` does not already exist for the code. A workaround for source that contains such large constants is to convert them to `0x` hexadecimal form as it has no limit.

Test your application thoroughly if you use a low limit. Ensure your tests run with the limit set early via the environment or flag so that it applies during startup and even during any installation step that may invoke Python to precompile `.py` sources to `.pyc` files.

4.16.3 Recommended configuration

The default *sys.int_info.default_max_str_digits* is expected to be reasonable for most applications. If your application requires a different limit, set it from your main entry point using Python version agnostic code as these APIs were added in security patch releases in versions before 3.12.

Example:

```
>>> import sys
>>> if hasattr(sys, "set_int_max_str_digits"):
...     upper_bound = 68000
...     lower_bound = 4004
...     current_limit = sys.get_int_max_str_digits()
...     if current_limit == 0 or current_limit > upper_bound:
...         sys.set_int_max_str_digits(upper_bound)
...     elif current_limit < lower_bound:
...         sys.set_int_max_str_digits(lower_bound)
```

If you need to disable it entirely, set it to `0`.

CHAPTER

FIVE

BUILT-IN EXCEPTIONS

In Python, all exceptions must be instances of a class that derives from *BaseException*. In a `try` statement with an `except` clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which *it* is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed in this chapter can be generated by the interpreter or built-in functions. Except where mentioned, they have an "associated value" indicating the detailed cause of the error. This may be a string or a tuple of several items of information (e.g., an error code and a string explaining the code). The associated value is usually passed as arguments to the exception class's constructor.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition "just like" the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be subclassed to define new exceptions; programmers are encouraged to derive new exceptions from the *Exception* class or one of its subclasses, and not from *BaseException*. More information on defining exceptions is available in the Python Tutorial under tut-userexceptions.

5.1 Exception context

Three attributes on exception objects provide information about the context in which the exception was raised:

`BaseException.`**`__context__`**

`BaseException.`**`__cause__`**

`BaseException.`**`__suppress_context__`**

When raising a new exception while another exception is already being handled, the new exception's `__context__` attribute is automatically set to the handled exception. An exception may be handled when an `except` or `finally` clause, or a `with` statement, is used.

This implicit exception context can be supplemented with an explicit cause by using `from` with `raise`:

```
raise new_exc from original_exc
```

The expression following `from` must be an exception or `None`. It will be set as `__cause__` on the raised exception. Setting `__cause__` also implicitly sets the `__suppress_context__` attribute to `True`, so that using `raise new_exc from None` effectively replaces the old exception with the new one for display purposes (e.g. converting *KeyError* to *AttributeError*), while leaving the old exception available in `__context__` for introspection when debugging.

The default traceback display code shows these chained exceptions in addition to the traceback for the exception itself. An explicitly chained exception in `__cause__` is always shown when present. An implicitly chained exception in `__context__` is shown only if `__cause__` is *None* and `__suppress_context__` is false.

In either case, the exception itself is always shown after any chained exceptions so that the final line of the traceback always shows the last exception that was raised.

5.2 Inheriting from built-in exceptions

User code can create subclasses that inherit from an exception type. It's recommended to only subclass one exception type at a time to avoid any possible conflicts between how the bases handle the `args` attribute, as well as due to possible memory layout incompatibilities.

CPython implementation detail: Most built-in exceptions are implemented in C for efficiency, see: Objects/exceptions.c. Some have custom memory layouts which makes it impossible to create a subclass that inherits from multiple exception types. The memory layout of a type is an implementation detail and might change between Python versions, leading to new conflicts in the future. Therefore, it's recommended to avoid subclassing multiple exception types altogether.

5.3 Base classes

The following exceptions are used mostly as base classes for other exceptions.

exception BaseException

The base class for all built-in exceptions. It is not meant to be directly inherited by user-defined classes (for that, use *Exception*). If *str()* is called on an instance of this class, the representation of the argument(s) to the instance are returned, or the empty string when there were no arguments.

args

The tuple of arguments given to the exception constructor. Some built-in exceptions (like *OSError*) expect a certain number of arguments and assign a special meaning to the elements of this tuple, while others are usually called only with a single string giving an error message.

with_traceback(*tb*)

This method sets *tb* as the new traceback for the exception and returns the exception object. It was more commonly used before the exception chaining features of **PEP 3134** became available. The following example shows how we can convert an instance of `SomeException` into an instance of `OtherException` while preserving the traceback. Once raised, the current frame is pushed onto the traceback of the `OtherException`, as would have happened to the traceback of the original `SomeException` had we allowed it to propagate to the caller.

```
try:
    ...
except SomeException:
    tb = sys.exception().__traceback__
    raise OtherException(...).with_traceback(tb)
```

__traceback__

A writable field that holds the traceback object associated with this exception. See also: raise.

add_note(*note*)

Add the string `note` to the exception's notes which appear in the standard traceback after the exception string. A *TypeError* is raised if `note` is not a string.

Added in version 3.11.

__notes__

A list of the notes of this exception, which were added with *add_note()*. This attribute is created when *add_note()* is called.

Added in version 3.11.

exception Exception

All built-in, non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class.

exception ArithmeticError

The base class for those built-in exceptions that are raised for various arithmetic errors: *OverflowError*, *ZeroDivisionError*, *FloatingPointError*.

exception BufferError

Raised when a buffer related operation cannot be performed.

exception LookupError

The base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid: *IndexError*, *KeyError*. This can be raised directly by *codecs.lookup()*.

5.4 Concrete exceptions

The following exceptions are the exceptions that are usually raised.

exception AssertionError

Raised when an `assert` statement fails.

exception AttributeError

Raised when an attribute reference (see attribute-references) or assignment fails. (When an object does not support attribute references or attribute assignments at all, *TypeError* is raised.)

The `name` and `obj` attributes can be set using keyword-only arguments to the constructor. When set they represent the name of the attribute that was attempted to be accessed and the object that was accessed for said attribute, respectively.

Changed in version 3.10: Added the `name` and `obj` attributes.

exception EOFError

Raised when the *input()* function hits an end-of-file condition (EOF) without reading any data. (N.B.: the `io.IOBase.read()` and *io.IOBase.readline()* methods return an empty string when they hit EOF.)

exception FloatingPointError

Not currently used.

exception GeneratorExit

Raised when a *generator* or *coroutine* is closed; see `generator.close()` and `coroutine.close()`. It directly inherits from *BaseException* instead of *Exception* since it is technically not an error.

exception ImportError

Raised when the `import` statement has troubles trying to load a module. Also raised when the "from list" in `from ... import` has a name that cannot be found.

The optional *name* and *path* keyword-only arguments set the corresponding attributes:

name

The name of the module that was attempted to be imported.

path

The path to any file which triggered the exception.

Changed in version 3.3: Added the *name* and *path* attributes.

exception ModuleNotFoundError

A subclass of *ImportError* which is raised by `import` when a module could not be located. It is also raised when `None` is found in *sys.modules*.

Added in version 3.6.

exception IndexError

Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not an integer, *TypeError* is raised.)

exception KeyError

Raised when a mapping (dictionary) key is not found in the set of existing keys.

exception KeyboardInterrupt

Raised when the user hits the interrupt key (normally `Control-C` or `Delete`). During execution, a check for interrupts is made regularly. The exception inherits from *BaseException* so as to not be accidentally caught by code that catches *Exception* and thus prevent the interpreter from exiting.

> **Note**
>
> Catching a *KeyboardInterrupt* requires special consideration. Because it can be raised at unpredictable points, it may, in some circumstances, leave the running program in an inconsistent state. It is generally best to allow *KeyboardInterrupt* to end the program as quickly as possible or avoid raising it entirely. (See *Note on Signal Handlers and Exceptions*.)

exception MemoryError

Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating what kind of (internal) operation ran out of memory. Note that because of the underlying memory management architecture (C's `malloc()` function), the interpreter may not always be able to completely recover from this situation; it nevertheless raises an exception so that a stack traceback can be printed, in case a run-away program was the cause.

exception NameError

Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.

The `name` attribute can be set using a keyword-only argument to the constructor. When set it represent the name of the variable that was attempted to be accessed.

Changed in version 3.10: Added the `name` attribute.

exception NotImplementedError

This exception is derived from *RuntimeError*. In user defined base classes, abstract methods should raise this exception when they require derived classes to override the method, or while the class is being developed to indicate that the real implementation still needs to be added.

> **Note**
>
> It should not be used to indicate that an operator or method is not meant to be supported at all – in that case either leave the operator / method undefined or, if a subclass, set it to `None`.

> **Note**
>
> `NotImplementedError` and `NotImplemented` are not interchangeable, even though they have similar names and purposes. See `NotImplemented` for details on when to use it.

exception OSError([*arg*])

exception OSError(*errno*, *strerror*[, *filename*[, *winerror*[, *filename2*]]])

This exception is raised when a system function returns a system-related error, including I/O failures such as "file not found" or "disk full" (not for illegal argument types or other incidental errors).

The second form of the constructor sets the corresponding attributes, described below. The attributes default to `None` if not specified. For backwards compatibility, if three arguments are passed, the `args` attribute contains only a 2-tuple of the first two constructor arguments.

The constructor often actually returns a subclass of `OSError`, as described in *OS exceptions* below. The particular subclass depends on the final `errno` value. This behaviour only occurs when constructing `OSError` directly or via an alias, and is not inherited when subclassing.

errno

A numeric error code from the C variable `errno`.

winerror

Under Windows, this gives you the native Windows error code. The `errno` attribute is then an approximate translation, in POSIX terms, of that native error code.

Under Windows, if the *winerror* constructor argument is an integer, the `errno` attribute is determined from the Windows error code, and the *errno* argument is ignored. On other platforms, the *winerror* argument is ignored, and the `winerror` attribute does not exist.

strerror

The corresponding error message, as provided by the operating system. It is formatted by the C functions `perror()` under POSIX, and `FormatMessage()` under Windows.

filename

filename2

For exceptions that involve a file system path (such as `open()` or `os.unlink()`), `filename` is the file name passed to the function. For functions that involve two file system paths (such as `os.rename()`), `filename2` corresponds to the second file name passed to the function.

Changed in version 3.3: `EnvironmentError`, `IOError`, `WindowsError`, `socket.error`, `select.error` and `mmap.error` have been merged into `OSError`, and the constructor may return a subclass.

Changed in version 3.4: The `filename` attribute is now the original file name passed to the function, instead of the name encoded to or decoded from the *filesystem encoding and error handler*. Also, the *filename2* constructor argument and attribute was added.

exception OverflowError

Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for integers

(which would rather raise *MemoryError* than give up). However, for historical reasons, OverflowError is sometimes raised for integers that are outside a required range. Because of the lack of standardization of floating-point exception handling in C, most floating-point operations are not checked.

exception PythonFinalizationError

This exception is derived from *RuntimeError*. It is raised when an operation is blocked during interpreter shutdown also known as *Python finalization*.

Examples of operations which can be blocked with a *PythonFinalizationError* during the Python finalization:

- Creating a new Python thread.
- *os.fork()*.

See also the *sys.is_finalizing()* function.

Added in version 3.13: Previously, a plain *RuntimeError* was raised.

exception RecursionError

This exception is derived from *RuntimeError*. It is raised when the interpreter detects that the maximum recursion depth (see *sys.getrecursionlimit()*) is exceeded.

Added in version 3.5: Previously, a plain *RuntimeError* was raised.

exception ReferenceError

This exception is raised when a weak reference proxy, created by the *weakref.proxy()* function, is used to access an attribute of the referent after it has been garbage collected. For more information on weak references, see the *weakref* module.

exception RuntimeError

Raised when an error is detected that doesn't fall in any of the other categories. The associated value is a string indicating what precisely went wrong.

exception StopIteration

Raised by built-in function *next()* and an *iterator*'s *__next__()* method to signal that there are no further items produced by the iterator.

value

The exception object has a single attribute `value`, which is given as an argument when constructing the exception, and defaults to *None*.

When a *generator* or *coroutine* function returns, a new *StopIteration* instance is raised, and the value returned by the function is used as the *value* parameter to the constructor of the exception.

If a generator code directly or indirectly raises *StopIteration*, it is converted into a *RuntimeError* (retaining the *StopIteration* as the new exception's cause).

Changed in version 3.3: Added `value` attribute and the ability for generator functions to use it to return a value.

Changed in version 3.5: Introduced the RuntimeError transformation via `from __future__ import generator_stop`, see **PEP 479**.

Changed in version 3.7: Enable **PEP 479** for all code by default: a *StopIteration* error raised in a generator is transformed into a *RuntimeError*.

exception StopAsyncIteration

Must be raised by `__anext__()` method of an *asynchronous iterator* object to stop the iteration.

Added in version 3.5.

exception SyntaxError(*message*, *details*)

Raised when the parser encounters a syntax error. This may occur in an `import` statement, in a call to the built-in functions *compile()*, *exec()*, or *eval()*, or when reading the initial script or standard input (also interactively).

The *str()* of the exception instance returns only the error message. Details is a tuple whose members are also available as separate attributes.

filename

The name of the file the syntax error occurred in.

lineno

Which line number in the file the error occurred in. This is 1-indexed: the first line in the file has a `lineno` of 1.

offset

The column in the line where the error occurred. This is 1-indexed: the first character in the line has an `offset` of 1.

text

The source code text involved in the error.

end_lineno

Which line number in the file the error occurred ends in. This is 1-indexed: the first line in the file has a `lineno` of 1.

end_offset

The column in the end line where the error occurred finishes. This is 1-indexed: the first character in the line has an `offset` of 1.

For errors in f-string fields, the message is prefixed by "f-string: " and the offsets are offsets in a text constructed from the replacement expression. For example, compiling f'Bad {a b} field' results in this args attribute: ('f-string: …', (", 1, 2, '(a b)n', 1, 5)).

Changed in version 3.10: Added the *end_lineno* and *end_offset* attributes.

exception IndentationError

Base class for syntax errors related to incorrect indentation. This is a subclass of *SyntaxError*.

exception TabError

Raised when indentation contains an inconsistent use of tabs and spaces. This is a subclass of *IndentationError*.

exception SystemError

Raised when the interpreter finds an internal error, but the situation does not look so serious to cause it to abandon all hope. The associated value is a string indicating what went wrong (in low-level terms).

You should report this to the author or maintainer of your Python interpreter. Be sure to report the version of the Python interpreter (`sys.version`; it is also printed at the start of an interactive Python session), the exact error message (the exception's associated value) and if possible the source of the program that triggered the error.

exception SystemExit

This exception is raised by the *sys.exit()* function. It inherits from *BaseException* instead of *Exception* so that it is not accidentally caught by code that catches *Exception*. This allows the exception to properly propagate up and cause the interpreter to exit. When it is not handled, the Python interpreter exits; no stack traceback is printed. The constructor accepts the same optional argument passed to *sys.exit()*. If the value is an integer, it specifies the system exit status (passed to C's `exit()` function); if it is `None`, the exit status is zero; if it has another type (such as a string), the object's value is printed and the exit status is one.

A call to *sys.exit()* is translated into an exception so that clean-up handlers (`finally` clauses of `try` statements) can be executed, and so that a debugger can execute a script without running the risk of losing control. The *os._exit()* function can be used if it is absolutely positively necessary to exit immediately (for example, in the child process after a call to *os.fork()*).

code

The exit status or error message that is passed to the constructor. (Defaults to `None`.)

exception TypeError

Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

This exception may be raised by user code to indicate that an attempted operation on an object is not supported, and is not meant to be. If an object is meant to support a given operation but has not yet provided an implementation, *NotImplementedError* is the proper exception to raise.

Passing arguments of the wrong type (e.g. passing a *list* when an *int* is expected) should result in a *TypeError*, but passing arguments with the wrong value (e.g. a number outside expected boundaries) should result in a *ValueError*.

exception UnboundLocalError

Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. This is a subclass of *NameError*.

exception UnicodeError

Raised when a Unicode-related encoding or decoding error occurs. It is a subclass of *ValueError*.

UnicodeError has attributes that describe the encoding or decoding error. For example, `err.object[err.start:err.end]` gives the particular invalid input that the codec failed on.

encoding

The name of the encoding that raised the error.

reason

A string describing the specific codec error.

object

The object the codec was attempting to encode or decode.

start

The first index of invalid data in *object*.

end

The index after the last invalid data in *object*.

exception UnicodeEncodeError

Raised when a Unicode-related error occurs during encoding. It is a subclass of *UnicodeError*.

exception UnicodeDecodeError

Raised when a Unicode-related error occurs during decoding. It is a subclass of *UnicodeError*.

exception UnicodeTranslateError

Raised when a Unicode-related error occurs during translating. It is a subclass of *UnicodeError*.

exception ValueError

Raised when an operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as *IndexError*.

exception ZeroDivisionError

Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

The following exceptions are kept for compatibility with previous versions; starting from Python 3.3, they are aliases of *OSError*.

exception EnvironmentError

exception IOError

exception WindowsError

Only available on Windows.

5.4.1 OS exceptions

The following exceptions are subclasses of *OSError*, they get raised depending on the system error code.

exception BlockingIOError

Raised when an operation would block on an object (e.g. socket) set for non-blocking operation. Corresponds to errno *EAGAIN*, *EALREADY*, *EWOULDBLOCK* and *EINPROGRESS*.

In addition to those of *OSError*, *BlockingIOError* can have one more attribute:

characters_written

An integer containing the number of characters written to the stream before it blocked. This attribute is available when using the buffered I/O classes from the *io* module.

exception ChildProcessError

Raised when an operation on a child process failed. Corresponds to errno *ECHILD*.

exception ConnectionError

A base class for connection-related issues.

Subclasses are *BrokenPipeError*, *ConnectionAbortedError*, *ConnectionRefusedError* and *ConnectionResetError*.

exception BrokenPipeError

A subclass of *ConnectionError*, raised when trying to write on a pipe while the other end has been closed, or trying to write on a socket which has been shutdown for writing. Corresponds to errno *EPIPE* and *ESHUTDOWN*.

exception ConnectionAbortedError

A subclass of *ConnectionError*, raised when a connection attempt is aborted by the peer. Corresponds to errno *ECONNABORTED*.

exception ConnectionRefusedError

A subclass of *ConnectionError*, raised when a connection attempt is refused by the peer. Corresponds to errno *ECONNREFUSED*.

exception ConnectionResetError

A subclass of *ConnectionError*, raised when a connection is reset by the peer. Corresponds to errno *ECONNRESET*.

exception FileExistsError

Raised when trying to create a file or directory which already exists. Corresponds to errno *EEXIST*.

exception FileNotFoundError

Raised when a file or directory is requested but doesn't exist. Corresponds to `errno` *ENOENT*.

exception InterruptedError

Raised when a system call is interrupted by an incoming signal. Corresponds to `errno` *EINTR*.

Changed in version 3.5: Python now retries system calls when a syscall is interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

exception IsADirectoryError

Raised when a file operation (such as *os.remove()*) is requested on a directory. Corresponds to `errno` *EISDIR*.

exception NotADirectoryError

Raised when a directory operation (such as *os.listdir()*) is requested on something which is not a directory. On most POSIX platforms, it may also be raised if an operation attempts to open or traverse a non-directory file as if it were a directory. Corresponds to `errno` *ENOTDIR*.

exception PermissionError

Raised when trying to run an operation without the adequate access rights - for example filesystem permissions. Corresponds to `errno` *EACCES*, *EPERM*, and *ENOTCAPABLE*.

Changed in version 3.11.1: WASI's *ENOTCAPABLE* is now mapped to *PermissionError*.

exception ProcessLookupError

Raised when a given process doesn't exist. Corresponds to `errno` *ESRCH*.

exception TimeoutError

Raised when a system function timed out at the system level. Corresponds to `errno` *ETIMEDOUT*.

Added in version 3.3: All the above *OSError* subclasses were added.

See also

PEP 3151 - Reworking the OS and IO exception hierarchy

5.5 Warnings

The following exceptions are used as warning categories; see the *Warning Categories* documentation for more details.

exception Warning

Base class for warning categories.

exception UserWarning

Base class for warnings generated by user code.

exception DeprecationWarning

Base class for warnings about deprecated features when those warnings are intended for other Python developers.

Ignored by the default warning filters, except in the `__main__` module (**PEP 565**). Enabling the *Python Development Mode* shows this warning.

The deprecation policy is described in **PEP 387**.

exception PendingDeprecationWarning

Base class for warnings about features which are obsolete and expected to be deprecated in the future, but are not deprecated at the moment.

This class is rarely used as emitting a warning about a possible upcoming deprecation is unusual, and *DeprecationWarning* is preferred for already active deprecations.

Ignored by the default warning filters. Enabling the *Python Development Mode* shows this warning.

The deprecation policy is described in **PEP 387**.

exception SyntaxWarning

Base class for warnings about dubious syntax.

exception RuntimeWarning

Base class for warnings about dubious runtime behavior.

exception FutureWarning

Base class for warnings about deprecated features when those warnings are intended for end users of applications that are written in Python.

exception ImportWarning

Base class for warnings about probable mistakes in module imports.

Ignored by the default warning filters. Enabling the *Python Development Mode* shows this warning.

exception UnicodeWarning

Base class for warnings related to Unicode.

exception EncodingWarning

Base class for warnings related to encodings.

See *Opt-in EncodingWarning* for details.

Added in version 3.10.

exception BytesWarning

Base class for warnings related to *bytes* and *bytearray*.

exception ResourceWarning

Base class for warnings related to resource usage.

Ignored by the default warning filters. Enabling the *Python Development Mode* shows this warning.

Added in version 3.2.

5.6 Exception groups

The following are used when it is necessary to raise multiple unrelated exceptions. They are part of the exception hierarchy so they can be handled with `except` like all other exceptions. In addition, they are recognised by `except*`, which matches their subgroups based on the types of the contained exceptions.

exception ExceptionGroup(*msg*, *excs*)

exception BaseExceptionGroup(*msg*, *excs*)

Both of these exception types wrap the exceptions in the sequence `excs`. The `msg` parameter must be a string. The difference between the two classes is that *BaseExceptionGroup* extends *BaseException* and it can wrap any exception, while *ExceptionGroup* extends *Exception* and it can only wrap subclasses of *Exception*. This design is so that `except Exception` catches an *ExceptionGroup* but not *BaseExceptionGroup*.

The *BaseExceptionGroup* constructor returns an *ExceptionGroup* rather than a *BaseExceptionGroup* if all contained exceptions are *Exception* instances, so it can be used to make the selection automatic. The *ExceptionGroup* constructor, on the other hand, raises a *TypeError* if any contained exception is not an *Exception* subclass.

message

The `msg` argument to the constructor. This is a read-only attribute.

exceptions

A tuple of the exceptions in the `excs` sequence given to the constructor. This is a read-only attribute.

subgroup(*condition*)

Returns an exception group that contains only the exceptions from the current group that match *condition*, or `None` if the result is empty.

The condition can be an exception type or tuple of exception types, in which case each exception is checked for a match using the same check that is used in an `except` clause. The condition can also be a callable (other than a type object) that accepts an exception as its single argument and returns true for the exceptions that should be in the subgroup.

The nesting structure of the current exception is preserved in the result, as are the values of its *message*, *__traceback__*, *__cause__*, *__context__* and *__notes__* fields. Empty nested groups are omitted from the result.

The condition is checked for all exceptions in the nested exception group, including the top-level and any nested exception groups. If the condition is true for such an exception group, it is included in the result in full.

Added in version 3.13: `condition` can be any callable which is not a type object.

split(*condition*)

Like *subgroup()*, but returns the pair `(match, rest)` where `match` is `subgroup(condition)` and `rest` is the remaining non-matching part.

derive(*excs*)

Returns an exception group with the same *message*, but which wraps the exceptions in `excs`.

This method is used by *subgroup()* and *split()*, which are used in various contexts to break up an exception group. A subclass needs to override it in order to make *subgroup()* and *split()* return instances of the subclass rather than *ExceptionGroup*.

subgroup() and *split()* copy the *__traceback__*, *__cause__*, *__context__* and *__notes__* fields from the original exception group to the one returned by *derive()*, so these fields do not need to be updated by *derive()*.

```
>>> class MyGroup(ExceptionGroup):
...     def derive(self, excs):
...         return MyGroup(self.message, excs)
...
>>> e = MyGroup("eg", [ValueError(1), TypeError(2)])
>>> e.add_note("a note")
>>> e.__context__ = Exception("context")
>>> e.__cause__ = Exception("cause")
>>> try:
...     raise e
... except Exception as e:
...     exc = e
...
>>> match, rest = exc.split(ValueError)
```

(continues on next page)

(continued from previous page)

```
>>> exc, exc.__context__, exc.__cause__, exc.__notes__
(MyGroup('eg', [ValueError(1), TypeError(2)]), Exception('context'),␣
↪Exception('cause'), ['a note'])
>>> match, match.__context__, match.__cause__, match.__notes__
(MyGroup('eg', [ValueError(1)]), Exception('context'), Exception('cause'), [
↪'a note'])
>>> rest, rest.__context__, rest.__cause__, rest.__notes__
(MyGroup('eg', [TypeError(2)]), Exception('context'), Exception('cause'), ['a␣
↪note'])
>>> exc.__traceback__ is match.__traceback__ is rest.__traceback__
True
```

Note that *BaseExceptionGroup* defines `__new__()`, so subclasses that need a different constructor signature need to override that rather than `__init__()`. For example, the following defines an exception group subclass which accepts an exit_code and and constructs the group's message from it.

```
class Errors(ExceptionGroup):
   def __new__(cls, errors, exit_code):
      self = super().__new__(Errors, f"exit code: {exit_code}", errors)
      self.exit_code = exit_code
      return self

   def derive(self, excs):
      return Errors(excs, self.exit_code)
```

Like *ExceptionGroup*, any subclass of *BaseExceptionGroup* which is also a subclass of *Exception* can only wrap instances of *Exception*.

Added in version 3.11.

5.7 Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
      ├── EOFError
      ├── ExceptionGroup [BaseExceptionGroup]
      ├── ImportError
      │    └── ModuleNotFoundError
      ├── LookupError
      │    ├── IndexError
      │    └── KeyError
```

(continues on next page)

(continued from previous page)

```
      ├── MemoryError
      ├── NameError
      │    └── UnboundLocalError
      ├── OSError
      │    ├── BlockingIOError
      │    ├── ChildProcessError
      │    ├── ConnectionError
      │    │    ├── BrokenPipeError
      │    │    ├── ConnectionAbortedError
      │    │    ├── ConnectionRefusedError
      │    │    └── ConnectionResetError
      │    ├── FileExistsError
      │    ├── FileNotFoundError
      │    ├── InterruptedError
      │    ├── IsADirectoryError
      │    ├── NotADirectoryError
      │    ├── PermissionError
      │    ├── ProcessLookupError
      │    └── TimeoutError
      ├── ReferenceError
      ├── RuntimeError
      │    ├── NotImplementedError
      │    ├── PythonFinalizationError
      │    └── RecursionError
      ├── StopAsyncIteration
      ├── StopIteration
      ├── SyntaxError
      │    └── IndentationError
      │         └── TabError
      ├── SystemError
      ├── TypeError
      ├── ValueError
      │    └── UnicodeError
      │         ├── UnicodeDecodeError
      │         ├── UnicodeEncodeError
      │         └── UnicodeTranslateError
      └── Warning
           ├── BytesWarning
           ├── DeprecationWarning
           ├── EncodingWarning
           ├── FutureWarning
           ├── ImportWarning
           ├── PendingDeprecationWarning
           ├── ResourceWarning
           ├── RuntimeWarning
           ├── SyntaxWarning
           ├── UnicodeWarning
           └── UserWarning
```

CHAPTER

SIX

TEXT PROCESSING SERVICES

The modules described in this chapter provide a wide range of string manipulation operations and other text processing services.

The `codecs` module described under *Binary Data Services* is also highly relevant to text processing. In addition, see the documentation for Python's built-in string type in *Text Sequence Type — str*.

6.1 `string` — Common string operations

Source code: Lib/string.py

See also

Text Sequence Type — str

String Methods

6.1.1 String constants

The constants defined in this module are:

`string.`**`ascii_letters`**

The concatenation of the `ascii_lowercase` and `ascii_uppercase` constants described below. This value is not locale-dependent.

`string.`**`ascii_lowercase`**

The lowercase letters `'abcdefghijklmnopqrstuvwxyz'`. This value is not locale-dependent and will not change.

`string.`**`ascii_uppercase`**

The uppercase letters `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. This value is not locale-dependent and will not change.

`string.`**`digits`**

The string `'0123456789'`.

`string.`**`hexdigits`**

The string `'0123456789abcdefABCDEF'`.

string.**octdigits**

The string '01234567'.

string.**punctuation**

String of ASCII characters which are considered punctuation characters in the C locale: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~.

string.**printable**

String of ASCII characters which are considered printable. This is a combination of *digits*, *ascii_letters*, *punctuation*, and *whitespace*.

string.**whitespace**

A string containing all ASCII characters that are considered whitespace. This includes the characters space, tab, linefeed, return, formfeed, and vertical tab.

6.1.2 Custom String Formatting

The built-in string class provides the ability to do complex variable substitutions and value formatting via the *format()* method described in **PEP 3101**. The *Formatter* class in the *string* module allows you to create and customize your own string formatting behaviors using the same implementation as the built-in *format()* method.

class string.**Formatter**

The *Formatter* class has the following public methods:

format(*format_string*, /, **args*, ***kwargs*)

The primary API method. It takes a format string and an arbitrary set of positional and keyword arguments. It is just a wrapper that calls *vformat()*.

Changed in version 3.7: A format string argument is now *positional-only*.

vformat(*format_string*, *args*, *kwargs*)

This function does the actual work of formatting. It is exposed as a separate function for cases where you want to pass in a predefined dictionary of arguments, rather than unpacking and repacking the dictionary as individual arguments using the *args and **kwargs syntax. *vformat()* does the work of breaking up the format string into character data and replacement fields. It calls the various methods described below.

In addition, the *Formatter* defines a number of methods that are intended to be replaced by subclasses:

parse(*format_string*)

Loop over the format_string and return an iterable of tuples (*literal_text*, *field_name*, *format_spec*, *conversion*). This is used by *vformat()* to break the string into either literal text, or replacement fields.

The values in the tuple conceptually represent a span of literal text followed by a single replacement field. If there is no literal text (which can happen if two replacement fields occur consecutively), then *literal_text* will be a zero-length string. If there is no replacement field, then the values of *field_name*, *format_spec* and *conversion* will be None.

get_field(*field_name*, *args*, *kwargs*)

Given *field_name* as returned by *parse()* (see above), convert it to an object to be formatted. Returns a tuple (obj, used_key). The default version takes strings of the form defined in **PEP 3101**, such as "0[name]" or "label.title". *args* and *kwargs* are as passed in to *vformat()*. The return value *used_key* has the same meaning as the *key* parameter to *get_value()*.

get_value(*key*, *args*, *kwargs*)

Retrieve a given field value. The *key* argument will be either an integer or a string. If it is an integer, it represents the index of the positional argument in *args*; if it is a string, then it represents a named argument in *kwargs*.

The *args* parameter is set to the list of positional arguments to *vformat()*, and the *kwargs* parameter is set to the dictionary of keyword arguments.

For compound field names, these functions are only called for the first component of the field name; subsequent components are handled through normal attribute and indexing operations.

So for example, the field expression '0.name' would cause *get_value()* to be called with a *key* argument of 0. The `name` attribute will be looked up after *get_value()* returns by calling the built-in *getattr()* function.

If the index or keyword refers to an item that does not exist, then an *IndexError* or *KeyError* should be raised.

check_unused_args(*used_args*, *args*, *kwargs*)

Implement checking for unused arguments if desired. The arguments to this function is the set of all argument keys that were actually referred to in the format string (integers for positional arguments, and strings for named arguments), and a reference to the *args* and *kwargs* that was passed to vformat. The set of unused args can be calculated from these parameters. *check_unused_args()* is assumed to raise an exception if the check fails.

format_field(*value*, *format_spec*)

format_field() simply calls the global *format()* built-in. The method is provided so that subclasses can override it.

convert_field(*value*, *conversion*)

Converts the value (returned by *get_field()*) given a conversion type (as in the tuple returned by the *parse()* method). The default version understands 's' (str), 'r' (repr) and 'a' (ascii) conversion types.

6.1.3 Format String Syntax

The *str.format()* method and the *Formatter* class share the same syntax for format strings (although in the case of *Formatter*, subclasses can define their own format string syntax). The syntax is related to that of formatted string literals, but it is less sophisticated and, in particular, does not support arbitrary expressions.

Format strings contain "replacement fields" surrounded by curly braces `{}`. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: `{{` and `}}`.

The grammar for a replacement field is as follows:

```
replacement_field  ::=  "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name         ::=  arg_name ("." attribute_name | "[" element_index "]")*
arg_name           ::=  [identifier | digit+]
attribute_name     ::=  identifier
element_index      ::=  digit+ | index_string
index_string       ::=  <any source character except "]"> +
conversion         ::=  "r" | "s" | "a"
format_spec        ::=  format-spec:format_spec
```

In less formal terms, the replacement field can start with a *field_name* that specifies the object whose value is to be formatted and inserted into the output instead of the replacement field. The *field_name* is optionally followed by a *conversion* field, which is preceded by an exclamation point `'!'`, and a *format_spec*, which is preceded by a colon `':'`. These specify a non-default format for the replacement value.

See also the *Format Specification Mini-Language* section.

The *field_name* itself begins with an *arg_name* that is either a number or a keyword. If it's a number, it refers to a positional argument, and if it's a keyword, it refers to a named keyword argument. An *arg_name* is treated as a number if a call to *str.isdecimal()* on the string would return true. If the numerical arg_names in a format string are 0, 1, 2, … in sequence, they can all be omitted (not just some) and the numbers 0, 1, 2, … will be automatically inserted in that order. Because *arg_name* is not quote-delimited, it is not possible to specify arbitrary dictionary keys (e.g., the strings `'10'` or `':-]'`) within a format string. The *arg_name* can be followed by any number of index or attribute expressions. An expression of the form `'.name'` selects the named attribute using *getattr()*, while an expression of the form `'[index]'` does an index lookup using `__getitem__()`.

Changed in version 3.1: The positional argument specifiers can be omitted for *str.format()*, so `'{} {}'.format(a, b)` is equivalent to `'{0} {1}'.format(a, b)`.

Changed in version 3.4: The positional argument specifiers can be omitted for *Formatter*.

Some simple format string examples:

```
"First, thou shalt count to {0}"  # References first positional argument
"Bring me a {}"                   # Implicitly references the first positional␣
↪argument
"From {} to {}"                   # Same as "From {0} to {1}"
"My quest is {name}"              # References keyword argument 'name'
"Weight in tons {0.weight}"       # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}"   # First element of keyword argument 'players'.
```

The *conversion* field causes a type coercion before formatting. Normally, the job of formatting a value is done by the `__format__()` method of the value itself. However, in some cases it is desirable to force a type to be formatted as a string, overriding its own definition of formatting. By converting the value to a string before calling `__format__()`, the normal formatting logic is bypassed.

Three conversion flags are currently supported: `'!s'` which calls *str()* on the value, `'!r'` which calls *repr()* and `'!a'` which calls *ascii()*.

Some examples:

```
"Harold's a clever {0!s}"         # Calls str() on the argument first
"Bring out the holy {name!r}"     # Calls repr() on the argument first
"More {!a}"                       # Calls ascii() on the argument first
```

The *format_spec* field contains a specification of how the value should be presented, including such details as field width, alignment, padding, decimal precision and so on. Each value type can define its own "formatting mini-language" or interpretation of the *format_spec*.

Most built-in types support a common formatting mini-language, which is described in the next section.

A *format_spec* field can also include nested replacement fields within it. These nested replacement fields may contain a field name, conversion flag and format specification, but deeper nesting is not allowed. The replacement fields within the format_spec are substituted before the *format_spec* string is interpreted. This allows the formatting of a value to be dynamically specified.

See the *Format examples* section for some examples.

Format Specification Mini-Language

"Format specifications" are used within replacement fields contained within a format string to define how individual values are presented (see *Format String Syntax* and f-strings). They can also be passed directly to the built-in `format()` function. Each formattable type may define how the format specification is to be interpreted.

Most built-in types implement the following options for format specifications, although some of the formatting options are only supported by the numeric types.

A general convention is that an empty format specification produces the same result as if you had called `str()` on the value. A non-empty format specification typically modifies the result.

The general form of a *standard format specifier* is:

```
format_spec     ::=  [[fill]align][sign]["z"]["#"]["0"][width][grouping_option]["." preci
fill            ::=  <any character>
align           ::=  "<" | ">" | "=" | "^"
sign            ::=  "+" | "-" | " "
width           ::=  digit+
grouping_option ::=  "_" | ","
precision       ::=  digit+
type            ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s
```

If a valid *align* value is specified, it can be preceded by a *fill* character that can be any character and defaults to a space if omitted. It is not possible to use a literal curly brace ("`{`" or "`}`") as the *fill* character in a formatted string literal or when using the `str.format()` method. However, it is possible to insert a curly brace with a nested replacement field. This limitation doesn't affect the `format()` function.

The meaning of the various alignment options is as follows:

Option	Meaning
`'<'`	Forces the field to be left-aligned within the available space (this is the default for most objects).
`'>'`	Forces the field to be right-aligned within the available space (this is the default for numbers).
`'='`	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form '+000000120'. This alignment option is only valid for numeric types. It becomes the default for numbers when '0' immediately precedes the field width.
`'^'`	Forces the field to be centered within the available space.

Note that unless a minimum field width is defined, the field width will always be the same size as the data to fill it, so that the alignment option has no meaning in this case.

The *sign* option is only valid for number types, and can be one of the following:

Option	Meaning
`'+'`	indicates that a sign should be used for both positive as well as negative numbers.
`'-'`	indicates that a sign should be used only for negative numbers (this is the default behavior).
space	indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.

The `'z'` option coerces negative zero floating-point values to positive zero after rounding to the format precision. This option is only valid for floating-point presentation types.

Changed in version 3.11: Added the `'z'` option (see also **PEP 682**).

The '#' option causes the "alternate form" to be used for the conversion. The alternate form is defined differently for different types. This option is only valid for integer, float and complex types. For integers, when binary, octal, or hexadecimal output is used, this option adds the respective prefix '0b', '0o', '0x', or '0X' to the output value. For float and complex the alternate form causes the result of the conversion to always contain a decimal-point character, even if no digits follow it. Normally, a decimal-point character appears in the result of these conversions only if a digit follows it. In addition, for 'g' and 'G' conversions, trailing zeros are not removed from the result.

The ',' option signals the use of a comma for a thousands separator. For a locale aware separator, use the 'n' integer presentation type instead.

Changed in version 3.1: Added the ',' option (see also **PEP 378**).

The '_' option signals the use of an underscore for a thousands separator for floating-point presentation types and for integer presentation type 'd'. For integer presentation types 'b', 'o', 'x', and 'X', underscores will be inserted every 4 digits. For other presentation types, specifying this option is an error.

Changed in version 3.6: Added the '_' option (see also **PEP 515**).

width is a decimal integer defining the minimum total field width, including any prefixes, separators, and other formatting characters. If not specified, then the field width will be determined by the content.

When no explicit alignment is given, preceding the *width* field by a zero ('0') character enables sign-aware zero-padding for numeric types. This is equivalent to a *fill* character of '0' with an *alignment* type of '='.

Changed in version 3.10: Preceding the *width* field by '0' no longer affects the default alignment for strings.

The *precision* is a decimal integer indicating how many digits should be displayed after the decimal point for presentation types 'f' and 'F', or before and after the decimal point for presentation types 'g' or 'G'. For string presentation types the field indicates the maximum field size - in other words, how many characters will be used from the field content. The *precision* is not allowed for integer presentation types.

Finally, the *type* determines how the data should be presented.

The available string presentation types are:

Type	Meaning
's'	String format. This is the default type for strings and may be omitted.
None	The same as 's'.

The available integer presentation types are:

Type	Meaning
'b'	Binary format. Outputs the number in base 2.
'c'	Character. Converts the integer to the corresponding unicode character before printing.
'd'	Decimal Integer. Outputs the number in base 10.
'o'	Octal format. Outputs the number in base 8.
'x'	Hex format. Outputs the number in base 16, using lower-case letters for the digits above 9.
'X'	Hex format. Outputs the number in base 16, using upper-case letters for the digits above 9. In case '#' is specified, the prefix '0x' will be upper-cased to '0X' as well.
'n'	Number. This is the same as 'd', except that it uses the current locale setting to insert the appropriate number separator characters.
None	The same as 'd'.

In addition to the above presentation types, integers can be formatted with the floating-point presentation types listed below (except 'n' and None). When doing so, *float()* is used to convert the integer to a floating-point number

before formatting.

The available presentation types for *float* and *Decimal* values are:

Type	Meaning
'e'	Scientific notation. For a given precision p, formats the number in scientific notation with the letter 'e' separating the coefficient from the exponent. The coefficient has one digit before and p digits after the decimal point, for a total of p + 1 significant digits. With no precision given, uses a precision of 6 digits after the decimal point for *float*, and shows all coefficient digits for *Decimal*. If no digits follow the decimal point, the decimal point is also removed unless the # option is used.
'E'	Scientific notation. Same as 'e' except it uses an upper case 'E' as the separator character.
'f'	Fixed-point notation. For a given precision p, formats the number as a decimal number with exactly p digits following the decimal point. With no precision given, uses a precision of 6 digits after the decimal point for *float*, and uses a precision large enough to show all coefficient digits for *Decimal*. If no digits follow the decimal point, the decimal point is also removed unless the # option is used.
'F'	Fixed-point notation. Same as 'f', but converts nan to NAN and inf to INF.
'g'	General format. For a given precision p >= 1, this rounds the number to p significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude. A precision of 0 is treated as equivalent to a precision of 1. The precise rules are as follows: suppose that the result formatted with presentation type 'e' and precision p-1 would have exponent exp. Then, if m <= exp < p, where m is -4 for floats and -6 for *Decimals*, the number is formatted with presentation type 'f' and precision p-1-exp. Otherwise, the number is formatted with presentation type 'e' and precision p-1. In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it, unless the '#' option is used. With no precision given, uses a precision of 6 significant digits for *float*. For *Decimal*, the coefficient of the result is formed from the coefficient digits of the value; scientific notation is used for values smaller than 1e-6 in absolute value and values where the place value of the least significant digit is larger than 1, and fixed-point notation is used otherwise. Positive and negative infinity, positive and negative zero, and nans, are formatted as inf, -inf, 0, -0 and nan respectively, regardless of the precision.
'G'	General format. Same as 'g' except switches to 'E' if the number gets too large. The representations of infinity and NaN are uppercased, too.
'n'	Number. This is the same as 'g', except that it uses the current locale setting to insert the appropriate number separator characters.
'%'	Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign.
None	For *float* this is the same as 'g', except that when fixed-point notation is used to format the result, it always includes at least one digit past the decimal point. The precision used is as large as needed to represent the given value faithfully. For *Decimal*, this is the same as either 'g' or 'G' depending on the value of context.capitals for the current decimal context. The overall effect is to match the output of *str()* as altered by the other format modifiers.

Format examples

This section contains examples of the *str.format()* syntax and comparison with the old `%`-formatting.

In most of the cases the syntax is similar to the old `%`-formatting, with the addition of the `{}` and with `:` used instead of `%`. For example, `'%03.2f'` can be translated to `'{:03.2f}'`.

The new format syntax also supports new and different options, shown in the following examples.

Accessing arguments by position:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 3.1+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')      # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # arguments' indices can be repeated
'abracadabra'
```

Accessing arguments by name:

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.
↪81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

Accessing arguments' attributes:

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part -5.
↪0.'
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

Accessing arguments' items:

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
```

Replacing `%s` and `%r`:

```
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

Aligning the text and specifying a width:

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'***********centered***********'
```

Replacing `%+f`, `%-f`, and `% f` and specifying a sign:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # show a space for positive numbers
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # show only the minus -- same as '{:f}; {:f}'
'3.140000; -3.140000'
```

Replacing `%x` and `%o` and converting the value to different bases:

```
>>> # format also supports binary numbers
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42)
'int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010'
```

Using the comma as a thousands separator:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

Expressing a percentage:

```
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'
```

Using type-specific formatting:

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

Nesting arguments and more complex examples:

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
'left<<<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>>right'
>>>
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
```

(continues on next page)

(continued from previous page)

```
'C0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print('{0:{width}{base}}'.format(num, base=base, width=width), end=' ')
...     print()
...
    5     5     5   101
    6     6     6   110
    7     7     7   111
    8     8    10  1000
    9     9    11  1001
   10     A    12  1010
   11     B    13  1011
```

6.1.4 Template strings

Template strings provide simpler string substitutions as described in **PEP 292**. A primary use case for template strings is for internationalization (i18n) since in that context, the simpler syntax and functionality makes it easier to translate than other built-in string formatting facilities in Python. As an example of a library built on template strings for i18n, see the flufl.i18n package.

Template strings support `$`-based substitutions, using the following rules:

- `$$` is an escape; it is replaced with a single `$`.
- `$identifier` names a substitution placeholder matching a mapping key of `"identifier"`. By default, `"identifier"` is restricted to any case-insensitive ASCII alphanumeric string (including underscores) that starts with an underscore or ASCII letter. The first non-identifier character after the `$` character terminates this placeholder specification.
- `${identifier}` is equivalent to `$identifier`. It is required when valid identifier characters follow the placeholder but are not part of the placeholder, such as `"${noun}ification"`.

Any other appearance of `$` in the string will result in a *`ValueError`* being raised.

The *`string`* module provides a *`Template`* class that implements these rules. The methods of *`Template`* are:

class string.**Template**(*template*)

The constructor takes a single argument which is the template string.

substitute(*mapping={}*, */*, ***kwds*)

Performs the template substitution, returning a new string. *mapping* is any dictionary-like object with keys that match the placeholders in the template. Alternatively, you can provide keyword arguments, where the keywords are the placeholders. When both *mapping* and *kwds* are given and there are duplicates, the placeholders from *kwds* take precedence.

safe_substitute(*mapping={}*, */*, ***kwds*)

Like *`substitute()`*, except that if placeholders are missing from *mapping* and *kwds*, instead of raising a *`KeyError`* exception, the original placeholder will appear in the resulting string intact. Also, unlike with *`substitute()`*, any other appearances of the `$` will simply return `$` instead of raising *`ValueError`*.

While other exceptions may still occur, this method is called "safe" because it always tries to return a usable string instead of raising an exception. In another sense, *`safe_substitute()`* may be anything other than

safe, since it will silently ignore malformed templates containing dangling delimiters, unmatched braces, or placeholders that are not valid Python identifiers.

is_valid()

Returns false if the template has invalid placeholders that will cause *substitute()* to raise *ValueError*.

Added in version 3.11.

get_identifiers()

Returns a list of the valid identifiers in the template, in the order they first appear, ignoring any invalid identifiers.

Added in version 3.11.

Template instances also provide one public data attribute:

template

This is the object passed to the constructor's *template* argument. In general, you shouldn't change it, but read-only access is not enforced.

Here is an example of how to use a Template:

```
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
...
ValueError: Invalid placeholder in string: line 1, col 11
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
...
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

Advanced usage: you can derive subclasses of *Template* to customize the placeholder syntax, delimiter character, or the entire regular expression used to parse template strings. To do this, you can override these class attributes:

- *delimiter* – This is the literal string describing a placeholder introducing delimiter. The default value is `$`. Note that this should *not* be a regular expression, as the implementation will call *re.escape()* on this string as needed. Note further that you cannot change the delimiter after class creation (i.e. a different delimiter must be set in the subclass's class namespace).
- *idpattern* – This is the regular expression describing the pattern for non-braced placeholders. The default value is the regular expression `(?a:[_a-z][_a-z0-9]*)`. If this is given and *braceidpattern* is `None` this pattern will also apply to braced placeholders.

> **Note**
>
> Since default *flags* is `re.IGNORECASE`, pattern `[a-z]` can match with some non-ASCII characters. That's why we use the local `a` flag here.

Changed in version 3.7: *braceidpattern* can be used to define separate patterns used inside and outside the braces.

- *braceidpattern* – This is like *idpattern* but describes the pattern for braced placeholders. Defaults to `None` which means to fall back to *idpattern* (i.e. the same pattern is used both inside and outside braces). If given, this allows you to define different patterns for braced and unbraced placeholders.

  Added in version 3.7.

- *flags* – The regular expression flags that will be applied when compiling the regular expression used for recognizing substitutions. The default value is `re.IGNORECASE`. Note that `re.VERBOSE` will always be added to the flags, so custom *idpattern*s must follow conventions for verbose regular expressions.

  Added in version 3.2.

Alternatively, you can provide the entire regular expression pattern by overriding the class attribute *pattern*. If you do this, the value must be a regular expression object with four named capturing groups. The capturing groups correspond to the rules given above, along with the invalid placeholder rule:

- *escaped* – This group matches the escape sequence, e.g. `$$`, in the default pattern.
- *named* – This group matches the unbraced placeholder name; it should not include the delimiter in capturing group.
- *braced* – This group matches the brace enclosed placeholder name; it should not include either the delimiter or braces in the capturing group.
- *invalid* – This group matches any other delimiter pattern (usually a single delimiter), and it should appear last in the regular expression.

The methods on this class will raise *ValueError* if the pattern matches the template without one of these named groups matching.

6.1.5 Helper functions

`string.`**`capwords`**(*s*, *sep=None*)

Split the argument into words using *str.split()*, capitalize each word using *str.capitalize()*, and join the capitalized words using *str.join()*. If the optional second argument *sep* is absent or `None`, runs of whitespace characters are replaced by a single space and leading and trailing whitespace are removed, otherwise *sep* is used to split and join the words.

6.2 `re` — Regular expression operations

Source code: Lib/re/

This module provides regular expression matching operations similar to those found in Perl.

Both patterns and strings to be searched can be Unicode strings (*str*) as well as 8-bit strings (*bytes*). However, Unicode strings and 8-bit strings cannot be mixed: that is, you cannot match a Unicode string with a bytes pattern or vice-versa; similarly, when asking for a substitution, the replacement string must be of the same type as both the pattern and the search string.

Regular expressions use the backslash character (`'\'`) to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python's usage of the same character for the same purpose in string literals; for example, to match a literal backslash, one might have to write `'\\\\'` as the pattern string, because the regular expression must be `\\`, and each backslash must be expressed as `\\` inside a regular Python string literal. Also, please note that any invalid escape sequences in Python's usage of the backslash in string literals now generate a *SyntaxWarning* and in the future this will become a *SyntaxError*. This behaviour will happen even if it is a valid escape sequence for a regular expression.

The solution is to use Python's raw string notation for regular expression patterns; backslashes are not handled in any special way in a string literal prefixed with `'r'`. So `r"\n"` is a two-character string containing `'\'` and `'n'`, while `"\n"` is a one-character string containing a newline. Usually patterns will be expressed in Python code using this raw string notation.

It is important to note that most regular expression operations are available as module-level functions and methods on *compiled regular expressions*. The functions are shortcuts that don't require you to compile a regex object first, but miss some fine-tuning parameters.

See also

The third-party regex module, which has an API compatible with the standard library `re` module, but offers additional functionality and a more thorough Unicode support.

6.2.1 Regular Expression Syntax

A regular expression (or RE) specifies a set of strings that matches it; the functions in this module let you check if a particular string matches a given regular expression (or if a given regular expression matches a particular string, which comes down to the same thing).

Regular expressions can be concatenated to form new regular expressions; if *A* and *B* are both regular expressions, then *AB* is also a regular expression. In general, if a string *p* matches *A* and another string *q* matches *B*, the string *pq* will match AB. This holds unless *A* or *B* contain low precedence operations; boundary conditions between *A* and *B*; or have numbered group references. Thus, complex expressions can easily be constructed from simpler primitive expressions like the ones described here. For details of the theory and implementation of regular expressions, consult the Friedl book [Frie09], or almost any textbook about compiler construction.

A brief explanation of the format of regular expressions follows. For further information and a gentler presentation, consult the regex-howto.

Regular expressions can contain both special and ordinary characters. Most ordinary characters, like `'A'`, `'a'`, or `'0'`, are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so `last` matches the string `'last'`. (In the rest of this section, we'll write RE's in `this special style`, usually without quotes, and strings to be matched `'in single quotes'`.)

Some characters, like `'|'` or `'('`, are special. Special characters either stand for classes of ordinary characters, or affect how the regular expressions around them are interpreted.

Repetition operators or quantifiers (`*`, `+`, `?`, `{m,n}`, etc) cannot be directly nested. This avoids ambiguity with the non-greedy modifier suffix `?`, and with other modifiers in other implementations. To apply a second repetition to an inner repetition, parentheses may be used. For example, the expression `(?:a{6})*` matches any multiple of six `'a'` characters.

The special characters are:

.
(Dot.) In the default mode, this matches any character except a newline. If the *DOTALL* flag has been specified, this matches any character including a newline. `(?s:.)` matches any character regardless of flags.

^
(Caret.) Matches the start of the string, and in *MULTILINE* mode also matches immediately after each newline.

$
Matches the end of the string or just before the newline at the end of the string, and in *MULTILINE* mode also matches before a newline. `foo` matches both 'foo' and 'foobar', while the regular expression `foo$` matches only 'foo'. More interestingly, searching for `foo.$` in `'foo1\nfoo2\n'` matches 'foo2' normally, but 'foo1' in

MULTILINE mode; searching for a single `$` in `'foo\n'` will find two (empty) matches: one just before the newline, and one at the end of the string.

*****

Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. `ab*` will match 'a', 'ab', or 'a' followed by any number of 'b's.

+

Causes the resulting RE to match 1 or more repetitions of the preceding RE. `ab+` will match 'a' followed by any non-zero number of 'b's; it will not match just 'a'.

?

Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. `ab?` will match either 'a' or 'ab'.

***?, +?, ??**

The `'*'`, `'+'`, and `'?'` quantifiers are all *greedy*; they match as much text as possible. Sometimes this behaviour isn't desired; if the RE `<.*>` is matched against `'<a> b <c>'`, it will match the entire string, and not just `'<a>'`. Adding `?` after the quantifier makes it perform the match in *non-greedy* or *minimal* fashion; as *few* characters as possible will be matched. Using the RE `<.*?>` will match only `'<a>'`.

***+, ++, ?+**

Like the `'*'`, `'+'`, and `'?'` quantifiers, those where `'+'` is appended also match as many times as possible. However, unlike the true greedy quantifiers, these do not allow back-tracking when the expression following it fails to match. These are known as *possessive* quantifiers. For example, `a*a` will match `'aaaa'` because the `a*` will match all 4 `'a'`s, but, when the final `'a'` is encountered, the expression is backtracked so that in the end the `a*` ends up matching 3 `'a'`s total, and the fourth `'a'` is matched by the final `'a'`. However, when `a*+a` is used to match `'aaaa'`, the `a*+` will match all 4 `'a'`, but when the final `'a'` fails to find any more characters to match, the expression cannot be backtracked and will thus fail to match. `x*+`, `x++` and `x?+` are equivalent to `(?>x*)`, `(?>x+)` and `(?>x?)` correspondingly.

Added in version 3.11.

{m}

Specifies that exactly *m* copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, `a{6}` will match exactly six `'a'` characters, but not five.

{m,n}

Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, `a{3,5}` will match from 3 to 5 `'a'` characters. Omitting *m* specifies a lower bound of zero, and omitting *n* specifies an infinite upper bound. As an example, `a{4,}b` will match `'aaaab'` or a thousand `'a'` characters followed by a `'b'`, but not `'aaab'`. The comma may not be omitted or the modifier would be confused with the previously described form.

{m,n}?

Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as *few* repetitions as possible. This is the non-greedy version of the previous quantifier. For example, on the 6-character string `'aaaaaa'`, `a{3,5}` will match 5 `'a'` characters, while `a{3,5}?` will only match 3 characters.

{m,n}+

Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as many repetitions as possible *without* establishing any backtracking points. This is the possessive version of the quantifier above. For example, on the 6-character string `'aaaaaa'`, `a{3,5}+aa` attempt to match 5 `'a'` characters, then, requiring 2 more `'a'`s, will need more characters than available and thus fail, while `a{3,5}aa` will match with `a{3,5}` capturing 5, then 4 `'a'`s by backtracking and then the final 2 `'a'`s are matched by the final `aa` in the pattern. `x{m,n}+` is equivalent to `(?>x{m,n})`.

Added in version 3.11.

****

Either escapes special characters (permitting you to match characters like `'*'`, `'?'`, and so forth), or signals a

special sequence; special sequences are discussed below.

If you're not using a raw string to express the pattern, remember that Python also uses the backslash as an escape sequence in string literals; if the escape sequence isn't recognized by Python's parser, the backslash and subsequent character are included in the resulting string. However, if Python would recognize the resulting sequence, the backslash should be repeated twice. This is complicated and hard to understand, so it's highly recommended that you use raw strings for all but the simplest expressions.

[]

Used to indicate a set of characters. In a set:

- Characters can be listed individually, e.g. `[amk]` will match `'a'`, `'m'`, or `'k'`.
- Ranges of characters can be indicated by giving two characters and separating them by a `'-'`, for example `[a-z]` will match any lowercase ASCII letter, `[0-5][0-9]` will match all the two-digits numbers from `00` to `59`, and `[0-9A-Fa-f]` will match any hexadecimal digit. If `-` is escaped (e.g. `[a\-z]`) or if it's placed as the first or last character (e.g. `[-a]` or `[a-]`), it will match a literal `'-'`.
- Special characters lose their special meaning inside sets. For example, `[(+*)]` will match any of the literal characters `'('`, `'+'`, `'*'`, or `')'`.
- Character classes such as `\w` or `\S` (defined below) are also accepted inside a set, although the characters they match depend on the *flags* used.
- Characters that are not within a range can be matched by *complementing* the set. If the first character of the set is `'^'`, all the characters that are *not* in the set will be matched. For example, `[^5]` will match any character except `'5'`, and `[^^]` will match any character except `'^'`. `^` has no special meaning if it's not the first character in the set.
- To match a literal `']'` inside a set, precede it with a backslash, or place it at the beginning of the set. For example, both `[()[\]{}]` and `[]()[{}]` will match a right bracket, as well as left bracket, braces, and parentheses.
- Support of nested sets and set operations as in Unicode Technical Standard #18 might be added in the future. This would change the syntax, so to facilitate this change a *FutureWarning* will be raised in ambiguous cases for the time being. That includes sets starting with a literal `'['` or containing literal character sequences `'--'`, `'&&'`, `'~~'`, and `'||'`. To avoid a warning escape them with a backslash.

Changed in version 3.7: *FutureWarning* is raised if a character set contains constructs that will change semantically in the future.

|

`A|B`, where *A* and *B* can be arbitrary REs, creates a regular expression that will match either *A* or *B*. An arbitrary number of REs can be separated by the `'|'` in this way. This can be used inside groups (see below) as well. As the target string is scanned, REs separated by `'|'` are tried from left to right. When one pattern completely matches, that branch is accepted. This means that once *A* matches, *B* will not be tested further, even if it would produce a longer overall match. In other words, the `'|'` operator is never greedy. To match a literal `'|'`, use `\|`, or enclose it inside a character class, as in `[|]`.

(...)

Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed, and can be matched later in the string with the `\number` special sequence, described below. To match the literals `'('` or `')'`, use `\(` or `\)`, or enclose them inside a character class: `[(]`, `[)]`.

(?...)

This is an extension notation (a `'?'` following a `'('` is not meaningful otherwise). The first character after the `'?'` determines what the meaning and further syntax of the construct is. Extensions usually do not create a new group; `(?P<name>...)` is the only exception to this rule. Following are the currently supported extensions.

(?aiLmsux)
(One or more letters from the set `'a'`, `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'`.) The group matches the empty string; the letters set the corresponding flags for the entire regular expression:

- *re.A* (ASCII-only matching)
- *re.I* (ignore case)
- *re.L* (locale dependent)
- *re.M* (multi-line)
- *re.S* (dot matches all)
- *re.U* (Unicode matching)
- *re.X* (verbose)

(The flags are described in *Module Contents*.) This is useful if you wish to include the flags as part of the regular expression, instead of passing a *flag* argument to the *re.compile()* function. Flags should be used first in the expression string.

Changed in version 3.11: This construction can only be used at the start of the expression.

(?:...)
A non-capturing version of regular parentheses. Matches whatever regular expression is inside the parentheses, but the substring matched by the group *cannot* be retrieved after performing a match or referenced later in the pattern.

(?aiLmsux-imsx:...)
(Zero or more letters from the set `'a'`, `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'`, optionally followed by `'-'` followed by one or more letters from the `'i'`, `'m'`, `'s'`, `'x'`.) The letters set or remove the corresponding flags for the part of the expression:

- *re.A* (ASCII-only matching)
- *re.I* (ignore case)
- *re.L* (locale dependent)
- *re.M* (multi-line)
- *re.S* (dot matches all)
- *re.U* (Unicode matching)
- *re.X* (verbose)

(The flags are described in *Module Contents*.)

The letters `'a'`, `'L'` and `'u'` are mutually exclusive when used as inline flags, so they can't be combined or follow `'-'`. Instead, when one of them appears in an inline group, it overrides the matching mode in the enclosing group. In Unicode patterns `(?a:...)` switches to ASCII-only matching, and `(?u:...)` switches to Unicode matching (default). In bytes patterns `(?L:...)` switches to locale dependent matching, and `(?a:...)` switches to ASCII-only matching (default). This override is only in effect for the narrow inline group, and the original matching mode is restored outside of the group.

Added in version 3.6.

Changed in version 3.7: The letters `'a'`, `'L'` and `'u'` also can be used in a group.

(?>...)
Attempts to match `...` as if it was a separate regular expression, and if successful, continues to match the rest of the pattern following it. If the subsequent pattern fails to match, the stack can only be unwound to a point *before* the `(?>...)` because once exited, the expression, known as an *atomic group*, has thrown away all stack points within itself. Thus, `(?>.*).` would never match anything because first the `.*` would match all characters possible, then,

having nothing left to match, the final `.` would fail to match. Since there are no stack points saved in the Atomic Group, and there is no stack point before it, the entire expression would thus fail to match.

Added in version 3.11.

(?P<name>...)
Similar to regular parentheses, but the substring matched by the group is accessible via the symbolic group name *name*. Group names must be valid Python identifiers, and in *bytes* patterns they can only contain bytes in the ASCII range. Each group name must be defined only once within a regular expression. A symbolic group is also a numbered group, just as if the group were not named.

Named groups can be referenced in three contexts. If the pattern is `(?P<quote>['"]).*?(?P=quote)` (i.e. matching a string quoted with either single or double quotes):

Context of reference to group "quote"	Ways to reference it
in the same pattern itself	• `(?P=quote)` (as shown) • `\1`
when processing match object *m*	• `m.group('quote')` • `m.end('quote')` (etc.)
in a string passed to the *repl* argument of `re.sub()`	• `\g<quote>` • `\g<1>` • `\1`

Changed in version 3.12: In *bytes* patterns, group *name* can only contain bytes in the ASCII range (`b'\x00'`-`b'\x7f'`).

(?P=name)
A backreference to a named group; it matches whatever text was matched by the earlier group named *name*.

(?#...)
A comment; the contents of the parentheses are simply ignored.

(?=...)
Matches if `...` matches next, but doesn't consume any of the string. This is called a *lookahead assertion*. For example, `Isaac (?=Asimov)` will match `'Isaac '` only if it's followed by `'Asimov'`.

(?!...)
Matches if `...` doesn't match next. This is a *negative lookahead assertion*. For example, `Isaac (?!Asimov)` will match `'Isaac '` only if it's *not* followed by `'Asimov'`.

(?<=...)
Matches if the current position in the string is preceded by a match for `...` that ends at the current position. This is called a *positive lookbehind assertion*. `(?<=abc)def` will find a match in `'abcdef'`, since the lookbehind will back up 3 characters and check if the contained pattern matches. The contained pattern must only match strings of some fixed length, meaning that `abc` or `a|b` are allowed, but `a*` and `a{3,4}` are not. Note that patterns which start with positive lookbehind assertions will not match at the beginning of the string being searched; you will most likely want to use the *search()* function rather than the *match()* function:

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

This example looks for a word following a hyphen:

```
>>> m = re.search(r'(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

Changed in version 3.5: Added support for group references of fixed length.

(?<!...)
Matches if the current position in the string is not preceded by a match for `...`. This is called a *negative lookbehind assertion*. Similar to positive lookbehind assertions, the contained pattern must only match strings of some fixed length. Patterns which start with negative lookbehind assertions may match at the beginning of the string being searched.

(?(id/name)yes-pattern|no-pattern)
Will try to match with `yes-pattern` if the group with given *id* or *name* exists, and with `no-pattern` if it doesn't. `no-pattern` is optional and can be omitted. For example, `(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)` is a poor email matching pattern, which will match with `'<user@host.com>'` as well as `'user@host.com'`, but not with `'<user@host.com'` nor `'user@host.com>'`.

Changed in version 3.12: Group *id* can only contain ASCII digits. In *bytes* patterns, group *name* can only contain bytes in the ASCII range (`b'\x00'-b'\x7f'`).

The special sequences consist of `'\'` and a character from the list below. If the ordinary character is not an ASCII digit or an ASCII letter, then the resulting RE will match the second character. For example, `\$` matches the character `'$'`.

\number
Matches the contents of the group of the same number. Groups are numbered starting from 1. For example, `(.+) \1` matches `'the the'` or `'55 55'`, but not `'thethe'` (note the space after the group). This special sequence can only be used to match one of the first 99 groups. If the first digit of *number* is 0, or *number* is 3 octal digits long, it will not be interpreted as a group match, but as the character with octal value *number*. Inside the `'['` and `']'` of a character class, all numeric escapes are treated as characters.

\A
Matches only at the start of the string.

\b
Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of word characters. Note that formally, `\b` is defined as the boundary between a `\w` and a `\W` character (or vice versa), or between `\w` and the beginning or end of the string. This means that `r'\bat\b'` matches `'at'`, `'at.'`, `'(at)'`, and `'as at ay'` but not `'attempt'` or `'atlas'`.

The default word characters in Unicode (str) patterns are Unicode alphanumerics and the underscore, but this can be changed by using the *ASCII* flag. Word boundaries are determined by the current locale if the *LOCALE* flag is used.

> **Note**
> Inside a character range, `\b` represents the backspace character, for compatibility with Python's string literals.

\B
Matches the empty string, but only when it is *not* at the beginning or end of a word. This means that `r'at\B'` matches `'athens'`, `'atom'`, `'attorney'`, but not `'at'`, `'at.'`, or `'at!'`. `\B` is the opposite of `\b`, so word characters in Unicode (str) patterns are Unicode alphanumerics or the underscore, although this can be changed by using the *ASCII* flag. Word boundaries are determined by the current locale if the *LOCALE* flag is used.

\d

For Unicode (str) patterns:
Matches any Unicode decimal digit (that is, any character in Unicode character category [Nd]). This includes `[0-9]`, and also many other digit characters.

Matches `[0-9]` if the *ASCII* flag is used.

For 8-bit (bytes) patterns:
Matches any decimal digit in the ASCII character set; this is equivalent to `[0-9]`.

\D

Matches any character which is not a decimal digit. This is the opposite of `\d`.

Matches `[^0-9]` if the *ASCII* flag is used.

\s

For Unicode (str) patterns:
Matches Unicode whitespace characters (which includes `[ \t\n\r\f\v]`, and also many other characters, for example the non-breaking spaces mandated by typography rules in many languages).

Matches `[ \t\n\r\f\v]` if the *ASCII* flag is used.

For 8-bit (bytes) patterns:
Matches characters considered whitespace in the ASCII character set; this is equivalent to `[ \t\n\r\f\v]`.

\S

Matches any character which is not a whitespace character. This is the opposite of `\s`.

Matches `[^ \t\n\r\f\v]` if the *ASCII* flag is used.

\w

For Unicode (str) patterns:
Matches Unicode word characters; this includes all Unicode alphanumeric characters (as defined by *str.isalnum()*), as well as the underscore (_).

Matches `[a-zA-Z0-9_]` if the *ASCII* flag is used.

For 8-bit (bytes) patterns:
Matches characters considered alphanumeric in the ASCII character set; this is equivalent to `[a-zA-Z0-9_]`. If the *LOCALE* flag is used, matches characters considered alphanumeric in the current locale and the underscore.

\W

Matches any character which is not a word character. This is the opposite of `\w`. By default, matches non-underscore (_) characters for which *str.isalnum()* returns `False`.

Matches `[^a-zA-Z0-9_]` if the *ASCII* flag is used.

If the *LOCALE* flag is used, matches characters which are neither alphanumeric in the current locale nor the underscore.

\Z

Matches only at the end of the string.

Most of the escape sequences supported by Python string literals are also accepted by the regular expression parser:

```
\a      \b      \f      \n
\N      \r      \t      \u
\U      \v      \x      \\
```

(Note that `\b` is used to represent word boundaries, and means "backspace" only inside character classes.)

'\u', '\U', and '\N' escape sequences are only recognized in Unicode (str) patterns. In bytes patterns they are errors. Unknown escapes of ASCII letters are reserved for future use and treated as errors.

Octal escapes are included in a limited form. If the first digit is a 0, or if there are three octal digits, it is considered an octal escape. Otherwise, it is a group reference. As for string literals, octal escapes are always at most three digits in length.

Changed in version 3.3: The '\u' and '\U' escape sequences have been added.

Changed in version 3.6: Unknown escapes consisting of '\' and an ASCII letter now are errors.

Changed in version 3.8: The '\N{*name*}' escape sequence has been added. As in string literals, it expands to the named Unicode character (e.g. '\N{EM DASH}').

6.2.2 Module Contents

The module defines several functions, constants, and an exception. Some of the functions are simplified versions of the full featured methods for compiled regular expressions. Most non-trivial applications always use the compiled form.

Flags

Changed in version 3.6: Flag constants are now instances of *RegexFlag*, which is a subclass of *enum.IntFlag*.

class re.**RegexFlag**

An *enum.IntFlag* class containing the regex options listed below.

Added in version 3.11: - added to __all__

re.**A**

re.**ASCII**

Make \w, \W, \b, \B, \d, \D, \s and \S perform ASCII-only matching instead of full Unicode matching. This is only meaningful for Unicode (str) patterns, and is ignored for bytes patterns.

Corresponds to the inline flag (?a).

> **Note**
>
> The *U* flag still exists for backward compatibility, but is redundant in Python 3 since matches are Unicode by default for str patterns, and Unicode matching isn't allowed for bytes patterns. *UNICODE* and the inline flag (?u) are similarly redundant.

re.**DEBUG**

Display debug information about compiled expression.

No corresponding inline flag.

re.**I**

re.**IGNORECASE**

Perform case-insensitive matching; expressions like [A-Z] will also match lowercase letters. Full Unicode matching (such as Ü matching ü) also works unless the *ASCII* flag is used to disable non-ASCII matches. The current locale does not change the effect of this flag unless the *LOCALE* flag is also used.

Corresponds to the inline flag (?i).

Note that when the Unicode patterns [a-z] or [A-Z] are used in combination with the *IGNORECASE* flag, they will match the 52 ASCII letters and 4 additional non-ASCII letters: 'İ' (U+0130, Latin capital letter I with dot

above), 'ı' (U+0131, Latin small letter dotless i), 'ſ' (U+017F, Latin small letter long s) and 'K' (U+212A, Kelvin sign). If the *ASCII* flag is used, only letters 'a' to 'z' and 'A' to 'Z' are matched.

re.**L**

re.**LOCALE**

Make `\w`, `\W`, `\b`, `\B` and case-insensitive matching dependent on the current locale. This flag can be used only with bytes patterns.

Corresponds to the inline flag `(?L)`.

> **Warning**
>
> This flag is discouraged; consider Unicode matching instead. The locale mechanism is very unreliable as it only handles one "culture" at a time and only works with 8-bit locales. Unicode matching is enabled by default for Unicode (str) patterns and it is able to handle different locales and languages.

Changed in version 3.6: *LOCALE* can be used only with bytes patterns and is not compatible with *ASCII*.

Changed in version 3.7: Compiled regular expression objects with the *LOCALE* flag no longer depend on the locale at compile time. Only the locale at matching time affects the result of matching.

re.**M**

re.**MULTILINE**

When specified, the pattern character `'^'` matches at the beginning of the string and at the beginning of each line (immediately following each newline); and the pattern character `'$'` matches at the end of the string and at the end of each line (immediately preceding each newline). By default, `'^'` matches only at the beginning of the string, and `'$'` only at the end of the string and immediately before the newline (if any) at the end of the string.

Corresponds to the inline flag `(?m)`.

re.**NOFLAG**

Indicates no flag being applied, the value is `0`. This flag may be used as a default value for a function keyword argument or as a base value that will be conditionally ORed with other flags. Example of use as a default value:

```
def myfunc(text, flag=re.NOFLAG):
    return re.match(text, flag)
```

Added in version 3.11.

re.**S**

re.**DOTALL**

Make the `'.'` special character match any character at all, including a newline; without this flag, `'.'` will match anything *except* a newline.

Corresponds to the inline flag `(?s)`.

re.**U**

re.**UNICODE**

In Python 3, Unicode characters are matched by default for `str` patterns. This flag is therefore redundant with **no effect** and is only kept for backward compatibility.

See *ASCII* to restrict matching to ASCII characters instead.

re.**X**

re.**VERBOSE**

This flag allows you to write regular expressions that look nicer and are more readable by allowing you to visually separate logical sections of the pattern and add comments. Whitespace within the pattern is ignored, except when in a character class, or when preceded by an unescaped backslash, or within tokens like `*?`, `(?:` or `(?P<...>`. For example, `(? :` and `* ?` are not allowed. When a line contains a `#` that is not in a character class and is not preceded by an unescaped backslash, all characters from the leftmost such `#` through the end of the line are ignored.

This means that the two following regular expression objects that match a decimal number are functionally equal:

```
a = re.compile(r"""\d +  # the integral part
                   \.    # the decimal point
                   \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

Corresponds to the inline flag `(?x)`.

Functions

re.**compile**(*pattern*, *flags=0*)

Compile a regular expression pattern into a *regular expression object*, which can be used for matching using its `match()`, `search()` and other methods, described below.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

The sequence

```
prog = re.compile(pattern)
result = prog.match(string)
```

is equivalent to

```
result = re.match(pattern, string)
```

but using `re.compile()` and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.

> **Note**
>
> The compiled versions of the most recent patterns passed to `re.compile()` and the module-level matching functions are cached, so programs that use only a few regular expressions at a time needn't worry about compiling regular expressions.

re.**search**(*pattern*, *string*, *flags=0*)

Scan through *string* looking for the first location where the regular expression *pattern* produces a match, and return a corresponding `Match`. Return `None` if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

re.**match**(*pattern*, *string*, *flags=0*)

If zero or more characters at the beginning of *string* match the regular expression *pattern*, return a corresponding `Match`. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

Note that even in *MULTILINE* mode, *re.match()* will only match at the beginning of the string and not at the beginning of each line.

If you want to locate a match anywhere in *string*, use *search()* instead (see also *search() vs. match()*).

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the | operator).

re.**fullmatch**(*pattern*, *string*, *flags=0*)

If the whole *string* matches the regular expression *pattern*, return a corresponding *Match*. Return None if the string does not match the pattern; note that this is different from a zero-length match.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the | operator).

Added in version 3.4.

re.**split**(*pattern*, *string*, *maxsplit=0*, *flags=0*)

Split *string* by the occurrences of *pattern*. If capturing parentheses are used in *pattern*, then the text of all groups in the pattern are also returned as part of the resulting list. If *maxsplit* is nonzero, at most *maxsplit* splits occur, and the remainder of the string is returned as the final element of the list.

```
>>> re.split(r'\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split(r'\W+', 'Words, words, words.', maxsplit=1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

If there are capturing groups in the separator and it matches at the start of the string, the result will start with an empty string. The same holds for the end of the string:

```
>>> re.split(r'(\W+)', '...words, words...')
['', '...', 'words', ', ', 'words', '...', '']
```

That way, separator components are always found at the same relative indices within the result list.

Empty matches for the pattern split the string only when not adjacent to a previous empty match.

```
>>> re.split(r'\b', 'Words, words, words.')
['', 'Words', ', ', 'words', ', ', 'words', '.']
>>> re.split(r'\W*', '...words...')
['', '', 'w', 'o', 'r', 'd', 's', '', '']
>>> re.split(r'(\W*)', '...words...')
['', '...', '', '', 'w', '', 'o', '', 'r', '', 'd', '', 's', '...', '', '', '']
```

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the | operator).

Changed in version 3.1: Added the optional flags argument.

Changed in version 3.7: Added support of splitting on a pattern that could match an empty string.

Deprecated since version 3.13: Passing *maxsplit* and *flags* as positional arguments is deprecated. In future Python versions they will be *keyword-only parameters*.

re.**findall**(*pattern*, *string*, *flags=0*)

Return all non-overlapping matches of *pattern* in *string*, as a list of strings or tuples. The *string* is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result.

The result depends on the number of capturing groups in the pattern. If there are no groups, return a list of strings matching the whole pattern. If there is exactly one group, return a list of strings matching that group. If multiple groups are present, return a list of tuples of strings matching the groups. Non-capturing groups do not affect the form of the result.

```
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.findall(r'(\w+)=(\d+)', 'set width=20 and height=10')
[('width', '20'), ('height', '10')]
```

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

Changed in version 3.7: Non-empty matches can now start just after a previous empty match.

`re.`**`finditer`**(*pattern*, *string*, *flags=0*)

Return an *iterator* yielding `Match` objects over all non-overlapping matches for the RE *pattern* in *string*. The *string* is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

Changed in version 3.7: Non-empty matches can now start just after a previous empty match.

`re.`**`sub`**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Return the string obtained by replacing the leftmost non-overlapping occurrences of *pattern* in *string* by the replacement *repl*. If the pattern isn't found, *string* is returned unchanged. *repl* can be a string or a function; if it is a string, any backslash escapes in it are processed. That is, `\n` is converted to a single newline character, `\r` is converted to a carriage return, and so forth. Unknown escapes of ASCII letters are reserved for future use and treated as errors. Other unknown escapes such as `\&` are left alone. Backreferences, such as `\6`, are replaced with the substring matched by group 6 in the pattern. For example:

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
...        r'static PyObject*\npy_\1(void)\n{',
...        'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

If *repl* is a function, it is called for every non-overlapping occurrence of *pattern*. The function takes a single `Match` argument, and returns the replacement string. For example:

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
...
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

The pattern may be a string or a `Pattern`.

The optional argument *count* is the maximum number of pattern occurrences to be replaced; *count* must be a non-negative integer. If omitted or zero, all occurrences will be replaced. Empty matches for the pattern are replaced only when not adjacent to a previous empty match, so `sub('x*', '-', 'abxd')` returns `'-a-b--d-'`.

In string-type *repl* arguments, in addition to the character escapes and backreferences described above, `\g<name>` will use the substring matched by the group named `name`, as defined by the `(?P<name>...)` syntax. `\g<number>` uses the corresponding group number; `\g<2>` is therefore equivalent to `\2`, but isn't ambiguous in a replacement such as `\g<2>0`. `\20` would be interpreted as a reference to group 20, not a reference to group

2 followed by the literal character `'0'`. The backreference `\g<0>` substitutes in the entire substring matched by the RE.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

Changed in version 3.1: Added the optional flags argument.

Changed in version 3.5: Unmatched groups are replaced with an empty string.

Changed in version 3.6: Unknown escapes in *pattern* consisting of `'\'` and an ASCII letter now are errors.

Changed in version 3.7: Unknown escapes in *repl* consisting of `'\'` and an ASCII letter now are errors. Empty matches for the pattern are replaced when adjacent to a previous non-empty match.

Changed in version 3.12: Group *id* can only contain ASCII digits. In *bytes* replacement strings, group *name* can only contain bytes in the ASCII range (`b'\x00'-b'\x7f'`).

Deprecated since version 3.13: Passing *count* and *flags* as positional arguments is deprecated. In future Python versions they will be *keyword-only parameters*.

`re.`**`subn`**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Perform the same operation as *sub()*, but return a tuple `(new_string, number_of_subs_made)`.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

`re.`**`escape`**(*pattern*)

Escape special characters in *pattern*. This is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it. For example:

```
>>> print(re.escape('https://www.python.org'))
https://www\.python\.org

>>> legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|~:"
>>> print('[%s]+' % re.escape(legal_chars))
[abcdefghijklmnopqrstuvwxyz0123456789!\#\$\%\&\'\*\+\-\.\^_`\|\~:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print('|'.join(map(re.escape, sorted(operators, reverse=True))))
/|\-|\+|\*\*|\*
```

This function must not be used for the replacement string in *sub()* and *subn()*, only backslashes should be escaped. For example:

```
>>> digits_re = r'\d+'
>>> sample = '/usr/sbin/sendmail - 0 errors, 12 warnings'
>>> print(re.sub(digits_re, digits_re.replace('\\', r'\\'), sample))
/usr/sbin/sendmail - \d+ errors, \d+ warnings
```

Changed in version 3.3: The `'_'` character is no longer escaped.

Changed in version 3.7: Only characters that can have special meaning in a regular expression are escaped. As a result, `'!'`, `'"'`, `'%'`, `"'"`, `','`, `'/'`, `':'`, `';'`, `'<'`, `'='`, `'>'`, `'@'`, and `"`` " are no longer escaped.

`re.`**`purge`**()

Clear the regular expression cache.

Exceptions

exception re.**PatternError**(*msg*, *pattern=None*, *pos=None*)

Exception raised when a string passed to one of the functions here is not a valid regular expression (for example, it might contain unmatched parentheses) or when some other error occurs during compilation or matching. It is never an error if a string contains no match for a pattern. The `PatternError` instance has the following additional attributes:

msg

The unformatted error message.

pattern

The regular expression pattern.

pos

The index in *pattern* where compilation failed (may be `None`).

lineno

The line corresponding to *pos* (may be `None`).

colno

The column corresponding to *pos* (may be `None`).

Changed in version 3.5: Added additional attributes.

Changed in version 3.13: `PatternError` was originally named `error`; the latter is kept as an alias for backward compatibility.

6.2.3 Regular Expression Objects

class re.**Pattern**

Compiled regular expression object returned by *re.compile()*.

Changed in version 3.9: *re.Pattern* supports `[]` to indicate a Unicode (str) or bytes pattern. See *Generic Alias Type*.

Pattern.**search**(*string*[, *pos*[, *endpos*]])

Scan through *string* looking for the first location where this regular expression produces a match, and return a corresponding *Match*. Return `None` if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

The optional second parameter *pos* gives an index in the string where the search is to start; it defaults to `0`. This is not completely equivalent to slicing the string; the `'^'` pattern character matches at the real beginning of the string and at positions just after a newline, but not necessarily at the index where the search is to start.

The optional parameter *endpos* limits how far the string will be searched; it will be as if the string is *endpos* characters long, so only the characters from *pos* to `endpos - 1` will be searched for a match. If *endpos* is less than *pos*, no match will be found; otherwise, if *rx* is a compiled regular expression object, `rx.search(string, 0, 50)` is equivalent to `rx.search(string[:50], 0)`.

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")     # Match at index 0
<re.Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1)  # No match; search doesn't include the "d"
```

Pattern.**match**(*string*[, *pos*[, *endpos*]])

If zero or more characters at the *beginning* of *string* match this regular expression, return a corresponding *Match*. Return None if the string does not match the pattern; note that this is different from a zero-length match.

The optional *pos* and *endpos* parameters have the same meaning as for the *search()* method.

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.match("dog", 1)   # Match as "o" is the 2nd character of "dog".
<re.Match object; span=(1, 2), match='o'>
```

If you want to locate a match anywhere in *string*, use *search()* instead (see also *search() vs. match()*).

Pattern.**fullmatch**(*string*[, *pos*[, *endpos*]])

If the whole *string* matches this regular expression, return a corresponding *Match*. Return None if the string does not match the pattern; note that this is different from a zero-length match.

The optional *pos* and *endpos* parameters have the same meaning as for the *search()* method.

```
>>> pattern = re.compile("o[gh]")
>>> pattern.fullmatch("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.fullmatch("ogre")     # No match as not the full string matches.
>>> pattern.fullmatch("doggie", 1, 3)   # Matches within given limits.
<re.Match object; span=(1, 3), match='og'>
```

Added in version 3.4.

Pattern.**split**(*string*, *maxsplit=0*)

Identical to the *split()* function, using the compiled pattern.

Pattern.**findall**(*string*[, *pos*[, *endpos*]])

Similar to the *findall()* function, using the compiled pattern, but also accepts optional *pos* and *endpos* parameters that limit the search region like for *search()*.

Pattern.**finditer**(*string*[, *pos*[, *endpos*]])

Similar to the *finditer()* function, using the compiled pattern, but also accepts optional *pos* and *endpos* parameters that limit the search region like for *search()*.

Pattern.**sub**(*repl*, *string*, *count=0*)

Identical to the *sub()* function, using the compiled pattern.

Pattern.**subn**(*repl*, *string*, *count=0*)

Identical to the *subn()* function, using the compiled pattern.

Pattern.**flags**

The regex matching flags. This is a combination of the flags given to *compile()*, any (?...) inline flags in the pattern, and implicit flags such as *UNICODE* if the pattern is a Unicode string.

Pattern.**groups**

The number of capturing groups in the pattern.

Pattern.**groupindex**

A dictionary mapping any symbolic group names defined by (?P<id>) to group numbers. The dictionary is empty if no symbolic groups were used in the pattern.

Pattern.**pattern**

The pattern string from which the pattern object was compiled.

Changed in version 3.7: Added support of *copy.copy()* and *copy.deepcopy()*. Compiled regular expression objects are considered atomic.

6.2.4 Match Objects

Match objects always have a boolean value of `True`. Since *match()* and *search()* return `None` when there is no match, you can test whether there was a match with a simple `if` statement:

```
match = re.search(pattern, string)
if match:
    process(match)
```

class re.**Match**

Match object returned by successful `match`es and `search`es.

Changed in version 3.9: *re.Match* supports `[]` to indicate a Unicode (str) or bytes match. See *Generic Alias Type*.

Match.**expand**(*template*)

Return the string obtained by doing backslash substitution on the template string *template*, as done by the *sub()* method. Escapes such as `\n` are converted to the appropriate characters, and numeric backreferences (`\1`, `\2`) and named backreferences (`\g<1>`, `\g<name>`) are replaced by the contents of the corresponding group. The backreference `\g<0>` will be replaced by the entire match.

Changed in version 3.5: Unmatched groups are replaced with an empty string.

Match.**group**([*group1*, ...])

Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, *group1* defaults to zero (the whole match is returned). If a *groupN* argument is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1..99], it is the string matching the corresponding parenthesized group. If a group number is negative or larger than the number of groups defined in the pattern, an *IndexError* exception is raised. If a group is contained in a part of the pattern that did not match, the corresponding result is `None`. If a group is contained in a part of the pattern that matched multiple times, the last match is returned.

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       # The entire match
'Isaac Newton'
>>> m.group(1)       # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)       # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)    # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

If the regular expression uses the `(?P<name>...)` syntax, the *groupN* arguments may also be strings identifying groups by their group name. If a string argument is not used as a group name in the pattern, an *IndexError* exception is raised.

A moderately complicated example:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

Named groups can also be referred to by their index:

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

If a group matches multiple times, only the last match is accessible:

```
>>> m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
>>> m.group(1)                        # Returns only the last match.
'c3'
```

`Match.__getitem__`(*g*)

This is identical to `m.group(g)`. This allows easier access to an individual group from a match:

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m[0]       # The entire match
'Isaac Newton'
>>> m[1]       # The first parenthesized subgroup.
'Isaac'
>>> m[2]       # The second parenthesized subgroup.
'Newton'
```

Named groups are supported as well:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Isaac Newton")
>>> m['first_name']
'Isaac'
>>> m['last_name']
'Newton'
```

Added in version 3.6.

`Match.groups`(*default=None*)

Return a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The *default* argument is used for groups that did not participate in the match; it defaults to `None`.

For example:

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

If we make the decimal place and everything after it optional, not all groups might participate in the match. These groups will default to `None` unless the *default* argument is given:

```
>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()      # Second group defaults to None.
('24', None)
>>> m.groups('0')   # Now, the second group defaults to '0'.
('24', '0')
```

`Match.groupdict`(*default=None*)

Return a dictionary containing all the *named* subgroups of the match, keyed by the subgroup name. The *default* argument is used for groups that did not participate in the match; it defaults to `None`. For example:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

Match.**start**([*group*])

Match.**end**([*group*])

Return the indices of the start and end of the substring matched by *group*; *group* defaults to zero (meaning the whole matched substring). Return `-1` if *group* exists but did not contribute to the match. For a match object *m*, and a group *g* that did contribute to the match, the substring matched by group *g* (equivalent to `m.group(g)`) is

```
m.string[m.start(g):m.end(g)]
```

Note that `m.start(group)` will equal `m.end(group)` if *group* matched a null string. For example, after `m = re.search('b(c?)', 'cba')`, `m.start(0)` is 1, `m.end(0)` is 2, `m.start(1)` and `m.end(1)` are both 2, and `m.start(2)` raises an *IndexError* exception.

An example that will remove *remove_this* from email addresses:

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

Match.**span**([*group*])

For a match *m*, return the 2-tuple `(m.start(group), m.end(group))`. Note that if *group* did not contribute to the match, this is `(-1, -1)`. *group* defaults to zero, the entire match.

Match.**pos**

The value of *pos* which was passed to the *search()* or *match()* method of a *regex object*. This is the index into the string at which the RE engine started looking for a match.

Match.**endpos**

The value of *endpos* which was passed to the *search()* or *match()* method of a *regex object*. This is the index into the string beyond which the RE engine will not go.

Match.**lastindex**

The integer index of the last matched capturing group, or `None` if no group was matched at all. For example, the expressions `(a)b`, `((a)(b))`, and `((ab))` will have `lastindex == 1` if applied to the string `'ab'`, while the expression `(a)(b)` will have `lastindex == 2`, if applied to the same string.

Match.**lastgroup**

The name of the last matched capturing group, or `None` if the group didn't have a name, or if no group was matched at all.

Match.**re**

The *regular expression object* whose *match()* or *search()* method produced this match instance.

Match.**string**

The string passed to *match()* or *search()*.

Changed in version 3.7: Added support of *copy.copy()* and *copy.deepcopy()*. Match objects are considered atomic.

6.2.5 Regular Expression Examples

Checking for a Pair

In this example, we'll use the following helper function to display match objects a little more gracefully:

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

Suppose you are writing a poker program where a player's hand is represented as a 5-character string with each character representing a card, "a" for ace, "k" for king, "q" for queen, "j" for jack, "t" for 10, and "2" through "9" representing the card with that value.

To see if a given string is a valid hand, one could do the following:

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q"))  # Valid.
"<Match: 'akt5q', groups=()>"
>>> displaymatch(valid.match("akt5e"))  # Invalid.
>>> displaymatch(valid.match("akt"))    # Invalid.
>>> displaymatch(valid.match("727ak"))  # Valid.
"<Match: '727ak', groups=()>"
```

That last hand, `"727ak"`, contained a pair, or two of the same valued cards. To match this with a regular expression, one could use backreferences as such:

```
>>> pair = re.compile(r".*(.).*\1")
>>> displaymatch(pair.match("717ak"))     # Pair of 7s.
"<Match: '717', groups=('7',)>"
>>> displaymatch(pair.match("718ak"))     # No pairs.
>>> displaymatch(pair.match("354aa"))     # Pair of aces.
"<Match: '354aa', groups=('a',)>"
```

To find out what card the pair consists of, one could use the *group()* method of the match object in the following manner:

```
>>> pair = re.compile(r".*(.).*\1")
>>> pair.match("717ak").group(1)
'7'

# Error because re.match() returns None, which doesn't have a group() method:
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(.).*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'

>>> pair.match("354aa").group(1)
'a'
```

Simulating scanf()

Python does not currently have an equivalent to `scanf()`. Regular expressions are generally more powerful, though also more verbose, than `scanf()` format strings. The table below offers some more-or-less equivalent mappings between `scanf()` format tokens and regular expressions.

`scanf()` Token	Regular Expression
`%c`	`.`
`%5c`	`.{5}`
`%d`	`[-+]?\d+`
`%e, %E, %f, %g`	`[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?`
`%i`	`[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)`
`%o`	`[-+]?[0-7]+`
`%s`	`\S+`
`%u`	`\d+`
`%x, %X`	`[-+]?(0[xX])?[\dA-Fa-f]+`

To extract the filename and numbers from a string like

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

you would use a `scanf()` format like

```
%s - %d errors, %d warnings
```

The equivalent regular expression would be

```
(\S+) - (\d+) errors, (\d+) warnings
```

search() vs. match()

Python offers different primitive operations based on regular expressions:

- *re.match()* checks for a match only at the beginning of the string
- *re.search()* checks for a match anywhere in the string (this is what Perl does by default)
- *re.fullmatch()* checks for entire string to be a match

For example:

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("c", "abcdef")   # Match
<re.Match object; span=(2, 3), match='c'>
>>> re.fullmatch("p.*n", "python") # Match
<re.Match object; span=(0, 6), match='python'>
>>> re.fullmatch("r.*n", "python") # No match
```

Regular expressions beginning with `'^'` can be used with *search()* to restrict the match at the beginning of the string:

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("^c", "abcdef")  # No match
>>> re.search("^a", "abcdef")  # Match
<re.Match object; span=(0, 1), match='a'>
```

Note however that in *MULTILINE* mode *match()* only matches at the beginning of the string, whereas using *search()* with a regular expression beginning with `'^'` will match at the beginning of each line.

```
>>> re.match("X", "A\nB\nX", re.MULTILINE)  # No match
>>> re.search("^X", "A\nB\nX", re.MULTILINE)  # Match
<re.Match object; span=(4, 5), match='X'>
```

Making a Phonebook

split() splits a string into a list delimited by the passed pattern. The method is invaluable for converting textual data into data structures that can be easily read and modified by Python as demonstrated in the following example that creates a phonebook.

First, here is the input. Normally it may come from a file, here we are using triple-quoted string syntax

```
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
...
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
...
...
... Heather Albrecht: 548.326.4584 919 Park Place"""
```

The entries are separated by one or more newlines. Now we convert the string into a list with each nonempty line having its own entry:

```
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street',
'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
'Frank Burger: 925.541.7625 662 South Dogwood Way',
'Heather Albrecht: 548.326.4584 919 Park Place']
```

Finally, split each entry into a list with first name, last name, telephone number, and address. We use the `maxsplit` parameter of *split()* because the address has spaces, our splitting pattern, in it:

```
>>> [re.split(":? ", entry, maxsplit=3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

The `:?` pattern matches the colon after the last name, so that it does not occur in the result list. With a `maxsplit` of `4`, we could separate the house number from the street name:

```
>>> [re.split(":? ", entry, maxsplit=4) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]
```

Text Munging

sub() replaces every occurrence of a pattern with a string or the result of a function. This example demonstrates using *sub()* with a function to "munge" text, or randomize the order of all the characters in each word of a sentence except for the first and last characters:

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
...
>>> text = "Professor Abdolmalek, please report your absences promptly."
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reoprt yuor asnebces potlmrpy.'
```

Finding all Adverbs

findall() matches *all* occurrences of a pattern, not just the first one as *search()* does. For example, if a writer wanted to find all of the adverbs in some text, they might use *findall()* in the following manner:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly\b", text)
['carefully', 'quickly']
```

Finding all Adverbs and their Positions

If one wants more information about all matches of a pattern than the matched text, *finditer()* is useful as it provides *Match* objects instead of strings. Continuing with the previous example, if a writer wanted to find all of the adverbs *and their positions* in some text, they would use *finditer()* in the following manner:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly\b", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
07-16: carefully
40-47: quickly
```

Raw String Notation

Raw string notation (`r"text"`) keeps regular expressions sane. Without it, every backslash (`'\'`) in a regular expression would have to be prefixed with another one to escape it. For example, the two following lines of code are functionally identical:

```
>>> re.match(r"\W(.)\1\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
>>> re.match("\\W(.)\\1\\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
```

When one wants to match a literal backslash, it must be escaped in the regular expression. With raw string notation, this means `r"\\"`. Without raw string notation, one must use `"\\\\"`, making the following lines of code functionally identical:

```
>>> re.match(r"\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
>>> re.match("\\\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
```

Writing a Tokenizer

A tokenizer or scanner analyzes a string to categorize groups of characters. This is a useful first step in writing a compiler or interpreter.

The text categories are specified with regular expressions. The technique is to combine those into a single master regular expression and to loop over successive matches:

```
from typing import NamedTuple
import re

class Token(NamedTuple):
    type: str
    value: str
    line: int
    column: int

def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('ASSIGN',   r':='),           # Assignment operator
        ('END',      r';'),            # Statement terminator
        ('ID',       r'[A-Za-z]+'),    # Identifiers
        ('OP',       r'[+\-*/]'),      # Arithmetic operators
        ('NEWLINE',  r'\n'),           # Line endings
        ('SKIP',     r'[ \t]+'),       # Skip over spaces and tabs
        ('MISMATCH', r'.'),            # Any other character
    ]
    tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    line_num = 1
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        column = mo.start() - line_start
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'ID' and value in keywords:
            kind = value
        elif kind == 'NEWLINE':
            line_start = mo.end()
            line_num += 1
            continue
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise RuntimeError(f'{value!r} unexpected on line {line_num}')
        yield Token(kind, value, line_num, column)

statements = '''
```

(continues on next page)

(continued from previous page)

```
    IF quantity THEN
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
'''

for token in tokenize(statements):
    print(token)
```

The tokenizer produces the following output:

```
Token(type='IF', value='IF', line=2, column=4)
Token(type='ID', value='quantity', line=2, column=7)
Token(type='THEN', value='THEN', line=2, column=16)
Token(type='ID', value='total', line=3, column=8)
Token(type='ASSIGN', value=':=', line=3, column=14)
Token(type='ID', value='total', line=3, column=17)
Token(type='OP', value='+', line=3, column=23)
Token(type='ID', value='price', line=3, column=25)
Token(type='OP', value='*', line=3, column=31)
Token(type='ID', value='quantity', line=3, column=33)
Token(type='END', value=';', line=3, column=41)
Token(type='ID', value='tax', line=4, column=8)
Token(type='ASSIGN', value=':=', line=4, column=12)
Token(type='ID', value='price', line=4, column=15)
Token(type='OP', value='*', line=4, column=21)
Token(type='NUMBER', value=0.05, line=4, column=23)
Token(type='END', value=';', line=4, column=27)
Token(type='ENDIF', value='ENDIF', line=5, column=4)
Token(type='END', value=';', line=5, column=9)
```

6.3 `difflib` — Helpers for computing deltas

Source code: Lib/difflib.py

This module provides classes and functions for comparing sequences. It can be used for example, for comparing files, and can produce information about file differences in various formats, including HTML and context and unified diffs. For comparing directories and files, see also, the *filecmp* module.

class difflib.**SequenceMatcher**

This is a flexible class for comparing pairs of sequences of any type, so long as the sequence elements are *hashable*. The basic algorithm predates, and is a little fancier than, an algorithm published in the late 1980's by Ratcliff and Obershelp under the hyperbolic name "gestalt pattern matching." The idea is to find the longest contiguous matching subsequence that contains no "junk" elements; these "junk" elements are ones that are uninteresting in some sense, such as blank lines or whitespace. (Handling junk is an extension to the Ratcliff and Obershelp algorithm.) The same idea is then applied recursively to the pieces of the sequences to the left and to the right of the matching subsequence. This does not yield minimal edit sequences, but does tend to yield matches that "look right" to people.

Timing: The basic Ratcliff-Obershelp algorithm is cubic time in the worst case and quadratic time in the expected case. *SequenceMatcher* is quadratic time for the worst case and has expected-case behavior dependent in a complicated way on how many elements the sequences have in common; best case time is linear.

Automatic junk heuristic: *SequenceMatcher* supports a heuristic that automatically treats certain sequence items as junk. The heuristic counts how many times each individual item appears in the sequence. If an item's duplicates (after the first one) account for more than 1% of the sequence and the sequence is at least 200 items long, this item is marked as "popular" and is treated as junk for the purpose of sequence matching. This heuristic can be turned off by setting the `autojunk` argument to `False` when creating the *SequenceMatcher*.

Changed in version 3.2: Added the *autojunk* parameter.

class difflib.**Differ**

This is a class for comparing sequences of lines of text, and producing human-readable differences or deltas. Differ uses *SequenceMatcher* both to compare sequences of lines, and to compare sequences of characters within similar (near-matching) lines.

Each line of a *Differ* delta begins with a two-letter code:

Code	Meaning
`'- '`	line unique to sequence 1
`'+ '`	line unique to sequence 2
`'  '`	line common to both sequences
`'? '`	line not present in either input sequence

Lines beginning with '`?`' attempt to guide the eye to intraline differences, and were not present in either input sequence. These lines can be confusing if the sequences contain whitespace characters, such as spaces, tabs or line breaks.

class difflib.**HtmlDiff**

This class can be used to create an HTML table (or a complete HTML file containing the table) showing a side by side, line by line comparison of text with inter-line and intra-line change highlights. The table can be generated in either full or contextual difference mode.

The constructor for this class is:

__init__(*tabsize=8*, *wrapcolumn=None*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

Initializes instance of *HtmlDiff*.

tabsize is an optional keyword argument to specify tab stop spacing and defaults to `8`.

wrapcolumn is an optional keyword to specify column number where lines are broken and wrapped, defaults to `None` where lines are not wrapped.

linejunk and *charjunk* are optional keyword arguments passed into *ndiff()* (used by *HtmlDiff* to generate the side by side HTML differences). See *ndiff()* documentation for argument default values and descriptions.

The following methods are public:

make_file(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*, ***, *charset='utf-8'*)

Compares *fromlines* and *tolines* (lists of strings) and returns a string which is a complete HTML file containing a table showing line by line differences with inter-line and intra-line changes highlighted.

fromdesc and *todesc* are optional keyword arguments to specify from/to file column header strings (both default to an empty string).

context and *numlines* are both optional keyword arguments. Set *context* to `True` when contextual differences are to be shown, else the default is `False` to show the full files. *numlines* defaults to `5`. When *context* is `True` *numlines* controls the number of context lines which surround the difference highlights. When *context* is `False` *numlines* controls the number of lines which are shown before a difference highlight when using

the "next" hyperlinks (setting to zero would cause the "next" hyperlinks to place the next difference highlight at the top of the browser without any leading context).

> **Note**
>
> *fromdesc* and *todesc* are interpreted as unescaped HTML and should be properly escaped while receiving input from untrusted sources.

Changed in version 3.5: *charset* keyword-only argument was added. The default charset of HTML document changed from `'ISO-8859-1'` to `'utf-8'`.

make_table(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*)

Compares *fromlines* and *tolines* (lists of strings) and returns a string which is a complete HTML table showing line by line differences with inter-line and intra-line changes highlighted.

The arguments for this method are the same as those for the *make_file()* method.

difflib.**context_diff**(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)

Compare *a* and *b* (lists of strings); return a delta (a *generator* generating the delta lines) in context diff format.

Context diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in a before/after style. The number of context lines is set by *n* which defaults to three.

By default, the diff control lines (those with `***` or `---`) are created with a trailing newline. This is helpful so that inputs created from *io.IOBase.readlines()* result in diffs that are suitable for use with *io.IOBase.writelines()* since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the *lineterm* argument to `""` so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for *fromfile*, *tofile*, *fromfiledate*, and *tofiledate*. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```
>>> import sys
>>> from difflib import *
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(context_diff(s1, s2, fromfile='before.py',
...                       tofile='after.py'))
*** before.py
--- after.py
***************
*** 1,4 ****
! bacon
! eggs
! ham
  guido
--- 1,4 ----
! python
! eggy
! hamster
  guido
```

See *A command-line interface to difflib* for a more detailed example.

difflib.**get_close_matches**(*word*, *possibilities*, *n=3*, *cutoff=0.6*)

Return a list of the best "good enough" matches. *word* is a sequence for which close matches are desired (typically a string), and *possibilities* is a list of sequences against which to match *word* (typically a list of strings).

Optional argument *n* (default `3`) is the maximum number of close matches to return; *n* must be greater than `0`.

Optional argument *cutoff* (default `0.6`) is a float in the range [0, 1]. Possibilities that don't score at least that similar to *word* are ignored.

The best (no more than *n*) matches among the possibilities are returned in a list, sorted by similarity score, most similar first.

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('pineapple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

difflib.**ndiff**(*a*, *b*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

Compare *a* and *b* (lists of strings); return a *Differ*-style delta (a *generator* generating the delta lines).

Optional keyword parameters *linejunk* and *charjunk* are filtering functions (or `None`):

linejunk: A function that accepts a single string argument, and returns true if the string is junk, or false if not. The default is `None`. There is also a module-level function *IS_LINE_JUNK()*, which filters out lines without visible characters, except for at most one pound character (`'#'`) – however the underlying *SequenceMatcher* class does a dynamic analysis of which lines are so frequent as to constitute noise, and this usually works better than using this function.

charjunk: A function that accepts a character (a string of length 1), and returns if the character is junk, or false if not. The default is module-level function *IS_CHARACTER_JUNK()*, which filters out whitespace characters (a blank or tab; it's a bad idea to include newline in this!).

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> print(''.join(diff), end="")
- one
?  ^
+ ore
?  ^
- two
- three
?  -
+ tree
+ emu
```

difflib.**restore**(*sequence*, *which*)

Return one of the two sequences that generated a delta.

Given a *sequence* produced by *Differ.compare()* or *ndiff()*, extract lines originating from file 1 or 2 (parameter *which*), stripping off line prefixes.

Example:

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print(''.join(restore(diff, 1)), end="")
one
two
three
>>> print(''.join(restore(diff, 2)), end="")
ore
tree
emu
```

difflib.**unified_diff**(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)

Compare *a* and *b* (lists of strings); return a delta (a *generator* generating the delta lines) in unified diff format.

Unified diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in an inline style (instead of separate before/after blocks). The number of context lines is set by *n* which defaults to three.

By default, the diff control lines (those with `---`, `+++`, or `@@`) are created with a trailing newline. This is helpful so that inputs created from *io.IOBase.readlines()* result in diffs that are suitable for use with *io.IOBase.writelines()* since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the *lineterm* argument to `""` so that the output will be uniformly newline free.

The unified diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for *fromfile*, *tofile*, *fromfiledate*, and *tofiledate*. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(unified_diff(s1, s2, fromfile='before.py', tofile=
↪'after.py'))
--- before.py
+++ after.py
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
+python
+eggy
+hamster
 guido
```

See *A command-line interface to difflib* for a more detailed example.

difflib.**diff_bytes**(*dfunc*, *a*, *b*, *fromfile=b''*, *tofile=b''*, *fromfiledate=b''*, *tofiledate=b''*, *n=3*, *lineterm=b'\n'*)

Compare *a* and *b* (lists of bytes objects) using *dfunc*; yield a sequence of delta lines (also bytes) in the format returned by *dfunc*. *dfunc* must be a callable, typically either *unified_diff()* or *context_diff()*.

Allows you to compare data with unknown or inconsistent encoding. All inputs except *n* must be bytes objects, not str. Works by losslessly converting all inputs (except *n*) to str, and calling `dfunc(a, b, fromfile, tofile, fromfiledate, tofiledate, n, lineterm)`. The output of *dfunc* is then converted back to bytes, so the delta lines that you receive have the same unknown/inconsistent encodings as *a* and *b*.

Added in version 3.5.

difflib.**IS_LINE_JUNK**(*line*)

Return `True` for ignorable lines. The line *line* is ignorable if *line* is blank or contains a single `'#'`, otherwise it is not ignorable. Used as a default for parameter *linejunk* in *ndiff()* in older versions.

difflib.**IS_CHARACTER_JUNK**(*ch*)

Return `True` for ignorable characters. The character *ch* is ignorable if *ch* is a space or tab, otherwise it is not ignorable. Used as a default for parameter *charjunk* in *ndiff()*.

See also

Pattern Matching: The Gestalt Approach
Discussion of a similar algorithm by John W. Ratcliff and D. E. Metzener. This was published in Dr. Dobb's Journal in July, 1988.

6.3.1 SequenceMatcher Objects

The *SequenceMatcher* class has this constructor:

class difflib.**SequenceMatcher**(*isjunk=None*, *a=''*, *b=''*, *autojunk=True*)

Optional argument *isjunk* must be `None` (the default) or a one-argument function that takes a sequence element and returns true if and only if the element is "junk" and should be ignored. Passing `None` for *isjunk* is equivalent to passing `lambda x: False`; in other words, no elements are ignored. For example, pass:

```
lambda x: x in " \t"
```

if you're comparing lines as sequences of characters, and don't want to synch up on blanks or hard tabs.

The optional arguments *a* and *b* are sequences to be compared; both default to empty strings. The elements of both sequences must be *hashable*.

The optional argument *autojunk* can be used to disable the automatic junk heuristic.

Changed in version 3.2: Added the *autojunk* parameter.

SequenceMatcher objects get three data attributes: *bjunk* is the set of elements of *b* for which *isjunk* is `True`; *bpopular* is the set of non-junk elements considered popular by the heuristic (if it is not disabled); *b2j* is a dict mapping the remaining elements of *b* to a list of positions where they occur. All three are reset whenever *b* is reset with *set_seqs()* or *set_seq2()*.

Added in version 3.2: The *bjunk* and *bpopular* attributes.

SequenceMatcher objects have the following methods:

set_seqs(*a*, *b*)

Set the two sequences to be compared.

SequenceMatcher computes and caches detailed information about the second sequence, so if you want to compare one sequence against many sequences, use *set_seq2()* to set the commonly used sequence once and call *set_seq1()* repeatedly, once for each of the other sequences.

set_seq1(*a*)

Set the first sequence to be compared. The second sequence to be compared is not changed.

set_seq2(*b*)

Set the second sequence to be compared. The first sequence to be compared is not changed.

find_longest_match(*alo=0*, *ahi=None*, *blo=0*, *bhi=None*)

Find longest matching block in `a[alo:ahi]` and `b[blo:bhi]`.

If *isjunk* was omitted or `None`, *find_longest_match()* returns `(i, j, k)` such that `a[i:i+k]` is equal to `b[j:j+k]`, where `alo <= i <= i+k <= ahi` and `blo <= j <= j+k <= bhi`. For all `(i', j', k')` meeting those conditions, the additional conditions `k >= k'`, `i <= i'`, and if `i == i'`, `j <= j'` are also met. In other words, of all maximal matching blocks, return one that starts earliest in *a*, and of all those maximal matching blocks that start earliest in *a*, return the one that starts earliest in *b*.

```
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

If *isjunk* was provided, first the longest matching block is determined as above, but with the additional restriction that no junk element appears in the block. Then that block is extended as far as possible by matching (only) junk elements on both sides. So the resulting block never matches on junk except as identical junk happens to be adjacent to an interesting match.

Here's the same example as before, but considering blanks to be junk. That prevents `' abcd'` from matching the `'  abcd'` at the tail end of the second sequence directly. Instead only the `'abcd'` can match, and matches the leftmost `'abcd'` in the second sequence:

```
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

If no blocks match, this returns `(alo, blo, 0)`.

This method returns a *named tuple* `Match(a, b, size)`.

Changed in version 3.9: Added default arguments.

get_matching_blocks()

Return list of triples describing non-overlapping matching subsequences. Each triple is of the form `(i, j, n)`, and means that `a[i:i+n] == b[j:j+n]`. The triples are monotonically increasing in *i* and *j*.

The last triple is a dummy, and has the value `(len(a), len(b), 0)`. It is the only triple with `n == 0`. If `(i, j, n)` and `(i', j', n')` are adjacent triples in the list, and the second is not the last triple in the list, then `i+n < i'` or `j+n < j'`; in other words, adjacent triples always describe non-adjacent equal blocks.

```
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(a=5, b=4, size=0)]
```

get_opcodes()

Return list of 5-tuples describing how to turn *a* into *b*. Each tuple is of the form `(tag, i1, i2, j1, j2)`. The first tuple has `i1 == j1 == 0`, and remaining tuples have *i1* equal to the *i2* from the preceding tuple, and, likewise, *j1* equal to the previous *j2*.

The *tag* values are strings, with these meanings:

Value	Meaning
`'replace'`	`a[i1:i2]` should be replaced by `b[j1:j2]`.
`'delete'`	`a[i1:i2]` should be deleted. Note that `j1 == j2` in this case.
`'insert'`	`b[j1:j2]` should be inserted at `a[i1:i1]`. Note that `i1 == i2` in this case.
`'equal'`	`a[i1:i2] == b[j1:j2]` (the sub-sequences are equal).

For example:

```
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print('{:7}   a[{}:{}] --> b[{}:{}] {!r:>8} --> {!r}'.format(
...         tag, i1, i2, j1, j2, a[i1:i2], b[j1:j2]))
delete    a[0:1] --> b[0:0]      'q' --> ''
equal     a[1:3] --> b[0:2]     'ab' --> 'ab'
replace   a[3:4] --> b[2:3]      'x' --> 'y'
equal     a[4:6] --> b[3:5]     'cd' --> 'cd'
insert    a[6:6] --> b[5:6]       '' --> 'f'
```

get_grouped_opcodes(*n=3*)

Return a *generator* of groups with up to *n* lines of context.

Starting with the groups returned by *get_opcodes()*, this method splits out smaller change clusters and eliminates intervening ranges which have no changes.

The groups are returned in the same format as *get_opcodes()*.

ratio()

Return a measure of the sequences' similarity as a float in the range [0, 1].

Where T is the total number of elements in both sequences, and M is the number of matches, this is 2.0*M / T. Note that this is `1.0` if the sequences are identical, and `0.0` if they have nothing in common.

This is expensive to compute if *get_matching_blocks()* or *get_opcodes()* hasn't already been called, in which case you may want to try *quick_ratio()* or *real_quick_ratio()* first to get an upper bound.

Note

Caution: The result of a *ratio()* call may depend on the order of the arguments. For instance:

```
>>> SequenceMatcher(None, 'tide', 'diet').ratio()
0.25
>>> SequenceMatcher(None, 'diet', 'tide').ratio()
0.5
```

quick_ratio()

Return an upper bound on *ratio()* relatively quickly.

real_quick_ratio()

Return an upper bound on *ratio()* very quickly.

The three methods that return the ratio of matching to total characters can give different results due to differing levels of approximation, although *quick_ratio()* and *real_quick_ratio()* are always at least as large as *ratio()*:

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

6.3.2 SequenceMatcher Examples

This example compares two strings, considering blanks to be "junk":

```
>>> s = SequenceMatcher(lambda x: x == " ",
...                     "private Thread currentThread;",
...                     "private volatile Thread currentThread;")
```

ratio() returns a float in [0, 1], measuring the similarity of the sequences. As a rule of thumb, a *ratio()* value over 0.6 means the sequences are close matches:

```
>>> print(round(s.ratio(), 3))
0.866
```

If you're only interested in where the sequences match, *get_matching_blocks()* is handy:

```
>>> for block in s.get_matching_blocks():
...     print("a[%d] and b[%d] match for %d elements" % block)
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 21 elements
a[29] and b[38] match for 0 elements
```

Note that the last tuple returned by *get_matching_blocks()* is always a dummy, `(len(a), len(b), 0)`, and this is the only case in which the last tuple element (number of elements matched) is `0`.

If you want to know how to change the first sequence into the second, use *get_opcodes()*:

```
>>> for opcode in s.get_opcodes():
...     print("%6s a[%d:%d] b[%d:%d]" % opcode)
 equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
 equal a[8:29] b[17:38]
```

See also

- The *get_close_matches()* function in this module which shows how simple code building on *SequenceMatcher* can be used to do useful work.
- Simple version control recipe for a small application built with *SequenceMatcher*.

6.3.3 Differ Objects

Note that *Differ*-generated deltas make no claim to be **minimal** diffs. To the contrary, minimal diffs are often counter-intuitive, because they synch up anywhere possible, sometimes accidental matches 100 pages apart. Restricting synch points to contiguous matches preserves some notion of locality, at the occasional cost of producing a longer diff.

The *Differ* class has this constructor:

class difflib.**Differ**(*linejunk=None*, *charjunk=None*)

Optional keyword parameters *linejunk* and *charjunk* are for filter functions (or `None`):

linejunk: A function that accepts a single string argument, and returns true if the string is junk. The default is `None`, meaning that no line is considered junk.

charjunk: A function that accepts a single character argument (a string of length 1), and returns true if the character is junk. The default is `None`, meaning that no character is considered junk.

These junk-filtering functions speed up matching to find differences and do not cause any differing lines or characters to be ignored. Read the description of the *find_longest_match()* method's *isjunk* parameter for an explanation.

Differ objects are used (deltas generated) via a single method:

compare(*a*, *b*)

Compare two sequences of lines, and generate the delta (a sequence of lines).

Each sequence must contain individual single-line strings ending with newlines. Such sequences can be obtained from the *readlines()* method of file-like objects. The delta generated also consists of newline-terminated strings, ready to be printed as-is via the *writelines()* method of a file-like object.

6.3.4 Differ Example

This example compares two texts. First we set up the texts, sequences of individual single-line strings ending with newlines (such sequences can also be obtained from the *readlines()* method of file-like objects):

```
>>> text1 = '''  1. Beautiful is better than ugly.
...   2. Explicit is better than implicit.
...   3. Simple is better than complex.
...   4. Complex is better than complicated.
... '''.splitlines(keepends=True)
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = '''  1. Beautiful is better than ugly.
...   3.   Simple is better than complex.
...   4. Complicated is better than complex.
...   5. Flat is better than nested.
... '''.splitlines(keepends=True)
```

Next we instantiate a Differ object:

```
>>> d = Differ()
```

Note that when instantiating a *Differ* object we may pass functions to filter out line and character "junk." See the *Differ()* constructor for details.

Finally, we compare the two:

```
>>> result = list(d.compare(text1, text2))
```

`result` is a list of strings, so let's pretty-print it:

```
>>> from pprint import pprint
>>> pprint(result)
['    1. Beautiful is better than ugly.\n',
 '-   2. Explicit is better than implicit.\n',
 '-   3. Simple is better than complex.\n',
 '+   3.   Simple is better than complex.\n',
 '?     ++\n',
 '-   4. Complex is better than complicated.\n',
 '?            ^                     ---- ^\n',
 '+   4. Complicated is better than complex.\n',
 '?           ++++ ^                     ^\n',
 '+   5. Flat is better than nested.\n']
```

As a single multi-line string it looks like this:

```
>>> import sys
>>> sys.stdout.writelines(result)
    1. Beautiful is better than ugly.
-   2. Explicit is better than implicit.
-   3. Simple is better than complex.
+   3.   Simple is better than complex.
?     ++
-   4. Complex is better than complicated.
?           ^                     ---- ^
+   4. Complicated is better than complex.
?          ++++ ^                      ^
+   5. Flat is better than nested.
```

6.3.5 A command-line interface to difflib

This example shows how to use difflib to create a `diff`-like utility.

```
""" Command line interface to difflib.py providing diffs in four formats:

* ndiff:    lists every line and highlights interline changes.
* context:  highlights clusters of changes in a before/after format.
* unified:  highlights clusters of changes in an inline format.
* html:     generates side by side comparison with change highlights.

"""

import sys, os, difflib, argparse
from datetime import datetime, timezone

def file_mtime(path):
    t = datetime.fromtimestamp(os.stat(path).st_mtime,
                               timezone.utc)
    return t.astimezone().isoformat()

def main():

    parser = argparse.ArgumentParser()
    parser.add_argument('-c', action='store_true', default=False,
                        help='Produce a context format diff (default)')
    parser.add_argument('-u', action='store_true', default=False,
                        help='Produce a unified format diff')
    parser.add_argument('-m', action='store_true', default=False,
                        help='Produce HTML side by side diff '
                             '(can use -c and -l in conjunction)')
    parser.add_argument('-n', action='store_true', default=False,
                        help='Produce a ndiff format diff')
    parser.add_argument('-l', '--lines', type=int, default=3,
                        help='Set number of context lines (default 3)')
    parser.add_argument('fromfile')
    parser.add_argument('tofile')
    options = parser.parse_args()

    n = options.lines
    fromfile = options.fromfile
```

(continues on next page)

(continued from previous page)

```
    tofile = options.tofile

    fromdate = file_mtime(fromfile)
    todate = file_mtime(tofile)
    with open(fromfile) as ff:
        fromlines = ff.readlines()
    with open(tofile) as tf:
        tolines = tf.readlines()

    if options.u:
        diff = difflib.unified_diff(fromlines, tolines, fromfile, tofile, fromdate,␣
↪todate, n=n)
    elif options.n:
        diff = difflib.ndiff(fromlines, tolines)
    elif options.m:
        diff = difflib.HtmlDiff().make_file(fromlines,tolines,fromfile,tofile,
↪context=options.c,numlines=n)
    else:
        diff = difflib.context_diff(fromlines, tolines, fromfile, tofile, fromdate,␣
↪todate, n=n)

    sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()
```

6.3.6 ndiff example

This example shows how to use *difflib.ndiff()*.

```
"""ndiff [-q] file1 file2
    or
ndiff (-r1 | -r2) < ndiff_output > file1_or_file2

Print a human-friendly file difference report to stdout.  Both inter-
and intra-line differences are noted.  In the second form, recreate file1
(-r1) or file2 (-r2) on stdout, from an ndiff report on stdin.

In the first form, if -q ("quiet") is not specified, the first two lines
of output are

-: file1
+: file2

Each remaining line begins with a two-letter code:

    "- "    line unique to file1
    "+ "    line unique to file2
    "  "    line common to both files
    "? "    line not present in either input file

Lines beginning with "? " attempt to guide the eye to intraline
differences, and were not present in either input file.  These lines can be
confusing if the source files contain tab characters.
```

(continues on next page)

(continued from previous page)

```
The first file can be recovered by retaining only lines that begin with
"  " or "- ", and deleting those 2-character prefixes; use ndiff with -r1.

The second file can be recovered similarly, but by retaining only "  " and
"+ " lines; use ndiff with -r2; or, on Unix, the second file can be
recovered by piping the output through

    sed -n '/^[+ ] /s/^..//p'
"""

__version__ = 1, 7, 0

import difflib, sys

def fail(msg):
    out = sys.stderr.write
    out(msg + "\n\n")
    out(__doc__)
    return 0

# open a file & return the file object; gripe and return 0 if it
# couldn't be opened
def fopen(fname):
    try:
        return open(fname)
    except IOError as detail:
        return fail("couldn't open " + fname + ": " + str(detail))

# open two files & spray the diff to stdout; return false iff a problem
def fcompare(f1name, f2name):
    f1 = fopen(f1name)
    f2 = fopen(f2name)
    if not f1 or not f2:
        return 0

    a = f1.readlines(); f1.close()
    b = f2.readlines(); f2.close()
    for line in difflib.ndiff(a, b):
        print(line, end=' ')

    return 1

# crack args (sys.argv[1:] is normal) & compare;
# return false iff a problem

def main(args):
    import getopt
    try:
        opts, args = getopt.getopt(args, "qr:")
    except getopt.error as detail:
        return fail(str(detail))
    noisy = 1
    qseen = rseen = 0
    for opt, val in opts:
        if opt == "-q":
            qseen = 1
            noisy = 0
```

(continues on next page)

(continued from previous page)

```
        elif opt == "-r":
            rseen = 1
            whichfile = val
    if qseen and rseen:
        return fail("can't specify both -q and -r")
    if rseen:
        if args:
            return fail("no args allowed with -r option")
        if whichfile in ("1", "2"):
            restore(whichfile)
            return 1
        return fail("-r value must be 1 or 2")
    if len(args) != 2:
        return fail("need 2 filename args")
    f1name, f2name = args
    if noisy:
        print('-:', f1name)
        print('+:', f2name)
    return fcompare(f1name, f2name)

# read ndiff output from stdin, and print file1 (which=='1') or
# file2 (which=='2') to stdout

def restore(which):
    restored = difflib.restore(sys.stdin.readlines(), which)
    sys.stdout.writelines(restored)

if __name__ == '__main__':
    main(sys.argv[1:])
```

6.4 `textwrap` — Text wrapping and filling

Source code: Lib/textwrap.py

The *textwrap* module provides some convenience functions, as well as *TextWrapper*, the class that does all the work. If you're just wrapping or filling one or two text strings, the convenience functions should be good enough; otherwise, you should use an instance of *TextWrapper* for efficiency.

`textwrap.`**`wrap`**(*text*, *width=70*, ***, *initial_indent=''*, *subsequent_indent=''*, *expand_tabs=True*, *replace_whitespace=True*, *fix_sentence_endings=False*, *break_long_words=True*, *drop_whitespace=True*, *break_on_hyphens=True*, *tabsize=8*, *max_lines=None*, *placeholder=' [...]'*)

Wraps the single paragraph in *text* (a string) so every line is at most *width* characters long. Returns a list of output lines, without final newlines.

Optional keyword arguments correspond to the instance attributes of *TextWrapper*, documented below.

See the *TextWrapper.wrap()* method for additional details on how *wrap()* behaves.

`textwrap.`**`fill`**(*text*, *width=70*, ***, *initial_indent=''*, *subsequent_indent=''*, *expand_tabs=True*, *replace_whitespace=True*, *fix_sentence_endings=False*, *break_long_words=True*, *drop_whitespace=True*, *break_on_hyphens=True*, *tabsize=8*, *max_lines=None*, *placeholder=' [...]'*)

Wraps the single paragraph in *text*, and returns a single string containing the wrapped paragraph. `fill()` is shorthand for

```
"\n".join(wrap(text, ...))
```

In particular, `fill()` accepts exactly the same keyword arguments as `wrap()`.

textwrap.**shorten**(*text*, *width*, *, *fix_sentence_endings=False*, *break_long_words=True*, *break_on_hyphens=True*, *placeholder='[...]'*)

Collapse and truncate the given *text* to fit in the given *width*.

First the whitespace in *text* is collapsed (all whitespace is replaced by single spaces). If the result fits in the *width*, it is returned. Otherwise, enough words are dropped from the end so that the remaining words plus the *placeholder* fit within *width*:

```
>>> textwrap.shorten("Hello  world!", width=12)
'Hello world!'
>>> textwrap.shorten("Hello  world!", width=11)
'Hello [...]'
>>> textwrap.shorten("Hello world", width=10, placeholder="...")
'Hello...'
```

Optional keyword arguments correspond to the instance attributes of `TextWrapper`, documented below. Note that the whitespace is collapsed before the text is passed to the `TextWrapper` `fill()` function, so changing the value of `tabsize`, `expand_tabs`, `drop_whitespace`, and `replace_whitespace` will have no effect.

Added in version 3.4.

textwrap.**dedent**(*text*)

Remove any common leading whitespace from every line in *text*.

This can be used to make triple-quoted strings line up with the left edge of the display, while still presenting them in the source code in indented form.

Note that tabs and spaces are both treated as whitespace, but they are not equal: the lines `"  hello"` and `"\thello"` are considered to have no common leading whitespace.

Lines containing only whitespace are ignored in the input and normalized to a single newline character in the output.

For example:

```
def test():
    # end first line with \ to avoid the empty line!
    s = '''\
    hello
      world
    '''
    print(repr(s))          # prints '    hello\n      world\n    '
    print(repr(dedent(s)))  # prints 'hello\n  world\n'
```

textwrap.**indent**(*text*, *prefix*, *predicate=None*)

Add *prefix* to the beginning of selected lines in *text*.

Lines are separated by calling `text.splitlines(True)`.

By default, *prefix* is added to all lines that do not consist solely of whitespace (including any line endings).

For example:

```
>>> s = 'hello\n\n \nworld'
>>> indent(s, '  ')
'  hello\n\n \n  world'
```

The optional *predicate* argument can be used to control which lines are indented. For example, it is easy to add *prefix* to even empty and whitespace-only lines:

```
>>> print(indent(s, '+ ', lambda line: True))
+ hello
+
+
+ world
```

Added in version 3.3.

wrap(), *fill()* and *shorten()* work by creating a *TextWrapper* instance and calling a single method on it. That instance is not reused, so for applications that process many text strings using *wrap()* and/or *fill()*, it may be more efficient to create your own *TextWrapper* object.

Text is preferably wrapped on whitespaces and right after the hyphens in hyphenated words; only then will long words be broken if necessary, unless *TextWrapper.break_long_words* is set to false.

class textwrap.**TextWrapper**(***kwargs*)

The *TextWrapper* constructor accepts a number of optional keyword arguments. Each keyword argument corresponds to an instance attribute, so for example

```
wrapper = TextWrapper(initial_indent="* ")
```

is the same as

```
wrapper = TextWrapper()
wrapper.initial_indent = "* "
```

You can reuse the same *TextWrapper* object many times, and you can change any of its options through direct assignment to instance attributes between uses.

The *TextWrapper* instance attributes (and keyword arguments to the constructor) are as follows:

width

(default: `70`) The maximum length of wrapped lines. As long as there are no individual words in the input text longer than *width*, *TextWrapper* guarantees that no output line will be longer than *width* characters.

expand_tabs

(default: `True`) If true, then all tab characters in *text* will be expanded to spaces using the *expandtabs()* method of *text*.

tabsize

(default: `8`) If *expand_tabs* is true, then all tab characters in *text* will be expanded to zero or more spaces, depending on the current column and the given tab size.

Added in version 3.3.

replace_whitespace

(default: `True`) If true, after tab expansion but before wrapping, the *wrap()* method will replace each whitespace character with a single space. The whitespace characters replaced are as follows: tab, newline, vertical tab, formfeed, and carriage return (`'\t\n\v\f\r'`).

> **Note**
>
> If *expand_tabs* is false and *replace_whitespace* is true, each tab character will be replaced by a single space, which is *not* the same as tab expansion.

> **Note**
>
> If *replace_whitespace* is false, newlines may appear in the middle of a line and cause strange output. For this reason, text should be split into paragraphs (using *str.splitlines()* or similar) which are wrapped separately.

drop_whitespace

(default: `True`) If true, whitespace at the beginning and ending of every line (after wrapping but before indenting) is dropped. Whitespace at the beginning of the paragraph, however, is not dropped if non-whitespace follows it. If whitespace being dropped takes up an entire line, the whole line is dropped.

initial_indent

(default: `''`) String that will be prepended to the first line of wrapped output. Counts towards the length of the first line. The empty string is not indented.

subsequent_indent

(default: `''`) String that will be prepended to all lines of wrapped output except the first. Counts towards the length of each line except the first.

fix_sentence_endings

(default: `False`) If true, *TextWrapper* attempts to detect sentence endings and ensure that sentences are always separated by exactly two spaces. This is generally desired for text in a monospaced font. However, the sentence detection algorithm is imperfect: it assumes that a sentence ending consists of a lowercase letter followed by one of `'.'`, `'!'`, or `'?'`, possibly followed by one of `'"'` or `"'"`, followed by a space. One problem with this algorithm is that it is unable to detect the difference between “Dr.” in

```
[...] Dr. Frankenstein's monster [...]
```

and “Spot.” in

```
[...] See Spot. See Spot run [...]
```

fix_sentence_endings is false by default.

Since the sentence detection algorithm relies on `string.lowercase` for the definition of “lowercase letter”, and a convention of using two spaces after a period to separate sentences on the same line, it is specific to English-language texts.

break_long_words

(default: `True`) If true, then words longer than *width* will be broken in order to ensure that no lines are longer than *width*. If it is false, long words will not be broken, and some lines may be longer than *width*. (Long words will be put on a line by themselves, in order to minimize the amount by which *width* is exceeded.)

break_on_hyphens

(default: `True`) If true, wrapping will occur preferably on whitespaces and right after hyphens in compound words, as it is customary in English. If false, only whitespaces will be considered as potentially good places for line breaks, but you need to set *break_long_words* to false if you want truly insecable words. Default behaviour in previous versions was to always allow breaking hyphenated words.

max_lines

(default: `None`) If not `None`, then the output will contain at most *max_lines* lines, with *placeholder* appearing at the end of the output.

Added in version 3.4.

placeholder

(default: `' [...]'`) String that will appear at the end of the output text if it has been truncated.

Added in version 3.4.

TextWrapper also provides some public methods, analogous to the module-level convenience functions:

wrap(*text*)

Wraps the single paragraph in *text* (a string) so every line is at most *width* characters long. All wrapping options are taken from instance attributes of the *TextWrapper* instance. Returns a list of output lines, without final newlines. If the wrapped output has no content, the returned list is empty.

fill(*text*)

Wraps the single paragraph in *text*, and returns a single string containing the wrapped paragraph.

6.5 unicodedata — Unicode Database

This module provides access to the Unicode Character Database (UCD) which defines character properties for all Unicode characters. The data contained in this database is compiled from the UCD version 15.1.0.

The module uses the same names and symbols as defined by Unicode Standard Annex #44, "Unicode Character Database". It defines the following functions:

unicodedata.**lookup**(*name*)

Look up character by name. If a character with the given name is found, return the corresponding character. If not found, *KeyError* is raised.

Changed in version 3.3: Support for name aliases[1] and named sequences[2] has been added.

unicodedata.**name**(*chr*[, *default*])

Returns the name assigned to the character *chr* as a string. If no name is defined, *default* is returned, or, if not given, *ValueError* is raised.

unicodedata.**decimal**(*chr*[, *default*])

Returns the decimal value assigned to the character *chr* as integer. If no such value is defined, *default* is returned, or, if not given, *ValueError* is raised.

unicodedata.**digit**(*chr*[, *default*])

Returns the digit value assigned to the character *chr* as integer. If no such value is defined, *default* is returned, or, if not given, *ValueError* is raised.

unicodedata.**numeric**(*chr*[, *default*])

Returns the numeric value assigned to the character *chr* as float. If no such value is defined, *default* is returned, or, if not given, *ValueError* is raised.

[1] https://www.unicode.org/Public/15.1.0/ucd/NameAliases.txt
[2] https://www.unicode.org/Public/15.1.0/ucd/NamedSequences.txt

unicodedata.**category**(*chr*)

Returns the general category assigned to the character *chr* as string.

unicodedata.**bidirectional**(*chr*)

Returns the bidirectional class assigned to the character *chr* as string. If no such value is defined, an empty string is returned.

unicodedata.**combining**(*chr*)

Returns the canonical combining class assigned to the character *chr* as integer. Returns `0` if no combining class is defined.

unicodedata.**east_asian_width**(*chr*)

Returns the east asian width assigned to the character *chr* as string.

unicodedata.**mirrored**(*chr*)

Returns the mirrored property assigned to the character *chr* as integer. Returns `1` if the character has been identified as a "mirrored" character in bidirectional text, `0` otherwise.

unicodedata.**decomposition**(*chr*)

Returns the character decomposition mapping assigned to the character *chr* as string. An empty string is returned in case no such mapping is defined.

unicodedata.**normalize**(*form*, *unistr*)

Return the normal form *form* for the Unicode string *unistr*. Valid values for *form* are 'NFC', 'NFKC', 'NFD', and 'NFKD'.

The Unicode standard defines various normalization forms of a Unicode string, based on the definition of canonical equivalence and compatibility equivalence. In Unicode, several characters can be expressed in various way. For example, the character U+00C7 (LATIN CAPITAL LETTER C WITH CEDILLA) can also be expressed as the sequence U+0043 (LATIN CAPITAL LETTER C) U+0327 (COMBINING CEDILLA).

For each character, there are two normal forms: normal form C and normal form D. Normal form D (NFD) is also known as canonical decomposition, and translates each character into its decomposed form. Normal form C (NFC) first applies a canonical decomposition, then composes pre-combined characters again.

In addition to these two forms, there are two additional normal forms based on compatibility equivalence. In Unicode, certain characters are supported which normally would be unified with other characters. For example, U+2160 (ROMAN NUMERAL ONE) is really the same thing as U+0049 (LATIN CAPITAL LETTER I). However, it is supported in Unicode for compatibility with existing character sets (e.g. gb2312).

The normal form KD (NFKD) will apply the compatibility decomposition, i.e. replace all compatibility characters with their equivalents. The normal form KC (NFKC) first applies the compatibility decomposition, followed by the canonical composition.

Even if two unicode strings are normalized and look the same to a human reader, if one has combining characters and the other doesn't, they may not compare equal.

unicodedata.**is_normalized**(*form*, *unistr*)

Return whether the Unicode string *unistr* is in the normal form *form*. Valid values for *form* are 'NFC', 'NFKC', 'NFD', and 'NFKD'.

Added in version 3.8.

In addition, the module exposes the following constant:

unicodedata.**unidata_version**

The version of the Unicode database used in this module.

unicodedata.**ucd_3_2_0**

This is an object that has the same methods as the entire module, but uses the Unicode database version 3.2 instead, for applications that require this specific version of the Unicode database (such as IDNA).

Examples:

```
>>> import unicodedata
>>> unicodedata.lookup('LEFT CURLY BRACKET')
'{'
>>> unicodedata.name('/')
'SOLIDUS'
>>> unicodedata.decimal('9')
9
>>> unicodedata.decimal('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not a decimal
>>> unicodedata.category('A')  # 'L'etter, 'u'ppercase
'Lu'
>>> unicodedata.bidirectional('\u0660') # 'A'rabic, 'N'umber
'AN'
```

6.6 stringprep — Internet String Preparation

Source code: Lib/stringprep.py

When identifying things (such as host names) in the internet, it is often necessary to compare such identifications for "equality". Exactly how this comparison is executed may depend on the application domain, e.g. whether it should be case-insensitive or not. It may be also necessary to restrict the possible identifications, to allow only identifications consisting of "printable" characters.

RFC 3454 defines a procedure for "preparing" Unicode strings in internet protocols. Before passing strings onto the wire, they are processed with the preparation procedure, after which they have a certain normalized form. The RFC defines a set of tables, which can be combined into profiles. Each profile must define which tables it uses, and what other optional parts of the stringprep procedure are part of the profile. One example of a stringprep profile is nameprep, which is used for internationalized domain names.

The module *stringprep* only exposes the tables from RFC 3454. As these tables would be very large to represent as dictionaries or lists, the module uses the Unicode character database internally. The module source code itself was generated using the mkstringprep.py utility.

As a result, these tables are exposed as functions, not as data structures. There are two kinds of tables in the RFC: sets and mappings. For a set, *stringprep* provides the "characteristic function", i.e. a function that returns True if the parameter is part of the set. For mappings, it provides the mapping function: given the key, it returns the associated value. Below is a list of all functions available in the module.

stringprep.**in_table_a1**(*code*)

Determine whether *code* is in tableA.1 (Unassigned code points in Unicode 3.2).

stringprep.**in_table_b1**(*code*)

Determine whether *code* is in tableB.1 (Commonly mapped to nothing).

stringprep.**map_table_b2**(*code*)

Return the mapped value for *code* according to tableB.2 (Mapping for case-folding used with NFKC).

stringprep.**map_table_b3**(*code*)
Return the mapped value for *code* according to tableB.3 (Mapping for case-folding used with no normalization).

stringprep.**in_table_c11**(*code*)
Determine whether *code* is in tableC.1.1 (ASCII space characters).

stringprep.**in_table_c12**(*code*)
Determine whether *code* is in tableC.1.2 (Non-ASCII space characters).

stringprep.**in_table_c11_c12**(*code*)
Determine whether *code* is in tableC.1 (Space characters, union of C.1.1 and C.1.2).

stringprep.**in_table_c21**(*code*)
Determine whether *code* is in tableC.2.1 (ASCII control characters).

stringprep.**in_table_c22**(*code*)
Determine whether *code* is in tableC.2.2 (Non-ASCII control characters).

stringprep.**in_table_c21_c22**(*code*)
Determine whether *code* is in tableC.2 (Control characters, union of C.2.1 and C.2.2).

stringprep.**in_table_c3**(*code*)
Determine whether *code* is in tableC.3 (Private use).

stringprep.**in_table_c4**(*code*)
Determine whether *code* is in tableC.4 (Non-character code points).

stringprep.**in_table_c5**(*code*)
Determine whether *code* is in tableC.5 (Surrogate codes).

stringprep.**in_table_c6**(*code*)
Determine whether *code* is in tableC.6 (Inappropriate for plain text).

stringprep.**in_table_c7**(*code*)
Determine whether *code* is in tableC.7 (Inappropriate for canonical representation).

stringprep.**in_table_c8**(*code*)
Determine whether *code* is in tableC.8 (Change display properties or are deprecated).

stringprep.**in_table_c9**(*code*)
Determine whether *code* is in tableC.9 (Tagging characters).

stringprep.**in_table_d1**(*code*)
Determine whether *code* is in tableD.1 (Characters with bidirectional property “R” or “AL”).

stringprep.**in_table_d2**(*code*)
Determine whether *code* is in tableD.2 (Characters with bidirectional property “L”).

6.7 readline — GNU readline interface

The *readline* module defines a number of functions to facilitate completion and reading/writing of history files from the Python interpreter. This module can be used directly, or via the *rlcompleter* module, which supports completion of Python identifiers at the interactive prompt. Settings made using this module affect the behaviour of both the interpreter's interactive prompt and the prompts offered by the built-in *input()* function.

Readline keybindings may be configured via an initialization file, typically `.inputrc` in your home directory. See Readline Init File in the GNU Readline manual for information about the format and allowable constructs of that file, and the capabilities of the Readline library in general.

Availability: not WASI, not iOS.

This module does not work or is not available on WebAssembly platforms, or on iOS. See *WebAssembly platforms* for more information on WASM availability; see *iOS* for more information on iOS availability.

Note

The underlying Readline library API may be implemented by the `editline` (`libedit`) library instead of GNU readline. On macOS the *readline* module detects which library is being used at run time.

The configuration file for `editline` is different from that of GNU readline. If you programmatically load configuration strings you can use *backend* to determine which library is being used.

If you use `editline`/`libedit` readline emulation on macOS, the initialization file located in your home directory is named `.editrc`. For example, the following content in `~/.editrc` will turn ON *vi* keybindings and TAB completion:

```
python:bind -v
python:bind ^I rl_complete
```

Also note that different libraries may use different history file formats. When switching the underlying library, existing history files may become unusable.

readline.**backend**

The name of the underlying Readline library being used, either `"readline"` or `"editline"`.

Added in version 3.13.

6.7.1 Init file

The following functions relate to the init file and user configuration:

readline.**parse_and_bind**(*string*)

Execute the init line provided in the *string* argument. This calls `rl_parse_and_bind()` in the underlying library.

readline.**read_init_file**([*filename*])

Execute a readline initialization file. The default filename is the last filename used. This calls `rl_read_init_file()` in the underlying library.

6.7.2 Line buffer

The following functions operate on the line buffer:

readline.**get_line_buffer**()

Return the current contents of the line buffer (rl_line_buffer in the underlying library).

readline.**insert_text**(*string*)

Insert text into the line buffer at the cursor position. This calls rl_insert_text() in the underlying library, but ignores the return value.

readline.**redisplay**()

Change what's displayed on the screen to reflect the current contents of the line buffer. This calls rl_redisplay() in the underlying library.

6.7.3 History file

The following functions operate on a history file:

readline.**read_history_file**([*filename*])

Load a readline history file, and append it to the history list. The default filename is ~/.history. This calls read_history() in the underlying library.

readline.**write_history_file**([*filename*])

Save the history list to a readline history file, overwriting any existing file. The default filename is ~/.history. This calls write_history() in the underlying library.

readline.**append_history_file**(*nelements*[, *filename*])

Append the last *nelements* items of history to a file. The default filename is ~/.history. The file must already exist. This calls append_history() in the underlying library. This function only exists if Python was compiled for a version of the library that supports it.

Added in version 3.5.

readline.**get_history_length**()

readline.**set_history_length**(*length*)

Set or return the desired number of lines to save in the history file. The *write_history_file()* function uses this value to truncate the history file, by calling history_truncate_file() in the underlying library. Negative values imply unlimited history file size.

6.7.4 History list

The following functions operate on a global history list:

readline.**clear_history**()

Clear the current history. This calls clear_history() in the underlying library. The Python function only exists if Python was compiled for a version of the library that supports it.

readline.**get_current_history_length**()

Return the number of items currently in the history. (This is different from *get_history_length()*, which returns the maximum number of lines that will be written to a history file.)

readline.**get_history_item**(*index*)

Return the current contents of history item at *index*. The item index is one-based. This calls history_get() in the underlying library.

readline.**remove_history_item**(*pos*)

Remove history item specified by its position from the history. The position is zero-based. This calls `remove_history()` in the underlying library.

readline.**replace_history_item**(*pos*, *line*)

Replace history item specified by its position with *line*. The position is zero-based. This calls `replace_history_entry()` in the underlying library.

readline.**add_history**(*line*)

Append *line* to the history buffer, as if it was the last line typed. This calls `add_history()` in the underlying library.

readline.**set_auto_history**(*enabled*)

Enable or disable automatic calls to `add_history()` when reading input via readline. The *enabled* argument should be a Boolean value that when true, enables auto history, and that when false, disables auto history.

Added in version 3.6.

CPython implementation detail: Auto history is enabled by default, and changes to this do not persist across multiple sessions.

6.7.5 Startup hooks

readline.**set_startup_hook**([*function*])

Set or remove the function invoked by the `rl_startup_hook` callback of the underlying library. If *function* is specified, it will be used as the new hook function; if omitted or `None`, any function already installed is removed. The hook is called with no arguments just before readline prints the first prompt.

readline.**set_pre_input_hook**([*function*])

Set or remove the function invoked by the `rl_pre_input_hook` callback of the underlying library. If *function* is specified, it will be used as the new hook function; if omitted or `None`, any function already installed is removed. The hook is called with no arguments after the first prompt has been printed and just before readline starts reading input characters. This function only exists if Python was compiled for a version of the library that supports it.

6.7.6 Completion

The following functions relate to implementing a custom word completion function. This is typically operated by the Tab key, and can suggest and automatically complete a word being typed. By default, Readline is set up to be used by *rlcompleter* to complete Python identifiers for the interactive interpreter. If the *readline* module is to be used with a custom completer, a different set of word delimiters should be set.

readline.**set_completer**([*function*])

Set or remove the completer function. If *function* is specified, it will be used as the new completer function; if omitted or `None`, any completer function already installed is removed. The completer function is called as `function(text, state)`, for *state* in `0`, `1`, `2`, ..., until it returns a non-string value. It should return the next possible completion starting with *text*.

The installed completer function is invoked by the *entry_func* callback passed to `rl_completion_matches()` in the underlying library. The *text* string comes from the first parameter to the `rl_attempted_completion_function` callback of the underlying library.

readline.**get_completer**()

Get the completer function, or `None` if no completer function has been set.

readline.**get_completion_type**()

Get the type of completion being attempted. This returns the `rl_completion_type` variable in the underlying library as an integer.

readline.**get_begidx**()

readline.**get_endidx**()

Get the beginning or ending index of the completion scope. These indexes are the *start* and *end* arguments passed to the `rl_attempted_completion_function` callback of the underlying library. The values may be different in the same input editing scenario based on the underlying C readline implementation. Ex: libedit is known to behave differently than libreadline.

readline.**set_completer_delims**(*string*)

readline.**get_completer_delims**()

Set or get the word delimiters for completion. These determine the start of the word to be considered for completion (the completion scope). These functions access the `rl_completer_word_break_characters` variable in the underlying library.

readline.**set_completion_display_matches_hook**([*function*])

Set or remove the completion display function. If *function* is specified, it will be used as the new completion display function; if omitted or `None`, any completion display function already installed is removed. This sets or clears the `rl_completion_display_matches_hook` callback in the underlying library. The completion display function is called as `function(substitution, [matches], longest_match_length)` once each time matches need to be displayed.

6.7.7 Example

The following example demonstrates how to use the *readline* module's history reading and writing functions to automatically load and save a history file named `.python_history` from the user's home directory. The code below would normally be executed automatically during interactive sessions from the user's `PYTHONSTARTUP` file.

```
import atexit
import os
import readline

histfile = os.path.join(os.path.expanduser("~"), ".python_history")
try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may grow unruly
    readline.set_history_length(1000)
except FileNotFoundError:
    pass

atexit.register(readline.write_history_file, histfile)
```

This code is actually automatically run when Python is run in interactive mode (see *Readline configuration*).

The following example achieves the same goal but supports concurrent interactive sessions, by only appending the new history.

```
import atexit
import os
import readline
histfile = os.path.join(os.path.expanduser("~"), ".python_history")

try:
```

(continues on next page)

(continued from previous page)

```
    readline.read_history_file(histfile)
    h_len = readline.get_current_history_length()
except FileNotFoundError:
    open(histfile, 'wb').close()
    h_len = 0

def save(prev_h_len, histfile):
    new_h_len = readline.get_current_history_length()
    readline.set_history_length(1000)
    readline.append_history_file(new_h_len - prev_h_len, histfile)
atexit.register(save, h_len, histfile)
```

The following example extends the *code.InteractiveConsole* class to support history save/restore.

```
import atexit
import code
import os
import readline

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/.console-history")):
        code.InteractiveConsole.__init__(self, locals, filename)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except FileNotFoundError:
                pass
            atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.set_history_length(1000)
        readline.write_history_file(histfile)
```

6.8 `rlcompleter` — Completion function for GNU readline

Source code: Lib/rlcompleter.py

The `rlcompleter` module defines a completion function suitable to be passed to *set_completer()* in the *readline* module.

When this module is imported on a Unix platform with the *readline* module available, an instance of the *Completer* class is automatically created and its *complete()* method is set as the *readline completer*. The method provides completion of valid Python identifiers and keywords.

Example:

```
>>> import rlcompleter
>>> import readline
```

(continues on next page)

(continued from previous page)

```
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
readline.__doc__          readline.get_line_buffer(  readline.read_init_file(
readline.__file__         readline.insert_text(      readline.set_completer(
readline.__name__         readline.parse_and_bind(
>>> readline.
```

The `rlcompleter` module is designed for use with Python's interactive mode. Unless Python is run with the `-S` option, the module is automatically imported and configured (see *Readline configuration*).

On platforms without *readline*, the *Completer* class defined by this module can still be used for custom purposes.

class rlcompleter.**Completer**

Completer objects have the following method:

complete(*text*, *state*)

Return the next possible completion for *text*.

When called by the *readline* module, this method is called successively with `state == 0, 1, 2, ...` until the method returns `None`.

If called for *text* that doesn't include a period character (`'.'`), it will complete from names currently defined in *__main__*, *builtins* and keywords (as defined by the *keyword* module).

If called for a dotted name, it will try to evaluate anything without obvious side-effects (functions will not be evaluated, but it can generate calls to `__getattr__()`) up to the last part, and find matches for the rest via the *dir()* function. Any exception raised during the evaluation of the expression is caught, silenced and *None* is returned.

CHAPTER

SEVEN

BINARY DATA SERVICES

The modules described in this chapter provide some basic services operations for manipulation of binary data. Other operations on binary data, specifically in relation to file formats and network protocols, are described in the relevant sections.

Some libraries described under *Text Processing Services* also work with either ASCII-compatible binary formats (for example, `re`) or all binary data (for example, `difflib`).

In addition, see the documentation for Python's built-in binary data types in *Binary Sequence Types — bytes, bytearray, memoryview*.

7.1 `struct` — Interpret bytes as packed binary data

Source code: Lib/struct.py

This module converts between Python values and C structs represented as Python `bytes` objects. Compact *format strings* describe the intended conversions to/from Python values. The module's functions and objects can be used for two largely distinct applications, data exchange with external sources (files or network connections), or data transfer between the Python application and the C layer.

> **Note**
>
> When no prefix character is given, native mode is the default. It packs or unpacks data based on the platform and compiler on which the Python interpreter was built. The result of packing a given C struct includes pad bytes which maintain proper alignment for the C types involved; similarly, alignment is taken into account when unpacking. In contrast, when communicating data between external sources, the programmer is responsible for defining byte ordering and padding between elements. See *Byte Order, Size, and Alignment* for details.

Several `struct` functions (and methods of `Struct`) take a *buffer* argument. This refers to objects that implement the bufferobjects and provide either a readable or read-writable buffer. The most common types used for that purpose are `bytes` and `bytearray`, but many other types that can be viewed as an array of bytes implement the buffer protocol, so that they can be read/filled without additional copying from a `bytes` object.

7.1.1 Functions and Exceptions

The module defines the following exception and functions:

exception struct.**error**

Exception raised on various occasions; argument is a string describing what is wrong.

struct.**pack**(*format*, *v1*, *v2*, ...)

Return a bytes object containing the values *v1*, *v2*, … packed according to the format string *format*. The arguments must match the values required by the format exactly.

struct.**pack_into**(*format*, *buffer*, *offset*, *v1*, *v2*, ...)

Pack the values *v1*, *v2*, … according to the format string *format* and write the packed bytes into the writable buffer *buffer* starting at position *offset*. Note that *offset* is a required argument.

struct.**unpack**(*format*, *buffer*)

Unpack from the buffer *buffer* (presumably packed by `pack(format, ...)`) according to the format string *format*. The result is a tuple even if it contains exactly one item. The buffer's size in bytes must match the size required by the format, as reflected by *calcsize()*.

struct.**unpack_from**(*format*, */*, *buffer*, *offset=0*)

Unpack from *buffer* starting at position *offset*, according to the format string *format*. The result is a tuple even if it contains exactly one item. The buffer's size in bytes, starting at position *offset*, must be at least the size required by the format, as reflected by *calcsize()*.

struct.**iter_unpack**(*format*, *buffer*)

Iteratively unpack from the buffer *buffer* according to the format string *format*. This function returns an iterator which will read equally sized chunks from the buffer until all its contents have been consumed. The buffer's size in bytes must be a multiple of the size required by the format, as reflected by *calcsize()*.

Each iteration yields a tuple as specified by the format string.

Added in version 3.4.

struct.**calcsize**(*format*)

Return the size of the struct (and hence of the bytes object produced by `pack(format, ...)`) corresponding to the format string *format*.

7.1.2 Format Strings

Format strings describe the data layout when packing and unpacking data. They are built up from *format characters*, which specify the type of data being packed/unpacked. In addition, special characters control the *byte order, size and alignment*. Each format string consists of an optional prefix character which describes the overall properties of the data and one or more format characters which describe the actual data values and padding.

Byte Order, Size, and Alignment

By default, C types are represented in the machine's native format and byte order, and properly aligned by skipping pad bytes if necessary (according to the rules used by the C compiler). This behavior is chosen so that the bytes of a packed struct correspond exactly to the memory layout of the corresponding C struct. Whether to use native byte ordering and padding or standard formats depends on the application.

Alternatively, the first character of the format string can be used to indicate the byte order, size and alignment of the packed data, according to the following table:

Character	Byte order	Size	Alignment
`@`	native	native	native
=	native	standard	none
<	little-endian	standard	none
>	big-endian	standard	none
!	network (= big-endian)	standard	none

If the first character is not one of these, `'@'` is assumed.

Note

The number 1023 (`0x3ff` in hexadecimal) has the following byte representations:

- `03 ff` in big-endian (>)
- `ff 03` in little-endian (<)

Python example:

```
>>> import struct
>>> struct.pack('>h', 1023)
b'\x03\xff'
>>> struct.pack('<h', 1023)
b'\xff\x03'
```

Native byte order is big-endian or little-endian, depending on the host system. For example, Intel x86, AMD64 (x86-64), and Apple M1 are little-endian; IBM z and many legacy architectures are big-endian. Use *`sys.byteorder`* to check the endianness of your system.

Native size and alignment are determined using the C compiler's `sizeof` expression. This is always combined with native byte order.

Standard size depends only on the format character; see the table in the *Format Characters* section.

Note the difference between `'@'` and `'='`: both use native byte order, but the size and alignment of the latter is standardized.

The form `'!'` represents the network byte order which is always big-endian as defined in IETF RFC 1700.

There is no way to indicate non-native byte order (force byte-swapping); use the appropriate choice of `'<'` or `'>'`.

Notes:

(1) Padding is only automatically added between successive structure members. No padding is added at the beginning or the end of the encoded struct.

(2) No padding is added when using non-native size and alignment, e.g. with '<', '>', '=', and '!'.

(3) To align the end of a structure to the alignment requirement of a particular type, end the format with the code for that type with a repeat count of zero. See *Examples*.

Format Characters

Format characters have the following meaning; the conversion between C and Python values should be obvious given their types. The 'Standard size' column refers to the size of the packed value in bytes when using standard size; that is, when the format string starts with one of `'<'`, `'>'`, `'!'` or `'='`. When using native size, the size of the packed value is platform-dependent.

Format	C Type	Python type	Standard size	Notes
`x`	pad byte	no value		(7)
`c`	`char`	bytes of length 1	1	
`b`	`signed char`	integer	1	(1), (2)
`B`	`unsigned char`	integer	1	(2)
`?`	`_Bool`	bool	1	(1)
`h`	`short`	integer	2	(2)
`H`	`unsigned short`	integer	2	(2)
`i`	`int`	integer	4	(2)
`I`	`unsigned int`	integer	4	(2)
`l`	`long`	integer	4	(2)
`L`	`unsigned long`	integer	4	(2)
`q`	`long long`	integer	8	(2)
`Q`	`unsigned long long`	integer	8	(2)
`n`	`ssize_t`	integer		(3)
`N`	`size_t`	integer		(3)
`e`	(6)	float	2	(4)
`f`	`float`	float	4	(4)
`d`	`double`	float	8	(4)
`s`	`char[]`	bytes		(9)
`p`	`char[]`	bytes		(8)
`P`	`void*`	integer		(5)

Changed in version 3.3: Added support for the `'n'` and `'N'` formats.

Changed in version 3.6: Added support for the `'e'` format.

Notes:

(1) The `'?'` conversion code corresponds to the `_Bool` type defined by C99. If this type is not available, it is simulated using a `char`. In standard mode, it is always represented by one byte.

(2) When attempting to pack a non-integer using any of the integer conversion codes, if the non-integer has a `__index__()` method then that method is called to convert the argument to an integer before packing.

Changed in version 3.2: Added use of the `__index__()` method for non-integers.

(3) The `'n'` and `'N'` conversion codes are only available for the native size (selected as the default or with the `'@'` byte order character). For the standard size, you can use whichever of the other integer formats fits your application.

(4) For the `'f'`, `'d'` and `'e'` conversion codes, the packed representation uses the IEEE 754 binary32, binary64 or binary16 format (for `'f'`, `'d'` or `'e'` respectively), regardless of the floating-point format used by the platform.

(5) The `'P'` format character is only available for the native byte ordering (selected as the default or with the `'@'` byte order character). The byte order character `'='` chooses to use little- or big-endian ordering based on the host system. The struct module does not interpret this as native ordering, so the `'P'` format is not available.

(6) The IEEE 754 binary16 "half precision" type was introduced in the 2008 revision of the IEEE 754 standard. It has a sign bit, a 5-bit exponent and 11-bit precision (with 10 bits explicitly stored), and can represent numbers between approximately `6.1e-05` and `6.5e+04` at full precision. This type is not widely supported by C compilers: on a

typical machine, an unsigned short can be used for storage, but not for math operations. See the Wikipedia page on the half-precision floating-point format for more information.

(7) When packing, `'x'` inserts one NUL byte.

(8) The `'p'` format character encodes a "Pascal string", meaning a short variable-length string stored in a *fixed number of bytes*, given by the count. The first byte stored is the length of the string, or 255, whichever is smaller. The bytes of the string follow. If the string passed in to *pack()* is too long (longer than the count minus 1), only the leading `count-1` bytes of the string are stored. If the string is shorter than `count-1`, it is padded with null bytes so that exactly count bytes in all are used. Note that for *unpack()*, the `'p'` format character consumes `count` bytes, but that the string returned can never contain more than 255 bytes.

(9) For the `'s'` format character, the count is interpreted as the length of the bytes, not a repeat count like for the other format characters; for example, `'10s'` means a single 10-byte string mapping to or from a single Python byte string, while `'10c'` means 10 separate one byte character elements (e.g., `cccccccccc`) mapping to or from ten different Python byte objects. (See *Examples* for a concrete demonstration of the difference.) If a count is not given, it defaults to 1. For packing, the string is truncated or padded with null bytes as appropriate to make it fit. For unpacking, the resulting bytes object always has exactly the specified number of bytes. As a special case, `'0s'` means a single, empty string (while `'0c'` means 0 characters).

A format character may be preceded by an integral repeat count. For example, the format string `'4h'` means exactly the same as `'hhhh'`.

Whitespace characters between formats are ignored; a count and its format must not contain whitespace though.

When packing a value `x` using one of the integer formats (`'b'`, `'B'`, `'h'`, `'H'`, `'i'`, `'I'`, `'l'`, `'L'`, `'q'`, `'Q'`), if `x` is outside the valid range for that format then *struct.error* is raised.

Changed in version 3.1: Previously, some of the integer formats wrapped out-of-range values and raised *DeprecationWarning* instead of *struct.error*.

For the `'?'` format character, the return value is either *True* or *False*. When packing, the truth value of the argument object is used. Either 0 or 1 in the native or standard bool representation will be packed, and any non-zero value will be `True` when unpacking.

Examples

Note

Native byte order examples (designated by the `'@'` format prefix or lack of any prefix character) may not match what the reader's machine produces as that depends on the platform and compiler.

Pack and unpack integers of three different sizes, using big endian ordering:

```
>>> from struct import *
>>> pack(">bhl", 1, 2, 3)
b'\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('>bhl', b'\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('>bhl')
7
```

Attempt to pack an integer which is too large for the defined field:

```
>>> pack(">h", 99999)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
struct.error: 'h' format requires -32768 <= number <= 32767
```

Demonstrate the difference between `'s'` and `'c'` format characters:

```
>>> pack("@ccc", b'1', b'2', b'3')
b'123'
>>> pack("@3s", b'123')
b'123'
```

Unpacked fields can be named by assigning them to variables or by wrapping the result in a named tuple:

```
>>> record = b'raymond   \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb', record)

>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school gradelevel')
>>> Student._make(unpack('<10sHHb', record))
Student(name=b'raymond   ', serialnum=4658, school=264, gradelevel=8)
```

The ordering of format characters may have an impact on size in native mode since padding is implicit. In standard mode, the user is responsible for inserting any desired padding. Note in the first `pack` call below that three NUL bytes were added after the packed `'#'` to align the following integer on a four-byte boundary. In this example, the output was produced on a little endian machine:

```
>>> pack('@ci', b'#', 0x12131415)
b'#\x00\x00\x00\x15\x14\x13\x12'
>>> pack('@ic', 0x12131415, b'#')
b'\x15\x14\x13\x12#'
>>> calcsize('@ci')
8
>>> calcsize('@ic')
5
```

The following format `'llh0l'` results in two pad bytes being added at the end, assuming the platform's longs are aligned on 4-byte boundaries:

```
>>> pack('@llh0l', 1, 2, 3)
b'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

See also

Module *array*
Packed binary storage of homogeneous data.

Module *json*
JSON encoder and decoder.

Module *pickle*
Python object serialization.

7.1.3 Applications

Two main applications for the *struct* module exist, data interchange between Python and C code within an application or another application compiled using the same compiler (*native formats*), and data interchange between applications using agreed upon data layout (*standard formats*). Generally speaking, the format strings constructed for these two domains are distinct.

Native Formats

When constructing format strings which mimic native layouts, the compiler and machine architecture determine byte ordering and padding. In such cases, the `@` format character should be used to specify native byte ordering and data sizes. Internal pad bytes are normally inserted automatically. It is possible that a zero-repeat format code will be needed at the end of a format string to round up to the correct byte boundary for proper alignment of consecutive chunks of data.

Consider these two simple examples (on a 64-bit, little-endian machine):

```
>>> calcsize('@lhl')
24
>>> calcsize('@llh')
18
```

Data is not padded to an 8-byte boundary at the end of the second format string without the use of extra padding. A zero-repeat format code solves that problem:

```
>>> calcsize('@llh0l')
24
```

The `'x'` format code can be used to specify the repeat, but for native formats it is better to use a zero-repeat format like `'0l'`.

By default, native byte ordering and alignment is used, but it is better to be explicit and use the `'@'` prefix character.

Standard Formats

When exchanging data beyond your process such as networking or storage, be precise. Specify the exact byte order, size, and alignment. Do not assume they match the native order of a particular machine. For example, network byte order is big-endian, while many popular CPUs are little-endian. By defining this explicitly, the user need not care about the specifics of the platform their code is running on. The first character should typically be < or > (or `!`). Padding is the responsibility of the programmer. The zero-repeat format character won't work. Instead, the user must explicitly add `'x'` pad bytes where needed. Revisiting the examples from the previous section, we have:

```
>>> calcsize('<qh6xq')
24
>>> pack('<qh6xq', 1, 2, 3) == pack('@lhl', 1, 2, 3)
True
>>> calcsize('@llh')
18
>>> pack('@llh', 1, 2, 3) == pack('<qqh', 1, 2, 3)
True
>>> calcsize('<qqh6x')
24
>>> calcsize('@llh0l')
24
>>> pack('@llh0l', 1, 2, 3) == pack('<qqh6x', 1, 2, 3)
True
```

The above results (executed on a 64-bit machine) aren't guaranteed to match when executed on different machines. For example, the examples below were executed on a 32-bit machine:

```
>>> calcsize('<qqh6x')
24
>>> calcsize('@llh0l')
12
>>> pack('@llh0l', 1, 2, 3) == pack('<qqh6x', 1, 2, 3)
False
```

7.1.4 Classes

The *struct* module also defines the following type:

class struct.**Struct**(*format*)

Return a new Struct object which writes and reads binary data according to the format string *format*. Creating a Struct object once and calling its methods is more efficient than calling module-level functions with the same format since the format string is only compiled once.

> **Note**
>
> The compiled versions of the most recent format strings passed to the module-level functions are cached, so programs that use only a few format strings needn't worry about reusing a single *Struct* instance.

Compiled Struct objects support the following methods and attributes:

pack(*v1*, *v2*, ...)

Identical to the *pack()* function, using the compiled format. (`len(result)` will equal *size*.)

pack_into(*buffer*, *offset*, *v1*, *v2*, ...)

Identical to the *pack_into()* function, using the compiled format.

unpack(*buffer*)

Identical to the *unpack()* function, using the compiled format. The buffer's size in bytes must equal *size*.

unpack_from(*buffer*, *offset=0*)

Identical to the *unpack_from()* function, using the compiled format. The buffer's size in bytes, starting at position *offset*, must be at least *size*.

iter_unpack(*buffer*)

Identical to the *iter_unpack()* function, using the compiled format. The buffer's size in bytes must be a multiple of *size*.

Added in version 3.4.

format

The format string used to construct this Struct object.

Changed in version 3.7: The format string type is now *str* instead of *bytes*.

size

The calculated size of the struct (and hence of the bytes object produced by the *pack()* method) corresponding to *format*.

Changed in version 3.13: The *repr()* of structs has changed. It is now:

```
>>> Struct('i')
Struct('i')
```

7.2 codecs — Codec registry and base classes

Source code: Lib/codecs.py

This module defines base classes for standard Python codecs (encoders and decoders) and provides access to the internal Python codec registry, which manages the codec and error handling lookup process. Most standard codecs are *text encodings*, which encode text to bytes (and decode bytes to text), but there are also codecs provided that encode text to text, and bytes to bytes. Custom codecs may encode and decode between arbitrary types, but some module features are restricted to be used specifically with *text encodings* or with codecs that encode to `bytes`.

The module defines the following functions for encoding and decoding with any codec:

`codecs.`**`encode`**(*obj*, *encoding='utf-8'*, *errors='strict'*)

Encodes *obj* using the codec registered for *encoding*.

Errors may be given to set the desired error handling scheme. The default error handler is `'strict'` meaning that encoding errors raise `ValueError` (or a more codec specific subclass, such as `UnicodeEncodeError`). Refer to *Codec Base Classes* for more information on codec error handling.

`codecs.`**`decode`**(*obj*, *encoding='utf-8'*, *errors='strict'*)

Decodes *obj* using the codec registered for *encoding*.

Errors may be given to set the desired error handling scheme. The default error handler is `'strict'` meaning that decoding errors raise `ValueError` (or a more codec specific subclass, such as `UnicodeDecodeError`). Refer to *Codec Base Classes* for more information on codec error handling.

The full details for each codec can also be looked up directly:

`codecs.`**`lookup`**(*encoding*)

Looks up the codec info in the Python codec registry and returns a `CodecInfo` object as defined below.

Encodings are first looked up in the registry's cache. If not found, the list of registered search functions is scanned. If no `CodecInfo` object is found, a `LookupError` is raised. Otherwise, the `CodecInfo` object is stored in the cache and returned to the caller.

`class` `codecs.`**`CodecInfo`**(*encode*, *decode*, *streamreader=None*, *streamwriter=None*, *incrementalencoder=None*, *incrementaldecoder=None*, *name=None*)

Codec details when looking up the codec registry. The constructor arguments are stored in attributes of the same name:

`name`

The name of the encoding.

`encode`

`decode`

The stateless encoding and decoding functions. These must be functions or methods which have the same interface as the `encode()` and `decode()` methods of Codec instances (see *Codec Interface*). The functions or methods are expected to work in a stateless mode.

`incrementalencoder`

incrementaldecoder

Incremental encoder and decoder classes or factory functions. These have to provide the interface defined by the base classes *IncrementalEncoder* and *IncrementalDecoder*, respectively. Incremental codecs can maintain state.

streamwriter

streamreader

Stream writer and reader classes or factory functions. These have to provide the interface defined by the base classes *StreamWriter* and *StreamReader*, respectively. Stream codecs can maintain state.

To simplify access to the various codec components, the module provides these additional functions which use *lookup()* for the codec lookup:

codecs.**getencoder**(*encoding*)

Look up the codec for the given encoding and return its encoder function.

Raises a *LookupError* in case the encoding cannot be found.

codecs.**getdecoder**(*encoding*)

Look up the codec for the given encoding and return its decoder function.

Raises a *LookupError* in case the encoding cannot be found.

codecs.**getincrementalencoder**(*encoding*)

Look up the codec for the given encoding and return its incremental encoder class or factory function.

Raises a *LookupError* in case the encoding cannot be found or the codec doesn't support an incremental encoder.

codecs.**getincrementaldecoder**(*encoding*)

Look up the codec for the given encoding and return its incremental decoder class or factory function.

Raises a *LookupError* in case the encoding cannot be found or the codec doesn't support an incremental decoder.

codecs.**getreader**(*encoding*)

Look up the codec for the given encoding and return its *StreamReader* class or factory function.

Raises a *LookupError* in case the encoding cannot be found.

codecs.**getwriter**(*encoding*)

Look up the codec for the given encoding and return its *StreamWriter* class or factory function.

Raises a *LookupError* in case the encoding cannot be found.

Custom codecs are made available by registering a suitable codec search function:

codecs.**register**(*search_function*)

Register a codec search function. Search functions are expected to take one argument, being the encoding name in all lower case letters with hyphens and spaces converted to underscores, and return a *CodecInfo* object. In case a search function cannot find a given encoding, it should return `None`.

Changed in version 3.9: Hyphens and spaces are converted to underscore.

codecs.**unregister**(*search_function*)

Unregister a codec search function and clear the registry's cache. If the search function is not registered, do nothing.

Added in version 3.10.

While the builtin *open()* and the associated *io* module are the recommended approach for working with encoded text files, this module provides additional utility functions and classes that allow the use of a wider range of codecs when working with binary files:

codecs.**open**(*filename*, *mode='r'*, *encoding=None*, *errors='strict'*, *buffering=-1*)

Open an encoded file using the given *mode* and return an instance of `StreamReaderWriter`, providing transparent encoding/decoding. The default file mode is `'r'`, meaning to open the file in read mode.

> **Note**
>
> If *encoding* is not `None`, then the underlying encoded files are always opened in binary mode. No automatic conversion of `'\n'` is done on reading and writing. The *mode* argument may be any binary mode acceptable to the built-in `open()` function; the `'b'` is automatically added.

encoding specifies the encoding which is to be used for the file. Any encoding that encodes to and decodes from bytes is allowed, and the data types supported by the file methods depend on the codec used.

errors may be given to define the error handling. It defaults to `'strict'` which causes a `ValueError` to be raised in case an encoding error occurs.

buffering has the same meaning as for the built-in `open()` function. It defaults to -1 which means that the default buffer size will be used.

Changed in version 3.11: The `'U'` mode has been removed.

codecs.**EncodedFile**(*file*, *data_encoding*, *file_encoding=None*, *errors='strict'*)

Return a `StreamRecoder` instance, a wrapped version of *file* which provides transparent transcoding. The original file is closed when the wrapped version is closed.

Data written to the wrapped file is decoded according to the given *data_encoding* and then written to the original file as bytes using *file_encoding*. Bytes read from the original file are decoded according to *file_encoding*, and the result is encoded using *data_encoding*.

If *file_encoding* is not given, it defaults to *data_encoding*.

errors may be given to define the error handling. It defaults to `'strict'`, which causes `ValueError` to be raised in case an encoding error occurs.

codecs.**iterencode**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)

Uses an incremental encoder to iteratively encode the input provided by *iterator*. This function is a *generator*. The *errors* argument (as well as any other keyword argument) is passed through to the incremental encoder.

This function requires that the codec accept text `str` objects to encode. Therefore it does not support bytes-to-bytes encoders such as `base64_codec`.

codecs.**iterdecode**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)

Uses an incremental decoder to iteratively decode the input provided by *iterator*. This function is a *generator*. The *errors* argument (as well as any other keyword argument) is passed through to the incremental decoder.

This function requires that the codec accept `bytes` objects to decode. Therefore it does not support text-to-text encoders such as `rot_13`, although `rot_13` may be used equivalently with `iterencode()`.

The module also provides the following constants which are useful for reading and writing to platform dependent files:

codecs.**BOM**

codecs.**BOM_BE**

codecs.**BOM_LE**

codecs.**BOM_UTF8**

codecs.**BOM_UTF16**

codecs.**BOM_UTF16_BE**

codecs.**BOM_UTF16_LE**

codecs.**BOM_UTF32**

codecs.**BOM_UTF32_BE**

codecs.**BOM_UTF32_LE**

These constants define various byte sequences, being Unicode byte order marks (BOMs) for several encodings. They are used in UTF-16 and UTF-32 data streams to indicate the byte order used, and in UTF-8 as a Unicode signature. *BOM_UTF16* is either *BOM_UTF16_BE* or *BOM_UTF16_LE* depending on the platform's native byte order, *BOM* is an alias for *BOM_UTF16*, *BOM_LE* for *BOM_UTF16_LE* and *BOM_BE* for *BOM_UTF16_BE*. The others represent the BOM in UTF-8 and UTF-32 encodings.

7.2.1 Codec Base Classes

The *codecs* module defines a set of base classes which define the interfaces for working with codec objects, and can also be used as the basis for custom codec implementations.

Each codec has to define four interfaces to make it usable as codec in Python: stateless encoder, stateless decoder, stream reader and stream writer. The stream reader and writers typically reuse the stateless encoder/decoder to implement the file protocols. Codec authors also need to define how the codec will handle encoding and decoding errors.

Error Handlers

To simplify and standardize error handling, codecs may implement different error handling schemes by accepting the *errors* string argument:

```
>>> 'German ß, ♬'.encode(encoding='ascii', errors='backslashreplace')
b'German \\xdf, \\u266c'
>>> 'German ß, ♬'.encode(encoding='ascii', errors='xmlcharrefreplace')
b'German &#223;, &#9836;'
```

The following error handlers can be used with all Python *Standard Encodings* codecs:

Value	Meaning
`'strict'`	Raise *UnicodeError* (or a subclass), this is the default. Implemented in *strict_errors()*.
`'ignore'`	Ignore the malformed data and continue without further notice. Implemented in *ignore_errors()*.
`'replace'`	Replace with a replacement marker. On encoding, use `?` (ASCII character). On decoding, use `�` (U+FFFD, the official REPLACEMENT CHARACTER). Implemented in *replace_errors()*.
`'backslashreplace'`	Replace with backslashed escape sequences. On encoding, use hexadecimal form of Unicode code point with formats `\xhh` `\uxxxx` `\Uxxxxxxxx`. On decoding, use hexadecimal form of byte value with format `\xhh`. Implemented in *backslashreplace_errors()*.
`'surrogateescape'`	On decoding, replace byte with individual surrogate code ranging from `U+DC80` to `U+DCFF`. This code will then be turned back into the same byte when the `'surrogateescape'` error handler is used when encoding the data. (See **PEP 383** for more.)

The following error handlers are only applicable to encoding (within *text encodings*):

Value	Meaning
`'xmlcharrefreplace'`	Replace with XML/HTML numeric character reference, which is a decimal form of Unicode code point with format `&#num;`. Implemented in *xmlcharrefreplace_errors()*.
`'namereplace'`	Replace with `\N{...}` escape sequences, what appears in the braces is the Name property from Unicode Character Database. Implemented in *namereplace_errors()*.

In addition, the following error handler is specific to the given codecs:

Value	Codecs	Meaning
`'surrogatepass'`	utf-8, utf-16, utf-32, utf-16-be, utf-16-le, utf-32-be, utf-32-le	Allow encoding and decoding surrogate code point (`U+D800` - `U+DFFF`) as normal code point. Otherwise these codecs treat the presence of surrogate code point in *str* as an error.

Added in version 3.1: The `'surrogateescape'` and `'surrogatepass'` error handlers.

Changed in version 3.4: The `'surrogatepass'` error handler now works with utf-16* and utf-32* codecs.

Added in version 3.5: The `'namereplace'` error handler.

Changed in version 3.5: The `'backslashreplace'` error handler now works with decoding and translating.

The set of allowed values can be extended by registering a new named error handler:

`codecs.`**`register_error`**(*name*, *error_handler*)

Register the error handling function *error_handler* under the name *name*. The *error_handler* argument will be called during encoding and decoding in case of an error, when *name* is specified as the errors parameter.

For encoding, *error_handler* will be called with a *UnicodeEncodeError* instance, which contains information about the location of the error. The error handler must either raise this or a different exception, or return a tuple with a replacement for the unencodable part of the input and a position where encoding should continue. The replacement may be either *str* or *bytes*. If the replacement is bytes, the encoder will simply copy them into the output buffer. If the replacement is a string, the encoder will encode the replacement. Encoding continues on original input at the specified position. Negative position values will be treated as being relative to the end of the input string. If the resulting position is out of bound an *IndexError* will be raised.

Decoding and translating works similarly, except *UnicodeDecodeError* or *UnicodeTranslateError* will be passed to the handler and that the replacement from the error handler will be put into the output directly.

Previously registered error handlers (including the standard error handlers) can be looked up by name:

`codecs.`**`lookup_error`**(*name*)

Return the error handler previously registered under the name *name*.

Raises a *LookupError* in case the handler cannot be found.

The following standard error handlers are also made available as module level functions:

`codecs.`**`strict_errors`**(*exception*)

Implements the `'strict'` error handling.

Each encoding or decoding error raises a *UnicodeError*.

`codecs.`**`ignore_errors`**(*exception*)

Implements the `'ignore'` error handling.

Malformed data is ignored; encoding or decoding is continued without further notice.

codecs.**replace_errors**(*exception*)

Implements the `'replace'` error handling.

Substitutes `?` (ASCII character) for encoding errors or `\ufffd` (U+FFFD, the official REPLACEMENT CHARACTER) for decoding errors.

codecs.**backslashreplace_errors**(*exception*)

Implements the `'backslashreplace'` error handling.

Malformed data is replaced by a backslashed escape sequence. On encoding, use the hexadecimal form of Unicode code point with formats `\x`*hh* `\u`*xxxx* `\U`*xxxxxxxx*. On decoding, use the hexadecimal form of byte value with format `\x`*hh*.

Changed in version 3.5: Works with decoding and translating.

codecs.**xmlcharrefreplace_errors**(*exception*)

Implements the `'xmlcharrefreplace'` error handling (for encoding within *text encoding* only).

The unencodable character is replaced by an appropriate XML/HTML numeric character reference, which is a decimal form of Unicode code point with format `&#`*num*`;` .

codecs.**namereplace_errors**(*exception*)

Implements the `'namereplace'` error handling (for encoding within *text encoding* only).

The unencodable character is replaced by a `\N{...}` escape sequence. The set of characters that appear in the braces is the Name property from Unicode Character Database. For example, the German lowercase letter `'ß'` will be converted to byte sequence `\N{LATIN SMALL LETTER SHARP S}`.

Added in version 3.5.

Stateless Encoding and Decoding

The base `Codec` class defines these methods which also define the function interfaces of the stateless encoder and decoder:

class codecs.**Codec**

encode(*input*, *errors='strict'*)

Encodes the object *input* and returns a tuple (output object, length consumed). For instance, *text encoding* converts a string object to a bytes object using a particular character set encoding (e.g., `cp1252` or `iso-8859-1`).

The *errors* argument defines the error handling to apply. It defaults to `'strict'` handling.

The method may not store state in the `Codec` instance. Use `StreamWriter` for codecs which have to keep state in order to make encoding efficient.

The encoder must be able to handle zero length input and return an empty object of the output object type in this situation.

decode(*input*, *errors='strict'*)

Decodes the object *input* and returns a tuple (output object, length consumed). For instance, for a *text encoding*, decoding converts a bytes object encoded using a particular character set encoding to a string object.

For text encodings and bytes-to-bytes codecs, *input* must be a bytes object or one which provides the read-only buffer interface – for example, buffer objects and memory mapped files.

The *errors* argument defines the error handling to apply. It defaults to `'strict'` handling.

The method may not store state in the `Codec` instance. Use `StreamReader` for codecs which have to keep state in order to make decoding efficient.

The decoder must be able to handle zero length input and return an empty object of the output object type in this situation.

Incremental Encoding and Decoding

The `IncrementalEncoder` and `IncrementalDecoder` classes provide the basic interface for incremental encoding and decoding. Encoding/decoding the input isn't done with one call to the stateless encoder/decoder function, but with multiple calls to the `encode()`/`decode()` method of the incremental encoder/decoder. The incremental encoder/decoder keeps track of the encoding/decoding process during method calls.

The joined output of calls to the `encode()`/`decode()` method is the same as if all the single inputs were joined into one, and this input was encoded/decoded with the stateless encoder/decoder.

IncrementalEncoder Objects

The `IncrementalEncoder` class is used for encoding an input in multiple steps. It defines the following methods which every incremental encoder must define in order to be compatible with the Python codec registry.

class codecs.**IncrementalEncoder**(*errors='strict'*)

Constructor for an `IncrementalEncoder` instance.

All incremental encoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The `IncrementalEncoder` may implement different error handling schemes by providing the *errors* keyword argument. See *Error Handlers* for possible values.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `IncrementalEncoder` object.

encode(*object*, *final=False*)

Encodes *object* (taking the current state of the encoder into account) and returns the resulting encoded object. If this is the last call to `encode()` *final* must be true (the default is false).

reset()

Reset the encoder to the initial state. The output is discarded: call `.encode(object, final=True)`, passing an empty byte or text string if necessary, to reset the encoder and to get the output.

getstate()

Return the current state of the encoder which must be an integer. The implementation should make sure that `0` is the most common state. (States that are more complicated than integers can be converted into an integer by marshaling/pickling the state and encoding the bytes of the resulting string into an integer.)

setstate(*state*)

Set the state of the encoder to *state*. *state* must be an encoder state returned by `getstate()`.

IncrementalDecoder Objects

The *IncrementalDecoder* class is used for decoding an input in multiple steps. It defines the following methods which every incremental decoder must define in order to be compatible with the Python codec registry.

class codecs.**IncrementalDecoder**(*errors='strict'*)

Constructor for an *IncrementalDecoder* instance.

All incremental decoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *IncrementalDecoder* may implement different error handling schemes by providing the *errors* keyword argument. See *Error Handlers* for possible values.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the *IncrementalDecoder* object.

decode(*object*, *final=False*)

Decodes *object* (taking the current state of the decoder into account) and returns the resulting decoded object. If this is the last call to *decode()* *final* must be true (the default is false). If *final* is true the decoder must decode the input completely and must flush all buffers. If this isn't possible (e.g. because of incomplete byte sequences at the end of the input) it must initiate error handling just like in the stateless case (which might raise an exception).

reset()

Reset the decoder to the initial state.

getstate()

Return the current state of the decoder. This must be a tuple with two items, the first must be the buffer containing the still undecoded input. The second must be an integer and can be additional state info. (The implementation should make sure that `0` is the most common additional state info.) If this additional state info is `0` it must be possible to set the decoder to the state which has no input buffered and `0` as the additional state info, so that feeding the previously buffered input to the decoder returns it to the previous state without producing any output. (Additional state info that is more complicated than integers can be converted into an integer by marshaling/pickling the info and encoding the bytes of the resulting string into an integer.)

setstate(*state*)

Set the state of the decoder to *state*. *state* must be a decoder state returned by *getstate()*.

Stream Encoding and Decoding

The *StreamWriter* and *StreamReader* classes provide generic working interfaces which can be used to implement new encoding submodules very easily. See `encodings.utf_8` for an example of how this is done.

StreamWriter Objects

The *StreamWriter* class is a subclass of *Codec* and defines the following methods which every stream writer must define in order to be compatible with the Python codec registry.

class codecs.**StreamWriter**(*stream*, *errors='strict'*)

Constructor for a *StreamWriter* instance.

All stream writers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *stream* argument must be a file-like object open for writing text or binary data, as appropriate for the specific codec.

The `StreamWriter` may implement different error handling schemes by providing the *errors* keyword argument. See *Error Handlers* for the standard error handlers the underlying stream codec may support.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `StreamWriter` object.

write(*object*)

Writes the object's contents encoded to the stream.

writelines(*list*)

Writes the concatenated iterable of strings to the stream (possibly by reusing the `write()` method). Infinite or very large iterables are not supported. The standard bytes-to-bytes codecs do not support this method.

reset()

Resets the codec buffers used for keeping internal state.

Calling this method should ensure that the data on the output is put into a clean state that allows appending of new fresh data without having to rescan the whole stream to recover state.

In addition to the above methods, the `StreamWriter` must also inherit all other methods and attributes from the underlying stream.

StreamReader Objects

The `StreamReader` class is a subclass of `Codec` and defines the following methods which every stream reader must define in order to be compatible with the Python codec registry.

class codecs.**StreamReader**(*stream*, *errors='strict'*)

Constructor for a `StreamReader` instance.

All stream readers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *stream* argument must be a file-like object open for reading text or binary data, as appropriate for the specific codec.

The `StreamReader` may implement different error handling schemes by providing the *errors* keyword argument. See *Error Handlers* for the standard error handlers the underlying stream codec may support.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `StreamReader` object.

The set of allowed values for the *errors* argument can be extended with `register_error()`.

read(*size=-1*, *chars=-1*, *firstline=False*)

Decodes data from the stream and returns the resulting object.

The *chars* argument indicates the number of decoded code points or bytes to return. The `read()` method will never return more data than requested, but it might return less, if there is not enough available.

The *size* argument indicates the approximate maximum number of encoded bytes or code points to read for decoding. The decoder can modify this setting as appropriate. The default value -1 indicates to read and decode as much as possible. This parameter is intended to prevent having to decode huge files in one step.

The *firstline* flag indicates that it would be sufficient to only return the first line, if there are decoding errors on later lines.

The method should use a greedy read strategy meaning that it should read as much data as is allowed within the definition of the encoding and the given size, e.g. if optional encoding endings or state markers are available on the stream, these should be read too.

readline(*size=None*, *keepends=True*)

Read one line from the input stream and return the decoded data.

size, if given, is passed as size argument to the stream's `read()` method.

If *keepends* is false line-endings will be stripped from the lines returned.

readlines(*sizehint=None*, *keepends=True*)

Read all lines available on the input stream and return them as a list of lines.

Line-endings are implemented using the codec's `decode()` method and are included in the list entries if *keepends* is true.

sizehint, if given, is passed as the *size* argument to the stream's `read()` method.

reset()

Resets the codec buffers used for keeping internal state.

Note that no stream repositioning should take place. This method is primarily intended to be able to recover from decoding errors.

In addition to the above methods, the `StreamReader` must also inherit all other methods and attributes from the underlying stream.

StreamReaderWriter Objects

The `StreamReaderWriter` is a convenience class that allows wrapping streams which work in both read and write modes.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.

class codecs.**StreamReaderWriter**(*stream*, *Reader*, *Writer*, *errors='strict'*)

Creates a `StreamReaderWriter` instance. *stream* must be a file-like object. *Reader* and *Writer* must be factory functions or classes providing the `StreamReader` and `StreamWriter` interface resp. Error handling is done in the same way as defined for the stream readers and writers.

`StreamReaderWriter` instances define the combined interfaces of `StreamReader` and `StreamWriter` classes. They inherit all other methods and attributes from the underlying stream.

StreamRecoder Objects

The `StreamRecoder` translates data from one encoding to another, which is sometimes useful when dealing with different encoding environments.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.

class codecs.**StreamRecoder**(*stream*, *encode*, *decode*, *Reader*, *Writer*, *errors='strict'*)

Creates a `StreamRecoder` instance which implements a two-way conversion: *encode* and *decode* work on the frontend — the data visible to code calling `read()` and `write()`, while *Reader* and *Writer* work on the backend — the data in *stream*.

You can use these objects to do transparent transcodings, e.g., from Latin-1 to UTF-8 and back.

The *stream* argument must be a file-like object.

The *encode* and *decode* arguments must adhere to the `Codec` interface. *Reader* and *Writer* must be factory functions or classes providing objects of the `StreamReader` and `StreamWriter` interface respectively.

Error handling is done in the same way as defined for the stream readers and writers.

`StreamRecoder` instances define the combined interfaces of `StreamReader` and `StreamWriter` classes. They inherit all other methods and attributes from the underlying stream.

7.2.2 Encodings and Unicode

Strings are stored internally as sequences of code points in range `U+0000`–`U+10FFFF`. (See **PEP 393** for more details about the implementation.) Once a string object is used outside of CPU and memory, endianness and how these arrays are stored as bytes become an issue. As with other codecs, serialising a string into a sequence of bytes is known as *encoding*, and recreating the string from the sequence of bytes is known as *decoding*. There are a variety of different text serialisation codecs, which are collectivity referred to as *text encodings*.

The simplest text encoding (called `'latin-1'` or `'iso-8859-1'`) maps the code points 0–255 to the bytes `0x0`–`0xff`, which means that a string object that contains code points above `U+00FF` can't be encoded with this codec. Doing so will raise a `UnicodeEncodeError` that looks like the following (although the details of the error message may differ): `UnicodeEncodeError: 'latin-1' codec can't encode character '\u1234' in position 3: ordinal not in range(256).`

There's another group of encodings (the so called charmap encodings) that choose a different subset of all Unicode code points and how these code points are mapped to the bytes `0x0`–`0xff`. To see how this is done simply open e.g. `encodings/cp1252.py` (which is an encoding that is used primarily on Windows). There's a string constant with 256 characters that shows you which character is mapped to which byte value.

All of these encodings can only encode 256 of the 1114112 code points defined in Unicode. A simple and straightforward way that can store each Unicode code point, is to store each code point as four consecutive bytes. There are two possibilities: store the bytes in big endian or in little endian order. These two encodings are called `UTF-32-BE` and `UTF-32-LE` respectively. Their disadvantage is that if e.g. you use `UTF-32-BE` on a little endian machine you will always have to swap bytes on encoding and decoding. `UTF-32` avoids this problem: bytes will always be in natural endianness. When these bytes are read by a CPU with a different endianness, then bytes have to be swapped though. To be able to detect the endianness of a `UTF-16` or `UTF-32` byte sequence, there's the so called BOM ("Byte Order Mark"). This is the Unicode character `U+FEFF`. This character can be prepended to every `UTF-16` or `UTF-32` byte sequence. The byte swapped version of this character (`0xFFFE`) is an illegal character that may not appear in a Unicode text. So when the first character in a `UTF-16` or `UTF-32` byte sequence appears to be a `U+FFFE` the bytes have to be swapped on decoding. Unfortunately the character `U+FEFF` had a second purpose as a `ZERO WIDTH NO-BREAK SPACE`: a character that has no width and doesn't allow a word to be split. It can e.g. be used to give hints to a ligature algorithm. With Unicode 4.0 using `U+FEFF` as a `ZERO WIDTH NO-BREAK SPACE` has been deprecated (with `U+2060` (`WORD JOINER`) assuming this role). Nevertheless Unicode software still must be able to handle `U+FEFF` in both roles: as a BOM it's a device to determine the storage layout of the encoded bytes, and vanishes once the byte sequence has been decoded into a string; as a `ZERO WIDTH NO-BREAK SPACE` it's a normal character that will be decoded like any other.

There's another encoding that is able to encode the full range of Unicode characters: UTF-8. UTF-8 is an 8-bit encoding, which means there are no issues with byte order in UTF-8. Each byte in a UTF-8 byte sequence consists of two parts: marker bits (the most significant bits) and payload bits. The marker bits are a sequence of zero to four `1` bits followed by a `0` bit. Unicode characters are encoded like this (with x being payload bits, which when concatenated give the Unicode character):

Range	Encoding
U-00000000 ... U-0000007F	0xxxxxxx
U-00000080 ... U-000007FF	110xxxxx 10xxxxxx
U-00000800 ... U-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
U-00010000 ... U-0010FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

The least significant bit of the Unicode character is the rightmost x bit.

As UTF-8 is an 8-bit encoding no BOM is required and any U+FEFF character in the decoded string (even if it's the first character) is treated as a ZERO WIDTH NO-BREAK SPACE.

Without external information it's impossible to reliably determine which encoding was used for encoding a string. Each charmap encoding can decode any random byte sequence. However that's not possible with UTF-8, as UTF-8 byte sequences have a structure that doesn't allow arbitrary byte sequences. To increase the reliability with which a UTF-8 encoding can be detected, Microsoft invented a variant of UTF-8 (that Python calls "utf-8-sig") for its Notepad program: Before any of the Unicode characters is written to the file, a UTF-8 encoded BOM (which looks like this as a byte sequence: 0xef, 0xbb, 0xbf) is written. As it's rather improbable that any charmap encoded file starts with these byte values (which would e.g. map to

LATIN SMALL LETTER I WITH DIAERESIS
RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
INVERTED QUESTION MARK

in iso-8859-1), this increases the probability that a utf-8-sig encoding can be correctly guessed from the byte sequence. So here the BOM is not used to be able to determine the byte order used for generating the byte sequence, but as a signature that helps in guessing the encoding. On encoding the utf-8-sig codec will write 0xef, 0xbb, 0xbf as the first three bytes to the file. On decoding utf-8-sig will skip those three bytes if they appear as the first three bytes in the file. In UTF-8, the use of the BOM is discouraged and should generally be avoided.

7.2.3 Standard Encodings

Python comes with a number of codecs built-in, either implemented as C functions or with dictionaries as mapping tables. The following table lists the codecs by name, together with a few common aliases, and the languages for which the encoding is likely used. Neither the list of aliases nor the list of languages is meant to be exhaustive. Notice that spelling alternatives that only differ in case or use a hyphen instead of an underscore are also valid aliases; therefore, e.g. 'utf-8' is a valid alias for the 'utf_8' codec.

CPython implementation detail: Some common encodings can bypass the codecs lookup machinery to improve performance. These optimization opportunities are only recognized by CPython for a limited set of (case insensitive) aliases: utf-8, utf8, latin-1, latin1, iso-8859-1, iso8859-1, mbcs (Windows only), ascii, us-ascii, utf-16, utf16, utf-32, utf32, and the same using underscores instead of dashes. Using alternative aliases for these encodings may result in slower execution.

Changed in version 3.6: Optimization opportunity recognized for us-ascii.

Many of the character sets support the same languages. They vary in individual characters (e.g. whether the EURO SIGN is supported or not), and in the assignment of characters to code positions. For the European languages in particular, the following variants typically exist:

- an ISO 8859 codeset
- a Microsoft Windows code page, which is typically derived from an 8859 codeset, but replaces control characters with additional graphic characters
- an IBM EBCDIC code page
- an IBM PC code page, which is ASCII compatible

Codec	Aliases	Languages
ascii	646, us-ascii	English
big5	big5-tw, csbig5	Traditional Chinese
big5hkscs	big5-hkscs, hkscs	Traditional Chinese
cp037	IBM037, IBM039	English
cp273	273, IBM273, csIBM273	German Added in version 3.4.
cp424	EBCDIC-CP-HE, IBM424	Hebrew
cp437	437, IBM437	English
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	Western Europe
cp720		Arabic
cp737		Greek
cp775	IBM775	Baltic languages
cp850	850, IBM850	Western Europe
cp852	852, IBM852	Central and Eastern Europe
cp855	855, IBM855	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
cp856		Hebrew
cp857	857, IBM857	Turkish
cp858	858, IBM858	Western Europe
cp860	860, IBM860	Portuguese
cp861	861, CP-IS, IBM861	Icelandic
cp862	862, IBM862	Hebrew
cp863	863, IBM863	Canadian
cp864	IBM864	Arabic
cp865	865, IBM865	Danish, Norwegian
cp866	866, IBM866	Russian
cp869	869, CP-GR, IBM869	Greek
cp874		Thai
cp875		Greek
cp932	932, ms932, mskanji, ms-kanji, windows-31j	Japanese
cp949	949, ms949, uhc	Korean
cp950	950, ms950	Traditional Chinese
cp1006		Urdu
cp1026	ibm1026	Turkish
cp1125	1125, ibm1125, cp866u, ruscii	Ukrainian Added in version 3.4.
cp1140	ibm1140	Western Europe
cp1250	windows-1250	Central and Eastern Europe
cp1251	windows-1251	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
cp1252	windows-1252	Western Europe
cp1253	windows-1253	Greek
cp1254	windows-1254	Turkish
cp1255	windows-1255	Hebrew
cp1256	windows-1256	Arabic
cp1257	windows-1257	Baltic languages
cp1258	windows-1258	Vietnamese
euc_jp	eucjp, ujis, u-jis	Japanese
euc_jis_2004	jisx0213, eucjis2004	Japanese

continues on next page

Table 1 – continued from previous page

Codec	Aliases	Languages
euc_jisx0213	eucjisx0213	Japanese
euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	Korean
gb2312	chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58	Simplified Chinese
gbk	936, cp936, ms936	Unified Chinese
gb18030	gb18030-2000	Unified Chinese
hz	hzgb, hz-gb, hz-gb-2312	Simplified Chinese
iso2022_jp	csiso2022jp, iso2022jp, iso-2022-jp	Japanese
iso2022_jp_1	iso2022jp-1, iso-2022-jp-1	Japanese
iso2022_jp_2	iso2022jp-2, iso-2022-jp-2	Japanese, Korean, Simplified Chinese, Western Europe, Greek
iso2022_jp_2004	iso2022jp-2004, iso-2022-jp-2004	Japanese
iso2022_jp_3	iso2022jp-3, iso-2022-jp-3	Japanese
iso2022_jp_ext	iso2022jp-ext, iso-2022-jp-ext	Japanese
iso2022_kr	csiso2022kr, iso2022kr, iso-2022-kr	Korean
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	Western Europe
iso8859_2	iso-8859-2, latin2, L2	Central and Eastern Europe
iso8859_3	iso-8859-3, latin3, L3	Esperanto, Maltese
iso8859_4	iso-8859-4, latin4, L4	Baltic languages
iso8859_5	iso-8859-5, cyrillic	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
iso8859_6	iso-8859-6, arabic	Arabic
iso8859_7	iso-8859-7, greek, greek8	Greek
iso8859_8	iso-8859-8, hebrew	Hebrew
iso8859_9	iso-8859-9, latin5, L5	Turkish
iso8859_10	iso-8859-10, latin6, L6	Nordic languages
iso8859_11	iso-8859-11, thai	Thai languages
iso8859_13	iso-8859-13, latin7, L7	Baltic languages
iso8859_14	iso-8859-14, latin8, L8	Celtic languages
iso8859_15	iso-8859-15, latin9, L9	Western Europe
iso8859_16	iso-8859-16, latin10, L10	South-Eastern Europe
johab	cp1361, ms1361	Korean
koi8_r		Russian
koi8_t		Tajik Added in version 3.5.
koi8_u		Ukrainian
kz1048	kz_1048, strk1048_2002, rk1048	Kazakh Added in version 3.5.
mac_cyrillic	maccyrillic	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
mac_greek	macgreek	Greek
mac_iceland	maciceland	Icelandic
mac_latin2	maclatin2, maccentraleurope, mac_centeuro	Central and Eastern Europe

continues on next page

Table 1 – continued from previous page

Codec	Aliases	Languages
mac_roman	macroman, macintosh	Western Europe
mac_turkish	macturkish	Turkish
ptcp154	csptcp154, pt154, cp154, cyrillic-asian	Kazakh
shift_jis	csshiftjis, shiftjis, sjis, s_jis	Japanese
shift_jis_2004	shiftjis2004, sjis_2004, sjis2004	Japanese
shift_jisx0213	shiftjisx0213, sjisx0213, s_jisx0213	Japanese
utf_32	U32, utf32	all languages
utf_32_be	UTF-32BE	all languages
utf_32_le	UTF-32LE	all languages
utf_16	U16, utf16	all languages
utf_16_be	UTF-16BE	all languages
utf_16_le	UTF-16LE	all languages
utf_7	U7, unicode-1-1-utf-7	all languages
utf_8	U8, UTF, utf8, cp65001	all languages
utf_8_sig		all languages

Changed in version 3.4: The utf-16* and utf-32* encoders no longer allow surrogate code points (`U+D800`–`U+DFFF`) to be encoded. The utf-32* decoders no longer decode byte sequences that correspond to surrogate code points.

Changed in version 3.8: `cp65001` is now an alias to `utf_8`.

7.2.4 Python Specific Encodings

A number of predefined codecs are specific to Python, so their codec names have no meaning outside Python. These are listed in the tables below based on the expected input and output types (note that while text encodings are the most common use case for codecs, the underlying codec infrastructure supports arbitrary data transforms rather than just text encodings). For asymmetric codecs, the stated meaning describes the encoding direction.

Text Encodings

The following codecs provide *str* to *bytes* encoding and *bytes-like object* to *str* decoding, similar to the Unicode text encodings.

Codec	Aliases	Meaning
idna		Implement **RFC 3490**, see also *encodings.idna*. Only errors='strict' is supported.
mbcs	ansi, dbcs	Windows only: Encode the operand according to the ANSI codepage (CP_ACP).
oem		Windows only: Encode the operand according to the OEM codepage (CP_OEMCP). Added in version 3.6.
palmos		Encoding of PalmOS 3.5.
punycode		Implement **RFC 3492**. Stateful codecs are not supported.
raw_unicode_escape		Latin-1 encoding with \u*XXXX* and \U*XXXXXXXX* for other code points. Existing backslashes are not escaped in any way. It is used in the Python pickle protocol.
undefined		Raise an exception for all conversions, even empty strings. The error handler is ignored.
unicode_escape		Encoding suitable as the contents of a Unicode literal in ASCII-encoded Python source code, except that quotes are not escaped. Decode from Latin-1 source code. Beware that Python source code actually uses UTF-8 by default.

Changed in version 3.8: "unicode_internal" codec is removed.

Binary Transforms

The following codecs provide binary transforms: *bytes-like object* to *bytes* mappings. They are not supported by *bytes.decode()* (which only produces *str* output).

Codec	Aliases	Meaning	Encoder / decoder
base64_codec[1]	base64, base_64	Convert the operand to multiline MIME base64 (the result always includes a trailing `'\n'`). Changed in version 3.4: accepts any *bytes-like object* as input for encoding and decoding	*base64.encodebytes()* / *base64.decodebytes()*
bz2_codec	bz2	Compress the operand using bz2.	*bz2.compress()* / *bz2.decompress()*
hex_codec	hex	Convert the operand to hexadecimal representation, with two digits per byte.	*binascii.b2a_hex()* / *binascii.a2b_hex()*
quopri_codec	quopri, quotedprintable, quoted_printal	Convert the operand to MIME quoted printable.	*quopri.encode()* with `quotetabs=True` / *quopri.decode()*
uu_codec	uu	Convert the operand using uuencode.	
zlib_codec	zip, zlib	Compress the operand using gzip.	*zlib.compress()* / *zlib.decompress()*

Added in version 3.2: Restoration of the binary transforms.

Changed in version 3.4: Restoration of the aliases for the binary transforms.

Text Transforms

The following codec provides a text transform: a *str* to *str* mapping. It is not supported by *str.encode()* (which only produces *bytes* output).

Codec	Aliases	Meaning
rot_13	rot13	Return the Caesar-cypher encryption of the operand.

Added in version 3.2: Restoration of the `rot_13` text transform.

Changed in version 3.4: Restoration of the `rot13` alias.

7.2.5 `encodings.idna` — Internationalized Domain Names in Applications

This module implements **RFC 3490** (Internationalized Domain Names in Applications) and **RFC 3492** (Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)). It builds upon the `punycode` encoding and *stringprep*.

If you need the IDNA 2008 standard from **RFC 5891** and **RFC 5895**, use the third-party idna module.

These RFCs together define a protocol to support non-ASCII characters in domain names. A domain name containing non-ASCII characters (such as `www.Alliancefrançaise.nu`) is converted into an ASCII-compatible encoding (ACE, such as `www.xn--alliancefranaise-npb.nu`). The ACE form of the domain name is then used in all places where arbitrary characters are not allowed by the protocol, such as DNS queries, HTTP *Host* fields, and so on.

[1] In addition to *bytes-like objects*, `'base64_codec'` also accepts ASCII-only instances of *str* for decoding

This conversion is carried out in the application; if possible invisible to the user: The application should transparently convert Unicode domain labels to IDNA on the wire, and convert back ACE labels to Unicode before presenting them to the user.

Python supports this conversion in several ways: the `idna` codec performs conversion between Unicode and ACE, separating an input string into labels based on the separator characters defined in **section 3.1 of RFC 3490** and converting each label to ACE as required, and conversely separating an input byte string into labels based on the `.` separator and converting any ACE labels found into unicode. Furthermore, the *`socket`* module transparently converts Unicode host names to ACE, so that applications need not be concerned about converting host names themselves when they pass them to the socket module. On top of that, modules that have host names as function parameters, such as *`http.client`* and *`ftplib`*, accept Unicode host names (*`http.client`* then also transparently sends an IDNA hostname in the *`Host`* field if it sends that field at all).

When receiving host names from the wire (such as in reverse name lookup), no automatic conversion to Unicode is performed: applications wishing to present such host names to the user should decode them to Unicode.

The module *`encodings.idna`* also implements the nameprep procedure, which performs certain normalizations on host names, to achieve case-insensitivity of international domain names, and to unify similar characters. The nameprep functions can be used directly if desired.

`encodings.idna.`**`nameprep`**(*label*)

Return the nameprepped version of *label*. The implementation currently assumes query strings, so `AllowUnassigned` is true.

`encodings.idna.`**`ToASCII`**(*label*)

Convert a label to ASCII, as specified in **RFC 3490**. `UseSTD3ASCIIRules` is assumed to be false.

`encodings.idna.`**`ToUnicode`**(*label*)

Convert a label to Unicode, as specified in **RFC 3490**.

7.2.6 `encodings.mbcs` — Windows ANSI codepage

This module implements the ANSI codepage (CP_ACP).

Availability: Windows.

Changed in version 3.2: Before 3.2, the *errors* argument was ignored; `'replace'` was always used to encode, and `'ignore'` to decode.

Changed in version 3.3: Support any error handler.

7.2.7 `encodings.utf_8_sig` — UTF-8 codec with BOM signature

This module implements a variant of the UTF-8 codec. On encoding, a UTF-8 encoded BOM will be prepended to the UTF-8 encoded bytes. For the stateful encoder this is only done once (on the first write to the byte stream). On decoding, an optional UTF-8 encoded BOM at the start of the data will be skipped.

CHAPTER

EIGHT

DATA TYPES

The modules described in this chapter provide a variety of specialized data types such as dates and times, fixed-type arrays, heap queues, double-ended queues, and enumerations.

Python also provides some built-in data types, in particular, `dict`, `list`, `set` and `frozenset`, and `tuple`. The `str` class is used to hold Unicode strings, and the `bytes` and `bytearray` classes are used to hold binary data.

The following modules are documented in this chapter:

8.1 `datetime` — Basic date and time types

Source code: Lib/datetime.py

The `datetime` module supplies classes for manipulating dates and times.

While date and time arithmetic is supported, the focus of the implementation is on efficient attribute extraction for output formatting and manipulation.

Tip

Skip to *the format codes*.

See also

Module `calendar`
General calendar related functions.

Module `time`
Time access and conversions.

Module `zoneinfo`
Concrete time zones representing the IANA time zone database.

Package dateutil
Third-party library with expanded time zone and parsing support.

Package DateType
Third-party library that introduces distinct static types to e.g. allow *static type checkers* to differentiate between naive and aware datetimes.

8.1.1 Aware and Naive Objects

Date and time objects may be categorized as "aware" or "naive" depending on whether or not they include time zone information.

With sufficient knowledge of applicable algorithmic and political time adjustments, such as time zone and daylight saving time information, an **aware** object can locate itself relative to other aware objects. An aware object represents a specific moment in time that is not open to interpretation.[1]

A **naive** object does not contain enough information to unambiguously locate itself relative to other date/time objects. Whether a naive object represents Coordinated Universal Time (UTC), local time, or time in some other time zone is purely up to the program, just like it is up to the program whether a particular number represents metres, miles, or mass. Naive objects are easy to understand and to work with, at the cost of ignoring some aspects of reality.

For applications requiring aware objects, *datetime* and *time* objects have an optional time zone information attribute, `tzinfo`, that can be set to an instance of a subclass of the abstract *tzinfo* class. These *tzinfo* objects capture information about the offset from UTC time, the time zone name, and whether daylight saving time is in effect.

Only one concrete *tzinfo* class, the *timezone* class, is supplied by the `datetime` module. The `timezone` class can represent simple time zones with fixed offsets from UTC, such as UTC itself or North American EST and EDT time zones. Supporting time zones at deeper levels of detail is up to the application. The rules for time adjustment across the world are more political than rational, change frequently, and there is no standard suitable for every application aside from UTC.

8.1.2 Constants

The `datetime` module exports the following constants:

`datetime.`**`MINYEAR`**

The smallest year number allowed in a *date* or *datetime* object. *MINYEAR* is 1.

`datetime.`**`MAXYEAR`**

The largest year number allowed in a *date* or *datetime* object. *MAXYEAR* is 9999.

`datetime.`**`UTC`**

Alias for the UTC time zone singleton *datetime.timezone.utc*.

Added in version 3.11.

8.1.3 Available Types

class `datetime.`**`date`**

An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: *year*, *month*, and *day*.

class `datetime.`**`time`**

An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds. (There is no notion of "leap seconds" here.) Attributes: *hour*, *minute*, *second*, *microsecond*, and *tzinfo*.

class `datetime.`**`datetime`**

A combination of a date and a time. Attributes: *year*, *month*, *day*, *hour*, *minute*, *second*, *microsecond*, and *tzinfo*.

[1] If, that is, we ignore the effects of Relativity

class datetime.**timedelta**

A duration expressing the difference between two *datetime* or *date* instances to microsecond resolution.

class datetime.**tzinfo**

An abstract base class for time zone information objects. These are used by the *datetime* and *time* classes to provide a customizable notion of time adjustment (for example, to account for time zone and/or daylight saving time).

class datetime.**timezone**

A class that implements the *tzinfo* abstract base class as a fixed offset from the UTC.

Added in version 3.2.

Objects of these types are immutable.

Subclass relationships:

```
object
    timedelta
    tzinfo
        timezone
    time
    date
        datetime
```

Common Properties

The *date*, *datetime*, *time*, and *timezone* types share these common features:

- Objects of these types are immutable.
- Objects of these types are *hashable*, meaning that they can be used as dictionary keys.
- Objects of these types support efficient pickling via the *pickle* module.

Determining if an Object is Aware or Naive

Objects of the *date* type are always naive.

An object of type *time* or *datetime* may be aware or naive.

A *datetime* object *d* is aware if both of the following hold:

1. `d.tzinfo` is not `None`
2. `d.tzinfo.utcoffset(d)` does not return `None`

Otherwise, *d* is naive.

A *time* object *t* is aware if both of the following hold:

1. `t.tzinfo` is not `None`
2. `t.tzinfo.utcoffset(None)` does not return `None`.

Otherwise, *t* is naive.

The distinction between aware and naive doesn't apply to *timedelta* objects.

8.1.4 `timedelta` Objects

A *timedelta* object represents a duration, the difference between two *datetime* or *date* instances.

class datetime.**timedelta**(*days=0*, *seconds=0*, *microseconds=0*, *milliseconds=0*, *minutes=0*, *hours=0*, *weeks=0*)

All arguments are optional and default to 0. Arguments may be integers or floats, and may be positive or negative.

Only *days*, *seconds* and *microseconds* are stored internally. Arguments are converted to those units:

- A millisecond is converted to 1000 microseconds.
- A minute is converted to 60 seconds.
- An hour is converted to 3600 seconds.
- A week is converted to 7 days.

and days, seconds and microseconds are then normalized so that the representation is unique, with

- `0 <= microseconds < 1000000`
- `0 <= seconds < 3600*24` (the number of seconds in one day)
- `-999999999 <= days <= 999999999`

The following example illustrates how any arguments besides *days*, *seconds* and *microseconds* are "merged" and normalized into those three resulting attributes:

```
>>> from datetime import timedelta
>>> delta = timedelta(
...     days=50,
...     seconds=27,
...     microseconds=10,
...     milliseconds=29000,
...     minutes=5,
...     hours=8,
...     weeks=2
... )
>>> # Only days, seconds, and microseconds remain
>>> delta
datetime.timedelta(days=64, seconds=29156, microseconds=10)
```

If any argument is a float and there are fractional microseconds, the fractional microseconds left over from all arguments are combined and their sum is rounded to the nearest microsecond using round-half-to-even tiebreaker. If no argument is a float, the conversion and normalization processes are exact (no information is lost).

If the normalized value of days lies outside the indicated range, *OverflowError* is raised.

Note that normalization of negative values may be surprising at first. For example:

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

Class attributes:

timedelta.**min**

The most negative *timedelta* object, `timedelta(-999999999)`.

timedelta.**max**

The most positive *timedelta* object, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

timedelta.**resolution**

The smallest possible difference between non-equal *timedelta* objects, `timedelta(microseconds=1)`.

Note that, because of normalization, `timedelta.max` is greater than `-timedelta.min`. `-timedelta.max` is not representable as a *timedelta* object.

Instance attributes (read-only):

Attribute	Value
`days`	Between -999999999 and 999999999 inclusive
`seconds`	Between 0 and 86399 inclusive
`microseconds`	Between 0 and 999999 inclusive

Supported operations:

Operation	Result
`t1 = t2 + t3`	Sum of `t2` and `t3`. Afterwards `t1 - t2 == t3` and `t1 - t3 == t2` are true. (1)
`t1 = t2 - t3`	Difference of `t2` and `t3`. Afterwards `t1 == t2 - t3` and `t2 == t1 + t3` are true. (1)(6)
`t1 = t2 * i or t1 = i * t2`	Delta multiplied by an integer. Afterwards `t1 // i == t2` is true, provided `i != 0`.
	In general, `t1 * i == t1 * (i-1) + t1` is true. (1)
`t1 = t2 * f or t1 = f * t2`	Delta multiplied by a float. The result is rounded to the nearest multiple of timedelta.resolution using round-half-to-even.
`f = t2 / t3`	Division (3) of overall duration `t2` by interval unit `t3`. Returns a *float* object.
`t1 = t2 / f or t1 = t2 / i`	Delta divided by a float or an int. The result is rounded to the nearest multiple of timedelta.resolution using round-half-to-even.
`t1 = t2 // i` or `t1 = t2 // t3`	The floor is computed and the remainder (if any) is thrown away. In the second case, an integer is returned. (3)
`t1 = t2 % t3`	The remainder is computed as a *timedelta* object. (3)
`q, r = divmod(t1, t2)`	Computes the quotient and the remainder: `q = t1 // t2` (3) and `r = t1 % t2`. q is an integer and r is a *timedelta* object.
`+t1`	Returns a *timedelta* object with the same value. (2)
`-t1`	Equivalent to `timedelta(-t1.days, -t1.seconds*, -t1.microseconds)`, and to `t1 * -1`. (1)(4)
`abs(t)`	Equivalent to `+t` when `t.days >= 0`, and to `-t` when `t.days < 0`. (2)
`str(t)`	Returns a string in the form `[D day[s], ][H]H:MM:SS[.UUUUUU]`, where D is negative for negative `t`. (5)
`repr(t)`	Returns a string representation of the *timedelta* object as a constructor call with canonical attribute values.

Notes:

(1) This is exact but may overflow.

(2) This is exact and cannot overflow.

(3) Division by zero raises *ZeroDivisionError*.

(4) `-timedelta.max` is not representable as a *timedelta* object.

(5) String representations of *timedelta* objects are normalized similarly to their internal representation. This leads to somewhat unusual results for negative timedeltas. For example:

```
>>> timedelta(hours=-5)
datetime.timedelta(days=-1, seconds=68400)
>>> print(_)
-1 day, 19:00:00
```

(6) The expression `t2 - t3` will always be equal to the expression `t2 + (-t3)` except when t3 is equal to `timedelta.max`; in that case the former will produce a result while the latter will overflow.

In addition to the operations listed above, *timedelta* objects support certain additions and subtractions with *date* and *datetime* objects (see below).

Changed in version 3.2: Floor division and true division of a *timedelta* object by another *timedelta* object are now supported, as are remainder operations and the *divmod()* function. True division and multiplication of a *timedelta* object by a *float* object are now supported.

timedelta objects support equality and order comparisons.

In Boolean contexts, a *timedelta* object is considered to be true if and only if it isn't equal to `timedelta(0)`.

Instance methods:

`timedelta.`**`total_seconds`**`()`

Return the total number of seconds contained in the duration. Equivalent to `td / timedelta(seconds=1)`. For interval units other than seconds, use the division form directly (e.g. `td / timedelta(microseconds=1)`).

Note that for very large time intervals (greater than 270 years on most platforms) this method will lose microsecond accuracy.

Added in version 3.2.

Examples of usage: `timedelta`

An additional example of normalization:

```
>>> # Components of another_year add up to exactly 365 days
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                          minutes=50, seconds=600)
>>> year == another_year
True
>>> year.total_seconds()
31536000.0
```

Examples of *timedelta* arithmetic:

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> ten_years = 10 * year
>>> ten_years
datetime.timedelta(days=3650)
>>> ten_years.days // 365
10
```

(continues on next page)

(continued from previous page)

```
>>> nine_years = ten_years - year
>>> nine_years
datetime.timedelta(days=3285)
>>> three_years = nine_years // 3
>>> three_years, three_years.days // 365
(datetime.timedelta(days=1095), 3)
```

8.1.5 `date` Objects

A *date* object represents a date (year, month and day) in an idealized calendar, the current Gregorian calendar indefinitely extended in both directions.

January 1 of year 1 is called day number 1, January 2 of year 1 is called day number 2, and so on.[2]

class datetime.**date**(*year*, *month*, *day*)

All arguments are required. Arguments must be integers, in the following ranges:

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= day <= number of days in the given month and year`

If an argument outside those ranges is given, *ValueError* is raised.

Other constructors, all class methods:

classmethod date.**today**()

Return the current local date.

This is equivalent to `date.fromtimestamp(time.time())`.

classmethod date.**fromtimestamp**(*timestamp*)

Return the local date corresponding to the POSIX timestamp, such as is returned by *time.time()*.

This may raise *OverflowError*, if the timestamp is out of the range of values supported by the platform C `localtime()` function, and *OSError* on `localtime()` failure. It's common for this to be restricted to years from 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by *fromtimestamp()*.

Changed in version 3.3: Raise *OverflowError* instead of *ValueError* if the timestamp is out of the range of values supported by the platform C `localtime()` function. Raise *OSError* instead of *ValueError* on `localtime()` failure.

classmethod date.**fromordinal**(*ordinal*)

Return the date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1.

ValueError is raised unless `1 <= ordinal <= date.max.toordinal()`. For any date *d*, `date.fromordinal(d.toordinal()) == d`.

classmethod date.**fromisoformat**(*date_string*)

Return a *date* corresponding to a *date_string* given in any valid ISO 8601 format, with the following exceptions:

1. Reduced precision dates are not currently supported (`YYYY-MM`, `YYYY`).
2. Extended date representations are not currently supported (`±YYYYYY-MM-DD`).

[2] This matches the definition of the "proleptic Gregorian" calendar in Dershowitz and Reingold's book *Calendrical Calculations*, where it's the base calendar for all computations. See the book for algorithms for converting between proleptic Gregorian ordinals and many other calendar systems.

3. Ordinal dates are not currently supported (`YYYY-OOO`).

Examples:

```
>>> from datetime import date
>>> date.fromisoformat('2019-12-04')
datetime.date(2019, 12, 4)
>>> date.fromisoformat('20191204')
datetime.date(2019, 12, 4)
>>> date.fromisoformat('2021-W01-1')
datetime.date(2021, 1, 4)
```

Added in version 3.7.

Changed in version 3.11: Previously, this method only supported the format `YYYY-MM-DD`.

classmethod date.**fromisocalendar**(*year*, *week*, *day*)

Return a *date* corresponding to the ISO calendar date specified by year, week and day. This is the inverse of the function *date.isocalendar()*.

Added in version 3.8.

Class attributes:

date.**min**

The earliest representable date, `date(MINYEAR, 1, 1)`.

date.**max**

The latest representable date, `date(MAXYEAR, 12, 31)`.

date.**resolution**

The smallest possible difference between non-equal date objects, `timedelta(days=1)`.

Instance attributes (read-only):

date.**year**

Between *MINYEAR* and *MAXYEAR* inclusive.

date.**month**

Between 1 and 12 inclusive.

date.**day**

Between 1 and the number of days in the given month of the given year.

Supported operations:

Operation	Result
`date2 = date1 + timedelta`	`date2` will be `timedelta.days` days after `date1`. (1)
`date2 = date1 - timedelta`	Computes `date2` such that `date2 + timedelta == date1`. (2)
`timedelta = date1 - date2`	(3)
`date1 == date2` `date1 != date2`	Equality comparison. (4)

`date1 < date2`
`date1 > date2`
`date1 <= date2`
`date1 >= date2`
	Order comparison. (5)

Notes:

(1) *date2* is moved forward in time if `timedelta.days > 0`, or backward if `timedelta.days < 0`. Afterward `date2 - date1 == timedelta.days`. `timedelta.seconds` and `timedelta.microseconds` are ignored. *OverflowError* is raised if `date2.year` would be smaller than *MINYEAR* or larger than *MAXYEAR*.

(2) `timedelta.seconds` and `timedelta.microseconds` are ignored.

(3) This is exact, and cannot overflow. `timedelta.seconds` and `timedelta.microseconds` are 0, and `date2 + timedelta == date1` after.

(4) *date* objects are equal if they represent the same date.

`date` objects that are not also *datetime* instances are never equal to `datetime` objects, even if they represent the same date.

(5) *date1* is considered less than *date2* when *date1* precedes *date2* in time. In other words, `date1 < date2` if and only if `date1.toordinal() < date2.toordinal()`.

Order comparison between a `date` object that is not also a *datetime* instance and a `datetime` object raises *TypeError*.

Changed in version 3.13: Comparison between *datetime* object and an instance of the *date* subclass that is not a `datetime` subclass no longer converts the latter to `date`, ignoring the time part and the time zone. The default behavior can be changed by overriding the special comparison methods in subclasses.

In Boolean contexts, all *date* objects are considered to be true.

Instance methods:

`date.`**`replace`**(*year=self.year*, *month=self.month*, *day=self.day*)

Return a date with the same value, except for those parameters given new values by whichever keyword arguments are specified.

Example:

```
>>> from datetime import date
>>> d = date(2002, 12, 31)
>>> d.replace(day=26)
datetime.date(2002, 12, 26)
```

date objects are also supported by generic function *copy.replace()*.

date.**timetuple**()

Return a *time.struct_time* such as returned by *time.localtime()*.

The hours, minutes and seconds are 0, and the DST flag is -1.

`d.timetuple()` is equivalent to:

```
time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))
```

where `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` is the day number within the current year starting with 1 for January 1st.

date.**toordinal**()

Return the proleptic Gregorian ordinal of the date, where January 1 of year 1 has ordinal 1. For any *date* object *d*, `date.fromordinal(d.toordinal()) == d`.

date.**weekday**()

Return the day of the week as an integer, where Monday is 0 and Sunday is 6. For example, `date(2002, 12, 4).weekday() == 2`, a Wednesday. See also *isoweekday()*.

date.**isoweekday**()

Return the day of the week as an integer, where Monday is 1 and Sunday is 7. For example, `date(2002, 12, 4).isoweekday() == 3`, a Wednesday. See also *weekday()*, *isocalendar()*.

date.**isocalendar**()

Return a *named tuple* object with three components: `year`, `week` and `weekday`.

The ISO calendar is a widely used variant of the Gregorian calendar.[3]

The ISO year consists of 52 or 53 full weeks, and where a week starts on a Monday and ends on a Sunday. The first week of an ISO year is the first (Gregorian) calendar week of a year containing a Thursday. This is called week number 1, and the ISO year of that Thursday is the same as its Gregorian year.

For example, 2004 begins on a Thursday, so the first week of ISO year 2004 begins on Monday, 29 Dec 2003 and ends on Sunday, 4 Jan 2004:

```
>>> from datetime import date
>>> date(2003, 12, 29).isocalendar()
datetime.IsoCalendarDate(year=2004, week=1, weekday=1)
>>> date(2004, 1, 4).isocalendar()
datetime.IsoCalendarDate(year=2004, week=1, weekday=7)
```

Changed in version 3.9: Result changed from a tuple to a *named tuple*.

date.**isoformat**()

Return a string representing the date in ISO 8601 format, `YYYY-MM-DD`:

```
>>> from datetime import date
>>> date(2002, 12, 4).isoformat()
'2002-12-04'
```

[3] See R. H. van Gent's guide to the mathematics of the ISO 8601 calendar for a good explanation.

date.**__str__**()

For a date *d*, `str(d)` is equivalent to `d.isoformat()`.

date.**ctime**()

Return a string representing the date:

```
>>> from datetime import date
>>> date(2002, 12, 4).ctime()
'Wed Dec  4 00:00:00 2002'
```

`d.ctime()` is equivalent to:

```
time.ctime(time.mktime(d.timetuple()))
```

on platforms where the native C `ctime()` function (which *time.ctime()* invokes, but which *date.ctime()* does not invoke) conforms to the C standard.

date.**strftime**(*format*)

Return a string representing the date, controlled by an explicit format string. Format codes referring to hours, minutes or seconds will see 0 values. See also *strftime() and strptime() Behavior* and *date.isoformat()*.

date.**__format__**(*format*)

Same as *date.strftime()*. This makes it possible to specify a format string for a *date* object in formatted string literals and when using *str.format()*. See also *strftime() and strptime() Behavior* and *date.isoformat()*.

Examples of Usage: `date`

Example of counting days to an event:

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
...
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

More examples of working with *date*:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)

>>> # Methods related to formatting string output
>>> d.isoformat()
```

(continues on next page)

(continued from previous page)

```
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> d.ctime()
'Mon Mar 11 00:00:00 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'

>>> # Methods for to extracting 'components' under different calendars
>>> t = d.timetuple()
>>> for i in t:
...     print(i)
2002                # year
3                   # month
11                  # day
0
0
0
0                   # weekday (0 = Monday)
70                  # 70th day in the year
-1
>>> ic = d.isocalendar()
>>> for i in ic:
...     print(i)
2002                # ISO year
11                  # ISO week number
1                   # ISO day number ( 1 = Monday )

>>> # A date object is immutable; all operations produce a new object
>>> d.replace(year=2005)
datetime.date(2005, 3, 11)
```

8.1.6 `datetime` Objects

A *datetime* object is a single object containing all the information from a *date* object and a *time* object.

Like a *date* object, *datetime* assumes the current Gregorian calendar extended in both directions; like a *time* object, *datetime* assumes there are exactly 3600*24 seconds in every day.

Constructor:

class datetime.**datetime**(*year*, *month*, *day*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, ***, *fold=0*)

The *year*, *month* and *day* arguments are required. *tzinfo* may be `None`, or an instance of a *tzinfo* subclass. The remaining arguments must be integers in the following ranges:

- `MINYEAR <= year <= MAXYEAR`,
- `1 <= month <= 12`,
- `1 <= day <= number of days in the given month and year`,
- `0 <= hour < 24`,
- `0 <= minute < 60`,

- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold in [0, 1]`.

If an argument outside those ranges is given, *ValueError* is raised.

Changed in version 3.6: Added the *fold* parameter.

Other constructors, all class methods:

classmethod `datetime.`**today**`()`

Return the current local date and time, with *tzinfo* `None`.

Equivalent to:

```
datetime.fromtimestamp(time.time())
```

See also *now()*, *fromtimestamp()*.

This method is functionally equivalent to *now()*, but without a `tz` parameter.

classmethod `datetime.`**now**`(`*tz=None*`)`

Return the current local date and time.

If optional argument *tz* is `None` or not specified, this is like *today()*, but, if possible, supplies more precision than can be gotten from going through a *time.time()* timestamp (for example, this may be possible on platforms supplying the C `gettimeofday()` function).

If *tz* is not `None`, it must be an instance of a *tzinfo* subclass, and the current date and time are converted to *tz*'s time zone.

This function is preferred over *today()* and *utcnow()*.

classmethod `datetime.`**utcnow**`()`

Return the current UTC date and time, with *tzinfo* `None`.

This is like *now()*, but returns the current UTC date and time, as a naive *datetime* object. An aware current UTC datetime can be obtained by calling `datetime.now(timezone.utc)`. See also *now()*.

> **Warning**
>
> Because naive `datetime` objects are treated by many `datetime` methods as local times, it is preferred to use aware datetimes to represent times in UTC. As such, the recommended way to create an object representing the current time in UTC is by calling `datetime.now(timezone.utc)`.

Deprecated since version 3.12: Use *datetime.now()* with *UTC* instead.

classmethod `datetime.`**fromtimestamp**`(`*timestamp*, *tz=None*`)`

Return the local date and time corresponding to the POSIX timestamp, such as is returned by *time.time()*. If optional argument *tz* is `None` or not specified, the timestamp is converted to the platform's local date and time, and the returned *datetime* object is naive.

If *tz* is not `None`, it must be an instance of a *tzinfo* subclass, and the timestamp is converted to *tz*'s time zone.

fromtimestamp() may raise *OverflowError*, if the timestamp is out of the range of values supported by the platform C `localtime()` or `gmtime()` functions, and *OSError* on `localtime()` or `gmtime()` failure. It's common for this to be restricted to years in 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by *fromtimestamp()*, and then

it's possible to have two timestamps differing by a second that yield identical *datetime* objects. This method is preferred over *utcfromtimestamp()*.

Changed in version 3.3: Raise *OverflowError* instead of *ValueError* if the timestamp is out of the range of values supported by the platform C `localtime()` or `gmtime()` functions. Raise *OSError* instead of *ValueError* on `localtime()` or `gmtime()` failure.

Changed in version 3.6: *fromtimestamp()* may return instances with *fold* set to 1.

classmethod datetime.**utcfromtimestamp**(*timestamp*)

Return the UTC *datetime* corresponding to the POSIX timestamp, with *tzinfo* `None`. (The resulting object is naive.)

This may raise *OverflowError*, if the timestamp is out of the range of values supported by the platform C `gmtime()` function, and *OSError* on `gmtime()` failure. It's common for this to be restricted to years in 1970 through 2038.

To get an aware *datetime* object, call *fromtimestamp()*:

```
datetime.fromtimestamp(timestamp, timezone.utc)
```

On the POSIX compliant platforms, it is equivalent to the following expression:

```
datetime(1970, 1, 1, tzinfo=timezone.utc) + timedelta(seconds=timestamp)
```

except the latter formula always supports the full years range: between *MINYEAR* and *MAXYEAR* inclusive.

Warning

Because naive `datetime` objects are treated by many `datetime` methods as local times, it is preferred to use aware datetimes to represent times in UTC. As such, the recommended way to create an object representing a specific timestamp in UTC is by calling `datetime.fromtimestamp(timestamp, tz=timezone.utc)`.

Changed in version 3.3: Raise *OverflowError* instead of *ValueError* if the timestamp is out of the range of values supported by the platform C `gmtime()` function. Raise *OSError* instead of *ValueError* on `gmtime()` failure.

Deprecated since version 3.12: Use *datetime.fromtimestamp()* with *UTC* instead.

classmethod datetime.**fromordinal**(*ordinal*)

Return the *datetime* corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. *ValueError* is raised unless `1 <= ordinal <= datetime.max.toordinal()`. The hour, minute, second and microsecond of the result are all 0, and *tzinfo* is `None`.

classmethod datetime.**combine**(*date*, *time*, *tzinfo=time.tzinfo*)

Return a new *datetime* object whose date components are equal to the given *date* object's, and whose time components are equal to the given *time* object's. If the *tzinfo* argument is provided, its value is used to set the *tzinfo* attribute of the result, otherwise the *tzinfo* attribute of the *time* argument is used. If the *date* argument is a *datetime* object, its time components and *tzinfo* attributes are ignored.

For any *datetime* object *d*, `d == datetime.combine(d.date(), d.time(), d.tzinfo)`.

Changed in version 3.6: Added the *tzinfo* argument.

classmethod datetime.**fromisoformat**(*date_string*)

Return a *datetime* corresponding to a *date_string* in any valid ISO 8601 format, with the following exceptions:

1. Time zone offsets may have fractional seconds.

2. The `T` separator may be replaced by any single unicode character.
3. Fractional hours and minutes are not supported.
4. Reduced precision dates are not currently supported (`YYYY-MM`, `YYYY`).
5. Extended date representations are not currently supported (`±YYYYYY-MM-DD`).
6. Ordinal dates are not currently supported (`YYYY-OOO`).

Examples:

```
>>> from datetime import datetime
>>> datetime.fromisoformat('2011-11-04')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('20111104')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('2011-11-04T00:05:23')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-11-04T00:05:23Z')
datetime.datetime(2011, 11, 4, 0, 5, 23, tzinfo=datetime.timezone.utc)
>>> datetime.fromisoformat('20111104T000523')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-W01-2T00:05:23.283')
datetime.datetime(2011, 1, 4, 0, 5, 23, 283000)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283+00:00')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000, tzinfo=datetime.timezone.utc)
>>> datetime.fromisoformat('2011-11-04T00:05:23+04:00')
datetime.datetime(2011, 11, 4, 0, 5, 23,
    tzinfo=datetime.timezone(datetime.timedelta(seconds=14400)))
```

Added in version 3.7.

Changed in version 3.11: Previously, this method only supported formats that could be emitted by *date.isoformat()* or *datetime.isoformat()*.

classmethod datetime.**fromisocalendar**(*year*, *week*, *day*)

Return a *datetime* corresponding to the ISO calendar date specified by year, week and day. The non-date components of the datetime are populated with their normal default values. This is the inverse of the function *datetime.isocalendar()*.

Added in version 3.8.

classmethod datetime.**strptime**(*date_string*, *format*)

Return a *datetime* corresponding to *date_string*, parsed according to *format*.

If *format* does not contain microseconds or time zone information, this is equivalent to:

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

ValueError is raised if the date_string and format can't be parsed by *time.strptime()* or if it returns a value which isn't a time tuple. See also *strftime() and strptime() Behavior* and *datetime.fromisoformat()*.

Changed in version 3.13: If *format* specifies a day of month without a year a *DeprecationWarning* is now emitted. This is to avoid a quadrennial leap year bug in code seeking to parse only a month and day as the default year used in absence of one in the format is not a leap year. Such *format* values may raise an error as of Python 3.15. The workaround is to always include a year in your *format*. If parsing *date_string* values that do not have a year, explicitly add a year that is a leap year before parsing:

```
>>> from datetime import datetime
>>> date_string = "02/29"
>>> when = datetime.strptime(f"{date_string};1984", "%m/%d;%Y")  # Avoids leap␣
↪year bug.
>>> when.strftime("%B %d")
'February 29'
```

Class attributes:

`datetime.`**`min`**

The earliest representable *datetime*, `datetime(MINYEAR, 1, 1, tzinfo=None)`.

`datetime.`**`max`**

The latest representable *datetime*, `datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)`.

`datetime.`**`resolution`**

The smallest possible difference between non-equal *datetime* objects, `timedelta(microseconds=1)`.

Instance attributes (read-only):

`datetime.`**`year`**

Between *MINYEAR* and *MAXYEAR* inclusive.

`datetime.`**`month`**

Between 1 and 12 inclusive.

`datetime.`**`day`**

Between 1 and the number of days in the given month of the given year.

`datetime.`**`hour`**

In `range(24)`.

`datetime.`**`minute`**

In `range(60)`.

`datetime.`**`second`**

In `range(60)`.

`datetime.`**`microsecond`**

In `range(1000000)`.

`datetime.`**`tzinfo`**

The object passed as the *tzinfo* argument to the *datetime* constructor, or `None` if none was passed.

`datetime.`**`fold`**

In `[0, 1]`. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The values 0 and 1 represent, respectively, the earlier and later of the two moments with the same wall time representation.

Added in version 3.6.

Supported operations:

Operation	Result
`datetime2 = datetime1 + timedelta`	(1)
`datetime2 = datetime1 - timedelta`	(2)
`timedelta = datetime1 - datetime2`	(3)
`datetime1 == datetime2` `datetime1 != datetime2`	Equality comparison. (4)

`datetime1 < datetime2`
`datetime1 > datetime2`
`datetime1 <= datetime2`
`datetime1 >= datetime2`
	Order comparison. (5)

(1) `datetime2` is a duration of `timedelta` removed from `datetime1`, moving forward in time if `timedelta.days > 0`, or backward if `timedelta.days < 0`. The result has the same *tzinfo* attribute as the input datetime, and `datetime2 - datetime1 == timedelta` after. *OverflowError* is raised if `datetime2.year` would be smaller than *MINYEAR* or larger than *MAXYEAR*. Note that no time zone adjustments are done even if the input is an aware object.

(2) Computes the `datetime2` such that `datetime2 + timedelta == datetime1`. As for addition, the result has the same *tzinfo* attribute as the input datetime, and no time zone adjustments are done even if the input is aware.

(3) Subtraction of a *datetime* from a *datetime* is defined only if both operands are naive, or if both are aware. If one is aware and the other is naive, *TypeError* is raised.

If both are naive, or both are aware and have the same *tzinfo* attribute, the *tzinfo* attributes are ignored, and the result is a *timedelta* object *t* such that `datetime2 + t == datetime1`. No time zone adjustments are done in this case.

If both are aware and have different *tzinfo* attributes, `a-b` acts as if *a* and *b* were first converted to naive UTC datetimes. The result is `(a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())` except that the implementation never overflows.

(4) *datetime* objects are equal if they represent the same date and time, taking into account the time zone.

Naive and aware `datetime` objects are never equal.

If both comparands are aware, and have the same `tzinfo` attribute, the `tzinfo` and *fold* attributes are ignored and the base datetimes are compared. If both comparands are aware and have different *tzinfo* attributes, the comparison acts as comparands were first converted to UTC datetimes except that the implementation never overflows. `datetime` instances in a repeated interval are never equal to `datetime` instances in other time zone.

(5) *datetime1* is considered less than *datetime2* when *datetime1* precedes *datetime2* in time, taking into account the time zone.

Order comparison between naive and aware *datetime* objects raises *TypeError*.

If both comparands are aware, and have the same `tzinfo` attribute, the `tzinfo` and *fold* attributes are ignored and the base datetimes are compared. If both comparands are aware and have different *tzinfo* attributes, the comparison acts as comparands were first converted to UTC datetimes except that the implementation never overflows.

Changed in version 3.3: Equality comparisons between aware and naive *datetime* instances don't raise *TypeError*.

Changed in version 3.13: Comparison between *datetime* object and an instance of the *date* subclass that is not a `datetime` subclass no longer converts the latter to `date`, ignoring the time part and the time zone. The default behavior can be changed by overriding the special comparison methods in subclasses.

Instance methods:

`datetime.`**`date`**`()`

Return *date* object with same year, month and day.

`datetime.`**`time`**`()`

Return *time* object with same hour, minute, second, microsecond and fold. *tzinfo* is `None`. See also method *timetz()*.

Changed in version 3.6: The fold value is copied to the returned *time* object.

`datetime.`**`timetz`**`()`

Return *time* object with same hour, minute, second, microsecond, fold, and tzinfo attributes. See also method *time()*.

Changed in version 3.6: The fold value is copied to the returned *time* object.

`datetime.`**`replace`**`(`*year=self.year*, *month=self.month*, *day=self.day*, *hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, *, *fold=0*`)`

Return a datetime with the same attributes, except for those attributes given new values by whichever keyword arguments are specified. Note that `tzinfo=None` can be specified to create a naive datetime from an aware datetime with no conversion of date and time data.

datetime objects are also supported by generic function *copy.replace()*.

Changed in version 3.6: Added the *fold* parameter.

`datetime.`**`astimezone`**`(`*tz=None*`)`

Return a *datetime* object with new *tzinfo* attribute *tz*, adjusting the date and time data so the result is the same UTC time as *self*, but in *tz*'s local time.

If provided, *tz* must be an instance of a *tzinfo* subclass, and its *utcoffset()* and *dst()* methods must not return `None`. If *self* is naive, it is presumed to represent time in the system time zone.

If called without arguments (or with `tz=None`) the system local time zone is assumed for the target time zone. The `.tzinfo` attribute of the converted datetime instance will be set to an instance of *timezone* with the zone name and offset obtained from the OS.

If `self.tzinfo` is *tz*, `self.astimezone(tz)` is equal to *self*: no adjustment of date or time data is performed. Else the result is local time in the time zone *tz*, representing the same UTC time as *self*: after `astz = dt.astimezone(tz)`, `astz - astz.utcoffset()` will have the same date and time data as `dt - dt.utcoffset()`.

If you merely want to attach a *timezone* object *tz* to a datetime *dt* without adjustment of date and time data, use `dt.replace(tzinfo=tz)`. If you merely want to remove the `timezone` object from an aware datetime *dt* without conversion of date and time data, use `dt.replace(tzinfo=None)`.

Note that the default *tzinfo.fromutc()* method can be overridden in a *tzinfo* subclass to affect the result returned by *astimezone()*. Ignoring error cases, *astimezone()* acts like:

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new timezone object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

Changed in version 3.3: *tz* now can be omitted.

Changed in version 3.6: The *astimezone()* method can now be called on naive instances that are presumed to represent system local time.

datetime.**utcoffset**()

If *tzinfo* is None, returns None, else returns self.tzinfo.utcoffset(self), and raises an exception if the latter doesn't return None or a *timedelta* object with magnitude less than one day.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

datetime.**dst**()

If *tzinfo* is None, returns None, else returns self.tzinfo.dst(self), and raises an exception if the latter doesn't return None or a *timedelta* object with magnitude less than one day.

Changed in version 3.7: The DST offset is not restricted to a whole number of minutes.

datetime.**tzname**()

If *tzinfo* is None, returns None, else returns self.tzinfo.tzname(self), raises an exception if the latter doesn't return None or a string object,

datetime.**timetuple**()

Return a *time.struct_time* such as returned by *time.localtime()*.

d.timetuple() is equivalent to:

```
time.struct_time((d.year, d.month, d.day,
                  d.hour, d.minute, d.second,
                  d.weekday(), yday, dst))
```

where yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1 is the day number within the current year starting with 1 for January 1st. The *tm_isdst* flag of the result is set according to the *dst()* method: *tzinfo* is None or *dst()* returns None, tm_isdst is set to −1; else if *dst()* returns a non-zero value, tm_isdst is set to 1; else tm_isdst is set to 0.

datetime.**utctimetuple**()

If *datetime* instance *d* is naive, this is the same as d.timetuple() except that *tm_isdst* is forced to 0 regardless of what d.dst() returns. DST is never in effect for a UTC time.

If *d* is aware, *d* is normalized to UTC time, by subtracting d.utcoffset(), and a *time.struct_time* for the normalized time is returned. tm_isdst is forced to 0. Note that an *OverflowError* may be raised if d.year was MINYEAR or MAXYEAR and UTC adjustment spills over a year boundary.

Warning

Because naive datetime objects are treated by many datetime methods as local times, it is preferred to use aware datetimes to represent times in UTC; as a result, using *datetime.utctimetuple()* may give misleading results. If you have a naive datetime representing UTC, use datetime.replace(tzinfo=timezone.utc) to make it aware, at which point you can use *datetime.timetuple()*.

datetime.**toordinal**()

Return the proleptic Gregorian ordinal of the date. The same as self.date().toordinal().

datetime.**timestamp**()

Return POSIX timestamp corresponding to the *datetime* instance. The return value is a *float* similar to that returned by *time.time()*.

Naive *datetime* instances are assumed to represent local time and this method relies on the platform C `mktime()` function to perform the conversion. Since *datetime* supports wider range of values than `mktime()` on many platforms, this method may raise *OverflowError* or *OSError* for times far in the past or far in the future.

For aware *datetime* instances, the return value is computed as:

```
(dt - datetime(1970, 1, 1, tzinfo=timezone.utc)).total_seconds()
```

Added in version 3.3.

Changed in version 3.6: The *timestamp()* method uses the *fold* attribute to disambiguate the times during a repeated interval.

Note

There is no method to obtain the POSIX timestamp directly from a naive *datetime* instance representing UTC time. If your application uses this convention and your system time zone is not set to UTC, you can obtain the POSIX timestamp by supplying `tzinfo=timezone.utc`:

```
timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
```

or by calculating the timestamp directly:

```
timestamp = (dt - datetime(1970, 1, 1)) / timedelta(seconds=1)
```

`datetime.`**`weekday`**`()`

Return the day of the week as an integer, where Monday is 0 and Sunday is 6. The same as `self.date().weekday()`. See also *isoweekday()*.

`datetime.`**`isoweekday`**`()`

Return the day of the week as an integer, where Monday is 1 and Sunday is 7. The same as `self.date().isoweekday()`. See also *weekday()*, *isocalendar()*.

`datetime.`**`isocalendar`**`()`

Return a *named tuple* with three components: `year`, `week` and `weekday`. The same as `self.date().isocalendar()`.

`datetime.`**`isoformat`**`(`*sep='T'*, *timespec='auto'*`)`

Return a string representing the date and time in ISO 8601 format:

- `YYYY-MM-DDTHH:MM:SS.ffffff`, if *microsecond* is not 0
- `YYYY-MM-DDTHH:MM:SS`, if *microsecond* is 0

If *utcoffset()* does not return `None`, a string is appended, giving the UTC offset:

- `YYYY-MM-DDTHH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]`, if *microsecond* is not 0
- `YYYY-MM-DDTHH:MM:SS+HH:MM[:SS[.ffffff]]`, if *microsecond* is 0

Examples:

```
>>> from datetime import datetime, timezone
>>> datetime(2019, 5, 18, 15, 17, 8, 132263).isoformat()
'2019-05-18T15:17:08.132263'
>>> datetime(2019, 5, 18, 15, 17, tzinfo=timezone.utc).isoformat()
'2019-05-18T15:17:00+00:00'
```

The optional argument *sep* (default `'T'`) is a one-character separator, placed between the date and time portions of the result. For example:

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     """A time zone with an arbitrary, constant -06:39 offset."""
...     def utcoffset(self, dt):
...         return timedelta(hours=-6, minutes=-39)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
>>> datetime(2009, 11, 27, microsecond=100, tzinfo=TZ()).isoformat()
'2009-11-27T00:00:00.000100-06:39'
```

The optional argument *timespec* specifies the number of additional components of the time to include (the default is `'auto'`). It can be one of the following:

- `'auto'`: Same as `'seconds'` if *microsecond* is 0, same as `'microseconds'` otherwise.
- `'hours'`: Include the *hour* in the two-digit `HH` format.
- `'minutes'`: Include *hour* and *minute* in `HH:MM` format.
- `'seconds'`: Include *hour*, *minute*, and *second* in `HH:MM:SS` format.
- `'milliseconds'`: Include full time, but truncate fractional second part to milliseconds. `HH:MM:SS.sss` format.
- `'microseconds'`: Include full time in `HH:MM:SS.ffffff` format.

> **Note**
>
> Excluded time components are truncated, not rounded.

ValueError will be raised on an invalid *timespec* argument:

```
>>> from datetime import datetime
>>> datetime.now().isoformat(timespec='minutes')
'2002-12-25T00:00'
>>> dt = datetime(2015, 1, 1, 12, 30, 59, 0)
>>> dt.isoformat(timespec='microseconds')
'2015-01-01T12:30:59.000000'
```

Changed in version 3.6: Added the *timespec* parameter.

`datetime.__str__()`

For a *datetime* instance *d*, `str(d)` is equivalent to `d.isoformat(' ')`.

`datetime.ctime()`

Return a string representing the date and time:

```
>>> from datetime import datetime
>>> datetime(2002, 12, 4, 20, 30, 40).ctime()
'Wed Dec  4 20:30:40 2002'
```

The output string will *not* include time zone information, regardless of whether the input is aware or naive.

`d.ctime()` is equivalent to:

```
time.ctime(time.mktime(d.timetuple()))
```

on platforms where the native C `ctime()` function (which *time.ctime()* invokes, but which *datetime.ctime()* does not invoke) conforms to the C standard.

`datetime.`**`strftime`**(*format*)

Return a string representing the date and time, controlled by an explicit format string. See also *strftime() and strptime() Behavior* and *datetime.isoformat()*.

`datetime.`**`__format__`**(*format*)

Same as *datetime.strftime()*. This makes it possible to specify a format string for a *datetime* object in formatted string literals and when using *str.format()*. See also *strftime() and strptime() Behavior* and *datetime.isoformat()*.

Examples of Usage: `datetime`

Examples of working with *datetime* objects:

```
>>> from datetime import datetime, date, time, timezone

>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)

>>> # Using datetime.now()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)   # GMT +1
>>> datetime.now(timezone.utc)
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060, tzinfo=datetime.timezone.utc)

>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)

>>> # Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print(it)
...
2006    # year
11      # month
21      # day
16      # hour
30      # minute
0       # second
1       # weekday (0 = Monday)
325     # number of days since 1st January
-1      # dst - method tzinfo.dst() returned None

>>> # Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print(it)
```

(continues on next page)

(continued from previous page)

```
...
2006    # ISO year
47      # ISO week
2       # ISO weekday

>>> # Formatting a datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}.'.format(dt, "day",
↪"month", "time")
'The day is 21, the month is November, the time is 04:30PM.'
```

The example below defines a *tzinfo* subclass capturing time zone information for Kabul, Afghanistan, which used +4 UTC until 1945 and then +4:30 UTC thereafter:

```
from datetime import timedelta, datetime, tzinfo, timezone

class KabulTz(tzinfo):
    # Kabul used +4 until 1945, when they moved to +4:30
    UTC_MOVE_DATE = datetime(1944, 12, 31, 20, tzinfo=timezone.utc)

    def utcoffset(self, dt):
        if dt.year < 1945:
            return timedelta(hours=4)
        elif (1945, 1, 1, 0, 0) <= dt.timetuple()[:5] < (1945, 1, 1, 0, 30):
            # An ambiguous ("imaginary") half-hour range representing
            # a 'fold' in time due to the shift from +4 to +4:30.
            # If dt falls in the imaginary range, use fold to decide how
            # to resolve. See PEP495.
            return timedelta(hours=4, minutes=(30 if dt.fold else 0))
        else:
            return timedelta(hours=4, minutes=30)

    def fromutc(self, dt):
        # Follow same validations as in datetime.tzinfo
        if not isinstance(dt, datetime):
            raise TypeError("fromutc() requires a datetime argument")
        if dt.tzinfo is not self:
            raise ValueError("dt.tzinfo is not self")

        # A custom implementation is required for fromutc as
        # the input to this function is a datetime with utc values
        # but with a tzinfo set to self.
        # See datetime.astimezone or fromtimestamp.
        if dt.replace(tzinfo=timezone.utc) >= self.UTC_MOVE_DATE:
            return dt + timedelta(hours=4, minutes=30)
        else:
            return dt + timedelta(hours=4)

    def dst(self, dt):
        # Kabul does not observe daylight saving time.
        return timedelta(0)

    def tzname(self, dt):
        if dt >= self.UTC_MOVE_DATE:
            return "+04:30"
        return "+04"
```

Usage of `KabulTz` from above:

```
>>> tz1 = KabulTz()

>>> # Datetime before the change
>>> dt1 = datetime(1900, 11, 21, 16, 30, tzinfo=tz1)
>>> print(dt1.utcoffset())
4:00:00

>>> # Datetime after the change
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=tz1)
>>> print(dt2.utcoffset())
4:30:00

>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(timezone.utc)
>>> dt3
datetime.datetime(2006, 6, 14, 8, 30, tzinfo=datetime.timezone.utc)
>>> dt2
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=KabulTz())
>>> dt2 == dt3
True
```

8.1.7 `time` Objects

A *time* object represents a (local) time of day, independent of any particular day, and subject to adjustment via a *tzinfo* object.

class datetime.**time**(*hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *, *fold=0*)

All arguments are optional. *tzinfo* may be `None`, or an instance of a *tzinfo* subclass. The remaining arguments must be integers in the following ranges:

- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold in [0, 1]`.

If an argument outside those ranges is given, *ValueError* is raised. All default to 0 except *tzinfo*, which defaults to `None`.

Class attributes:

time.**min**

The earliest representable *time*, `time(0, 0, 0, 0)`.

time.**max**

The latest representable *time*, `time(23, 59, 59, 999999)`.

time.**resolution**

The smallest possible difference between non-equal *time* objects, `timedelta(microseconds=1)`, although note that arithmetic on *time* objects is not supported.

Instance attributes (read-only):

time.**hour**
In range(24).

time.**minute**
In range(60).

time.**second**
In range(60).

time.**microsecond**
In range(1000000).

time.**tzinfo**
The object passed as the tzinfo argument to the *time* constructor, or None if none was passed.

time.**fold**
In [0, 1]. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The values 0 and 1 represent, respectively, the earlier and later of the two moments with the same wall time representation.

Added in version 3.6.

time objects support equality and order comparisons, where *a* is considered less than *b* when *a* precedes *b* in time.

Naive and aware time objects are never equal. Order comparison between naive and aware time objects raises *TypeError*.

If both comparands are aware, and have the same *tzinfo* attribute, the tzinfo and fold attributes are ignored and the base times are compared. If both comparands are aware and have different tzinfo attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from self.utcoffset()).

Changed in version 3.3: Equality comparisons between aware and naive *time* instances don't raise *TypeError*.

In Boolean contexts, a *time* object is always considered to be true.

Changed in version 3.5: Before Python 3.5, a *time* object was considered to be false if it represented midnight in UTC. This behavior was considered obscure and error-prone and has been removed in Python 3.5. See bpo-13936 for full details.

Other constructor:

classmethod time.**fromisoformat**(*time_string*)
Return a *time* corresponding to a *time_string* in any valid ISO 8601 format, with the following exceptions:

1. Time zone offsets may have fractional seconds.
2. The leading T, normally required in cases where there may be ambiguity between a date and a time, is not required.
3. Fractional seconds may have any number of digits (anything beyond 6 will be truncated).
4. Fractional hours and minutes are not supported.

Examples:

```
>>> from datetime import time
>>> time.fromisoformat('04:23:01')
datetime.time(4, 23, 1)
>>> time.fromisoformat('T04:23:01')
datetime.time(4, 23, 1)
>>> time.fromisoformat('T042301')
```

(continues on next page)

(continued from previous page)

```
datetime.time(4, 23, 1)
>>> time.fromisoformat('04:23:01.000384')
datetime.time(4, 23, 1, 384)
>>> time.fromisoformat('04:23:01,000384')
datetime.time(4, 23, 1, 384)
>>> time.fromisoformat('04:23:01+04:00')
datetime.time(4, 23, 1, tzinfo=datetime.timezone(datetime.
↪timedelta(seconds=14400)))
>>> time.fromisoformat('04:23:01Z')
datetime.time(4, 23, 1, tzinfo=datetime.timezone.utc)
>>> time.fromisoformat('04:23:01+00:00')
datetime.time(4, 23, 1, tzinfo=datetime.timezone.utc)
```

Added in version 3.7.

Changed in version 3.11: Previously, this method only supported formats that could be emitted by *time.isoformat()*.

Instance methods:

time.**replace**(*hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, ***, *fold=0*)

Return a *time* with the same value, except for those attributes given new values by whichever keyword arguments are specified. Note that `tzinfo=None` can be specified to create a naive *time* from an aware *time*, without conversion of the time data.

time objects are also supported by generic function *copy.replace()*.

Changed in version 3.6: Added the *fold* parameter.

time.**isoformat**(*timespec='auto'*)

Return a string representing the time in ISO 8601 format, one of:

- `HH:MM:SS.ffffff`, if *microsecond* is not 0
- `HH:MM:SS`, if *microsecond* is 0
- `HH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]`, if *utcoffset()* does not return `None`
- `HH:MM:SS+HH:MM[:SS[.ffffff]]`, if *microsecond* is 0 and *utcoffset()* does not return `None`

The optional argument *timespec* specifies the number of additional components of the time to include (the default is `'auto'`). It can be one of the following:

- `'auto'`: Same as `'seconds'` if *microsecond* is 0, same as `'microseconds'` otherwise.
- `'hours'`: Include the *hour* in the two-digit `HH` format.
- `'minutes'`: Include *hour* and *minute* in `HH:MM` format.
- `'seconds'`: Include *hour*, *minute*, and *second* in `HH:MM:SS` format.
- `'milliseconds'`: Include full time, but truncate fractional second part to milliseconds. `HH:MM:SS.sss` format.
- `'microseconds'`: Include full time in `HH:MM:SS.ffffff` format.

> **Note**
>
> Excluded time components are truncated, not rounded.

ValueError will be raised on an invalid *timespec* argument.

Example:

```
>>> from datetime import time
>>> time(hour=12, minute=34, second=56, microsecond=123456).isoformat(timespec=
↪'minutes')
'12:34'
>>> dt = time(hour=12, minute=34, second=56, microsecond=0)
>>> dt.isoformat(timespec='microseconds')
'12:34:56.000000'
>>> dt.isoformat(timespec='auto')
'12:34:56'
```

Changed in version 3.6: Added the *timespec* parameter.

time.**__str__**()

For a time *t*, `str(t)` is equivalent to `t.isoformat()`.

time.**strftime**(*format*)

Return a string representing the time, controlled by an explicit format string. See also *strftime() and strptime() Behavior* and *time.isoformat()*.

time.**__format__**(*format*)

Same as *time.strftime()*. This makes it possible to specify a format string for a *time* object in formatted string literals and when using *str.format()*. See also *strftime() and strptime() Behavior* and *time.isoformat()*.

time.**utcoffset**()

If *tzinfo* is `None`, returns `None`, else returns `self.tzinfo.utcoffset(None)`, and raises an exception if the latter doesn't return `None` or a *timedelta* object with magnitude less than one day.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

time.**dst**()

If *tzinfo* is `None`, returns `None`, else returns `self.tzinfo.dst(None)`, and raises an exception if the latter doesn't return `None`, or a *timedelta* object with magnitude less than one day.

Changed in version 3.7: The DST offset is not restricted to a whole number of minutes.

time.**tzname**()

If *tzinfo* is `None`, returns `None`, else returns `self.tzinfo.tzname(None)`, or raises an exception if the latter doesn't return `None` or a string object.

Examples of Usage: `time`

Examples of working with a *time* object:

```
>>> from datetime import time, tzinfo, timedelta
>>> class TZ1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self,dt):
...         return "+01:00"
...     def  __repr__(self):
```

(continues on next page)

(continued from previous page)

```
...         return f"{self.__class__.__name__}()"
...
>>> t = time(12, 10, 30, tzinfo=TZ1())
>>> t
datetime.time(12, 10, 30, tzinfo=TZ1())
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'+01:00'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 +01:00'
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

8.1.8 `tzinfo` Objects

class datetime.**tzinfo**

This is an abstract base class, meaning that this class should not be instantiated directly. Define a subclass of *tzinfo* to capture information about a particular time zone.

An instance of (a concrete subclass of) *tzinfo* can be passed to the constructors for *datetime* and *time* objects. The latter objects view their attributes as being in local time, and the *tzinfo* object supports methods revealing offset of local time from UTC, the name of the time zone, and DST offset, all relative to a date or time object passed to them.

You need to derive a concrete subclass, and (at least) supply implementations of the standard *tzinfo* methods needed by the *datetime* methods you use. The `datetime` module provides *timezone*, a simple concrete subclass of *tzinfo* which can represent time zones with fixed offset from UTC such as UTC itself or North American EST and EDT.

Special requirement for pickling: A *tzinfo* subclass must have an `__init__()` method that can be called with no arguments, otherwise it can be pickled but possibly not unpickled again. This is a technical requirement that may be relaxed in the future.

A concrete subclass of *tzinfo* may need to implement the following methods. Exactly which methods are needed depends on the uses made of aware `datetime` objects. If in doubt, simply implement all of them.

tzinfo.**utcoffset**(*dt*)

Return offset of local time from UTC, as a *timedelta* object that is positive east of UTC. If local time is west of UTC, this should be negative.

This represents the *total* offset from UTC; for example, if a *tzinfo* object represents both time zone and DST adjustments, *utcoffset()* should return their sum. If the UTC offset isn't known, return `None`. Else the value returned must be a *timedelta* object strictly between `-timedelta(hours=24)` and `timedelta(hours=24)` (the magnitude of the offset must be less than one day). Most implementations of *utcoffset()* will probably look like one of these two:

```
return CONSTANT                 # fixed-offset class
return CONSTANT + self.dst(dt)  # daylight-aware class
```

If *utcoffset()* does not return `None`, *dst()* should not return `None` either.

The default implementation of *utcoffset()* raises *NotImplementedError*.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

`tzinfo.`**`dst`**`(`*dt*`)`

Return the daylight saving time (DST) adjustment, as a *timedelta* object or `None` if DST information isn't known.

Return `timedelta(0)` if DST is not in effect. If DST is in effect, return the offset as a *timedelta* object (see *utcoffset()* for details). Note that DST offset, if applicable, has already been added to the UTC offset returned by *utcoffset()*, so there's no need to consult *dst()* unless you're interested in obtaining DST info separately. For example, *datetime.timetuple()* calls its *tzinfo* attribute's *dst()* method to determine how the *tm_isdst* flag should be set, and *tzinfo.fromutc()* calls *dst()* to account for DST changes when crossing time zones.

An instance *tz* of a *tzinfo* subclass that models both standard and daylight times must be consistent in this sense:

`tz.utcoffset(dt) - tz.dst(dt)`

must return the same result for every *datetime* *dt* with `dt.tzinfo == tz`. For sane *tzinfo* subclasses, this expression yields the time zone's "standard offset", which should not depend on the date or the time, but only on geographic location. The implementation of *datetime.astimezone()* relies on this, but cannot detect violations; it's the programmer's responsibility to ensure it. If a *tzinfo* subclass cannot guarantee this, it may be able to override the default implementation of *tzinfo.fromutc()* to work correctly with *astimezone()* regardless.

Most implementations of *dst()* will probably look like one of these two:

```
def dst(self, dt):
    # a fixed-offset class:  doesn't account for DST
    return timedelta(0)
```

or:

```
def dst(self, dt):
    # Code to set dston and dstoff to the time zone's DST
    # transition times based on the input dt.year, and expressed
    # in standard local time.

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

The default implementation of *dst()* raises *NotImplementedError*.

Changed in version 3.7: The DST offset is not restricted to a whole number of minutes.

`tzinfo.`**`tzname`**`(`*dt*`)`

Return the time zone name corresponding to the *datetime* object *dt*, as a string. Nothing about string names is defined by the `datetime` module, and there's no requirement that it mean anything in particular. For example, `"GMT"`, `"UTC"`, `"-500"`, `"-5:00"`, `"EDT"`, `"US/Eastern"`, `"America/New York"` are all valid replies. Return `None` if a string name isn't known. Note that this is a method rather than a fixed string primarily because some *tzinfo* subclasses will wish to return different names depending on the specific value of *dt* passed, especially if the *tzinfo* class is accounting for daylight time.

The default implementation of *tzname()* raises *NotImplementedError*.

These methods are called by a *datetime* or *time* object, in response to their methods of the same names. A *datetime* object passes itself as the argument, and a *time* object passes `None` as the argument. A *tzinfo* subclass's methods should therefore be prepared to accept a *dt* argument of `None`, or of class *datetime*.

When `None` is passed, it's up to the class designer to decide the best response. For example, returning `None` is appropriate if the class wishes to say that time objects don't participate in the *tzinfo* protocols. It may be more useful for `utcoffset(None)` to return the standard UTC offset, as there is no other convention for discovering the standard offset.

When a *datetime* object is passed in response to a *datetime* method, `dt.tzinfo` is the same object as *self*. *tzinfo* methods can rely on this, unless user code calls *tzinfo* methods directly. The intent is that the *tzinfo* methods interpret *dt* as being in local time, and not need worry about objects in other time zones.

There is one more *tzinfo* method that a subclass may wish to override:

`tzinfo.`**`fromutc`**`(`*dt*`)`

This is called from the default *datetime.astimezone()* implementation. When called from that, `dt.tzinfo` is *self*, and *dt*'s date and time data are to be viewed as expressing a UTC time. The purpose of *fromutc()* is to adjust the date and time data, returning an equivalent datetime in *self*'s local time.

Most *tzinfo* subclasses should be able to inherit the default *fromutc()* implementation without problems. It's strong enough to handle fixed-offset time zones, and time zones accounting for both standard and daylight time, and the latter even if the DST transition times differ in different years. An example of a time zone the default *fromutc()* implementation may not handle correctly in all cases is one where the standard offset (from UTC) depends on the specific date and time passed, which can happen for political reasons. The default implementations of *astimezone()* and *fromutc()* may not produce the result you want if the result is one of the hours straddling the moment the standard offset changes.

Skipping code for error cases, the default *fromutc()* implementation acts like:

```
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst  # this is self's standard offset
    if delta:
        dt += delta   # convert to standard local time
        dtdst = dt.dst()
        # raise ValueError if dtdst is None
    if dtdst:
        return dt + dtdst
    else:
        return dt
```

In the following `tzinfo_examples.py` file there are some examples of *tzinfo* classes:

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
SECOND = timedelta(seconds=1)

# A class capturing the platform's idea of local time.
# (May result in wrong values on historical times in
#  timezones where UTC offset and/or the DST rules had
#  changed in the past.)
import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds = -_time.altzone)
```

(continues on next page)

(continued from previous page)

```
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

    def fromutc(self, dt):
        assert dt.tzinfo is self
        stamp = (dt - datetime(1970, 1, 1, tzinfo=self)) // SECOND
        args = _time.localtime(stamp)[:6]
        dst_diff = DSTDIFF // SECOND
        # Detect fold
        fold = (args == _time.localtime(stamp - dst_diff))
        return datetime(*args, microsecond=dt.microsecond,
                        tzinfo=self, fold=fold)

    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET

    def dst(self, dt):
        if self._isdst(dt):
            return DSTDIFF
        else:
            return ZERO

    def tzname(self, dt):
        return _time.tzname[self._isdst(dt)]

    def _isdst(self, dt):
        tt = (dt.year, dt.month, dt.day,
              dt.hour, dt.minute, dt.second,
              dt.weekday(), 0, 0)
        stamp = _time.mktime(tt)
        tt = _time.localtime(stamp)
        return tt.tm_isdst > 0

Local = LocalTimezone()


# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt


# US DST Rules
#
# This is a simplified (i.e., wrong for a few cases) set of rules for US
# DST start and end times. For a complete and up-to-date set of DST rules
# and timezone definitions, visit the Olson Database (or try pytz):
```

(continues on next page)

(continued from previous page)

```
# http://www.twinsun.com/tz/tz-link.htm
# https://sourceforge.net/projects/pytz/ (might not be up-to-date)
#
# In the US, since 2007, DST starts at 2am (standard time) on the second
# Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 2)
# From 1987 to 2006, DST used to start at 2am (standard time) on the first
# Sunday in April and to end at 2am (DST time) on the last
# Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 25, 2)
# From 1967 to 1986, DST used to start at 2am (standard time) on the last
# Sunday in April (the one on or after April 24) and to end at 2am (DST time)
# on the last Sunday of October, which is the first Sunday
# on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)
DSTEND_1967_1986 = DSTEND_1987_2006

def us_dst_range(year):
    # Find start and end times for US DST. For years before 1967, return
    # start = end for no DST.
    if 2006 < year:
        dststart, dstend = DSTSTART_2007, DSTEND_2007
    elif 1986 < year < 2007:
        dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
    elif 1966 < year < 1987:
        dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
    else:
        return (datetime(year, 1, 1), ) * 2

    start = first_sunday_on_or_after(dststart.replace(year=year))
    end = first_sunday_on_or_after(dstend.replace(year=year))
    return start, end


class USTimeZone(tzinfo):

    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname

    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)
```

(continues on next page)

(continued from previous page)

```
    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # An exception may be sensible here, in one or both cases.
            # It depends on how you want to treat them.  The default
            # fromutc() implementation (called by the default astimezone()
            # implementation) passes a datetime with dt.tzinfo is self.
            return ZERO
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        # Can't compare naive to aware objects, so strip the timezone from
        # dt first.
        dt = dt.replace(tzinfo=None)
        if start + HOUR <= dt < end - HOUR:
            # DST is in effect.
            return HOUR
        if end - HOUR <= dt < end:
            # Fold (an ambiguous hour): use dt.fold to disambiguate.
            return ZERO if dt.fold else HOUR
        if start <= dt < start + HOUR:
            # Gap (a non-existent hour): reverse the fold rule.
            return HOUR if dt.fold else ZERO
        # DST is off.
        return ZERO

    def fromutc(self, dt):
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        start = start.replace(tzinfo=self)
        end = end.replace(tzinfo=self)
        std_time = dt + self.stdoffset
        dst_time = std_time + HOUR
        if end <= dst_time < end + HOUR:
            # Repeated hour
            return std_time.replace(fold=1)
        if std_time < start or dst_time >= end:
            # Standard time
            return std_time
        if start <= std_time < end - HOUR:
            # Daylight saving time
            return dst_time


Eastern  = USTimeZone(-5, "Eastern",  "EST", "EDT")
Central  = USTimeZone(-6, "Central",  "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific  = USTimeZone(-8, "Pacific",  "PST", "PDT")
```

Note that there are unavoidable subtleties twice per year in a *tzinfo* subclass accounting for both standard and daylight time, at the DST transition points. For concreteness, consider US Eastern (UTC -0500), where EDT begins the minute after 1:59 (EST) on the second Sunday in March, and ends the minute after 1:59 (EDT) on the first Sunday in November:

```
  UTC   3:MM  4:MM  5:MM  6:MM  7:MM  8:MM
  EST  22:MM 23:MM  0:MM  1:MM  2:MM  3:MM
  EDT  23:MM  0:MM  1:MM  2:MM  3:MM  4:MM

start  22:MM 23:MM  0:MM  1:MM  3:MM  4:MM
```

(continues on next page)

(continued from previous page)

```
  end   23:MM   0:MM   1:MM   1:MM   2:MM   3:MM
```

When DST starts (the "start" line), the local wall clock leaps from 1:59 to 3:00. A wall time of the form 2:MM doesn't really make sense on that day, so `astimezone(Eastern)` won't deliver a result with `hour == 2` on the day DST begins. For example, at the Spring forward transition of 2016, we get:

```
>>> from datetime import datetime, timezone
>>> from tzinfo_examples import HOUR, Eastern
>>> u0 = datetime(2016, 3, 13, 5, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname())
...
05:00:00 UTC = 00:00:00 EST
06:00:00 UTC = 01:00:00 EST
07:00:00 UTC = 03:00:00 EDT
08:00:00 UTC = 04:00:00 EDT
```

When DST ends (the "end" line), there's a potentially worse problem: there's an hour that can't be spelled unambiguously in local wall time: the last hour of daylight time. In Eastern, that's times of the form 5:MM UTC on the day daylight time ends. The local wall clock leaps from 1:59 (daylight time) back to 1:00 (standard time) again. Local times of the form 1:MM are ambiguous. *astimezone()* mimics the local clock's behavior by mapping two adjacent UTC hours into the same local hour then. In the Eastern example, UTC times of the form 5:MM and 6:MM both map to 1:MM when converted to Eastern, but earlier times have the *fold* attribute set to 0 and the later times have it set to 1. For example, at the Fall back transition of 2016, we get:

```
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname(), t.fold)
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

Note that the *datetime* instances that differ only by the value of the *fold* attribute are considered equal in comparisons.

Applications that can't bear wall-time ambiguities should explicitly check the value of the *fold* attribute or avoid using hybrid *tzinfo* subclasses; there are no ambiguities when using *timezone*, or any other fixed-offset *tzinfo* subclass (such as a class representing only EST (fixed offset -5 hours), or only EDT (fixed offset -4 hours)).

See also

zoneinfo
The `datetime` module has a basic *timezone* class (for handling arbitrary fixed offsets from UTC) and its *timezone.utc* attribute (a UTC `timezone` instance).

`zoneinfo` brings the *IANA time zone database* (also known as the Olson database) to Python, and its usage is recommended.

IANA time zone database
The Time Zone Database (often called tz, tzdata or zoneinfo) contains code and data that represent the history

of local time for many representative locations around the globe. It is updated periodically to reflect changes made by political bodies to time zone boundaries, UTC offsets, and daylight-saving rules.

8.1.9 `timezone` Objects

The *timezone* class is a subclass of *tzinfo*, each instance of which represents a time zone defined by a fixed offset from UTC.

Objects of this class cannot be used to represent time zone information in the locations where different offsets are used in different days of the year or where historical changes have been made to civil time.

class datetime.**timezone**(*offset*, *name=None*)

The *offset* argument must be specified as a *timedelta* object representing the difference between the local time and UTC. It must be strictly between `-timedelta(hours=24)` and `timedelta(hours=24)`, otherwise *ValueError* is raised.

The *name* argument is optional. If specified it must be a string that will be used as the value returned by the *datetime.tzname()* method.

Added in version 3.2.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

timezone.**utcoffset**(*dt*)

Return the fixed value specified when the *timezone* instance is constructed.

The *dt* argument is ignored. The return value is a *timedelta* instance equal to the difference between the local time and UTC.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

timezone.**tzname**(*dt*)

Return the fixed value specified when the *timezone* instance is constructed.

If *name* is not provided in the constructor, the name returned by `tzname(dt)` is generated from the value of the `offset` as follows. If *offset* is `timedelta(0)`, the name is "UTC", otherwise it is a string in the format `UTC±HH:MM`, where ± is the sign of `offset`, HH and MM are two digits of `offset.hours` and `offset.minutes` respectively.

Changed in version 3.6: Name generated from `offset=timedelta(0)` is now plain `'UTC'`, not `'UTC+00:00'`.

timezone.**dst**(*dt*)

Always returns `None`.

timezone.**fromutc**(*dt*)

Return `dt + offset`. The *dt* argument must be an aware *datetime* instance, with `tzinfo` set to `self`.

Class attributes:

timezone.**utc**

The UTC time zone, `timezone(timedelta(0))`.

8.1.10 `strftime()` and `strptime()` Behavior

date, *datetime*, and *time* objects all support a `strftime(format)` method, to create a string representing the time under the control of an explicit format string.

Conversely, the *datetime.strptime()* class method creates a *datetime* object from a string representing a date and time and a corresponding format string.

The table below provides a high-level comparison of *strftime()* versus *strptime()*:

	strftime	strptime
Usage	Convert object to a string according to a given format	Parse a string into a *datetime* object given a corresponding format
Type of method	Instance method	Class method
Method of	*date*; *datetime*; *time*	*datetime*
Signature	`strftime(format)`	`strptime(date_string, format)`

`strftime()` and `strptime()` Format Codes

These methods accept format codes that can be used to parse and format dates:

```
>>> datetime.strptime('31/01/22 23:59:59.999999',
...                   '%d/%m/%y %H:%M:%S.%f')
datetime.datetime(2022, 1, 31, 23, 59, 59, 999999)
>>> _.strftime('%a %d %b %Y, %I:%M%p')
'Mon 31 Jan 2022, 11:59PM'
```

The following is a list of all the format codes that the 1989 C standard requires, and these work on all platforms with a standard C implementation.

Directive	Meaning	Example	Notes
%a	Weekday as locale's abbreviated name.	Sun, Mon, …, Sat (en_US); So, Mo, …, Sa (de_DE)	(1)
%A	Weekday as locale's full name.	Sunday, Monday, …, Saturday (en_US); Sonntag, Montag, …, Samstag (de_DE)	(1)
%w	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0, 1, …, 6	
%d	Day of the month as a zero-padded decimal number.	01, 02, …, 31	(9)
%b	Month as locale's abbreviated name.	Jan, Feb, …, Dec (en_US); Jan, Feb, …, Dez (de_DE)	(1)
%B	Month as locale's full name.	January, February, …, December (en_US); Januar, Februar, …, Dezember (de_DE)	(1)
%m	Month as a zero-padded decimal number.	01, 02, …, 12	(9)
%y	Year without century as a zero-padded decimal number.	00, 01, …, 99	(9)
%Y	Year with century as a decimal number.	0001, 0002, …, 2013, 2014, …, 9998, 9999	(2)
%H	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, …, 23	(9)
%I	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, …, 12	(9)
%p	Locale's equivalent of either AM or PM.	AM, PM (en_US); am, pm (de_DE)	(1), (3)
%M	Minute as a zero-padded decimal number.	00, 01, …, 59	(9)
%S	Second as a zero-padded decimal number.	00, 01, …, 59	(4), (9)
%f	Microsecond as a decimal number, zero-padded to 6 digits.	000000, 000001, …, 999999	(5)
%z	UTC offset in the form ±HHMM[SS[.ffffff]] (empty string	(empty), +0000, -0400, +1030, +063415, -030712.345216	(6)

Several additional directives not required by the C89 standard are included for convenience. These parameters all correspond to ISO 8601 date values.

Directive	Meaning	Example	Notes
`%G`	ISO 8601 year with century representing the year that contains the greater part of the ISO week (`%V`).	0001, 0002, …, 2013, 2014, …, 9998, 9999	(8)
`%u`	ISO 8601 weekday as a decimal number where 1 is Monday.	1, 2, …, 7	
`%V`	ISO 8601 week as a decimal number with Monday as the first day of the week. Week 01 is the week containing Jan 4.	01, 02, …, 53	(8), (9)
`%:z`	UTC offset in the form `±HH:MM[:SS[.ffffff]]` (empty string if the object is naive).	(empty), +00:00, -04:00, +10:30, +06:34:15, -03:07:12.345216	(6)

These may not be available on all platforms when used with the *strftime()* method. The ISO 8601 year and ISO 8601 week directives are not interchangeable with the year and week number directives above. Calling *strptime()* with incomplete or ambiguous ISO 8601 directives will raise a *ValueError*.

The full set of format codes supported varies across platforms, because Python calls the platform C library's `strftime()` function, and platform variations are common. To see the full set of format codes supported on your platform, consult the *strftime(3)* documentation. There are also differences between platforms in handling of unsupported format specifiers.

Added in version 3.6: `%G`, `%u` and `%V` were added.

Added in version 3.12: `%:z` was added.

Technical Detail

Broadly speaking, `d.strftime(fmt)` acts like the *time* module's `time.strftime(fmt, d.timetuple())` although not all objects support a *timetuple()* method.

For the *datetime.strptime()* class method, the default value is `1900-01-01T00:00:00.000`: any components not specified in the format string will be pulled from the default value.[4]

Using `datetime.strptime(date_string, format)` is equivalent to:

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

except when the format includes sub-second components or time zone offset information, which are supported in `datetime.strptime` but are discarded by `time.strptime`.

For *time* objects, the format codes for year, month, and day should not be used, as `time` objects have no such values. If they're used anyway, 1900 is substituted for the year, and 1 for the month and day.

For *date* objects, the format codes for hours, minutes, seconds, and microseconds should not be used, as *date* objects have no such values. If they're used anyway, 0 is substituted for them.

For the same reason, handling of format strings containing Unicode code points that can't be represented in the charset of the current locale is also platform-dependent. On some platforms such code points are preserved intact in the output, while on others `strftime` may raise *UnicodeError* or return an empty string instead.

Notes:

[4] Passing `datetime.strptime('Feb 29', '%b %d')` will fail since 1900 is not a leap year.

(1) Because the format depends on the current locale, care should be taken when making assumptions about the output value. Field orderings will vary (for example, "month/day/year" versus "day/month/year"), and the output may contain non-ASCII characters.

(2) The *strptime()* method can parse years in the full [1, 9999] range, but years < 1000 must be zero-filled to 4-digit width.

Changed in version 3.2: In previous versions, *strftime()* method was restricted to years >= 1900.

Changed in version 3.3: In version 3.2, *strftime()* method was restricted to years >= 1000.

(3) When used with the *strptime()* method, the `%p` directive only affects the output hour field if the `%I` directive is used to parse the hour.

(4) Unlike the *time* module, the `datetime` module does not support leap seconds.

(5) When used with the *strptime()* method, the `%f` directive accepts from one to six digits and zero pads on the right. `%f` is an extension to the set of format characters in the C standard (but implemented separately in datetime objects, and therefore always available).

(6) For a naive object, the `%z`, `%:z` and `%Z` format codes are replaced by empty strings.

For an aware object:

%z

utcoffset() is transformed into a string of the form `±HHMM[SS[.ffffff]]`, where `HH` is a 2-digit string giving the number of UTC offset hours, `MM` is a 2-digit string giving the number of UTC offset minutes, `SS` is a 2-digit string giving the number of UTC offset seconds and `ffffff` is a 6-digit string giving the number of UTC offset microseconds. The `ffffff` part is omitted when the offset is a whole number of seconds and both the `ffffff` and the `SS` part is omitted when the offset is a whole number of minutes. For example, if *utcoffset()* returns `timedelta(hours=-3, minutes=-30)`, `%z` is replaced with the string `'-0330'`.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

Changed in version 3.7: When the `%z` directive is provided to the *strptime()* method, the UTC offsets can have a colon as a separator between hours, minutes and seconds. For example, `'+01:00:00'` will be parsed as an offset of one hour. In addition, providing `'Z'` is identical to `'+00:00'`.

%:z

Behaves exactly as `%z`, but has a colon separator added between hours, minutes and seconds.

%Z

In *strftime()*, `%Z` is replaced by an empty string if *tzname()* returns `None`; otherwise `%Z` is replaced by the returned value, which must be a string.

strptime() only accepts certain values for `%Z`:

1. any value in `time.tzname` for your machine's locale
2. the hard-coded values `UTC` and `GMT`

So someone living in Japan may have `JST`, `UTC`, and `GMT` as valid values, but probably not `EST`. It will raise `ValueError` for invalid values.

Changed in version 3.2: When the `%z` directive is provided to the *strptime()* method, an aware *datetime* object will be produced. The `tzinfo` of the result will be set to a *timezone* instance.

(7) When used with the *strptime()* method, `%U` and `%W` are only used in calculations when the day of the week and the calendar year (`%Y`) are specified.

(8) Similar to `%U` and `%W`, `%V` is only used in calculations when the day of the week and the ISO year (`%G`) are specified in a *strptime()* format string. Also note that `%G` and `%Y` are not interchangeable.

(9) When used with the *strptime()* method, the leading zero is optional for formats %d, %m, %H, %I, %M, %S, %j, %U, %W, and %V. Format %y does require a leading zero.

(10) When parsing a month and day using *strptime()*, always include a year in the format. If the value you need to parse lacks a year, append an explicit dummy leap year. Otherwise your code will raise an exception when it encounters leap day because the default year used by the parser is not a leap year. Users run into this bug every four years…

```
>>> month_day = "02/29"
>>> datetime.strptime(f"{month_day};1984", "%m/%d;%Y")  # No leap year bug.
datetime.datetime(1984, 2, 29, 0, 0)
```

Deprecated since version 3.13, will be removed in version 3.15: *strptime()* calls using a format string containing a day of month without a year now emit a *DeprecationWarning*. In 3.15 or later we may change this into an error or change the default year to a leap year. See gh-70647.

8.2 zoneinfo — IANA time zone support

Added in version 3.9.

Source code: Lib/zoneinfo

The *zoneinfo* module provides a concrete time zone implementation to support the IANA time zone database as originally specified in **PEP 615**. By default, *zoneinfo* uses the system's time zone data if available; if no system time zone data is available, the library will fall back to using the first-party tzdata package available on PyPI.

See also

Module: *datetime*
Provides the *time* and *datetime* types with which the *ZoneInfo* class is designed to be used.

Package tzdata
First-party package maintained by the CPython core developers to supply time zone data via PyPI.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

8.2.1 Using ZoneInfo

ZoneInfo is a concrete implementation of the *datetime.tzinfo* abstract base class, and is intended to be attached to tzinfo, either via the constructor, the *datetime.replace* method or *datetime.astimezone*:

```
>>> from zoneinfo import ZoneInfo
>>> from datetime import datetime, timedelta

>>> dt = datetime(2020, 10, 31, 12, tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
2020-10-31 12:00:00-07:00

>>> dt.tzname()
'PDT'
```

Datetimes constructed in this way are compatible with datetime arithmetic and handle daylight saving time transitions with no further intervention:

```
>>> dt_add = dt + timedelta(days=1)

>>> print(dt_add)
2020-11-01 12:00:00-08:00

>>> dt_add.tzname()
'PST'
```

These time zones also support the *fold* attribute introduced in **PEP 495**. During offset transitions which induce ambiguous times (such as a daylight saving time to standard time transition), the offset from *before* the transition is used when `fold=0`, and the offset *after* the transition is used when `fold=1`, for example:

```
>>> dt = datetime(2020, 11, 1, 1, tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
2020-11-01 01:00:00-07:00

>>> print(dt.replace(fold=1))
2020-11-01 01:00:00-08:00
```

When converting from another time zone, the fold will be set to the correct value:

```
>>> from datetime import timezone
>>> LOS_ANGELES = ZoneInfo("America/Los_Angeles")
>>> dt_utc = datetime(2020, 11, 1, 8, tzinfo=timezone.utc)

>>> # Before the PDT -> PST transition
>>> print(dt_utc.astimezone(LOS_ANGELES))
2020-11-01 01:00:00-07:00

>>> # After the PDT -> PST transition
>>> print((dt_utc + timedelta(hours=1)).astimezone(LOS_ANGELES))
2020-11-01 01:00:00-08:00
```

8.2.2 Data sources

The `zoneinfo` module does not directly provide time zone data, and instead pulls time zone information from the system time zone database or the first-party PyPI package tzdata, if available. Some systems, including notably Windows systems, do not have an IANA database available, and so for projects targeting cross-platform compatibility that require time zone data, it is recommended to declare a dependency on tzdata. If neither system data nor tzdata are available, all calls to *ZoneInfo* will raise *ZoneInfoNotFoundError*.

Configuring the data sources

When `ZoneInfo(key)` is called, the constructor first searches the directories specified in *TZPATH* for a file matching `key`, and on failure looks for a match in the tzdata package. This behavior can be configured in three ways:

1. The default *TZPATH* when not otherwise specified can be configured at *compile time*.
2. *TZPATH* can be configured using *an environment variable*.
3. At *runtime*, the search path can be manipulated using the *reset_tzpath()* function.

Compile-time configuration

The default *TZPATH* includes several common deployment locations for the time zone database (except on Windows, where there are no "well-known" locations for time zone data). On POSIX systems, downstream distributors and those building Python from source who know where their system time zone data is deployed may change the default time zone path by specifying the compile-time option `TZPATH` (or, more likely, the `configure flag --with-tzpath`), which should be a string delimited by *os.pathsep*.

On all platforms, the configured value is available as the `TZPATH` key in *sysconfig.get_config_var()*.

Environment configuration

When initializing *TZPATH* (either at import time or whenever *reset_tzpath()* is called with no arguments), the `zoneinfo` module will use the environment variable `PYTHONTZPATH`, if it exists, to set the search path.

PYTHONTZPATH

This is an *os.pathsep*-separated string containing the time zone search path to use. It must consist of only absolute rather than relative paths. Relative components specified in `PYTHONTZPATH` will not be used, but otherwise the behavior when a relative path is specified is implementation-defined; CPython will raise *InvalidTZPathWarning*, but other implementations are free to silently ignore the erroneous component or raise an exception.

To set the system to ignore the system data and use the tzdata package instead, set `PYTHONTZPATH=""`.

Runtime configuration

The TZ search path can also be configured at runtime using the *reset_tzpath()* function. This is generally not an advisable operation, though it is reasonable to use it in test functions that require the use of a specific time zone path (or require disabling access to the system time zones).

8.2.3 The `ZoneInfo` class

class zoneinfo.**ZoneInfo**(*key*)

A concrete *datetime.tzinfo* subclass that represents an IANA time zone specified by the string `key`. Calls to the primary constructor will always return objects that compare identically; put another way, barring cache invalidation via *ZoneInfo.clear_cache()*, for all values of `key`, the following assertion will always be true:

```
a = ZoneInfo(key)
b = ZoneInfo(key)
assert a is b
```

`key` must be in the form of a relative, normalized POSIX path, with no up-level references. The constructor will raise *ValueError* if a non-conforming key is passed.

If no file matching `key` is found, the constructor will raise *ZoneInfoNotFoundError*.

The `ZoneInfo` class has two alternate constructors:

classmethod ZoneInfo.**from_file**(*fobj*, */*, *key=None*)

Constructs a `ZoneInfo` object from a file-like object returning bytes (e.g. a file opened in binary mode or an *io.BytesIO* object). Unlike the primary constructor, this always constructs a new object.

The `key` parameter sets the name of the zone for the purposes of `__str__()` and `__repr__()`.

Objects created via this constructor cannot be pickled (see *pickling*).

classmethod ZoneInfo.**no_cache**(*key*)

An alternate constructor that bypasses the constructor's cache. It is identical to the primary constructor, but returns a new object on each call. This is most likely to be useful for testing or demonstration purposes, but it can also be used to create a system with a different cache invalidation strategy.

Objects created via this constructor will also bypass the cache of a deserializing process when unpickled.

> **Caution**
>
> Using this constructor may change the semantics of your datetimes in surprising ways, only use it if you know that you need to.

The following class methods are also available:

classmethod ZoneInfo.**clear_cache**(*, *only_keys=None*)

A method for invalidating the cache on the `ZoneInfo` class. If no arguments are passed, all caches are invalidated and the next call to the primary constructor for each key will return a new instance.

If an iterable of key names is passed to the `only_keys` parameter, only the specified keys will be removed from the cache. Keys passed to `only_keys` but not found in the cache are ignored.

> **Warning**
>
> Invoking this function may change the semantics of datetimes using `ZoneInfo` in surprising ways; this modifies module state and thus may have wide-ranging effects. Only use it if you know that you need to.

The class has one attribute:

ZoneInfo.**key**

This is a read-only *attribute* that returns the value of `key` passed to the constructor, which should be a lookup key in the IANA time zone database (e.g. `America/New_York`, `Europe/Paris` or `Asia/Tokyo`).

For zones constructed from file without specifying a `key` parameter, this will be set to `None`.

> **Note**
>
> Although it is a somewhat common practice to expose these to end users, these values are designed to be primary keys for representing the relevant zones and not necessarily user-facing elements. Projects like CLDR (the Unicode Common Locale Data Repository) can be used to get more user-friendly strings from these keys.

String representations

The string representation returned when calling *str* on a *ZoneInfo* object defaults to using the *ZoneInfo.key* attribute (see the note on usage in the attribute documentation):

```
>>> zone = ZoneInfo("Pacific/Kwajalein")
>>> str(zone)
'Pacific/Kwajalein'

>>> dt = datetime(2020, 4, 1, 3, 15, tzinfo=zone)
```

(continues on next page)

(continued from previous page)

```
>>> f"{dt.isoformat()} [{dt.tzinfo}]"
'2020-04-01T03:15:00+12:00 [Pacific/Kwajalein]'
```

For objects constructed from a file without specifying a `key` parameter, `str` falls back to calling *`repr()`*. `ZoneInfo`'s `repr` is implementation-defined and not necessarily stable between versions, but it is guaranteed not to be a valid `ZoneInfo` key.

Pickle serialization

Rather than serializing all transition data, `ZoneInfo` objects are serialized by key, and `ZoneInfo` objects constructed from files (even those with a value for `key` specified) cannot be pickled.

The behavior of a `ZoneInfo` file depends on how it was constructed:

1. `ZoneInfo(key)`: When constructed with the primary constructor, a `ZoneInfo` object is serialized by key, and when deserialized, the deserializing process uses the primary and thus it is expected that these are expected to be the same object as other references to the same time zone. For example, if `europe_berlin_pkl` is a string containing a pickle constructed from `ZoneInfo("Europe/Berlin")`, one would expect the following behavior:

   ```
   >>> a = ZoneInfo("Europe/Berlin")
   >>> b = pickle.loads(europe_berlin_pkl)
   >>> a is b
   True
   ```

2. `ZoneInfo.no_cache(key)`: When constructed from the cache-bypassing constructor, the `ZoneInfo` object is also serialized by key, but when deserialized, the deserializing process uses the cache bypassing constructor. If `europe_berlin_pkl_nc` is a string containing a pickle constructed from `ZoneInfo.no_cache("Europe/Berlin")`, one would expect the following behavior:

   ```
   >>> a = ZoneInfo("Europe/Berlin")
   >>> b = pickle.loads(europe_berlin_pkl_nc)
   >>> a is b
   False
   ```

3. `ZoneInfo.from_file(fobj, /, key=None)`: When constructed from a file, the `ZoneInfo` object raises an exception on pickling. If an end user wants to pickle a `ZoneInfo` constructed from a file, it is recommended that they use a wrapper type or a custom serialization function: either serializing by key or storing the contents of the file object and serializing that.

This method of serialization requires that the time zone data for the required key be available on both the serializing and deserializing side, similar to the way that references to classes and functions are expected to exist in both the serializing and deserializing environments. It also means that no guarantees are made about the consistency of results when unpickling a `ZoneInfo` pickled in an environment with a different version of the time zone data.

8.2.4 Functions

zoneinfo.**available_timezones**()

Get a set containing all the valid keys for IANA time zones available anywhere on the time zone path. This is recalculated on every call to the function.

This function only includes canonical zone names and does not include "special" zones such as those under the `posix/` and `right/` directories, or the `posixrules` zone.

Caution

This function may open a large number of files, as the best way to determine if a file on the time zone path is a valid time zone is to read the "magic string" at the beginning.

Note

These values are not designed to be exposed to end-users; for user facing elements, applications should use something like CLDR (the Unicode Common Locale Data Repository) to get more user-friendly strings. See also the cautionary note on *ZoneInfo.key*.

zoneinfo.**reset_tzpath**(*to=None*)

Sets or resets the time zone search path (*TZPATH*) for the module. When called with no arguments, *TZPATH* is set to the default value.

Calling `reset_tzpath` will not invalidate the *ZoneInfo* cache, and so calls to the primary `ZoneInfo` constructor will only use the new `TZPATH` in the case of a cache miss.

The `to` parameter must be a *sequence* of strings or *os.PathLike* and not a string, all of which must be absolute paths. *ValueError* will be raised if something other than an absolute path is passed.

8.2.5 Globals

zoneinfo.**TZPATH**

A read-only sequence representing the time zone search path – when constructing a `ZoneInfo` from a key, the key is joined to each entry in the `TZPATH`, and the first file found is used.

`TZPATH` may contain only absolute paths, never relative paths, regardless of how it is configured.

The object that `zoneinfo.TZPATH` points to may change in response to a call to *reset_tzpath()*, so it is recommended to use `zoneinfo.TZPATH` rather than importing `TZPATH` from `zoneinfo` or assigning a long-lived variable to `zoneinfo.TZPATH`.

For more information on configuring the time zone search path, see *Configuring the data sources*.

8.2.6 Exceptions and warnings

exception zoneinfo.**ZoneInfoNotFoundError**

Raised when construction of a *ZoneInfo* object fails because the specified key could not be found on the system. This is a subclass of *KeyError*.

exception zoneinfo.**InvalidTZPathWarning**

Raised when *PYTHONTZPATH* contains an invalid component that will be filtered out, such as a relative path.

8.3 calendar — General calendar-related functions

Source code: Lib/calendar.py

This module allows you to output calendars like the Unix **cal** program, and provides additional useful functions related to the calendar. By default, these calendars have Monday as the first day of the week, and Sunday as the last (the European convention). Use *setfirstweekday()* to set the first day of the week to Sunday (6) or to any other weekday. Parameters that specify dates are given as integers. For related functionality, see also the *datetime* and *time* modules.

The functions and classes defined in this module use an idealized calendar, the current Gregorian calendar extended indefinitely in both directions. This matches the definition of the "proleptic Gregorian" calendar in Dershowitz and Reingold's book "Calendrical Calculations", where it's the base calendar for all computations. Zero and negative years are interpreted as prescribed by the ISO 8601 standard. Year 0 is 1 BC, year -1 is 2 BC, and so on.

class calendar.**Calendar**(*firstweekday=0*)

Creates a *Calendar* object. *firstweekday* is an integer specifying the first day of the week. *MONDAY* is 0 (the default), *SUNDAY* is 6.

A *Calendar* object provides several methods that can be used for preparing the calendar data for formatting. This class doesn't do any formatting itself. This is the job of subclasses.

Calendar instances have the following methods:

iterweekdays()

Return an iterator for the week day numbers that will be used for one week. The first value from the iterator will be the same as the value of the *firstweekday* property.

itermonthdates(*year*, *month*)

Return an iterator for the month *month* (1–12) in the year *year*. This iterator will return all days (as *datetime.date* objects) for the month and all days before the start of the month or after the end of the month that are required to get a complete week.

itermonthdays(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*, but not restricted by the *datetime.date* range. Days returned will simply be day of the month numbers. For the days outside of the specified month, the day number is 0.

itermonthdays2(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*, but not restricted by the *datetime.date* range. Days returned will be tuples consisting of a day of the month number and a week day number.

itermonthdays3(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to `itermonthdates()`, but not restricted by the `datetime.date` range. Days returned will be tuples consisting of a year, a month and a day of the month numbers.

Added in version 3.7.

itermonthdays4(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to `itermonthdates()`, but not restricted by the `datetime.date` range. Days returned will be tuples consisting of a year, a month, a day of the month, and a day of the week numbers.

Added in version 3.7.

monthdatescalendar(*year*, *month*)

Return a list of the weeks in the month *month* of the *year* as full weeks. Weeks are lists of seven `datetime.date` objects.

monthdays2calendar(*year*, *month*)

Return a list of the weeks in the month *month* of the *year* as full weeks. Weeks are lists of seven tuples of day numbers and weekday numbers.

monthdayscalendar(*year*, *month*)

Return a list of the weeks in the month *month* of the *year* as full weeks. Weeks are lists of seven day numbers.

yeardatescalendar(*year*, *width=3*)

Return the data for the specified year ready for formatting. The return value is a list of month rows. Each month row contains up to *width* months (defaulting to 3). Each month contains between 4 and 6 weeks and each week contains 1–7 days. Days are `datetime.date` objects.

yeardays2calendar(*year*, *width=3*)

Return the data for the specified year ready for formatting (similar to `yeardatescalendar()`). Entries in the week lists are tuples of day numbers and weekday numbers. Day numbers outside this month are zero.

yeardayscalendar(*year*, *width=3*)

Return the data for the specified year ready for formatting (similar to `yeardatescalendar()`). Entries in the week lists are day numbers. Day numbers outside this month are zero.

class calendar.**TextCalendar**(*firstweekday=0*)

This class can be used to generate plain text calendars.

`TextCalendar` instances have the following methods:

formatmonth(*theyear*, *themonth*, *w=0*, *l=0*)

Return a month's calendar in a multi-line string. If *w* is provided, it specifies the width of the date columns, which are centered. If *l* is given, it specifies the number of lines that each week will use. Depends on the first weekday as specified in the constructor or set by the `setfirstweekday()` method.

prmonth(*theyear*, *themonth*, *w=0*, *l=0*)

Print a month's calendar as returned by `formatmonth()`.

formatyear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)

Return a *m*-column calendar for an entire year as a multi-line string. Optional parameters *w*, *l*, and *c* are for date column width, lines per week, and number of spaces between month columns, respectively. Depends on the first weekday as specified in the constructor or set by the `setfirstweekday()` method. The earliest year for which a calendar can be generated is platform-dependent.

pryear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)

Print the calendar for an entire year as returned by `formatyear()`.

class calendar.**HTMLCalendar**(*firstweekday=0*)

This class can be used to generate HTML calendars.

`HTMLCalendar` instances have the following methods:

formatmonth(*theyear*, *themonth*, *withyear=True*)

Return a month's calendar as an HTML table. If *withyear* is true the year will be included in the header, otherwise just the month name will be used.

formatyear(*theyear*, *width=3*)

Return a year's calendar as an HTML table. *width* (defaulting to 3) specifies the number of months per row.

formatyearpage(*theyear*, *width=3*, *css='calendar.css'*, *encoding=None*)

Return a year's calendar as a complete HTML page. *width* (defaulting to 3) specifies the number of months per row. *css* is the name for the cascading style sheet to be used. `None` can be passed if no style sheet should be used. *encoding* specifies the encoding to be used for the output (defaulting to the system default encoding).

formatmonthname(*theyear*, *themonth*, *withyear=True*)

Return a month name as an HTML table row. If *withyear* is true the year will be included in the row, otherwise just the month name will be used.

`HTMLCalendar` has the following attributes you can override to customize the CSS classes used by the calendar:

cssclasses

A list of CSS classes used for each weekday. The default class list is:

```
cssclasses = ["mon", "tue", "wed", "thu", "fri", "sat", "sun"]
```

more styles can be added for each day:

```
cssclasses = ["mon text-bold", "tue", "wed", "thu", "fri", "sat", "sun red"]
```

Note that the length of this list must be seven items.

cssclass_noday

The CSS class for a weekday occurring in the previous or coming month.

Added in version 3.7.

cssclasses_weekday_head

A list of CSS classes used for weekday names in the header row. The default is the same as `cssclasses`.

Added in version 3.7.

cssclass_month_head

The month's head CSS class (used by `formatmonthname()`). The default value is `"month"`.

Added in version 3.7.

cssclass_month

The CSS class for the whole month's table (used by `formatmonth()`). The default value is `"month"`.

Added in version 3.7.

cssclass_year

The CSS class for the whole year's table of tables (used by *formatyear()*). The default value is `"year"`.

Added in version 3.7.

cssclass_year_head

The CSS class for the table head for the whole year (used by *formatyear()*). The default value is `"year"`.

Added in version 3.7.

Note that although the naming for the above described class attributes is singular (e.g. `cssclass_month` `cssclass_noday`), one can replace the single CSS class with a space separated list of CSS classes, for example:

```
"text-bold text-red"
```

Here is an example how `HTMLCalendar` can be customized:

```
class CustomHTMLCal(calendar.HTMLCalendar):
    cssclasses = [style + " text-nowrap" for style in
                  calendar.HTMLCalendar.cssclasses]
    cssclass_month_head = "text-center month-head"
    cssclass_month = "text-center month"
    cssclass_year = "text-italic lead"
```

class calendar.**LocaleTextCalendar**(*firstweekday=0*, *locale=None*)

This subclass of *TextCalendar* can be passed a locale name in the constructor and will return month and weekday names in the specified locale.

class calendar.**LocaleHTMLCalendar**(*firstweekday=0*, *locale=None*)

This subclass of *HTMLCalendar* can be passed a locale name in the constructor and will return month and weekday names in the specified locale.

Note

The constructor, `formatweekday()` and `formatmonthname()` methods of these two classes temporarily change the `LC_TIME` locale to the given *locale*. Because the current locale is a process-wide setting, they are not thread-safe.

For simple text calendars this module provides the following functions.

calendar.**setfirstweekday**(*weekday*)

Sets the weekday (`0` is Monday, `6` is Sunday) to start each week. The values *MONDAY*, *TUESDAY*, *WEDNESDAY*, *THURSDAY*, *FRIDAY*, *SATURDAY*, and *SUNDAY* are provided for convenience. For example, to set the first weekday to Sunday:

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

calendar.**firstweekday**()

Returns the current setting for the weekday to start each week.

calendar.**isleap**(*year*)

Returns *True* if *year* is a leap year, otherwise *False*.

calendar.**leapdays**(*y1*, *y2*)

Returns the number of leap years in the range from *y1* to *y2* (exclusive), where *y1* and *y2* are years.

This function works for ranges spanning a century change.

calendar.**weekday**(*year*, *month*, *day*)

Returns the day of the week (`0` is Monday) for *year* (`1970`–…), *month* (`1`–`12`), *day* (`1`–`31`).

calendar.**weekheader**(*n*)

Return a header containing abbreviated weekday names. *n* specifies the width in characters for one weekday.

calendar.**monthrange**(*year*, *month*)

Returns weekday of first day of the month and number of days in month, for the specified *year* and *month*.

calendar.**monthcalendar**(*year*, *month*)

Returns a matrix representing a month's calendar. Each row represents a week; days outside of the month are represented by zeros. Each week begins with Monday unless set by *setfirstweekday()*.

calendar.**prmonth**(*theyear*, *themonth*, *w=0*, *l=0*)

Prints a month's calendar as returned by *month()*.

calendar.**month**(*theyear*, *themonth*, *w=0*, *l=0*)

Returns a month's calendar in a multi-line string using the *formatmonth()* of the *TextCalendar* class.

calendar.**prcal**(*year*, *w=0*, *l=0*, *c=6*, *m=3*)

Prints the calendar for an entire year as returned by *calendar()*.

calendar.**calendar**(*year*, *w=2*, *l=1*, *c=6*, *m=3*)

Returns a 3-column calendar for an entire year as a multi-line string using the *formatyear()* of the *TextCalendar* class.

calendar.**timegm**(*tuple*)

An unrelated but handy function that takes a time tuple such as returned by the *gmtime()* function in the *time* module, and returns the corresponding Unix timestamp value, assuming an epoch of 1970, and the POSIX encoding. In fact, *time.gmtime()* and *timegm()* are each others' inverse.

The *calendar* module exports the following data attributes:

calendar.**day_name**

An array that represents the days of the week in the current locale.

calendar.**day_abbr**

An array that represents the abbreviated days of the week in the current locale.

calendar.**MONDAY**

calendar.**TUESDAY**

calendar.**WEDNESDAY**

calendar.**THURSDAY**

calendar.**FRIDAY**

calendar.**SATURDAY**

calendar.**SUNDAY**

Aliases for the days of the week, where `MONDAY` is `0` and `SUNDAY` is `6`.

Added in version 3.12.

class calendar.**Day**

Enumeration defining days of the week as integer constants. The members of this enumeration are exported to the module scope as *MONDAY* through *SUNDAY*.

Added in version 3.12.

calendar.**month_name**

An array that represents the months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and `month_name[0]` is the empty string.

calendar.**month_abbr**

An array that represents the abbreviated months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and `month_abbr[0]` is the empty string.

calendar.**JANUARY**

calendar.**FEBRUARY**

calendar.**MARCH**

calendar.**APRIL**

calendar.**MAY**

calendar.**JUNE**

calendar.**JULY**

calendar.**AUGUST**

calendar.**SEPTEMBER**

calendar.**OCTOBER**

calendar.**NOVEMBER**

calendar.**DECEMBER**

Aliases for the months of the year, where `JANUARY` is `1` and `DECEMBER` is `12`.

Added in version 3.12.

class calendar.**Month**

Enumeration defining months of the year as integer constants. The members of this enumeration are exported to the module scope as *JANUARY* through *DECEMBER*.

Added in version 3.12.

The *calendar* module defines the following exceptions:

exception calendar.**IllegalMonthError**(*month*)

A subclass of *ValueError*, raised when the given month number is outside of the range 1-12 (inclusive).

month

The invalid month number.

exception calendar.**IllegalWeekdayError**(*weekday*)

A subclass of *ValueError*, raised when the given weekday number is outside of the range 0-6 (inclusive).

weekday

The invalid weekday number.

See also

Module *datetime*
Object-oriented interface to dates and times with similar functionality to the *time* module.

Module ***time***
Low-level time related functions.

8.3.1 Command-Line Usage

Added in version 2.5.

The *calendar* module can be executed as a script from the command line to interactively print a calendar.

```
python -m calendar [-h] [-L LOCALE] [-e ENCODING] [-t {text,html}]
                   [-w WIDTH] [-l LINES] [-s SPACING] [-m MONTHS] [-c CSS]
                   [-f FIRST_WEEKDAY] [year] [month]
```

For example, to print a calendar for the year 2000:

```
$ python -m calendar 2000
                                  2000

      January                   February                  March
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                1  2          1  2  3  4  5  6          1  2  3  4  5
 3  4  5  6  7  8  9       7  8  9 10 11 12 13       6  7  8  9 10 11 12
10 11 12 13 14 15 16      14 15 16 17 18 19 20      13 14 15 16 17 18 19
17 18 19 20 21 22 23      21 22 23 24 25 26 27      20 21 22 23 24 25 26
24 25 26 27 28 29 30      28 29                     27 28 29 30 31
31

       April                     May                       June
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                1  2       1  2  3  4  5  6  7                1  2  3  4
 3  4  5  6  7  8  9       8  9 10 11 12 13 14       5  6  7  8  9 10 11
10 11 12 13 14 15 16      15 16 17 18 19 20 21      12 13 14 15 16 17 18
17 18 19 20 21 22 23      22 23 24 25 26 27 28      19 20 21 22 23 24 25
24 25 26 27 28 29 30      29 30 31                  26 27 28 29 30

        July                    August                  September
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                1  2          1  2  3  4  5  6                   1  2  3
 3  4  5  6  7  8  9       7  8  9 10 11 12 13       4  5  6  7  8  9 10
10 11 12 13 14 15 16      14 15 16 17 18 19 20      11 12 13 14 15 16 17
17 18 19 20 21 22 23      21 22 23 24 25 26 27      18 19 20 21 22 23 24
24 25 26 27 28 29 30      28 29 30 31               25 26 27 28 29 30
31

      October                   November                  December
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                   1             1  2  3  4  5                   1  2  3
 2  3  4  5  6  7  8       6  7  8  9 10 11 12       4  5  6  7  8  9 10
 9 10 11 12 13 14 15      13 14 15 16 17 18 19      11 12 13 14 15 16 17
16 17 18 19 20 21 22      20 21 22 23 24 25 26      18 19 20 21 22 23 24
23 24 25 26 27 28 29      27 28 29 30               25 26 27 28 29 30 31
30 31
```

The following options are accepted:

--help, -h

Show the help message and exit.

--locale LOCALE, **-L** LOCALE

The locale to use for month and weekday names. Defaults to English.

--encoding ENCODING, **-e** ENCODING

The encoding to use for output. *--encoding* is required if *--locale* is set.

--type {text,html}, **-t** {text,html}

Print the calendar to the terminal as text, or as an HTML document.

--first-weekday FIRST_WEEKDAY, **-f** FIRST_WEEKDAY

The weekday to start each week. Must be a number between 0 (Monday) and 6 (Sunday). Defaults to 0.

Added in version 3.13.

year

The year to print the calendar for. Defaults to the current year.

month

The month of the specified *year* to print the calendar for. Must be a number between 1 and 12, and may only be used in text mode. Defaults to printing a calendar for the full year.

Text-mode options:

--width WIDTH, **-w** WIDTH

The width of the date column in terminal columns. The date is printed centred in the column. Any value lower than 2 is ignored. Defaults to 2.

--lines LINES, **-l** LINES

The number of lines for each week in terminal rows. The date is printed top-aligned. Any value lower than 1 is ignored. Defaults to 1.

--spacing SPACING, **-s** SPACING

The space between months in columns. Any value lower than 2 is ignored. Defaults to 6.

--months MONTHS, **-m** MONTHS

The number of months printed per row. Defaults to 3.

HTML-mode options:

--css CSS, **-c** CSS

The path of a CSS stylesheet to use for the calendar. This must either be relative to the generated HTML, or an absolute HTTP or `file:///` URL.

8.4 `collections` — Container datatypes

Source code: Lib/collections/__init__.py

This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, *dict*, *list*, *set*, and *tuple*.

namedtuple()	factory function for creating tuple subclasses with named fields
deque	list-like container with fast appends and pops on either end
ChainMap	dict-like class for creating a single view of multiple mappings
Counter	dict subclass for counting *hashable* objects
OrderedDict	dict subclass that remembers the order entries were added
defaultdict	dict subclass that calls a factory function to supply missing values
UserDict	wrapper around dictionary objects for easier dict subclassing
UserList	wrapper around list objects for easier list subclassing
UserString	wrapper around string objects for easier string subclassing

8.4.1 `ChainMap` objects

Added in version 3.3.

A *ChainMap* class is provided for quickly linking a number of mappings so they can be treated as a single unit. It is often much faster than creating a new dictionary and running multiple *update()* calls.

The class can be used to simulate nested scopes and is useful in templating.

class collections.**ChainMap**(**maps*)

A *ChainMap* groups multiple dicts or other mappings together to create a single, updateable view. If no *maps* are specified, a single empty dictionary is provided so that a new chain always has at least one mapping.

The underlying mappings are stored in a list. That list is public and can be accessed or updated using the *maps* attribute. There is no other state.

Lookups search the underlying mappings successively until a key is found. In contrast, writes, updates, and deletions only operate on the first mapping.

A *ChainMap* incorporates the underlying mappings by reference. So, if one of the underlying mappings gets updated, those changes will be reflected in *ChainMap*.

All of the usual dictionary methods are supported. In addition, there is a *maps* attribute, a method for creating new subcontexts, and a property for accessing all but the first mapping:

maps

A user updateable list of mappings. The list is ordered from first-searched to last-searched. It is the only stored state and can be modified to change which mappings are searched. The list should always contain at least one mapping.

new_child(*m=None*, ***kwargs*)

Returns a new *ChainMap* containing a new map followed by all of the maps in the current instance. If `m` is specified, it becomes the new map at the front of the list of mappings; if not specified, an empty dict is used, so that a call to `d.new_child()` is equivalent to: `ChainMap({}, *d.maps)`. If any keyword arguments are specified, they update passed map or new empty dict. This method is used for creating subcontexts that can be updated without altering values in any of the parent mappings.

Changed in version 3.4: The optional `m` parameter was added.

Changed in version 3.10: Keyword arguments support was added.

parents

Property returning a new *ChainMap* containing all of the maps in the current instance except the first one. This is useful for skipping the first map in the search. Use cases are similar to those for the `nonlocal` keyword used in *nested scopes*. The use cases also parallel those for the built-in *super()* function. A reference to `d.parents` is equivalent to: `ChainMap(*d.maps[1:])`.

Note, the iteration order of a *ChainMap* is determined by scanning the mappings last to first:

```
>>> baseline = {'music': 'bach', 'art': 'rembrandt'}
>>> adjustments = {'art': 'van gogh', 'opera': 'carmen'}
>>> list(ChainMap(adjustments, baseline))
['music', 'art', 'opera']
```

This gives the same ordering as a series of *dict.update()* calls starting with the last mapping:

```
>>> combined = baseline.copy()
>>> combined.update(adjustments)
>>> list(combined)
['music', 'art', 'opera']
```

Changed in version 3.9: Added support for | and |= operators, specified in **PEP 584**.

See also

- The MultiContext class in the Enthought CodeTools package has options to support writing to any mapping in the chain.
- Django's Context class for templating is a read-only chain of mappings. It also features pushing and popping of contexts similar to the *new_child()* method and the *parents* property.
- The Nested Contexts recipe has options to control whether writes and other mutations apply only to the first mapping or to any mapping in the chain.
- A greatly simplified read-only version of Chainmap.

`ChainMap` Examples and Recipes

This section shows various approaches to working with chained maps.

Example of simulating Python's internal lookup chain:

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

Example of letting user specified command-line arguments take precedence over environment variables which in turn take precedence over default values:

```
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k: v for k, v in vars(namespace).items() if v is not None}

combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

Example patterns for using the *ChainMap* class to simulate nested contexts:

```
c = ChainMap()          # Create root context
d = c.new_child()       # Create nested child context
e = c.new_child()       # Child of c, independent from d
e.maps[0]               # Current context dictionary -- like Python's locals()
e.maps[-1]              # Root context -- like Python's globals()
e.parents               # Enclosing context chain -- like Python's nonlocals

d['x'] = 1              # Set value in current context
d['x']                  # Get first key in the chain of contexts
del d['x']              # Delete from current context
list(d)                 # All nested values
k in d                  # Check all nested values
len(d)                  # Number of nested values
d.items()               # All nested items
dict(d)                 # Flatten into a regular dictionary
```

The *ChainMap* class only makes updates (writes and deletions) to the first mapping in the chain while lookups will search the full chain. However, if deep writes and deletions are desired, it is easy to make a subclass that updates keys found deeper in the chain:

```
class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner scopes'

    def __setitem__(self, key, value):
        for mapping in self.maps:
            if key in mapping:
                mapping[key] = value
                return
        self.maps[0][key] = value

    def __delitem__(self, key):
        for mapping in self.maps:
            if key in mapping:
                del mapping[key]
                return
        raise KeyError(key)

>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'blue'}, {'lion': 'yellow'})
>>> d['lion'] = 'orange'         # update an existing key two levels down
>>> d['snake'] = 'red'           # new keys get added to the topmost dict
>>> del d['elephant']            # remove an existing key one level down
>>> d                            # display result
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion': 'orange'})
```

8.4.2 `Counter` objects

A counter tool is provided to support convenient and rapid tallies. For example:

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
...
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})
```

(continues on next page)

(continued from previous page)

```
>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

class collections.**Counter**([*iterable-or-mapping*])

A *Counter* is a *dict* subclass for counting *hashable* objects. It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The *Counter* class is similar to bags or multisets in other languages.

Elements are counted from an *iterable* or initialized from another *mapping* (or counter):

```
>>> c = Counter()                           # a new, empty counter
>>> c = Counter('gallahad')                 # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)             # a new counter from keyword args
```

Counter objects have a dictionary interface except that they return a zero count for missing items instead of raising a *KeyError*:

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                              # count of a missing element is zero
0
```

Setting a count to zero does not remove an element from a counter. Use `del` to remove it entirely:

```
>>> c['sausage'] = 0                        # counter entry with a zero count
>>> del c['sausage']                        # del actually removes the entry
```

Added in version 3.1.

Changed in version 3.7: As a *dict* subclass, *Counter* inherited the capability to remember insertion order. Math operations on *Counter* objects also preserve order. Results are ordered according to when an element is first encountered in the left operand and then by the order encountered in the right operand.

Counter objects support additional methods beyond those available for all dictionaries:

elements()

Return an iterator over elements repeating each as many times as its count. Elements are returned in the order first encountered. If an element's count is less than one, *elements()* will ignore it.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common([*n*])

Return a list of the *n* most common elements and their counts from the most common to the least. If *n* is omitted or `None`, *most_common()* returns *all* elements in the counter. Elements with equal counts are ordered in the order first encountered:

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
```

subtract([*iterable-or-mapping*])

Elements are subtracted from an *iterable* or from another *mapping* (or counter). Like `dict.update()` but subtracts counts instead of replacing them. Both inputs and outputs may be zero or negative.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

Added in version 3.2.

total()

Compute the sum of the counts.

```
>>> c = Counter(a=10, b=5, c=0)
>>> c.total()
15
```

Added in version 3.10.

The usual dictionary methods are available for `Counter` objects except for two which work differently for counters.

fromkeys(*iterable*)

This class method is not implemented for `Counter` objects.

update([*iterable-or-mapping*])

Elements are counted from an *iterable* or added-in from another *mapping* (or counter). Like `dict.update()` but adds counts instead of replacing them. Also, the *iterable* is expected to be a sequence of elements, not a sequence of `(key, value)` pairs.

Counters support rich comparison operators for equality, subset, and superset relationships: ==, !=, <, <=, >, >=. All of those tests treat missing elements as having zero counts so that `Counter(a=1) == Counter(a=1, b=0)` returns true.

Changed in version 3.10: Rich comparison operations were added.

Changed in version 3.10: In equality tests, missing elements are treated as having zero counts. Formerly, `Counter(a=3)` and `Counter(a=3, b=0)` were considered distinct.

Common patterns for working with `Counter` objects:

```
c.total()                       # total of all counts
c.clear()                       # reset all counts
list(c)                         # list unique elements
set(c)                          # convert to a set
dict(c)                         # convert to a regular dictionary
c.items()                       # access the (elem, cnt) pairs
Counter(dict(list_of_pairs))    # convert from a list of (elem, cnt) pairs
c.most_common()[:-n-1:-1]       # n least common elements
+c                              # remove zero and negative counts
```

Several mathematical operations are provided for combining `Counter` objects to produce multisets (counters that have counts greater than zero). Addition and subtraction combine counters by adding or subtracting the counts of corresponding elements. Intersection and union return the minimum and maximum of corresponding counts. Equality and inclusion compare corresponding counts. Each operation can accept inputs with signed counts, but the output will exclude results with counts of zero or less.

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                       # add two counters together:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                       # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d                       # intersection:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d                       # union:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
>>> c == d                      # equality:  c[x] == d[x]
False
>>> c <= d                      # inclusion:  c[x] <= d[x]
False
```

Unary addition and subtraction are shortcuts for adding an empty counter or subtracting from an empty counter.

```
>>> c = Counter(a=2, b=-4)
>>> +c
Counter({'a': 2})
>>> -c
Counter({'b': 4})
```

Added in version 3.3: Added support for unary plus, unary minus, and in-place multiset operations.

Note

Counters were primarily designed to work with positive integers to represent running counts; however, care was taken to not unnecessarily preclude use cases needing other types or negative values. To help with those use cases, this section documents the minimum range and type restrictions.

- The `Counter` class itself is a dictionary subclass with no restrictions on its keys and values. The values are intended to be numbers representing counts, but you *could* store anything in the value field.
- The `most_common()` method requires only that the values be orderable.
- For in-place operations such as `c[key] += 1`, the value type need only support addition and subtraction. So fractions, floats, and decimals would work and negative values are supported. The same is also true for `update()` and `subtract()` which allow negative and zero values for both inputs and outputs.
- The multiset methods are designed only for use cases with positive values. The inputs may be negative or zero, but only outputs with positive values are created. There are no type restrictions, but the value type needs to support addition, subtraction, and comparison.
- The `elements()` method requires integer counts. It ignores zero and negative counts.

See also

- Bag class in Smalltalk.
- Wikipedia entry for Multisets.
- C++ multisets tutorial with examples.
- For mathematical operations on multisets and their use cases, see *Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19*.

- To enumerate all distinct multisets of a given size over a given set of elements, see *itertools.combinations_with_replacement()*:

```
map(Counter, combinations_with_replacement('ABC', 2)) # --> AA AB AC BB BC CC
```

8.4.3 deque objects

class collections.**deque**([*iterable*[, *maxlen*]])

Returns a new deque object initialized left-to-right (using *append()*) with data from *iterable*. If *iterable* is not specified, the new deque is empty.

Deques are a generalization of stacks and queues (the name is pronounced "deck" and is short for "double-ended queue"). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same $O(1)$ performance in either direction.

Though *list* objects support similar operations, they are optimized for fast fixed-length operations and incur $O(n)$ memory movement costs for `pop(0)` and `insert(0, v)` operations which change both the size and position of the underlying data representation.

If *maxlen* is not specified or is `None`, deques may grow to an arbitrary length. Otherwise, the deque is bounded to the specified maximum length. Once a bounded length deque is full, when new items are added, a corresponding number of items are discarded from the opposite end. Bounded length deques provide functionality similar to the `tail` filter in Unix. They are also useful for tracking transactions and other pools of data where only the most recent activity is of interest.

Deque objects support the following methods:

append(*x*)

Add *x* to the right side of the deque.

appendleft(*x*)

Add *x* to the left side of the deque.

clear()

Remove all elements from the deque leaving it with length 0.

copy()

Create a shallow copy of the deque.

Added in version 3.5.

count(*x*)

Count the number of deque elements equal to *x*.

Added in version 3.2.

extend(*iterable*)

Extend the right side of the deque by appending elements from the iterable argument.

extendleft(*iterable*)

Extend the left side of the deque by appending elements from *iterable*. Note, the series of left appends results in reversing the order of elements in the iterable argument.

index(*x*[, *start*[, *stop*]])

Return the position of *x* in the deque (at or after index *start* and before index *stop*). Returns the first match or raises *ValueError* if not found.

Added in version 3.5.

insert(*i*, *x*)

Insert *x* into the deque at position *i*.

If the insertion would cause a bounded deque to grow beyond *maxlen*, an *IndexError* is raised.

Added in version 3.5.

pop()

Remove and return an element from the right side of the deque. If no elements are present, raises an *IndexError*.

popleft()

Remove and return an element from the left side of the deque. If no elements are present, raises an *IndexError*.

remove(*value*)

Remove the first occurrence of *value*. If not found, raises a *ValueError*.

reverse()

Reverse the elements of the deque in-place and then return `None`.

Added in version 3.2.

rotate(*n=1*)

Rotate the deque *n* steps to the right. If *n* is negative, rotate to the left.

When the deque is not empty, rotating one step to the right is equivalent to `d.appendleft(d.pop())`, and rotating one step to the left is equivalent to `d.append(d.popleft())`.

Deque objects also provide one read-only attribute:

maxlen

Maximum size of a deque or `None` if unbounded.

Added in version 3.1.

In addition to the above, deques support iteration, pickling, `len(d)`, `reversed(d)`, `copy.copy(d)`, `copy.deepcopy(d)`, membership testing with the `in` operator, and subscript references such as `d[0]` to access the first element. Indexed access is *O*(1) at both ends but slows to *O*(*n*) in the middle. For fast random access, use lists instead.

Starting in version 3.5, deques support `__add__()`, `__mul__()`, and `__imul__()`.

Example:

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print(elem.upper())
G
H
I

>>> d.append('j')                    # add a new entry to the right side
>>> d.appendleft('f')                # add a new entry to the left side
>>> d                                # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the rightmost item
'j'
>>> d.popleft()                      # return and remove the leftmost item
'f'
```

(continues on next page)

(continued from previous page)

```
>>> list(d)                         # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                            # peek at leftmost item
'g'
>>> d[-1]                           # peek at rightmost item
'i'

>>> list(reversed(d))               # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                        # search the deque
True
>>> d.extend('jkl')                 # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                     # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                    # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))              # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                       # empty the deque
>>> d.pop()                         # cannot pop from an empty deque
Traceback (most recent call last):
    File "<pyshell#6>", line 1, in -toplevel-
        d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')             # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

deque Recipes

This section shows various approaches to working with deques.

Bounded length deques provide functionality similar to the `tail` filter in Unix:

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)
```

Another approach to using deques is to maintain a sequence of recently added elements by appending to the right and popping to the left:

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # https://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
```

(continues on next page)

(continued from previous page)

```
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

A round-robin scheduler can be implemented with input iterators stored in a *deque*. Values are yielded from the active iterator in position zero. If that iterator is exhausted, it can be removed with *popleft()*; otherwise, it can be cycled back to the end with the *rotate()* method:

```
def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    iterators = deque(map(iter, iterables))
    while iterators:
        try:
            while True:
                yield next(iterators[0])
                iterators.rotate(-1)
        except StopIteration:
            # Remove an exhausted iterator.
            iterators.popleft()
```

The *rotate()* method provides a way to implement *deque* slicing and deletion. For example, a pure Python implementation of `del d[n]` relies on the `rotate()` method to position elements to be popped:

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

To implement *deque* slicing, use a similar approach applying *rotate()* to bring a target element to the left side of the deque. Remove old entries with *popleft()*, add new entries with *extend()*, and then reverse the rotation. With minor variations on that approach, it is easy to implement Forth style stack manipulations such as `dup`, `drop`, `swap`, `over`, `pick`, `rot`, and `roll`.

8.4.4 `defaultdict` objects

class collections.**defaultdict**(*default_factory=None*, */*[, ...])

Return a new dictionary-like object. *defaultdict* is a subclass of the built-in *dict* class. It overrides one method and adds one writable instance variable. The remaining functionality is the same as for the *dict* class and is not documented here.

The first argument provides the initial value for the *default_factory* attribute; it defaults to `None`. All remaining arguments are treated the same as if they were passed to the *dict* constructor, including keyword arguments.

defaultdict objects support the following method in addition to the standard *dict* operations:

__missing__(*key*)

If the *default_factory* attribute is `None`, this raises a *KeyError* exception with the *key* as argument.

If *default_factory* is not `None`, it is called without arguments to provide a default value for the given *key*, this value is inserted in the dictionary for the *key*, and returned.

If calling *default_factory* raises an exception this exception is propagated unchanged.

This method is called by the `__getitem__()` method of the *dict* class when the requested key is not found; whatever it returns or raises is then returned or raised by `__getitem__()`.

Note that *__missing__()* is *not* called for any operations besides `__getitem__()`. This means that `get()` will, like normal dictionaries, return `None` as a default rather than using *default_factory*.

defaultdict objects support the following instance variable:

default_factory

This attribute is used by the *__missing__()* method; it is initialized from the first argument to the constructor, if present, or to `None`, if absent.

Changed in version 3.9: Added merge (`|`) and update (`|=`) operators, specified in **PEP 584**.

defaultdict Examples

Using *list* as the *default_factory*, it is easy to group a sequence of key-value pairs into a dictionary of lists:

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

When each key is encountered for the first time, it is not already in the mapping; so an entry is automatically created using the *default_factory* function which returns an empty *list*. The `list.append()` operation then attaches the value to the new list. When keys are encountered again, the look-up proceeds normally (returning the list for that key) and the `list.append()` operation adds another value to the list. This technique is simpler and faster than an equivalent technique using *dict.setdefault()*:

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Setting the *default_factory* to *int* makes the *defaultdict* useful for counting (like a bag or multiset in other languages):

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

When a letter is first encountered, it is missing from the mapping, so the *default_factory* function calls *int()* to supply a default count of zero. The increment operation then builds up the count for each letter.

The function *int()* which always returns zero is just a special case of constant functions. A faster and more flexible way to create constant functions is to use a lambda function which can supply any constant value (not just zero):

```
>>> def constant_factory(value):
...     return lambda: value
...
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

Setting the *default_factory* to *set* makes the *defaultdict* useful for building a dictionary of sets:

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> sorted(d.items())
[('blue', {2, 4}), ('red', {1, 3})]
```

8.4.5 `namedtuple()` Factory Function for Tuples with Named Fields

Named tuples assign meaning to each position in a tuple and allow for more readable, self-documenting code. They can be used wherever regular tuples are used, and they add the ability to access fields by name instead of position index.

`collections.`**`namedtuple`**(*typename*, *field_names*, *, *rename=False*, *defaults=None*, *module=None*)

Returns a new tuple subclass named *typename*. The new subclass is used to create tuple-like objects that have fields accessible by attribute lookup as well as being indexable and iterable. Instances of the subclass also have a helpful docstring (with typename and field_names) and a helpful `__repr__()` method which lists the tuple contents in a `name=value` format.

The *field_names* are a sequence of strings such as `['x', 'y']`. Alternatively, *field_names* can be a single string with each fieldname separated by whitespace and/or commas, for example `'x y'` or `'x, y'`.

Any valid Python identifier may be used for a fieldname except for names starting with an underscore. Valid identifiers consist of letters, digits, and underscores but do not start with a digit or underscore and cannot be a *keyword* such as *class*, *for*, *return*, *global*, *pass*, or *raise*.

If *rename* is true, invalid fieldnames are automatically replaced with positional names. For example, `['abc', 'def', 'ghi', 'abc']` is converted to `['abc', '_1', 'ghi', '_3']`, eliminating the keyword `def` and the duplicate fieldname `abc`.

defaults can be `None` or an *iterable* of default values. Since fields with a default value must come after any fields without a default, the *defaults* are applied to the rightmost parameters. For example, if the fieldnames are `['x', 'y', 'z']` and the defaults are `(1, 2)`, then `x` will be a required argument, `y` will default to `1`, and `z` will default to `2`.

If *module* is defined, the `__module__` attribute of the named tuple is set to that value.

Named tuple instances do not have per-instance dictionaries, so they are lightweight and require no more memory than regular tuples.

To support pickling, the named tuple class should be assigned to a variable that matches *typename*.

Changed in version 3.1: Added support for *rename*.

Changed in version 3.6: The *verbose* and *rename* parameters became *keyword-only arguments*.

Changed in version 3.6: Added the *module* parameter.

Changed in version 3.7: Removed the *verbose* parameter and the `_source` attribute.

Changed in version 3.7: Added the *defaults* parameter and the `_field_defaults` attribute.

```
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)     # instantiate with positional or keyword arguments
>>> p[0] + p[1]             # indexable like the plain tuple (11, 22)
33
>>> x, y = p                # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y               # fields also accessible by name
33
>>> p                       # readable __repr__ with a name=value style
Point(x=11, y=22)
```

Named tuples are especially useful for assigning field names to result tuples returned by the *csv* or *sqlite3* modules:

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade
↪')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
    print(emp.name, emp.title)

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print(emp.name, emp.title)
```

In addition to the methods inherited from tuples, named tuples support three additional methods and two attributes. To prevent conflicts with field names, the method and attribute names start with an underscore.

classmethod somenamedtuple.**_make**(*iterable*)

Class method that makes a new instance from an existing sequence or iterable.

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

somenamedtuple.**_asdict**()

Return a new *dict* which maps field names to their corresponding values:

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
{'x': 11, 'y': 22}
```

Changed in version 3.1: Returns an *OrderedDict* instead of a regular *dict*.

Changed in version 3.8: Returns a regular *dict* instead of an *OrderedDict*. As of Python 3.7, regular dicts are guaranteed to be ordered. If the extra features of *OrderedDict* are required, the suggested remediation is to cast the result to the desired type: `OrderedDict(nt._asdict())`.

somenamedtuple.**_replace**(***kwargs*)

Return a new instance of the named tuple replacing specified fields with new values:

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)

>>> for partnum, record in inventory.items():
...     inventory[partnum] = record._replace(price=newprices[partnum],␣
↪timestamp=time.now())
```

Named tuples are also supported by generic function *copy.replace()*.

Changed in version 3.13: Raise *TypeError* instead of *ValueError* for invalid keyword arguments.

somenamedtuple.**_fields**

Tuple of strings listing the field names. Useful for introspection and for creating new named tuple types from existing named tuples.

```
>>> p._fields            # view the field names
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

somenamedtuple.**_field_defaults**

Dictionary mapping field names to default values.

```
>>> Account = namedtuple('Account', ['type', 'balance'], defaults=[0])
>>> Account._field_defaults
{'balance': 0}
>>> Account('premium')
Account(type='premium', balance=0)
```

To retrieve a field whose name is stored in a string, use the *getattr()* function:

```
>>> getattr(p, 'x')
11
```

To convert a dictionary to a named tuple, use the double-star-operator (as described in tut-unpacking-arguments):

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

Since a named tuple is a regular Python class, it is easy to add or change functionality with a subclass. Here is how to add a calculated field and a fixed-width print format:

```
>>> class Point(namedtuple('Point', ['x', 'y'])):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...         return 'Point: x=%6.3f  y=%6.3f  hypot=%6.3f' % (self.x, self.y, self.
↪hypot)

>>> for p in Point(3, 4), Point(14, 5/7):
```

(continues on next page)

(continued from previous page)

```
...     print(p)
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018
```

The subclass shown above sets `__slots__` to an empty tuple. This helps keep memory requirements low by preventing the creation of instance dictionaries.

Subclassing is not useful for adding new, stored fields. Instead, simply create a new named tuple type from the *_fields* attribute:

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

Docstrings can be customized by making direct assignments to the `__doc__` fields:

```
>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection'
>>> Book.id.__doc__ = '13-digit ISBN'
>>> Book.title.__doc__ = 'Title of first printing'
>>> Book.authors.__doc__ = 'List of authors sorted by last name'
```

Changed in version 3.5: Property docstrings became writeable.

See also

- See *typing.NamedTuple* for a way to add type hints for named tuples. It also provides an elegant notation using the `class` keyword:

  ```
  class Component(NamedTuple):
      part_number: int
      weight: float
      description: Optional[str] = None
  ```

- See *types.SimpleNamespace()* for a mutable namespace based on an underlying dictionary instead of a tuple.
- The *dataclasses* module provides a decorator and functions for automatically adding generated special methods to user-defined classes.

8.4.6 `OrderedDict` objects

Ordered dictionaries are just like regular dictionaries but have some extra capabilities relating to ordering operations. They have become less important now that the built-in *dict* class gained the ability to remember insertion order (this new behavior became guaranteed in Python 3.7).

Some differences from *dict* still remain:

- The regular *dict* was designed to be very good at mapping operations. Tracking insertion order was secondary.
- The *OrderedDict* was designed to be good at reordering operations. Space efficiency, iteration speed, and the performance of update operations were secondary.
- The *OrderedDict* algorithm can handle frequent reordering operations better than *dict*. As shown in the recipes below, this makes it suitable for implementing various kinds of LRU caches.
- The equality operation for *OrderedDict* checks for matching order.

A regular *dict* can emulate the order sensitive equality test with `p == q and all(k1 == k2 for k1, k2 in zip(p, q))`.

- The `popitem()` method of *OrderedDict* has a different signature. It accepts an optional argument to specify which item is popped.

  A regular *dict* can emulate OrderedDict's `od.popitem(last=True)` with `d.popitem()` which is guaranteed to pop the rightmost (last) item.

  A regular *dict* can emulate OrderedDict's `od.popitem(last=False)` with `(k := next(iter(d)), d.pop(k))` which will return and remove the leftmost (first) item if it exists.

- *OrderedDict* has a `move_to_end()` method to efficiently reposition an element to an endpoint.

  A regular *dict* can emulate OrderedDict's `od.move_to_end(k, last=True)` with `d[k] = d.pop(k)` which will move the key and its associated value to the rightmost (last) position.

  A regular *dict* does not have an efficient equivalent for OrderedDict's `od.move_to_end(k, last=False)` which moves the key and its associated value to the leftmost (first) position.

- Until Python 3.8, *dict* lacked a `__reversed__()` method.

class collections.**OrderedDict**([*items*])

Return an instance of a *dict* subclass that has methods specialized for rearranging dictionary order.

Added in version 3.1.

popitem(*last=True*)

The *popitem()* method for ordered dictionaries returns and removes a (key, value) pair. The pairs are returned in LIFO order if *last* is true or FIFO (first-in, first-out) order if false.

move_to_end(*key*, *last=True*)

Move an existing *key* to either end of an ordered dictionary. The item is moved to the right end if *last* is true (the default) or to the beginning if *last* is false. Raises *KeyError* if the *key* does not exist:

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d)
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d)
'bacde'
```

Added in version 3.2.

In addition to the usual mapping methods, ordered dictionaries also support reverse iteration using *reversed()*.

Equality tests between *OrderedDict* objects are order-sensitive and are implemented as `list(od1.items())==list(od2.items())`. Equality tests between *OrderedDict* objects and other *Mapping* objects are order-insensitive like regular dictionaries. This allows *OrderedDict* objects to be substituted anywhere a regular dictionary is used.

Changed in version 3.5: The items, keys, and values *views* of *OrderedDict* now support reverse iteration using *reversed()*.

Changed in version 3.6: With the acceptance of **PEP 468**, order is retained for keyword arguments passed to the *OrderedDict* constructor and its `update()` method.

Changed in version 3.9: Added merge (`|`) and update (`|=`) operators, specified in **PEP 584**.

`OrderedDict` Examples and Recipes

It is straightforward to create an ordered dictionary variant that remembers the order the keys were *last* inserted. If a new entry overwrites an existing entry, the original insertion position is changed and moved to the end:

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        self.move_to_end(key)
```

An *OrderedDict* would also be useful for implementing variants of *functools.lru_cache()*:

```
from collections import OrderedDict
from time import time

class TimeBoundedLRU:
    "LRU Cache that invalidates and refreshes old entries."

    def __init__(self, func, maxsize=128, maxage=30):
        self.cache = OrderedDict()      # { args : (timestamp, result)}
        self.func = func
        self.maxsize = maxsize
        self.maxage = maxage

    def __call__(self, *args):
        if args in self.cache:
            self.cache.move_to_end(args)
            timestamp, result = self.cache[args]
            if time() - timestamp <= self.maxage:
                return result
        result = self.func(*args)
        self.cache[args] = time(), result
        if len(self.cache) > self.maxsize:
            self.cache.popitem(last=False)
        return result
```

```
class MultiHitLRUCache:
    """ LRU cache that defers caching a result until
        it has been requested multiple times.

        To avoid flushing the LRU cache with one-time requests,
        we don't cache until a request has been made more than once.

    """

    def __init__(self, func, maxsize=128, maxrequests=4096, cache_after=1):
        self.requests = OrderedDict()   # { uncached_key : request_count }
        self.cache = OrderedDict()      # { cached_key : function_result }
        self.func = func
        self.maxrequests = maxrequests  # max number of uncached requests
        self.maxsize = maxsize          # max number of stored return values
        self.cache_after = cache_after

    def __call__(self, *args):
        if args in self.cache:
```

(continues on next page)

(continued from previous page)

```
            self.cache.move_to_end(args)
            return self.cache[args]
        result = self.func(*args)
        self.requests[args] = self.requests.get(args, 0) + 1
        if self.requests[args] <= self.cache_after:
            self.requests.move_to_end(args)
            if len(self.requests) > self.maxrequests:
                self.requests.popitem(last=False)
        else:
            self.requests.pop(args, None)
            self.cache[args] = result
            if len(self.cache) > self.maxsize:
                self.cache.popitem(last=False)
        return result
```

8.4.7 `UserDict` objects

The class, *UserDict* acts as a wrapper around dictionary objects. The need for this class has been partially supplanted by the ability to subclass directly from *dict*; however, this class can be easier to work with because the underlying dictionary is accessible as an attribute.

class collections.**UserDict**([*initialdata*])

Class that simulates a dictionary. The instance's contents are kept in a regular dictionary, which is accessible via the *data* attribute of *UserDict* instances. If *initialdata* is provided, *data* is initialized with its contents; note that a reference to *initialdata* will not be kept, allowing it to be used for other purposes.

In addition to supporting the methods and operations of mappings, *UserDict* instances provide the following attribute:

data

A real dictionary used to store the contents of the *UserDict* class.

8.4.8 `UserList` objects

This class acts as a wrapper around list objects. It is a useful base class for your own list-like classes which can inherit from them and override existing methods or add new ones. In this way, one can add new behaviors to lists.

The need for this class has been partially supplanted by the ability to subclass directly from *list*; however, this class can be easier to work with because the underlying list is accessible as an attribute.

class collections.**UserList**([*list*])

Class that simulates a list. The instance's contents are kept in a regular list, which is accessible via the *data* attribute of *UserList* instances. The instance's contents are initially set to a copy of *list*, defaulting to the empty list `[]`. *list* can be any iterable, for example a real Python list or a *UserList* object.

In addition to supporting the methods and operations of mutable sequences, *UserList* instances provide the following attribute:

data

A real *list* object used to store the contents of the *UserList* class.

Subclassing requirements: Subclasses of *UserList* are expected to offer a constructor which can be called with either no arguments or one argument. List operations which return a new sequence attempt to create an instance of the actual implementation class. To do so, it assumes that the constructor can be called with a single parameter, which is a sequence object used as a data source.

If a derived class does not wish to comply with this requirement, all of the special methods supported by this class will need to be overridden; please consult the sources for information about the methods which need to be provided in that case.

8.4.9 `UserString` objects

The class, *UserString* acts as a wrapper around string objects. The need for this class has been partially supplanted by the ability to subclass directly from *str*; however, this class can be easier to work with because the underlying string is accessible as an attribute.

class collections.**UserString**(*seq*)

Class that simulates a string object. The instance's content is kept in a regular string object, which is accessible via the *data* attribute of *UserString* instances. The instance's contents are initially set to a copy of *seq*. The *seq* argument can be any object which can be converted into a string using the built-in *str()* function.

In addition to supporting the methods and operations of strings, *UserString* instances provide the following attribute:

data

A real *str* object used to store the contents of the *UserString* class.

Changed in version 3.5: New methods `__getnewargs__`, `__rmod__`, `casefold`, `format_map`, `isprintable`, and `maketrans`.

8.5 `collections.abc` — Abstract Base Classes for Containers

Added in version 3.3: Formerly, this module was part of the *collections* module.

Source code: Lib/_collections_abc.py

This module provides *abstract base classes* that can be used to test whether a class provides a particular interface; for example, whether it is *hashable* or whether it is a *mapping*.

An *issubclass()* or *isinstance()* test for an interface works in one of three ways.

1) A newly written class can inherit directly from one of the abstract base classes. The class must supply the required abstract methods. The remaining mixin methods come from inheritance and can be overridden if desired. Other methods may be added as needed:

```
class C(Sequence):                      # Direct inheritance
    def __init__(self): ...             # Extra method not required by the ABC
    def __getitem__(self, index):  ...  # Required abstract method
    def __len__(self):  ...             # Required abstract method
    def count(self, value): ...         # Optionally override a mixin method
```

```
>>> issubclass(C, Sequence)
True
>>> isinstance(C(), Sequence)
True
```

2) Existing classes and built-in classes can be registered as "virtual subclasses" of the ABCs. Those classes should define the full API including all of the abstract methods and all of the mixin methods. This lets users rely on *issubclass()* or *isinstance()* tests to determine whether the full interface is supported. The exception to this rule is for methods that are automatically inferred from the rest of the API:

```
class D:                                    # No inheritance
    def __init__(self): ...                 # Extra method not required by the ABC
    def __getitem__(self, index):  ...      # Abstract method
    def __len__(self):  ...                 # Abstract method
    def count(self, value): ...             # Mixin method
    def index(self, value): ...             # Mixin method

Sequence.register(D)                        # Register instead of inherit
```

```
>>> issubclass(D, Sequence)
True
>>> isinstance(D(), Sequence)
True
```

In this example, class `D` does not need to define `__contains__`, `__iter__`, and `__reversed__` because the in-operator, the *iteration* logic, and the `reversed()` function automatically fall back to using `__getitem__` and `__len__`.

3) Some simple interfaces are directly recognizable by the presence of the required methods (unless those methods have been set to `None`):

```
class E:
    def __iter__(self): ...
    def __next__(self): ...
```

```
>>> issubclass(E, Iterable)
True
>>> isinstance(E(), Iterable)
True
```

Complex interfaces do not support this last technique because an interface is more than just the presence of method names. Interfaces specify semantics and relationships between methods that cannot be inferred solely from the presence of specific method names. For example, knowing that a class supplies `__getitem__`, `__len__`, and `__iter__` is insufficient for distinguishing a `Sequence` from a `Mapping`.

Added in version 3.9: These abstract classes now support `[]`. See *Generic Alias Type* and **PEP 585**.

8.5.1 Collections Abstract Base Classes

The collections module offers the following *ABCs*:

ABC	Inherits from	Abstract Methods	Mixin Methods
Container[1]		`__contains__`	
Hashable[Page 283, 1]		`__hash__`	
Iterable[Page 283, 12]		`__iter__`	
Iterator[Page 283, 1]	*Iterable*	`__next__`	`__iter__`
Reversible[Page 283, 1]	*Iterable*	`__reversed__`	
Generator[Page 283, 1]	*Iterator*	`send`, `throw`	`close`, `__iter__`, `__next__`
Sized[Page 283, 1]		`__len__`	
Callable[Page 283, 1]		`__call__`	
Collection[Page 283, 1]	*Sized*, *Iterable*, *Container*	`__contains__`, `__iter__`, `__len__`	
Sequence	*Reversible*, *Collection*	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `index`, and `count`
MutableSequence	*Sequence*	`__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `insert`	Inherited *Sequence* methods and `append`, `clear`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`
ByteString	*Sequence*	`__getitem__`, `__len__`	Inherited *Sequence* methods
Set	*Collection*	`__contains__`, `__iter__`, `__len__`	`__le__`, `__lt__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`, `__and__`, `__or__`, `__sub__`, `__xor__`, and `isdisjoint`
MutableSet	*Set*	`__contains__`, `__iter__`, `__len__`, `add`, `discard`	Inherited *Set* methods and `clear`, `pop`, `remove`, `__ior__`, `__iand__`, `__ixor__`, and `__isub__`
Mapping	*Collection*	`__getitem__`, `__iter__`, `__len__`	`__contains__`, `keys`, `items`, `values`, `get`, `__eq__`, and `__ne__`
MutableMapping	*Mapping*	`__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__`	Inherited *Mapping* methods and `pop`, `popitem`, `clear`, `update`, and `setdefault`
MappingView	*Sized*		`__len__`
ItemsView	*MappingView*, *Set*		`__contains__`, `__iter__`
KeysView	*MappingView*, *Set*		`__contains__`, `__iter__`
ValuesView	*MappingView*, *Collection*		`__contains__`, `__iter__`
Awaitable[Page 283, 1]		`__await__`	
Coroutine[Page 283, 1]	*Awaitable*	`send`, `throw`	`close`
AsyncIterable[Page 283, 1]		`__aiter__`	
AsyncIterator[Page 283, 1]	*AsyncIterable*	`__anext__`	`__aiter__`

8.5.2 Collections Abstract Base Classes – Detailed Descriptions

class collections.abc.**Container**

ABC for classes that provide the `__contains__()` method.

class collections.abc.**Hashable**

ABC for classes that provide the `__hash__()` method.

class collections.abc.**Sized**

ABC for classes that provide the `__len__()` method.

class collections.abc.**Callable**

ABC for classes that provide the `__call__()` method.

class collections.abc.**Iterable**

ABC for classes that provide the *`__iter__()`* method.

Checking `isinstance(obj, Iterable)` detects classes that are registered as *`Iterable`* or that have an *`__iter__()`* method, but it does not detect classes that iterate with the `__getitem__()` method. The only reliable way to determine whether an object is *iterable* is to call `iter(obj)`.

class collections.abc.**Collection**

ABC for sized iterable container classes.

Added in version 3.6.

class collections.abc.**Iterator**

ABC for classes that provide the *`__iter__()`* and *`__next__()`* methods. See also the definition of *iterator*.

class collections.abc.**Reversible**

ABC for iterable classes that also provide the `__reversed__()` method.

Added in version 3.6.

class collections.abc.**Generator**

ABC for *generator* classes that implement the protocol defined in **PEP 342** that extends *iterators* with the `send()`, `throw()` and `close()` methods.

Added in version 3.5.

class collections.abc.**Sequence**

class collections.abc.**MutableSequence**

class collections.abc.**ByteString**

ABCs for read-only and mutable *sequences*.

Implementation note: Some of the mixin methods, such as *`__iter__()`*, `__reversed__()` and `index()`, make repeated calls to the underlying `__getitem__()` method. Consequently, if `__getitem__()` is implemented with constant access speed, the mixin methods will have linear performance; however, if the underlying method is linear (as it would be with a linked list), the mixins will have quadratic performance and will likely need to be overridden.

Changed in version 3.5: The index() method added support for *stop* and *start* arguments.

[1] These ABCs override *`__subclasshook__()`* to support testing an interface by verifying the required methods are present and have not been set to *`None`*. This only works for simple interfaces. More complex interfaces require registration or direct subclassing.

[2] Checking `isinstance(obj, Iterable)` detects classes that are registered as *`Iterable`* or that have an *`__iter__()`* method, but it does not detect classes that iterate with the `__getitem__()` method. The only reliable way to determine whether an object is *iterable* is to call `iter(obj)`.

Deprecated since version 3.12, will be removed in version 3.14: The *ByteString* ABC has been deprecated. For use in typing, prefer a union, like `bytes | bytearray`, or *collections.abc.Buffer*. For use as an ABC, prefer *Sequence* or *collections.abc.Buffer*.

class collections.abc.**Set**

class collections.abc.**MutableSet**

ABCs for read-only and mutable *sets*.

class collections.abc.**Mapping**

class collections.abc.**MutableMapping**

ABCs for read-only and mutable *mappings*.

class collections.abc.**MappingView**

class collections.abc.**ItemsView**

class collections.abc.**KeysView**

class collections.abc.**ValuesView**

ABCs for mapping, items, keys, and values *views*.

class collections.abc.**Awaitable**

ABC for *awaitable* objects, which can be used in `await` expressions. Custom implementations must provide the `__await__()` method.

Coroutine objects and instances of the *Coroutine* ABC are all instances of this ABC.

> **Note**
>
> In CPython, generator-based coroutines (*generators* decorated with *@types.coroutine*) are *awaitables*, even though they do not have an `__await__()` method. Using `isinstance(gencoro, Awaitable)` for them will return `False`. Use *inspect.isawaitable()* to detect them.

Added in version 3.5.

class collections.abc.**Coroutine**

ABC for *coroutine* compatible classes. These implement the following methods, defined in coroutine-objects: `send()`, `throw()`, and `close()`. Custom implementations must also implement `__await__()`. All *Coroutine* instances are also instances of *Awaitable*.

> **Note**
>
> In CPython, generator-based coroutines (*generators* decorated with *@types.coroutine*) are *awaitables*, even though they do not have an `__await__()` method. Using `isinstance(gencoro, Coroutine)` for them will return `False`. Use *inspect.isawaitable()* to detect them.

Added in version 3.5.

class collections.abc.**AsyncIterable**

ABC for classes that provide an `__aiter__` method. See also the definition of *asynchronous iterable*.

Added in version 3.5.

class collections.abc.**AsyncIterator**

ABC for classes that provide `__aiter__` and `__anext__` methods. See also the definition of *asynchronous iterator*.

Added in version 3.5.

class collections.abc.**AsyncGenerator**

ABC for *asynchronous generator* classes that implement the protocol defined in **PEP 525** and **PEP 492**.

Added in version 3.6.

class collections.abc.**Buffer**

ABC for classes that provide the `__buffer__()` method, implementing the buffer protocol. See **PEP 688**.

Added in version 3.12.

8.5.3 Examples and Recipes

ABCs allow us to ask classes or instances if they provide particular functionality, for example:

```
size = None
if isinstance(myvar, collections.abc.Sized):
    size = len(myvar)
```

Several of the ABCs are also useful as mixins that make it easier to develop classes supporting container APIs. For example, to write a class supporting the full *Set* API, it is only necessary to supply the three underlying abstract methods: `__contains__()`, *__iter__()*, and `__len__()`. The ABC supplies the remaining methods such as `__and__()` and *isdisjoint()*:

```
class ListBasedSet(collections.abc.Set):
    ''' Alternate set implementation favoring space over speed
        and not requiring the set elements to be hashable. '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)

    def __iter__(self):
        return iter(self.elements)

    def __contains__(self, value):
        return value in self.elements

    def __len__(self):
        return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2            # The __and__() method is supported automatically
```

Notes on using *Set* and *MutableSet* as a mixin:

(1) Since some set operations create new sets, the default mixin methods need a way to create new instances from an *iterable*. The class constructor is assumed to have a signature in the form `ClassName(iterable)`. That assumption is factored-out to an internal *classmethod* called `_from_iterable()` which calls `cls(iterable)` to produce a new set. If the *Set* mixin is being used in a class with a different constructor signature, you will need to override `_from_iterable()` with a classmethod or regular method that can construct new instances from an iterable argument.

(2) To override the comparisons (presumably for speed, as the semantics are fixed), redefine `__le__()` and `__ge__()`, then the other operations will automatically follow suit.

(3) The *Set* mixin provides a `_hash()` method to compute a hash value for the set; however, `__hash__()` is not defined because not all sets are *hashable* or immutable. To add set hashability using mixins, inherit from both *Set()* and *Hashable()*, then define `__hash__ = Set._hash`.

See also

- OrderedSet recipe for an example built on *MutableSet*.
- For more about ABCs, see the *abc* module and **PEP 3119**.

8.6 `heapq` — Heap queue algorithm

Source code: Lib/heapq.py

This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

Heaps are binary trees for which every parent node has a value less than or equal to any of its children. We refer to this condition as the heap invariant.

This implementation uses arrays for which `heap[k] <= heap[2*k+1]` and `heap[k] <= heap[2*k+2]` for all *k*, counting elements from zero. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that its smallest element is always the root, `heap[0]`.

The API below differs from textbook heap algorithms in two aspects: (a) We use zero-based indexing. This makes the relationship between the index for a node and the indexes for its children slightly less obvious, but is more suitable since Python uses zero-based indexing. (b) Our pop method returns the smallest item, not the largest (called a "min heap" in textbooks; a "max heap" is more common in texts because of its suitability for in-place sorting).

These two make it possible to view the heap as a regular Python list without surprises: `heap[0]` is the smallest item, and `heap.sort()` maintains the heap invariant!

To create a heap, use a list initialized to `[]`, or you can transform a populated list into a heap via function *heapify()*.

The following functions are provided:

`heapq.`**`heappush`**(*heap*, *item*)

Push the value *item* onto the *heap*, maintaining the heap invariant.

`heapq.`**`heappop`**(*heap*)

Pop and return the smallest item from the *heap*, maintaining the heap invariant. If the heap is empty, *IndexError* is raised. To access the smallest item without popping it, use `heap[0]`.

`heapq.`**`heappushpop`**(*heap*, *item*)

Push *item* on the heap, then pop and return the smallest item from the *heap*. The combined action runs more efficiently than *heappush()* followed by a separate call to *heappop()*.

`heapq.`**`heapify`**(*x*)

Transform list *x* into a heap, in-place, in linear time.

`heapq.`**`heapreplace`**(*heap*, *item*)

Pop and return the smallest item from the *heap*, and also push the new *item*. The heap size doesn't change. If the heap is empty, *IndexError* is raised.

This one step operation is more efficient than a *heappop()* followed by *heappush()* and can be more appropriate when using a fixed-size heap. The pop/push combination always returns an element from the heap and replaces it with *item*.

The value returned may be larger than the *item* added. If that isn't desired, consider using `heappushpop()` instead. Its push/pop combination returns the smaller of the two values, leaving the larger value on the heap.

The module also offers three general purpose functions based on heaps.

`heapq.merge`(**iterables*, *key=None*, *reverse=False*)

Merge multiple sorted inputs into a single sorted output (for example, merge timestamped entries from multiple log files). Returns an *iterator* over the sorted values.

Similar to `sorted(itertools.chain(*iterables))` but returns an iterable, does not pull the data into memory all at once, and assumes that each of the input streams is already sorted (smallest to largest).

Has two optional arguments which must be specified as keyword arguments.

key specifies a *key function* of one argument that is used to extract a comparison key from each input element. The default value is `None` (compare the elements directly).

reverse is a boolean value. If set to `True`, then the input elements are merged as if each comparison were reversed. To achieve behavior similar to `sorted(itertools.chain(*iterables), reverse=True)`, all iterables must be sorted from largest to smallest.

Changed in version 3.5: Added the optional *key* and *reverse* parameters.

`heapq.nlargest`(*n*, *iterable*, *key=None*)

Return a list with the *n* largest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). Equivalent to: `sorted(iterable, key=key, reverse=True)[:n]`.

`heapq.nsmallest`(*n*, *iterable*, *key=None*)

Return a list with the *n* smallest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). Equivalent to: `sorted(iterable, key=key)[:n]`.

The latter two functions perform best for smaller values of *n*. For larger values, it is more efficient to use the `sorted()` function. Also, when `n==1`, it is more efficient to use the built-in `min()` and `max()` functions. If repeated usage of these functions is required, consider turning the iterable into an actual heap.

8.6.1 Basic Examples

A heapsort can be implemented by pushing all values onto a heap and then popping off the smallest values one at a time:

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This is similar to `sorted(iterable)`, but unlike `sorted()`, this implementation is not stable.

Heap elements can be tuples. This is useful for assigning comparison values (such as task priorities) alongside the main record being tracked:

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
```

(continues on next page)

(continued from previous page)

```
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

8.6.2 Priority Queue Implementation Notes

A priority queue is common use for a heap, and it presents several implementation challenges:

- Sort stability: how do you get two tasks with equal priorities to be returned in the order they were originally added?
- Tuple comparison breaks for (priority, task) pairs if the priorities are equal and the tasks do not have a default comparison order.
- If the priority of a task changes, how do you move it to a new position in the heap?
- Or if a pending task needs to be deleted, how do you find it and remove it from the queue?

A solution to the first two challenges is to store entries as 3-element list including the priority, an entry count, and the task. The entry count serves as a tie-breaker so that two tasks with the same priority are returned in the order they were added. And since no two entry counts are the same, the tuple comparison will never attempt to directly compare two tasks.

Another solution to the problem of non-comparable tasks is to create a wrapper class that ignores the task item and only compares the priority field:

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

The remaining challenges revolve around finding a pending task and making changes to its priority or removing it entirely. Finding a task can be done with a dictionary pointing to an entry in the queue.

Removing the entry or changing its priority is more difficult because it would break the heap structure invariants. So, a possible solution is to mark the entry as removed and add a new entry with the revised priority:

```
pq = []                         # list of entries arranged in a heap
entry_finder = {}               # mapping of tasks to entries
REMOVED = '<removed-task>'      # placeholder for a removed task
counter = itertools.count()     # unique sequence count

def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    'Mark an existing task as REMOVED.  Raise KeyError if not found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED
```

(continues on next page)

(continued from previous page)

```
def pop_task():
    'Remove and return the lowest priority task. Raise KeyError if empty.'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')
```

8.6.3 Theory

Heaps are arrays for which `a[k] <= a[2*k+1]` and `a[k] <= a[2*k+2]` for all *k*, counting elements from 0. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that `a[0]` is always its smallest element.

The strange invariant above is meant to be an efficient memory representation for a tournament. The numbers below are *k*, not `a[k]`:

```
                                0

               1                                2

       3               4                5               6

   7       8       9       10      11      12      13      14

15 16   17 18   19 20   21 22   23 24   25 26   27 28   29 30
```

In the tree above, each cell *k* is topping `2*k+1` and `2*k+2`. In a usual binary tournament we see in sports, each cell is the winner over the two cells it tops, and we can trace the winner down the tree to see all opponents s/he had. However, in many computer applications of such tournaments, we do not need to trace the history of a winner. To be more memory efficient, when a winner is promoted, we try to replace it by something else at a lower level, and the rule becomes that a cell and the two cells it tops contain three different items, but the top cell "wins" over the two topped cells.

If this heap invariant is protected at all time, index 0 is clearly the overall winner. The simplest algorithmic way to remove it and find the "next" winner is to move some loser (let's say cell 30 in the diagram above) into the 0 position, and then percolate this new 0 down the tree, exchanging values, until the invariant is re-established. This is clearly logarithmic on the total number of items in the tree. By iterating over all items, you get an $O(n \log n)$ sort.

A nice feature of this sort is that you can efficiently insert new items while the sort is going on, provided that the inserted items are not "better" than the last 0'th element you extracted. This is especially useful in simulation contexts, where the tree holds all incoming events, and the "win" condition means the smallest scheduled time. When an event schedules other events for execution, they are scheduled into the future, so they can easily go into the heap. So, a heap is a good structure for implementing schedulers (this is what I used for my MIDI sequencer :-).

Various structures for implementing schedulers have been extensively studied, and heaps are good for this, as they are reasonably speedy, the speed is almost constant, and the worst case is not much different than the average case. However, there are other representations which are more efficient overall, yet the worst cases might be terrible.

Heaps are also very useful in big disk sorts. You most probably all know that a big sort implies producing "runs" (which are pre-sorted sequences, whose size is usually related to the amount of CPU memory), followed by a merging passes for these runs, which merging is often very cleverly organised[1]. It is very important that the initial sort produces the longest

[1] The disk balancing algorithms which are current, nowadays, are more annoying than clever, and this is a consequence of the seeking capabilities of the disks. On devices which cannot seek, like big tape drives, the story was quite different, and one had to be very clever to ensure (far in advance)

runs possible. Tournaments are a good way to achieve that. If, using all the memory available to hold a tournament, you replace and percolate items that happen to fit the current run, you'll produce runs which are twice the size of the memory for random input, and much better for input fuzzily ordered.

Moreover, if you output the 0'th item on disk and get an input which may not fit in the current tournament (because the value "wins" over the last output value), it cannot fit in the heap, so the size of the heap decreases. The freed memory could be cleverly reused immediately for progressively building a second heap, which grows at exactly the same rate the first heap is melting. When the first heap completely vanishes, you switch heaps and start a new run. Clever and quite effective!

In a word, heaps are useful memory structures to know. I use them in a few applications, and I think it is good to keep a 'heap' module around. :-)

8.7 `bisect` — Array bisection algorithm

Source code: Lib/bisect.py

This module provides support for maintaining a list in sorted order without having to sort the list after each insertion. For long lists of items with expensive comparison operations, this can be an improvement over linear searches or frequent resorting.

The module is called *bisect* because it uses a basic bisection algorithm to do its work. Unlike other bisection tools that search for a specific value, the functions in this module are designed to locate an insertion point. Accordingly, the functions never call an `__eq__()` method to determine whether a value has been found. Instead, the functions only call the `__lt__()` method and will return an insertion point between values in an array.

The following functions are provided:

`bisect.`**`bisect_left`**(*a*, *x*, *lo=0*, *hi=len(a)*, ***, *key=None*)

Locate the insertion point for *x* in *a* to maintain sorted order. The parameters *lo* and *hi* may be used to specify a subset of the list which should be considered; by default the entire list is used. If *x* is already present in *a*, the insertion point will be before (to the left of) any existing entries. The return value is suitable for use as the first parameter to `list.insert()` assuming that *a* is already sorted.

The returned insertion point *ip* partitions the array *a* into two slices such that `all(elem < x for elem in a[lo : ip])` is true for the left slice and `all(elem >= x for elem in a[ip : hi])` is true for the right slice.

key specifies a *key function* of one argument that is used to extract a comparison key from each element in the array. To support searching complex records, the key function is not applied to the *x* value.

If *key* is `None`, the elements are compared directly and no key function is called.

Changed in version 3.10: Added the *key* parameter.

`bisect.`**`bisect_right`**(*a*, *x*, *lo=0*, *hi=len(a)*, ***, *key=None*)

`bisect.`**`bisect`**(*a*, *x*, *lo=0*, *hi=len(a)*, ***, *key=None*)

Similar to *bisect_left()*, but returns an insertion point which comes after (to the right of) any existing entries of *x* in *a*.

The returned insertion point *ip* partitions the array *a* into two slices such that `all(elem <= x for elem in a[lo : ip])` is true for the left slice and `all(elem > x for elem in a[ip : hi])` is true for the right slice.

that each tape movement will be the most effective possible (that is, will best participate at "progressing" the merge). Some tapes were even able to read backwards, and this was also used to avoid the rewinding time. Believe me, real good tape sorts were quite spectacular to watch! From all times, sorting has always been a Great Art! :-)

Changed in version 3.10: Added the *key* parameter.

bisect.**insort_left**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

Insert *x* in *a* in sorted order.

This function first runs *bisect_left()* to locate an insertion point. Next, it runs the `insert()` method on *a* to insert *x* at the appropriate position to maintain sort order.

To support inserting records in a table, the *key* function (if any) is applied to *x* for the search step but not for the insertion step.

Keep in mind that the *O*(log *n*) search is dominated by the slow *O*(*n*) insertion step.

Changed in version 3.10: Added the *key* parameter.

bisect.**insort_right**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

bisect.**insort**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

Similar to *insort_left()*, but inserting *x* in *a* after any existing entries of *x*.

This function first runs *bisect_right()* to locate an insertion point. Next, it runs the `insert()` method on *a* to insert *x* at the appropriate position to maintain sort order.

To support inserting records in a table, the *key* function (if any) is applied to *x* for the search step but not for the insertion step.

Keep in mind that the *O*(log *n*) search is dominated by the slow *O*(*n*) insertion step.

Changed in version 3.10: Added the *key* parameter.

8.7.1 Performance Notes

When writing time sensitive code using *bisect()* and *insort()*, keep these thoughts in mind:

- Bisection is effective for searching ranges of values. For locating specific values, dictionaries are more performant.
- The *insort()* functions are *O*(*n*) because the logarithmic search step is dominated by the linear time insertion step.
- The search functions are stateless and discard key function results after they are used. Consequently, if the search functions are used in a loop, the key function may be called again and again on the same array elements. If the key function isn't fast, consider wrapping it with *functools.cache()* to avoid duplicate computations. Alternatively, consider searching an array of precomputed keys to locate the insertion point (as shown in the examples section below).

See also

- Sorted Collections is a high performance module that uses *bisect* to managed sorted collections of data.
- The SortedCollection recipe uses bisect to build a full-featured collection class with straight-forward search methods and support for a key-function. The keys are precomputed to save unnecessary calls to the key function during searches.

8.7.2 Searching Sorted Lists

The above *bisect functions* are useful for finding insertion points but can be tricky or awkward to use for common searching tasks. The following five functions show how to transform them into the standard lookups for sorted lists:

```
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```

8.7.3 Examples

The `bisect()` function can be useful for numeric table lookups. This example uses `bisect()` to look up a letter grade for an exam score (say) based on a set of ordered numeric breakpoints: 90 and up is an 'A', 80 to 89 is a 'B', and so on:

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
...     i = bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

The `bisect()` and `insort()` functions also work with lists of tuples. The *key* argument can serve to extract the field used for ordering records in a table:

```
>>> from collections import namedtuple
>>> from operator import attrgetter
>>> from bisect import bisect, insort
>>> from pprint import pprint

>>> Movie = namedtuple('Movie', ('name', 'released', 'director'))

>>> movies = [
...     Movie('Jaws', 1975, 'Spielberg'),
...     Movie('Titanic', 1997, 'Cameron'),
...     Movie('The Birds', 1963, 'Hitchcock'),
...     Movie('Aliens', 1986, 'Cameron')
... ]

>>> # Find the first movie released after 1960
>>> by_year = attrgetter('released')
>>> movies.sort(key=by_year)
>>> movies[bisect(movies, 1960, key=by_year)]
Movie(name='The Birds', released=1963, director='Hitchcock')

>>> # Insert a movie while maintaining sort order
>>> romance = Movie('Love Story', 1970, 'Hiller')
>>> insort(movies, romance, key=by_year)
>>> pprint(movies)
[Movie(name='The Birds', released=1963, director='Hitchcock'),
 Movie(name='Love Story', released=1970, director='Hiller'),
 Movie(name='Jaws', released=1975, director='Spielberg'),
 Movie(name='Aliens', released=1986, director='Cameron'),
 Movie(name='Titanic', released=1997, director='Cameron')]
```

If the key function is expensive, it is possible to avoid repeated function calls by searching a list of precomputed keys to find the index of a record:

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])       # Or use operator.itemgetter(1).
>>> keys = [r[1] for r in data]         # Precompute a list of keys.
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

8.8 `array` — Efficient arrays of numeric values

This module defines an object type which can compactly represent an array of basic values: characters, integers, floating-point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a *type code*, which is a single character. The following type codes are defined:

Type code	C Type	Python Type	Minimum size in bytes	Notes
'b'	signed char	int	1	
'B'	unsigned char	int	1	
'u'	wchar_t	Unicode character	2	(1)
'w'	Py_UCS4	Unicode character	4	
'h'	signed short	int	2	
'H'	unsigned short	int	2	
'i'	signed int	int	2	
'I'	unsigned int	int	2	
'l'	signed long	int	4	
'L'	unsigned long	int	4	
'q'	signed long long	int	8	
'Q'	unsigned long long	int	8	
'f'	float	float	4	
'd'	double	float	8	

Notes:

(1) It can be 16 bits or 32 bits depending on the platform.

Changed in version 3.9: `array('u')` now uses `wchar_t` as C type instead of deprecated `Py_UNICODE`. This change doesn't affect its behavior because `Py_UNICODE` is alias of `wchar_t` since Python 3.3.

Deprecated since version 3.3, will be removed in version 3.16: Please migrate to `'w'` typecode.

The actual representation of values is determined by the machine architecture (strictly speaking, by the C implementation). The actual size can be accessed through the *array.itemsize* attribute.

The module defines the following item:

array.**typecodes**

A string with all available type codes.

The module defines the following type:

class array.**array**(*typecode*[, *initializer*])

A new array whose items are restricted by *typecode*, and initialized from the optional *initializer* value, which must be a *bytes* or *bytearray* object, a Unicode string, or iterable over elements of the appropriate type.

If given a *bytes* or *bytearray* object, the initializer is passed to the new array's *frombytes()* method; if given a Unicode string, the initializer is passed to the *fromunicode()* method; otherwise, the initializer's iterator is passed to the *extend()* method to add initial items to the array.

Array objects support the ordinary sequence operations of indexing, slicing, concatenation, and multiplication. When using slice assignment, the assigned value must be an array object with the same type code; in all other cases, *TypeError* is raised. Array objects also implement the buffer interface, and may be used wherever *bytes-like objects* are supported.

Raises an *auditing event* `array.__new__` with arguments `typecode`, `initializer`.

typecode

The typecode character used to create the array.

itemsize

The length in bytes of one array item in the internal representation.

append(*x*)

Append a new item with value *x* to the end of the array.

buffer_info()

Return a tuple `(address, length)` giving the current memory address and the length in elements of the buffer used to hold array's contents. The size of the memory buffer in bytes can be computed as `array.buffer_info()[1] * array.itemsize`. This is occasionally useful when working with low-level (and inherently unsafe) I/O interfaces that require memory addresses, such as certain `ioctl()` operations. The returned numbers are valid as long as the array exists and no length-changing operations are applied to it.

> **Note**
>
> When using array objects from code written in C or C++ (the only way to effectively make use of this information), it makes more sense to use the buffer interface supported by array objects. This method is maintained for backward compatibility and should be avoided in new code. The buffer interface is documented in bufferobjects.

byteswap()

"Byteswap" all items of the array. This is only supported for values which are 1, 2, 4, or 8 bytes in size; for other types of values, *RuntimeError* is raised. It is useful when reading data from a file written on a machine with a different byte order.

count(*x*)

Return the number of occurrences of *x* in the array.

extend(*iterable*)

Append items from *iterable* to the end of the array. If *iterable* is another array, it must have *exactly* the same type code; if not, *TypeError* will be raised. If *iterable* is not an array, it must be iterable and its elements must be the right type to be appended to the array.

frombytes(*buffer*)

Appends items from the *bytes-like object*, interpreting its content as an array of machine values (as if it had been read from a file using the *fromfile()* method).

Added in version 3.2: `fromstring()` is renamed to *frombytes()* for clarity.

fromfile(*f*, *n*)

Read *n* items (as machine values) from the *file object* *f* and append them to the end of the array. If less than *n* items are available, *EOFError* is raised, but the items that were available are still inserted into the array.

fromlist(*list*)

Append items from the list. This is equivalent to `for x in list: a.append(x)` except that if there is a type error, the array is unchanged.

fromunicode(*s*)

Extends this array with data from the given Unicode string. The array must have type code `'u'` or `'w'`; otherwise a *ValueError* is raised. Use `array.frombytes(unicodestring.encode(enc))` to append Unicode data to an array of some other type.

index(*x*[, *start*[, *stop*]])

Return the smallest *i* such that *i* is the index of the first occurrence of *x* in the array. The optional arguments *start* and *stop* can be specified to search for *x* within a subsection of the array. Raise *ValueError* if *x* is not found.

Changed in version 3.10: Added optional *start* and *stop* parameters.

insert(*i*, *x*)

Insert a new item with value *x* in the array before position *i*. Negative values are treated as being relative to the end of the array.

pop([*i*])

Removes the item with the index *i* from the array and returns it. The optional argument defaults to `-1`, so that by default the last item is removed and returned.

remove(*x*)

Remove the first occurrence of *x* from the array.

clear()

Remove all elements from the array.

Added in version 3.13.

reverse()

Reverse the order of the items in the array.

tobytes()

Convert the array to an array of machine values and return the bytes representation (the same sequence of bytes that would be written to a file by the *tofile()* method.)

Added in version 3.2: `tostring()` is renamed to *tobytes()* for clarity.

tofile(*f*)

Write all items (as machine values) to the *file object* *f*.

tolist()

Convert the array to an ordinary list with the same items.

tounicode()

Convert the array to a Unicode string. The array must have a type `'u'` or `'w'`; otherwise a *ValueError* is raised. Use `array.tobytes().decode(enc)` to obtain a Unicode string from an array of some other type.

The string representation of array objects has the form `array(typecode, initializer)`. The *initializer* is omitted if the array is empty, otherwise it is a Unicode string if the *typecode* is `'u'` or `'w'`, otherwise it is a list of numbers. The string representation is guaranteed to be able to be converted back to an array with the same type and value using *eval()*, so long as the *array* class has been imported using `from array import array`. Variables `inf` and `nan` must also be defined if it contains corresponding floating-point values. Examples:

```
array('l')
array('w', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14, -inf, nan])
```

See also

Module *struct*
Packing and unpacking of heterogeneous binary data.

NumPy
The NumPy package defines another array type.

8.9 weakref — Weak references

Source code: Lib/weakref.py

The *weakref* module allows the Python programmer to create *weak references* to objects.

In the following, the term *referent* means the object which is referred to by a weak reference.

A weak reference to an object is not enough to keep the object alive: when the only remaining references to a referent are weak references, *garbage collection* is free to destroy the referent and reuse its memory for something else. However, until the object is actually destroyed the weak reference may return the object even if there are no strong references to it.

A primary use for weak references is to implement caches or mappings holding large objects, where it's desired that a large object not be kept alive solely because it appears in a cache or mapping.

For example, if you have a number of large binary image objects, you may wish to associate a name with each. If you used a Python dictionary to map names to images, or images to names, the image objects would remain alive just because they appeared as values or keys in the dictionaries. The *WeakKeyDictionary* and *WeakValueDictionary* classes supplied by the *weakref* module are an alternative, using weak references to construct mappings that don't keep objects alive solely because they appear in the mapping objects. If, for example, an image object is a value in a *WeakValueDictionary*, then when the last remaining references to that image object are the weak references held by weak mappings, garbage collection can reclaim the object, and its corresponding entries in weak mappings are simply deleted.

WeakKeyDictionary and *WeakValueDictionary* use weak references in their implementation, setting up callback functions on the weak references that notify the weak dictionaries when a key or value has been reclaimed by garbage collection. *WeakSet* implements the *set* interface, but keeps weak references to its elements, just like a *WeakKeyDictionary* does.

finalize provides a straight forward way to register a cleanup function to be called when an object is garbage collected. This is simpler to use than setting up a callback function on a raw weak reference, since the module automatically ensures that the finalizer remains alive until the object is collected.

Most programs should find that using one of these weak container types or *finalize* is all they need – it's not usually necessary to create your own weak references directly. The low-level machinery is exposed by the *weakref* module for the benefit of advanced uses.

Not all objects can be weakly referenced. Objects which support weak references include class instances, functions written in Python (but not in C), instance methods, sets, frozensets, some *file objects*, *generators*, type objects, sockets, arrays, deques, regular expression pattern objects, and code objects.

Changed in version 3.2: Added support for thread.lock, threading.Lock, and code objects.

Several built-in types such as *list* and *dict* do not directly support weak references but can add support through subclassing:

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # this object is weak referenceable
```

CPython implementation detail: Other built-in types such as *tuple* and *int* do not support weak references even when subclassed.

Extension types can easily be made to support weak references; see weakref-support.

When `__slots__` are defined for a given type, weak reference support is disabled unless a `'__weakref__'` string is also present in the sequence of strings in the `__slots__` declaration. See __slots__ documentation for details.

class weakref.**ref**(*object*[, *callback*])

Return a weak reference to *object*. The original object can be retrieved by calling the reference object if the referent is still alive; if the referent is no longer alive, calling the reference object will cause *None* to be returned. If *callback* is provided and not *None*, and the returned weakref object is still alive, the callback will be called when the object is about to be finalized; the weak reference object will be passed as the only parameter to the callback; the referent will no longer be available.

It is allowable for many weak references to be constructed for the same object. Callbacks registered for each weak reference will be called from the most recently registered callback to the oldest registered callback.

Exceptions raised by the callback will be noted on the standard error output, but cannot be propagated; they are handled in exactly the same way as exceptions raised from an object's `__del__()` method.

Weak references are *hashable* if the *object* is hashable. They will maintain their hash value even after the *object* was deleted. If *hash()* is called the first time only after the *object* was deleted, the call will raise *TypeError*.

Weak references support tests for equality, but not ordering. If the referents are still alive, two references have the same equality relationship as their referents (regardless of the *callback*). If either referent has been deleted, the references are equal only if the reference objects are the same object.

This is a subclassable type rather than a factory function.

__callback__

This read-only attribute returns the callback currently associated to the weakref. If there is no callback or if the referent of the weakref is no longer alive then this attribute will have value `None`.

Changed in version 3.4: Added the *__callback__* attribute.

weakref.**proxy**(*object*[, *callback*])

Return a proxy to *object* which uses a weak reference. This supports use of the proxy in most contexts instead of requiring the explicit dereferencing used with weak reference objects. The returned object will have a type of either `ProxyType` or `CallableProxyType`, depending on whether *object* is callable. Proxy objects are not *hashable* regardless of the referent; this avoids a number of problems related to their fundamentally mutable nature, and prevents their use as dictionary keys. *callback* is the same as the parameter of the same name to the *ref()* function.

Accessing an attribute of the proxy object after the referent is garbage collected raises *ReferenceError*.

Changed in version 3.8: Extended the operator support on proxy objects to include the matrix multiplication operators `@` and `@=`.

weakref.**getweakrefcount**(*object*)

Return the number of weak references and proxies which refer to *object*.

weakref.**getweakrefs**(*object*)

Return a list of all weak reference and proxy objects which refer to *object*.

class weakref.**WeakKeyDictionary**([*dict*])

Mapping class that references keys weakly. Entries in the dictionary will be discarded when there is no longer a strong reference to the key. This can be used to associate additional data with an object owned by other parts of an application without adding attributes to those objects. This can be especially useful with objects that override attribute accesses.

Note that when a key with equal value to an existing key (but not equal identity) is inserted into the dictionary, it replaces the value but does not replace the existing key. Due to this, when the reference to the original key is deleted, it also deletes the entry in the dictionary:

```
>>> class T(str): pass
...
```

(continues on next page)

(continued from previous page)

```
>>> k1, k2 = T(), T()
>>> d = weakref.WeakKeyDictionary()
>>> d[k1] = 1   # d = {k1: 1}
>>> d[k2] = 2   # d = {k1: 2}
>>> del k1      # d = {}
```

A workaround would be to remove the key prior to reassignment:

```
>>> class T(str): pass
...
>>> k1, k2 = T(), T()
>>> d = weakref.WeakKeyDictionary()
>>> d[k1] = 1   # d = {k1: 1}
>>> del d[k1]
>>> d[k2] = 2   # d = {k2: 2}
>>> del k1      # d = {k2: 2}
```

Changed in version 3.9: Added support for | and |= operators, specified in **PEP 584**.

WeakKeyDictionary objects have an additional method that exposes the internal references directly. The references are not guaranteed to be "live" at the time they are used, so the result of calling the references needs to be checked before being used. This can be used to avoid creating references that will cause the garbage collector to keep the keys around longer than needed.

WeakKeyDictionary.**keyrefs**()

Return an iterable of the weak references to the keys.

class weakref.**WeakValueDictionary**([*dict*])

Mapping class that references values weakly. Entries in the dictionary will be discarded when no strong reference to the value exists any more.

Changed in version 3.9: Added support for | and |= operators, as specified in **PEP 584**.

WeakValueDictionary objects have an additional method that has the same issues as the *WeakKeyDictionary.keyrefs()* method.

WeakValueDictionary.**valuerefs**()

Return an iterable of the weak references to the values.

class weakref.**WeakSet**([*elements*])

Set class that keeps weak references to its elements. An element will be discarded when no strong reference to it exists any more.

class weakref.**WeakMethod**(*method*[, *callback*])

A custom *ref* subclass which simulates a weak reference to a bound method (i.e., a method defined on a class and looked up on an instance). Since a bound method is ephemeral, a standard weak reference cannot keep hold of it. *WeakMethod* has special code to recreate the bound method until either the object or the original function dies:

```
>>> class C:
...     def method(self):
...         print("method called!")
...
>>> c = C()
>>> r = weakref.ref(c.method)
>>> r()
>>> r = weakref.WeakMethod(c.method)
>>> r()
```

(continues on next page)

(continued from previous page)

```
<bound method C.method of <__main__.C object at 0x7fc859830220>>
>>> r()()
method called!
>>> del c
>>> gc.collect()
0
>>> r()
>>>
```

callback is the same as the parameter of the same name to the `ref()` function.

Added in version 3.4.

class weakref.**finalize**(*obj*, *func*, */*, **args*, ***kwargs*)

Return a callable finalizer object which will be called when *obj* is garbage collected. Unlike an ordinary weak reference, a finalizer will always survive until the reference object is collected, greatly simplifying lifecycle management.

A finalizer is considered *alive* until it is called (either explicitly or at garbage collection), and after that it is *dead*. Calling a live finalizer returns the result of evaluating `func(*arg, **kwargs)`, whereas calling a dead finalizer returns `None`.

Exceptions raised by finalizer callbacks during garbage collection will be shown on the standard error output, but cannot be propagated. They are handled in the same way as exceptions raised from an object's `__del__()` method or a weak reference's callback.

When the program exits, each remaining live finalizer is called unless its `atexit` attribute has been set to false. They are called in reverse order of creation.

A finalizer will never invoke its callback during the later part of the *interpreter shutdown* when module globals are liable to have been replaced by `None`.

__call__()

If *self* is alive then mark it as dead and return the result of calling `func(*args, **kwargs)`. If *self* is dead then return `None`.

detach()

If *self* is alive then mark it as dead and return the tuple `(obj, func, args, kwargs)`. If *self* is dead then return `None`.

peek()

If *self* is alive then return the tuple `(obj, func, args, kwargs)`. If *self* is dead then return `None`.

alive

Property which is true if the finalizer is alive, false otherwise.

atexit

A writable boolean property which by default is true. When the program exits, it calls all remaining live finalizers for which `atexit` is true. They are called in reverse order of creation.

> **Note**
>
> It is important to ensure that *func*, *args* and *kwargs* do not own any references to *obj*, either directly or indirectly, since otherwise *obj* will never be garbage collected. In particular, *func* should not be a bound method of *obj*.

Added in version 3.4.

weakref.**ReferenceType**
: The type object for weak references objects.

weakref.**ProxyType**
: The type object for proxies of objects which are not callable.

weakref.**CallableProxyType**
: The type object for proxies of callable objects.

weakref.**ProxyTypes**
: Sequence containing all the type objects for proxies. This can make it simpler to test if an object is a proxy without being dependent on naming both proxy types.

See also

PEP 205 - Weak References
The proposal and rationale for this feature, including links to earlier implementations and information about similar features in other languages.

8.9.1 Weak Reference Objects

Weak reference objects have no methods and no attributes besides *ref.__callback__*. A weak reference object allows the referent to be obtained, if it still exists, by calling it:

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

If the referent no longer exists, calling the reference object returns *None*:

```
>>> del o, o2
>>> print(r())
None
```

Testing that a weak reference object is still live should be done using the expression `ref() is not None`. Normally, application code that needs to use a reference object should follow this pattern:

```
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print("Object has been deallocated; can't frobnicate.")
else:
    print("Object is still live!")
    o.do_something_useful()
```

Using a separate test for "liveness" creates race conditions in threaded applications; another thread can cause a weak reference to become invalidated before the weak reference is called; the idiom shown above is safe in threaded applications as well as single-threaded applications.

Specialized versions of *ref* objects can be created through subclassing. This is used in the implementation of the *WeakValueDictionary* to reduce the memory overhead for each entry in the mapping. This may be most useful to associate additional information with a reference, but could also be used to insert additional processing on calls to retrieve the referent.

This example shows how a subclass of *ref* can be used to store additional information about an object and affect the value that's returned when the referent is accessed:

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, /, **annotations):
        super().__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.items():
            setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number of
        times the reference has been called.
        """
        ob = super().__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

8.9.2 Example

This simple example shows how an application can use object IDs to retrieve objects that it has seen before. The IDs of the objects can then be used in other data structures without forcing the objects to remain alive, but the objects can still be retrieved by ID if they do.

```
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```

8.9.3 Finalizer Objects

The main benefit of using *finalize* is that it makes it simple to register a callback without needing to preserve the returned finalizer object. For instance

```
>>> import weakref
>>> class Object:
...     pass
...
>>> kenny = Object()
>>> weakref.finalize(kenny, print, "You killed Kenny!")
<finalize object at ...; for 'Object' at ...>
>>> del kenny
You killed Kenny!
```

The finalizer can be called directly as well. However the finalizer will invoke the callback at most once.

```
>>> def callback(x, y, z):
...     print("CALLBACK")
...     return x + y + z
...
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> assert f.alive
>>> assert f() == 6
CALLBACK
>>> assert not f.alive
>>> f()                     # callback not called because finalizer dead
>>> del obj                 # callback not called because finalizer dead
```

You can unregister a finalizer using its *detach()* method. This kills the finalizer and returns the arguments passed to the constructor when it was created.

```
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> f.detach()
(<...Object object ...>, <function callback ...>, (1, 2), {'z': 3})
>>> newobj, func, args, kwargs = _
>>> assert not f.alive
>>> assert newobj is obj
>>> assert func(*args, **kwargs) == 6
CALLBACK
```

Unless you set the *atexit* attribute to *False*, a finalizer will be called when the program exits if it is still alive. For instance

```
>>> obj = Object()
>>> weakref.finalize(obj, print, "obj dead or exiting")
<finalize object at ...; for 'Object' at ...>
>>> exit()
obj dead or exiting
```

8.9.4 Comparing finalizers with `__del__()` methods

Suppose we want to create a class whose instances represent temporary directories. The directories should be deleted with their contents when the first of the following events occurs:

- the object is garbage collected,
- the object's `remove()` method is called, or
- the program exits.

We might try to implement the class using a `__del__()` method as follows:

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()

    def remove(self):
        if self.name is not None:
            shutil.rmtree(self.name)
            self.name = None

    @property
    def removed(self):
        return self.name is None

    def __del__(self):
        self.remove()
```

Starting with Python 3.4, `__del__()` methods no longer prevent reference cycles from being garbage collected, and module globals are no longer forced to `None` during *interpreter shutdown*. So this code should work without any issues on CPython.

However, handling of `__del__()` methods is notoriously implementation specific, since it depends on internal details of the interpreter's garbage collector implementation.

A more robust alternative can be to define a finalizer which only references the specific functions and objects that it needs, rather than having access to the full state of the object:

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()
        self._finalizer = weakref.finalize(self, shutil.rmtree, self.name)

    def remove(self):
        self._finalizer()

    @property
    def removed(self):
        return not self._finalizer.alive
```

Defined like this, our finalizer only receives a reference to the details it needs to clean up the directory appropriately. If the object never gets garbage collected the finalizer will still be called at exit.

The other advantage of weakref based finalizers is that they can be used to register finalizers for classes where the definition is controlled by a third party, such as running code when a module is unloaded:

```
import weakref, sys
def unloading_module():
```

(continues on next page)

(continued from previous page)

```
    # implicit reference to the module globals from the function body
weakref.finalize(sys.modules[__name__], unloading_module)
```

Note

If you create a finalizer object in a daemonic thread just as the program exits then there is the possibility that the finalizer does not get called at exit. However, in a daemonic thread *atexit.register()*, `try: ... finally: ...` and `with: ...` do not guarantee that cleanup occurs either.

8.10 `types` — Dynamic type creation and names for built-in types

Source code: Lib/types.py

This module defines utility functions to assist in dynamic creation of new types.

It also defines names for some object types that are used by the standard Python interpreter, but not exposed as builtins like *int* or *str* are.

Finally, it provides some additional type-related utility classes and functions that are not fundamental enough to be builtins.

8.10.1 Dynamic Type Creation

`types.`**`new_class`**(*name*, *bases=()*, *kwds=None*, *exec_body=None*)

Creates a class object dynamically using the appropriate metaclass.

The first three arguments are the components that make up a class definition header: the class name, the base classes (in order), the keyword arguments (such as `metaclass`).

The *exec_body* argument is a callback that is used to populate the freshly created class namespace. It should accept the class namespace as its sole argument and update the namespace directly with the class contents. If no callback is provided, it has the same effect as passing in `lambda ns: None`.

Added in version 3.3.

`types.`**`prepare_class`**(*name*, *bases=()*, *kwds=None*)

Calculates the appropriate metaclass and creates the class namespace.

The arguments are the components that make up a class definition header: the class name, the base classes (in order) and the keyword arguments (such as `metaclass`).

The return value is a 3-tuple: `metaclass, namespace, kwds`

metaclass is the appropriate metaclass, *namespace* is the prepared class namespace and *kwds* is an updated copy of the passed in *kwds* argument with any `'metaclass'` entry removed. If no *kwds* argument is passed in, this will be an empty dict.

Added in version 3.3.

Changed in version 3.6: The default value for the `namespace` element of the returned tuple has changed. Now an insertion-order-preserving mapping is used when the metaclass does not have a `__prepare__` method.

See also

metaclasses
Full details of the class creation process supported by these functions

PEP 3115 - Metaclasses in Python 3000
Introduced the `__prepare__` namespace hook

`types.`**`resolve_bases`**(*bases*)

Resolve MRO entries dynamically as specified by **PEP 560**.

This function looks for items in *bases* that are not instances of *type*, and returns a tuple where each such object that has an `__mro_entries__()` method is replaced with an unpacked result of calling this method. If a *bases* item is an instance of *type*, or it doesn't have an `__mro_entries__()` method, then it is included in the return tuple unchanged.

Added in version 3.7.

`types.`**`get_original_bases`**(*cls*, `/`)

Return the tuple of objects originally given as the bases of *cls* before the `__mro_entries__()` method has been called on any bases (following the mechanisms laid out in **PEP 560**). This is useful for introspecting *Generics*.

For classes that have an `__orig_bases__` attribute, this function returns the value of `cls.__orig_bases__`. For classes without the `__orig_bases__` attribute, `cls.__bases__` is returned.

Examples:

```
from typing import TypeVar, Generic, NamedTuple, TypedDict

T = TypeVar("T")
class Foo(Generic[T]): ...
class Bar(Foo[int], float): ...
class Baz(list[str]): ...
Eggs = NamedTuple("Eggs", [("a", int), ("b", str)])
Spam = TypedDict("Spam", {"a": int, "b": str})

assert Bar.__bases__ == (Foo, float)
assert get_original_bases(Bar) == (Foo[int], float)

assert Baz.__bases__ == (list,)
assert get_original_bases(Baz) == (list[str],)

assert Eggs.__bases__ == (tuple,)
assert get_original_bases(Eggs) == (NamedTuple,)

assert Spam.__bases__ == (dict,)
assert get_original_bases(Spam) == (TypedDict,)

assert int.__bases__ == (object,)
assert get_original_bases(int) == (object,)
```

Added in version 3.12.

See also

PEP 560 - Core support for typing module and generic types

8.10.2 Standard Interpreter Types

This module provides names for many of the types that are required to implement a Python interpreter. It deliberately avoids including some of the types that arise only incidentally during processing such as the `listiterator` type.

Typical use of these names is for *isinstance()* or *issubclass()* checks.

If you instantiate any of these types, note that signatures may vary between Python versions.

Standard names are defined for the following types:

`types.`**NoneType**

The type of *None*.

Added in version 3.10.

`types.`**FunctionType**

`types.`**LambdaType**

The type of user-defined functions and functions created by `lambda` expressions.

Raises an *auditing event* `function.__new__` with argument `code`.

The audit event only occurs for direct instantiation of function objects, and is not raised for normal compilation.

`types.`**GeneratorType**

The type of *generator*-iterator objects, created by generator functions.

`types.`**CoroutineType**

The type of *coroutine* objects, created by `async def` functions.

Added in version 3.5.

`types.`**AsyncGeneratorType**

The type of *asynchronous generator*-iterator objects, created by asynchronous generator functions.

Added in version 3.6.

class `types.`**CodeType**(***kwargs*)

The type of code objects such as returned by *compile()*.

Raises an *auditing event* `code.__new__` with arguments `code`, `filename`, `name`, `argcount`, `posonlyargcount`, `kwonlyargcount`, `nlocals`, `stacksize`, `flags`.

Note that the audited arguments may not match the names or positions required by the initializer. The audit event only occurs for direct instantiation of code objects, and is not raised for normal compilation.

`types.`**CellType**

The type for cell objects: such objects are used as containers for a function's free variables.

Added in version 3.8.

`types.`**MethodType**

The type of methods of user-defined class instances.

`types.`**BuiltinFunctionType**

`types.`**BuiltinMethodType**

The type of built-in functions like *len()* or *sys.exit()*, and methods of built-in classes. (Here, the term "built-in" means "written in C".)

types.**WrapperDescriptorType**

The type of methods of some built-in data types and base classes such as `object.__init__()` or `object.__lt__()`.

Added in version 3.7.

types.**MethodWrapperType**

The type of *bound* methods of some built-in data types and base classes. For example it is the type of `object().__str__`.

Added in version 3.7.

types.**NotImplementedType**

The type of *NotImplemented*.

Added in version 3.10.

types.**MethodDescriptorType**

The type of methods of some built-in data types such as *str.join()*.

Added in version 3.7.

types.**ClassMethodDescriptorType**

The type of *unbound* class methods of some built-in data types such as `dict.__dict__['fromkeys']`.

Added in version 3.7.

class types.**ModuleType**(*name*, *doc=None*)

The type of *modules*. The constructor takes the name of the module to be created and optionally its *docstring*.

> **Note**
>
> Use *importlib.util.module_from_spec()* to create a new module if you wish to set the various import-controlled attributes.

__doc__

The *docstring* of the module. Defaults to `None`.

__loader__

The *loader* which loaded the module. Defaults to `None`.

This attribute is to match *importlib.machinery.ModuleSpec.loader* as stored in the `__spec__` object.

> **Note**
>
> A future version of Python may stop setting this attribute by default. To guard against this potential change, preferably read from the `__spec__` attribute instead or use `getattr(module, "__loader__", None)` if you explicitly need to use this attribute.

Changed in version 3.4: Defaults to `None`. Previously the attribute was optional.

__name__

The name of the module. Expected to match *importlib.machinery.ModuleSpec.name*.

__package__

Which *package* a module belongs to. If the module is top-level (i.e. not a part of any specific package) then the attribute should be set to `''`, else it should be set to the name of the package (which can be `__name__` if the module is a package itself). Defaults to `None`.

This attribute is to match *importlib.machinery.ModuleSpec.parent* as stored in the `__spec__` object.

> **Note**
>
> A future version of Python may stop setting this attribute by default. To guard against this potential change, preferably read from the `__spec__` attribute instead or use `getattr(module, "__package__", None)` if you explicitly need to use this attribute.

Changed in version 3.4: Defaults to `None`. Previously the attribute was optional.

__spec__

A record of the module's import-system-related state. Expected to be an instance of *importlib.machinery.ModuleSpec*.

Added in version 3.4.

`types.`**EllipsisType**

The type of *Ellipsis*.

Added in version 3.10.

class `types.`**GenericAlias**(*t_origin*, *t_args*)

The type of *parameterized generics* such as `list[int]`.

`t_origin` should be a non-parameterized generic class, such as `list`, `tuple` or `dict`. `t_args` should be a *tuple* (possibly of length 1) of types which parameterize `t_origin`:

```
>>> from types import GenericAlias

>>> list[int] == GenericAlias(list, (int,))
True
>>> dict[str, int] == GenericAlias(dict, (str, int))
True
```

Added in version 3.9.

Changed in version 3.9.2: This type can now be subclassed.

> **See also**
>
> *Generic Alias Types*
> In-depth documentation on instances of `types.GenericAlias`
>
> **PEP 585 - Type Hinting Generics In Standard Collections**
> Introducing the `types.GenericAlias` class

class `types.`**UnionType**

The type of *union type expressions*.

Added in version 3.10.

class types.**TracebackType**(*tb_next*, *tb_frame*, *tb_lasti*, *tb_lineno*)

The type of traceback objects such as found in `sys.exception().__traceback__`.

See the language reference for details of the available attributes and operations, and guidance on creating tracebacks dynamically.

types.**FrameType**

The type of frame objects such as found in `tb.tb_frame` if `tb` is a traceback object.

types.**GetSetDescriptorType**

The type of objects defined in extension modules with `PyGetSetDef`, such as `FrameType.f_locals` or `array.array.typecode`. This type is used as descriptor for object attributes; it has the same purpose as the *property* type, but for classes defined in extension modules.

types.**MemberDescriptorType**

The type of objects defined in extension modules with `PyMemberDef`, such as `datetime.timedelta.days`. This type is used as descriptor for simple C data members which use standard conversion functions; it has the same purpose as the *property* type, but for classes defined in extension modules.

In addition, when a class is defined with a `__slots__` attribute, then for each slot, an instance of `MemberDescriptorType` will be added as an attribute on the class. This allows the slot to appear in the class's *__dict__*.

CPython implementation detail: In other implementations of Python, this type may be identical to `GetSetDescriptorType`.

class types.**MappingProxyType**(*mapping*)

Read-only proxy of a mapping. It provides a dynamic view on the mapping's entries, which means that when the mapping changes, the view reflects these changes.

Added in version 3.3.

Changed in version 3.9: Updated to support the new union (`|`) operator from **PEP 584**, which simply delegates to the underlying mapping.

key in proxy

Return `True` if the underlying mapping has a key *key*, else `False`.

proxy[key]

Return the item of the underlying mapping with key *key*. Raises a *KeyError* if *key* is not in the underlying mapping.

iter(proxy)

Return an iterator over the keys of the underlying mapping. This is a shortcut for `iter(proxy.keys())`.

len(proxy)

Return the number of items in the underlying mapping.

copy()

Return a shallow copy of the underlying mapping.

get(*key*[, *default*])

Return the value for *key* if *key* is in the underlying mapping, else *default*. If *default* is not given, it defaults to `None`, so that this method never raises a *KeyError*.

items()

Return a new view of the underlying mapping's items (`(key, value)` pairs).

keys()

Return a new view of the underlying mapping's keys.

values()

Return a new view of the underlying mapping's values.

reversed(proxy)

Return a reverse iterator over the keys of the underlying mapping.

Added in version 3.9.

hash(proxy)

Return a hash of the underlying mapping.

Added in version 3.12.

class types.**CapsuleType**

The type of capsule objects.

Added in version 3.13.

8.10.3 Additional Utility Classes and Functions

class types.**SimpleNamespace**

A simple *object* subclass that provides attribute access to its namespace, as well as a meaningful repr.

Unlike *object*, with SimpleNamespace you can add and remove attributes.

SimpleNamespace objects may be initialized in the same way as *dict*: either with keyword arguments, with a single positional argument, or with both. When initialized with keyword arguments, those are directly added to the underlying namespace. Alternatively, when initialized with a positional argument, the underlying namespace will be updated with key-value pairs from that argument (either a mapping object or an *iterable* object producing key-value pairs). All such keys must be strings.

The type is roughly equivalent to the following code:

```
class SimpleNamespace:
    def __init__(self, mapping_or_iterable=(), /, **kwargs):
        self.__dict__.update(mapping_or_iterable)
        self.__dict__.update(kwargs)

    def __repr__(self):
        items = (f"{k}={v!r}" for k, v in self.__dict__.items())
        return "{}({})".format(type(self).__name__, ", ".join(items))

    def __eq__(self, other):
        if isinstance(self, SimpleNamespace) and isinstance(other,␣
↪SimpleNamespace):
            return self.__dict__ == other.__dict__
        return NotImplemented
```

SimpleNamespace may be useful as a replacement for `class NS: pass`. However, for a structured record type use *namedtuple()* instead.

SimpleNamespace objects are supported by *copy.replace()*.

Added in version 3.3.

Changed in version 3.9: Attribute order in the repr changed from alphabetical to insertion (like dict).

Changed in version 3.13: Added support for an optional positional argument.

types.**DynamicClassAttribute**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

Route attribute access on a class to __getattr__.

This is a descriptor, used to define attributes that act differently when accessed through an instance and through a class. Instance access remains normal, but access to an attribute through a class will be routed to the class's __getattr__ method; this is done by raising AttributeError.

This allows one to have properties active on an instance, and have virtual attributes on the class with the same name (see *enum.Enum* for an example).

Added in version 3.4.

8.10.4 Coroutine Utility Functions

types.**coroutine**(*gen_func*)

This function transforms a *generator* function into a *coroutine function* which returns a generator-based coroutine. The generator-based coroutine is still a *generator iterator*, but is also considered to be a *coroutine* object and is *awaitable*. However, it may not necessarily implement the `__await__()` method.

If *gen_func* is a generator function, it will be modified in-place.

If *gen_func* is not a generator function, it will be wrapped. If it returns an instance of *collections.abc.Generator*, the instance will be wrapped in an *awaitable* proxy object. All other types of objects will be returned as is.

Added in version 3.5.

8.11 copy — Shallow and deep copy operations

Source code: Lib/copy.py

Assignment statements in Python do not copy objects, they create bindings between a target and an object. For collections that are mutable or contain mutable items, a copy is sometimes needed so one can change one copy without changing the other. This module provides generic shallow and deep copy operations (explained below).

Interface summary:

copy.**copy**(*obj*)

Return a shallow copy of *obj*.

copy.**deepcopy**(*obj*[, *memo*])

Return a deep copy of *obj*.

copy.**replace**(*obj*, /, ***changes*)

Creates a new object of the same type as *obj*, replacing fields with values from *changes*.

Added in version 3.13.

exception copy.**Error**

Raised for module specific errors.

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A *shallow copy* constructs a new compound object and then (to the extent possible) inserts *references* into it to the objects found in the original.

- A *deep copy* constructs a new compound object and then, recursively, inserts *copies* into it of the objects found in the original.

Two problems often exist with deep copy operations that don't exist with shallow copy operations:

- Recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop.
- Because deep copy copies everything it may copy too much, such as data which is intended to be shared between copies.

The *deepcopy()* function avoids these problems by:

- keeping a `memo` dictionary of objects already copied during the current copying pass; and
- letting user-defined classes override the copying operation or the set of components copied.

This module does not copy types like module, method, stack trace, stack frame, file, socket, window, or any similar types. It does "copy" functions and classes (shallow and deeply), by returning the original object unchanged; this is compatible with the way these are treated by the *pickle* module.

Shallow copies of dictionaries can be made using *dict.copy()*, and of lists by assigning a slice of the entire list, for example, `copied_list = original_list[:]`.

Classes can use the same interfaces to control copying that they use to control pickling. See the description of module *pickle* for information on these methods. In fact, the *copy* module uses the registered pickle functions from the *copyreg* module.

In order for a class to define its own copy implementation, it can define special methods *__copy__()* and *__deepcopy__()*.

`object.`**`__copy__`**(*self*)

Called to implement the shallow copy operation; no additional arguments are passed.

`object.`**`__deepcopy__`**(*self*, *memo*)

Called to implement the deep copy operation; it is passed one argument, the *memo* dictionary. If the `__deepcopy__` implementation needs to make a deep copy of a component, it should call the *deepcopy()* function with the component as first argument and the *memo* dictionary as second argument. The *memo* dictionary should be treated as an opaque object.

Function `copy.replace()` is more limited than *copy()* and *deepcopy()*, and only supports named tuples created by *namedtuple()*, *dataclasses*, and other classes which define method *__replace__()*.

`object.`**`__replace__`**(*self*, */*, ***changes*)

This method should create a new object of the same type, replacing fields with values from *changes*.

See also

Module *pickle*
Discussion of the special methods used to support object state retrieval and restoration.

8.12 pprint — Data pretty printer

Source code: Lib/pprint.py

The *pprint* module provides a capability to "pretty-print" arbitrary Python data structures in a form which can be used as input to the interpreter. If the formatted structures include objects which are not fundamental Python types, the representation may not be loadable. This may be the case if objects such as files, sockets or classes are included, as well as many other objects which are not representable as Python literals.

The formatted representation keeps objects on a single line if it can, and breaks them onto multiple lines if they don't fit within the allowed width, adjustable by the *width* parameter defaulting to 80 characters.

Dictionaries are sorted by key before the display is computed.

Changed in version 3.9: Added support for pretty-printing *types.SimpleNamespace*.

Changed in version 3.10: Added support for pretty-printing *dataclasses.dataclass*.

8.12.1 Functions

pprint.**pp**(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*, *sort_dicts=False*, *underscore_numbers=False*)

Prints the formatted representation of *object*, followed by a newline. This function may be used in the interactive interpreter instead of the *print()* function for inspecting values. Tip: you can reassign `print = pprint.pp` for use within a scope.

Parameters

- **object** – The object to be printed.
- **stream** (*file-like object* | None) – A file-like object to which the output will be written by calling its `write()` method. If `None` (the default), *sys.stdout* is used.
- **indent** (`int`) – The amount of indentation added for each nesting level.
- **width** (`int`) – The desired maximum number of characters per line in the output. If a structure cannot be formatted within the width constraint, a best effort will be made.
- **depth** (`int` | *None*) – The number of nesting levels which may be printed. If the data structure being printed is too deep, the next contained level is replaced by `...`. If `None` (the default), there is no constraint on the depth of the objects being formatted.
- **compact** (`bool`) – Control the way long *sequences* are formatted. If `False` (the default), each item of a sequence will be formatted on a separate line, otherwise as many items as will fit within the *width* will be formatted on each output line.
- **sort_dicts** (`bool`) – If `True`, dictionaries will be formatted with their keys sorted, otherwise they will be displayed in insertion order (the default).
- **underscore_numbers** (`bool`) – If `True`, integers will be formatted with the _ character for a thousands separator, otherwise underscores are not displayed (the default).

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pp(stuff)
[<Recursion on list with id=...>,
```

(continues on next page)

(continued from previous page)

```
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
```

Added in version 3.8.

pprint.**pprint**(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*, *sort_dicts=True*, *underscore_numbers=False*)

Alias for *pp()* with *sort_dicts* set to `True` by default, which would automatically sort the dictionaries' keys, you might want to use *pp()* instead where it is `False` by default.

pprint.**pformat**(*object*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*, *sort_dicts=True*, *underscore_numbers=False*)

Return the formatted representation of *object* as a string. *indent*, *width*, *depth*, *compact*, *sort_dicts* and *underscore_numbers* are passed to the *PrettyPrinter* constructor as formatting parameters and their meanings are as described in the documentation above.

pprint.**isreadable**(*object*)

Determine if the formatted representation of *object* is "readable", or can be used to reconstruct the value using *eval()*. This always returns `False` for recursive objects.

```
>>> pprint.isreadable(stuff)
False
```

pprint.**isrecursive**(*object*)

Determine if *object* requires a recursive representation. This function is subject to the same limitations as noted in *saferepr()* below and may raise an *RecursionError* if it fails to detect a recursive object.

pprint.**saferepr**(*object*)

Return a string representation of *object*, protected against recursion in some common data structures, namely instances of *dict*, *list* and *tuple* or subclasses whose `__repr__` has not been overridden. If the representation of object exposes a recursive entry, the recursive reference will be represented as `<Recursion on typename with id=number>`. The representation is not otherwise formatted.

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'knights', 'ni']"
```

8.12.2 PrettyPrinter Objects

class pprint.**PrettyPrinter**(*indent=1*, *width=80*, *depth=None*, *stream=None*, ***, *compact=False*, *sort_dicts=True*, *underscore_numbers=False*)

Construct a *PrettyPrinter* instance.

Arguments have the same meaning as for *pp()*. Note that they are in a different order, and that *sort_dicts* defaults to `True`.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
```

(continues on next page)

(continued from previous page)

```
[   ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
    'spam',
    'eggs',
    'lumberjack',
    'knights',
    'ni']
>>> pp = pprint.PrettyPrinter(width=41, compact=True)
>>> pp.pprint(stuff)
[['spam', 'eggs', 'lumberjack',
  'knights', 'ni'],
 'spam', 'eggs', 'lumberjack', 'knights',
 'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
... ('parrot', ('fresh fruit',)))))))
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead', (...)))))))
```

Changed in version 3.4: Added the *compact* parameter.

Changed in version 3.8: Added the *sort_dicts* parameter.

Changed in version 3.10: Added the *underscore_numbers* parameter.

Changed in version 3.11: No longer attempts to write to `sys.stdout` if it is `None`.

PrettyPrinter instances have the following methods:

PrettyPrinter.**pformat**(*object*)

Return the formatted representation of *object*. This takes into account the options passed to the *PrettyPrinter* constructor.

PrettyPrinter.**pprint**(*object*)

Print the formatted representation of *object* on the configured stream, followed by a newline.

The following methods provide the implementations for the corresponding functions of the same names. Using these methods on an instance is slightly more efficient since new *PrettyPrinter* objects don't need to be created.

PrettyPrinter.**isreadable**(*object*)

Determine if the formatted representation of the object is "readable," or can be used to reconstruct the value using *eval()*. Note that this returns `False` for recursive objects. If the *depth* parameter of the *PrettyPrinter* is set and the object is deeper than allowed, this returns `False`.

PrettyPrinter.**isrecursive**(*object*)

Determine if the object requires a recursive representation.

This method is provided as a hook to allow subclasses to modify the way objects are converted to strings. The default implementation uses the internals of the *saferepr()* implementation.

PrettyPrinter.**format**(*object*, *context*, *maxlevels*, *level*)

Returns three values: the formatted version of *object* as a string, a flag indicating whether the result is readable, and a flag indicating whether recursion was detected. The first argument is the object to be presented. The second is a dictionary which contains the *id()* of objects that are part of the current presentation context (direct and indirect containers for *object* that are affecting the presentation) as the keys; if an object needs to be presented which is already represented in *context*, the third return value should be `True`. Recursive calls to the *format()* method should add additional entries for containers to this dictionary. The third argument, *maxlevels*, gives the requested limit to recursion; this will be `0` if there is no requested limit. This argument should be passed unmodified to recursive calls. The fourth argument, *level*, gives the current level; recursive calls should be passed a value less than that of the current call.

8.12.3 Example

To demonstrate several uses of the *pp()* function and its parameters, let's fetch information about a project from PyPI:

```
>>> import json
>>> import pprint
>>> from urllib.request import urlopen
>>> with urlopen('https://pypi.org/pypi/sampleproject/json') as resp:
...     project_info = json.load(resp)['info']
```

In its basic form, *pp()* shows the whole object:

```
>>> pprint.pp(project_info)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': ['Development Status :: 3 - Alpha',
                 'Intended Audience :: Developers',
                 'License :: OSI Approved :: MIT License',
                 'Programming Language :: Python :: 2',
                 'Programming Language :: Python :: 2.6',
                 'Programming Language :: Python :: 2.7',
                 'Programming Language :: Python :: 3',
                 'Programming Language :: Python :: 3.2',
                 'Programming Language :: Python :: 3.3',
                 'Programming Language :: Python :: 3.4',
                 'Topic :: Software Development :: Build Tools'],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {'last_day': -1, 'last_month': -1, 'last_week': -1},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
```

(continues on next page)

(continued from previous page)

```
 'project_urls': {'Download': 'UNKNOWN',
                  'Homepage': 'https://github.com/pypa/sampleproject'},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

The result can be limited to a certain *depth* (ellipsis is used for deeper contents):

```
>>> pprint.pp(project_info, depth=1)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

Additionally, maximum character *width* can be suggested. If a long object cannot be split, the specified width will be exceeded:

```
>>> pprint.pp(project_info, depth=1, width=60)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the '
                'project.\n'
                '\n'
                'The file should use UTF-8 encoding and be '
                'written using ReStructured Text. It\n'
                'will be used to generate the project '
                'webpage on PyPI, and should be written '
                'for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would '
                'include an overview of the project, '
                'basic\n'
                'usage examples, etc. Generally, including '
                'the project changelog in here is not\n'
                'a good idea, although a simple "What\'s '
                'New" section for the most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

8.13 reprlib — Alternate repr() implementation

Source code: Lib/reprlib.py

The reprlib module provides a means for producing object representations with limits on the size of the resulting strings. This is used in the Python debugger and may be useful in other contexts as well.

This module provides a class, an instance, and a function:

class reprlib.**Repr**(*, *maxlevel=6*, *maxtuple=6*, *maxlist=6*, *maxarray=5*, *maxdict=4*, *maxset=6*, *maxfrozenset=6*, *maxdeque=6*, *maxstring=30*, *maxlong=40*, *maxother=30*, *fillvalue='...'*, *indent=None*)

Class which provides formatting services useful in implementing functions similar to the built-in *repr()*; size limits for different object types are added to avoid the generation of representations which are excessively long.

The keyword arguments of the constructor can be used as a shortcut to set the attributes of the *Repr* instance. Which means that the following initialization:

```
aRepr = reprlib.Repr(maxlevel=3)
```

Is equivalent to:

```
aRepr = reprlib.Repr()
aRepr.maxlevel = 3
```

See section *Repr Objects* for more information about *Repr* attributes.

Changed in version 3.12: Allow attributes to be set via keyword arguments.

reprlib.**aRepr**

This is an instance of *Repr* which is used to provide the *repr()* function described below. Changing the attributes of this object will affect the size limits used by *repr()* and the Python debugger.

reprlib.**repr**(*obj*)

This is the *repr()* method of aRepr. It returns a string similar to that returned by the built-in function of the same name, but with limits on most sizes.

In addition to size-limiting tools, the module also provides a decorator for detecting recursive calls to __repr__() and substituting a placeholder string instead.

@reprlib.**recursive_repr**(*fillvalue='...'*)

Decorator for __repr__() methods to detect recursive calls within the same thread. If a recursive call is made, the *fillvalue* is returned, otherwise, the usual __repr__() call is made. For example:

```
>>> from reprlib import recursive_repr
>>> class MyList(list):
...     @recursive_repr()
...     def __repr__(self):
...         return '<' + '|'.join(map(repr, self)) + '>'
...
>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

Added in version 3.2.

8.13.1 Repr Objects

Repr instances provide several attributes which can be used to provide size limits for the representations of different object types, and methods which format specific object types.

Repr.**fillvalue**

This string is displayed for recursive references. It defaults to `...`.

Added in version 3.11.

Repr.**maxlevel**

Depth limit on the creation of recursive representations. The default is `6`.

Repr.**maxdict**

Repr.**maxlist**

Repr.**maxtuple**

Repr.**maxset**

Repr.**maxfrozenset**

Repr.**maxdeque**

Repr.**maxarray**

Limits on the number of entries represented for the named object type. The default is `4` for *maxdict*, `5` for *maxarray*, and `6` for the others.

Repr.**maxlong**

Maximum number of characters in the representation for an integer. Digits are dropped from the middle. The default is `40`.

Repr.**maxstring**

Limit on the number of characters in the representation of the string. Note that the "normal" representation of the string is used as the character source: if escape sequences are needed in the representation, these may be mangled when the representation is shortened. The default is `30`.

Repr.**maxother**

This limit is used to control the size of object types for which no specific formatting method is available on the *Repr* object. It is applied in a similar manner as *maxstring*. The default is `20`.

Repr.**indent**

If this attribute is set to `None` (the default), the output is formatted with no line breaks or indentation, like the standard *repr()*. For example:

```
>>> example = [
...     1, 'spam', {'a': 2, 'b': 'spam eggs', 'c': {3: 4.5, 6: []}}, 'ham']
>>> import reprlib
>>> aRepr = reprlib.Repr()
>>> print(aRepr.repr(example))
[1, 'spam', {'a': 2, 'b': 'spam eggs', 'c': {3: 4.5, 6: []}}, 'ham']
```

If *indent* is set to a string, each recursion level is placed on its own line, indented by that string:

```
>>> aRepr.indent = '-->'
>>> print(aRepr.repr(example))
[
-->1,
-->'spam',
-->{
```

(continues on next page)

(continued from previous page)

```
-->-->'a': 2,
-->-->'b': 'spam eggs',
-->-->'c': {
-->-->-->3: 4.5,
-->-->-->6: [],
-->-->},
-->},
-->'ham',
]
```

Setting *indent* to a positive integer value behaves as if it was set to a string with that number of spaces:

```
>>> aRepr.indent = 4
>>> print(aRepr.repr(example))
[
    1,
    'spam',
    {
        'a': 2,
        'b': 'spam eggs',
        'c': {
            3: 4.5,
            6: [],
        },
    },
    'ham',
]
```

Added in version 3.12.

Repr.**repr**(*obj*)

The equivalent to the built-in *repr()* that uses the formatting imposed by the instance.

Repr.**repr1**(*obj*, *level*)

Recursive implementation used by *repr()*. This uses the type of *obj* to determine which formatting method to call, passing it *obj* and *level*. The type-specific methods should call *repr1()* to perform recursive formatting, with `level - 1` for the value of *level* in the recursive call.

Repr.**repr_TYPE**(*obj*, *level*)

Formatting methods for specific types are implemented as methods with a name based on the type name. In the method name, **TYPE** is replaced by `'_'.join(type(obj).__name__.split())`. Dispatch to these methods is handled by *repr1()*. Type-specific methods which need to recursively format a value should call `self.repr1(subobj, level - 1)`.

8.13.2 Subclassing Repr Objects

The use of dynamic dispatching by *Repr.repr1()* allows subclasses of *Repr* to add support for additional built-in object types or to modify the handling of types already supported. This example shows how special support for file objects could be added:

```
import reprlib
import sys

class MyRepr(reprlib.Repr):
```

(continues on next page)

(continued from previous page)

```
    def repr_TextIOWrapper(self, obj, level):
        if obj.name in {'<stdin>', '<stdout>', '<stderr>'}:
            return obj.name
        return repr(obj)

aRepr = MyRepr()
print(aRepr.repr(sys.stdin))         # prints '<stdin>'
```

```
<stdin>
```

8.14 enum — Support for enumerations

Added in version 3.4.

Source code: Lib/enum.py

Important

This page contains the API reference information. For tutorial information and discussion of more advanced topics, see

- Basic Tutorial
- Advanced Tutorial
- Enum Cookbook

An enumeration:

- is a set of symbolic names (members) bound to unique values
- can be iterated over to return its canonical (i.e. non-alias) members in definition order
- uses *call* syntax to return members by value
- uses *index* syntax to return members by name

Enumerations are created either by using `class` syntax, or by using function-call syntax:

```
>>> from enum import Enum

>>> # class syntax
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3

>>> # functional syntax
>>> Color = Enum('Color', ['RED', 'GREEN', 'BLUE'])
```

Even though we can use `class` syntax to create Enums, Enums are not normal Python classes. See How are Enums different? for more details.

Note

Nomenclature

- The class `Color` is an *enumeration* (or *enum*)
- The attributes `Color.RED`, `Color.GREEN`, etc., are *enumeration members* (or *members*) and are functionally constants.
- The enum members have *names* and *values* (the name of `Color.RED` is `RED`, the value of `Color.BLUE` is `3`, etc.)

8.14.1 Module Contents

EnumType

The `type` for Enum and its subclasses.

Enum

Base class for creating enumerated constants.

IntEnum

Base class for creating enumerated constants that are also subclasses of *int*. (*Notes*)

StrEnum

Base class for creating enumerated constants that are also subclasses of *str*. (*Notes*)

Flag

Base class for creating enumerated constants that can be combined using the bitwise operations without losing their *Flag* membership.

IntFlag

Base class for creating enumerated constants that can be combined using the bitwise operators without losing their *IntFlag* membership. *IntFlag* members are also subclasses of *int*. (*Notes*)

ReprEnum

Used by *IntEnum*, *StrEnum*, and *IntFlag* to keep the *str()* of the mixed-in type.

EnumCheck

An enumeration with the values `CONTINUOUS`, `NAMED_FLAGS`, and `UNIQUE`, for use with *verify()* to ensure various constraints are met by a given enumeration.

FlagBoundary

An enumeration with the values `STRICT`, `CONFORM`, `EJECT`, and `KEEP` which allows for more fine-grained control over how invalid values are dealt with in an enumeration.

auto

Instances are replaced with an appropriate value for Enum members. *StrEnum* defaults to the lower-cased version of the member name, while other Enums default to 1 and increase from there.

property()

Allows *Enum* members to have attributes without conflicting with member names. The `value` and `name` attributes are implemented this way.

unique()

Enum class decorator that ensures only one name is bound to any one value.

verify()

Enum class decorator that checks user-selectable constraints on an enumeration.

member()

Make `obj` a member. Can be used as a decorator.

nonmember()

Do not make `obj` a member. Can be used as a decorator.

global_enum()

Modify the *str()* and *repr()* of an enum to show its members as belonging to the module instead of its class, and export the enum members to the global namespace.

show_flag_values()

Return a list of all power-of-two integers contained in a flag.

Added in version 3.6: `Flag`, `IntFlag`, `auto`

Added in version 3.11: `StrEnum`, `EnumCheck`, `ReprEnum`, `FlagBoundary`, `property`, `member`, `nonmember`, `global_enum`, `show_flag_values`

8.14.2 Data Types

class `enum.`**EnumType**

EnumType is the *metaclass* for *enum* enumerations. It is possible to subclass *EnumType* – see Subclassing EnumType for details.

`EnumType` is responsible for setting the correct `__repr__()`, `__str__()`, `__format__()`, and `__reduce__()` methods on the final *enum*, as well as creating the enum members, properly handling duplicates, providing iteration over the enum class, etc.

__call__(*cls*, *value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This method is called in two different ways:

- to look up an existing member:

  cls
  The enum class being called.

  value
  The value to lookup.

- to use the `cls` enum to create a new enum (only if the existing enum does not have any members):

  cls
  The enum class being called.

  value
  The name of the new Enum to create.

names
The names/values of the members for the new Enum.

module
The name of the module the new Enum is created in.

qualname
The actual location in the module where this Enum can be found.

type
A mix-in type for the new Enum.

start
The first integer value for the Enum (used by `auto`).

boundary
How to handle out-of-range values from bit operations (`Flag` only).

__contains__(*cls*, *member*)

Returns `True` if member belongs to the `cls`:

```
>>> some_var = Color.RED
>>> some_var in Color
True
>>> Color.RED.value in Color
True
```

Changed in version 3.12: Before Python 3.12, a `TypeError` is raised if a non-Enum-member is used in a containment check.

__dir__(*cls*)

Returns `['__class__', '__doc__', '__members__', '__module__']` and the names of the members in *cls*:

```
>>> dir(Color)
['BLUE', 'GREEN', 'RED', '__class__', '__contains__', '__doc__', '__getitem__
↪', '__init_subclass__', '__iter__', '__len__', '__members__', '__module__',
↪'__name__', '__qualname__']
```

__getitem__(*cls*, *name*)

Returns the Enum member in *cls* matching *name*, or raises a `KeyError`:

```
>>> Color['BLUE']
<Color.BLUE: 3>
```

__iter__(*cls*)

Returns each member in *cls* in definition order:

```
>>> list(Color)
[<Color.RED: 1>, <Color.GREEN: 2>, <Color.BLUE: 3>]
```

__len__(*cls*)

Returns the number of member in *cls*:

```
>>> len(Color)
3
```

__members__

Returns a mapping of every enum name to its member, including aliases

__reversed__(*cls*)

Returns each member in *cls* in reverse definition order:

```
>>> list(reversed(Color))
[<Color.BLUE: 3>, <Color.GREEN: 2>, <Color.RED: 1>]
```

_add_alias_()

Adds a new name as an alias to an existing member. Raises a *NameError* if the name is already assigned to a different member.

_add_value_alias_()

Adds a new value as an alias to an existing member. Raises a *ValueError* if the value is already linked with a different member.

Added in version 3.11: Before 3.11 `EnumType` was called `EnumMeta`, which is still available as an alias.

class enum.**Enum**

Enum is the base class for all *enum* enumerations.

name

The name used to define the `Enum` member:

```
>>> Color.BLUE.name
'BLUE'
```

value

The value given to the `Enum` member:

```
>>> Color.RED.value
1
```

Value of the member, can be set in *__new__()*.

> **Note**
>
> Enum member values
>
> Member values can be anything: *int*, *str*, etc. If the exact value is unimportant you may use *auto* instances and an appropriate value will be chosen for you. See *auto* for the details.
>
> While mutable/unhashable values, such as *dict*, *list* or a mutable *dataclass*, can be used, they will have a quadratic performance impact during creation relative to the total number of mutable/unhashable values in the enum.

name

Name of the member.

value

Value of the member, can be set in *__new__()*.

order

No longer used, kept for backward compatibility. (class attribute, removed during class creation).

ignore

ignore is only used during creation and is removed from the enumeration once creation is complete.

ignore is a list of names that will not become members, and whose names will also be removed from the completed enumeration. See TimePeriod for an example.

__dir__ (*self*)

Returns ['__class__', '__doc__', '__module__', 'name', 'value'] and any public methods defined on *self.__class__*:

```
>>> from datetime import date
>>> class Weekday(Enum):
...     MONDAY = 1
...     TUESDAY = 2
...     WEDNESDAY = 3
...     THURSDAY = 4
...     FRIDAY = 5
...     SATURDAY = 6
...     SUNDAY = 7
...     @classmethod
...     def today(cls):
...         print('today is %s' % cls(date.today().isoweekday()).name)
...
>>> dir(Weekday.SATURDAY)
['__class__', '__doc__', '__eq__', '__hash__', '__module__', 'name', 'today',
↪'value']
```

_generate_next_value_ (*name*, *start*, *count*, *last_values*)

name
The name of the member being defined (e.g. 'RED').

start
The start value for the Enum; the default is 1.

count
The number of members currently defined, not including this one.

last_values
A list of the previous values.

A *staticmethod* that is used to determine the next value returned by *auto*:

```
>>> from enum import auto
>>> class PowersOfThree(Enum):
...     @staticmethod
...     def _generate_next_value_(name, start, count, last_values):
...         return 3 ** (count + 1)
...     FIRST = auto()
...     SECOND = auto()
...
>>> PowersOfThree.SECOND.value
9
```

__init__ (*self*, **args*, ***kwds*)

By default, does nothing. If multiple values are given in the member assignment, those values become separate arguments to __init__; e.g.

```
>>> from enum import Enum
>>> class Weekday(Enum):
...     MONDAY = 1, 'Mon'
```

`Weekday.__init__()` would be called as `Weekday.__init__(self, 1, 'Mon')`

__init_subclass__(*cls*, ***kwds*)

A *classmethod* that is used to further configure subsequent subclasses. By default, does nothing.

missing(*cls*, *value*)

A *classmethod* for looking up values not found in *cls*. By default it does nothing, but can be overridden to implement custom search behavior:

```
>>> from enum import StrEnum
>>> class Build(StrEnum):
...     DEBUG = auto()
...     OPTIMIZED = auto()
...     @classmethod
...     def _missing_(cls, value):
...         value = value.lower()
...         for member in cls:
...             if member.value == value:
...                 return member
...         return None
...
>>> Build.DEBUG.value
'debug'
>>> Build('deBUG')
<Build.DEBUG: 'debug'>
```

__new__(*cls*, **args*, ***kwds*)

By default, doesn't exist. If specified, either in the enum class definition or in a mixin class (such as `int`), all values given in the member assignment will be passed; e.g.

```
>>> from enum import Enum
>>> class MyIntEnum(int, Enum):
...     TWENTYSIX = '1a', 16
```

results in the call `int('1a', 16)` and a value of `26` for the member.

> **Note**
>
> When writing a custom `__new__`, do not use `super().__new__` – call the appropriate `__new__` instead.

__repr__(*self*)

Returns the string used for *repr()* calls. By default, returns the *Enum* name, member name, and value, but can be overridden:

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __repr__(self):
...         cls_name = self.__class__.__name__
```

(continues on next page)

(continued from previous page)

```
...         return f'{cls_name}.{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.ALTERNATE}"
(OtherStyle.ALTERNATE, 'OtherStyle.ALTERNATE', 'OtherStyle.ALTERNATE')
```

__str__ (*self*)

Returns the string used for *str()* calls. By default, returns the *Enum* name and member name, but can be overridden:

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __str__(self):
...         return f'{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.ALTERNATE}"
(<OtherStyle.ALTERNATE: 1>, 'ALTERNATE', 'ALTERNATE')
```

__format__ (*self*)

Returns the string used for *format()* and *f-string* calls. By default, returns *__str__()* return value, but can be overridden:

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __format__(self, spec):
...         return f'{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.ALTERNATE}"
(<OtherStyle.ALTERNATE: 1>, 'OtherStyle.ALTERNATE', 'ALTERNATE')
```

Note

Using *auto* with *Enum* results in integers of increasing value, starting with `1`.

Changed in version 3.12: Added enum-dataclass-support

class enum.**IntEnum**

IntEnum is the same as *Enum*, but its members are also integers and can be used anywhere that an integer can be used. If any integer operation is performed with an *IntEnum* member, the resulting value loses its enumeration status.

```
>>> from enum import IntEnum
>>> class Number(IntEnum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...
>>> Number.THREE
<Number.THREE: 3>
>>> Number.ONE + Number.TWO
3
```

(continues on next page)

(continued from previous page)

```
>>> Number.THREE + 5
8
>>> Number.THREE == 3
True
```

> **Note**
>
> Using *auto* with *IntEnum* results in integers of increasing value, starting with 1.

Changed in version 3.11: `__str__()` is now `int.__str__()` to better support the *replacement of existing constants* use-case. `__format__()` was already `int.__format__()` for that same reason.

class enum.**StrEnum**

`StrEnum` is the same as *Enum*, but its members are also strings and can be used in most of the same places that a string can be used. The result of any string operation performed on or with a *StrEnum* member is not part of the enumeration.

> **Note**
>
> There are places in the stdlib that check for an exact *str* instead of a *str* subclass (i.e. `type(unknown) == str` instead of `isinstance(unknown, str)`), and in those locations you will need to use `str(StrEnum.member)`.

> **Note**
>
> Using *auto* with *StrEnum* results in the lower-cased member name as the value.

> **Note**
>
> `__str__()` is `str.__str__()` to better support the *replacement of existing constants* use-case. `__format__()` is likewise `str.__format__()` for that same reason.

Added in version 3.11.

class enum.**Flag**

`Flag` is the same as *Enum*, but its members support the bitwise operators `&` (*AND*), `|` (*OR*), `^` (*XOR*), and `~` (*INVERT*); the results of those operations are (aliases of) members of the enumeration.

__contains__(*self*, *value*)

Returns *True* if value is in self:

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> purple = Color.RED | Color.BLUE
>>> white = Color.RED | Color.GREEN | Color.BLUE
```

(continues on next page)

(continued from previous page)

```
>>> Color.GREEN in purple
False
>>> Color.GREEN in white
True
>>> purple in white
True
>>> white in purple
False
```

__iter__(self):

Returns all contained non-alias members:

```
>>> list(Color.RED)
[<Color.RED: 1>]
>>> list(purple)
[<Color.RED: 1>, <Color.BLUE: 4>]
```

Added in version 3.11.

__len__(self):

Returns number of members in flag:

```
>>> len(Color.GREEN)
1
>>> len(white)
3
```

__bool__(self):

Returns *True* if any members in flag, *False* otherwise:

```
>>> bool(Color.GREEN)
True
>>> bool(white)
True
>>> black = Color(0)
>>> bool(black)
False
```

__or__ (*self*, *other*)

Returns current flag binary or'ed with other:

```
>>> Color.RED | Color.GREEN
<Color.RED|GREEN: 3>
```

__and__ (*self*, *other*)

Returns current flag binary and'ed with other:

```
>>> purple & white
<Color.RED|BLUE: 5>
>>> purple & Color.GREEN
<Color: 0>
```

__xor__ (*self*, *other*)

Returns current flag binary xor'ed with other:

```
>>> purple ^ white
<Color.GREEN: 2>
>>> purple ^ Color.GREEN
<Color.RED|GREEN|BLUE: 7>
```

__invert__(self):

Returns all the flags in *type(self)* that are not in *self*:

```
>>> ~white
<Color: 0>
>>> ~purple
<Color.GREEN: 2>
>>> ~Color.RED
<Color.GREEN|BLUE: 6>
```

_numeric_repr_()

Function used to format any remaining unnamed numeric values. Default is the value's repr; common choices are *hex()* and *oct()*.

Note

Using *auto* with *Flag* results in integers that are powers of two, starting with `1`.

Changed in version 3.11: The *repr()* of zero-valued flags has changed. It is now::

```
>>> Color(0)
<Color: 0>
```

class enum.**IntFlag**

`IntFlag` is the same as *Flag*, but its members are also integers and can be used anywhere that an integer can be used.

```
>>> from enum import IntFlag, auto
>>> class Color(IntFlag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> Color.RED & 2
<Color: 0>
>>> Color.RED | 2
<Color.RED|GREEN: 3>
```

If any integer operation is performed with an *IntFlag* member, the result is not an *IntFlag*:

```
>>> Color.RED + 2
3
```

If a *Flag* operation is performed with an *IntFlag* member and:

- the result is a valid *IntFlag*: an *IntFlag* is returned
- the result is not a valid *IntFlag*: the result depends on the *FlagBoundary* setting

The *repr()* of unnamed zero-valued flags has changed. It is now:

```
>>> Color(0)
<Color: 0>
```

Note

Using *auto* with *IntFlag* results in integers that are powers of two, starting with `1`.

Changed in version 3.11: `__str__()` is now `int.__str__()` to better support the *replacement of existing constants* use-case. `__format__()` was already `int.__format__()` for that same reason.

Inversion of an `IntFlag` now returns a positive value that is the union of all flags not in the given flag, rather than a negative value. This matches the existing *Flag* behavior.

class enum.**ReprEnum**

`ReprEnum` uses the *repr()* of *Enum*, but the *str()* of the mixed-in data type:

- `int.__str__()` for *IntEnum* and *IntFlag*
- `str.__str__()` for *StrEnum*

Inherit from `ReprEnum` to keep the *str()* / *format()* of the mixed-in data type instead of using the *Enum*-default *str()*.

Added in version 3.11.

class enum.**EnumCheck**

EnumCheck contains the options used by the *verify()* decorator to ensure various constraints; failed constraints result in a *ValueError*.

UNIQUE

Ensure that each value has only one name:

```
>>> from enum import Enum, verify, UNIQUE
>>> @verify(UNIQUE)
... class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     CRIMSON = 1
Traceback (most recent call last):
...
ValueError: aliases found in <enum 'Color'>: CRIMSON -> RED
```

CONTINUOUS

Ensure that there are no missing values between the lowest-valued member and the highest-valued member:

```
>>> from enum import Enum, verify, CONTINUOUS
>>> @verify(CONTINUOUS)
... class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 5
Traceback (most recent call last):
...
ValueError: invalid enum 'Color': missing values 3, 4
```

NAMED_FLAGS

Ensure that any flag groups/masks contain only named flags – useful when values are specified instead of being generated by *auto()*:

```
>>> from enum import Flag, verify, NAMED_FLAGS
>>> @verify(NAMED_FLAGS)
... class Color(Flag):
...     RED = 1
...     GREEN = 2
...     BLUE = 4
...     WHITE = 15
...     NEON = 31
Traceback (most recent call last):
...
ValueError: invalid Flag 'Color': aliases WHITE and NEON are missing combined␣
↪values of 0x18 [use enum.show_flag_values(value) for details]
```

Note

CONTINUOUS and NAMED_FLAGS are designed to work with integer-valued members.

Added in version 3.11.

class enum.**FlagBoundary**

`FlagBoundary` controls how out-of-range values are handled in *Flag* and its subclasses.

STRICT

Out-of-range values cause a *ValueError* to be raised. This is the default for *Flag*:

```
>>> from enum import Flag, STRICT, auto
>>> class StrictFlag(Flag, boundary=STRICT):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> StrictFlag(2**2 + 2**4)
Traceback (most recent call last):
...
ValueError: <flag 'StrictFlag'> invalid value 20
    given 0b0 10100
  allowed 0b0 00111
```

CONFORM

Out-of-range values have invalid values removed, leaving a valid *Flag* value:

```
>>> from enum import Flag, CONFORM, auto
>>> class ConformFlag(Flag, boundary=CONFORM):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> ConformFlag(2**2 + 2**4)
<ConformFlag.BLUE: 4>
```

EJECT

Out-of-range values lose their *Flag* membership and revert to *int*.

```
>>> from enum import Flag, EJECT, auto
>>> class EjectFlag(Flag, boundary=EJECT):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> EjectFlag(2**2 + 2**4)
20
```

KEEP

Out-of-range values are kept, and the *Flag* membership is kept. This is the default for *IntFlag*:

```
>>> from enum import Flag, KEEP, auto
>>> class KeepFlag(Flag, boundary=KEEP):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> KeepFlag(2**2 + 2**4)
<KeepFlag.BLUE|16: 20>
```

Added in version 3.11.

Supported __dunder__ names

__members__ is a read-only ordered mapping of `member_name:member` items. It is only available on the class.

__new__(), if specified, must create and return the enum members; it is also a very good idea to set the member's `_value_` appropriately. Once all the members are created it is no longer used.

Supported _sunder_ names

- *_add_alias_()* – adds a new name as an alias to an existing member.
- *_add_value_alias_()* – adds a new value as an alias to an existing member.
- *_name_* – name of the member
- *_value_* – value of the member; can be set in `__new__`
- *_missing_()* – a lookup function used when a value is not found; may be overridden
- *_ignore_* – a list of names, either as a *list* or a *str*, that will not be transformed into members, and will be removed from the final class
- *_order_* – no longer used, kept for backward compatibility (class attribute, removed during class creation)
- *_generate_next_value_()* – used to get an appropriate value for an enum member; may be overridden

> **Note**
>
> For standard *Enum* classes the next value chosen is the highest value seen incremented by one.
>
> For *Flag* classes the next value chosen will be the next highest power-of-two.

- While _sunder_ names are generally reserved for the further development of the *Enum* class and can not be used, some are explicitly allowed:
  - _repr_* (e.g. _repr_html_), as used in IPython's rich display

Added in version 3.6: _missing_, _order_, _generate_next_value_

Added in version 3.7: _ignore_

Added in version 3.13: _add_alias_, _add_value_alias_, _repr_*

8.14.3 Utilities and Decorators

class enum.**auto**

auto can be used in place of a value. If used, the *Enum* machinery will call an *Enum*'s *_generate_next_value_()* to get an appropriate value. For *Enum* and *IntEnum* that appropriate value will be the last value plus one; for *Flag* and *IntFlag* it will be the first power-of-two greater than the highest value; for *StrEnum* it will be the lower-cased version of the member's name. Care must be taken if mixing *auto()* with manually specified values.

auto instances are only resolved when at the top level of an assignment:

- FIRST = auto() will work (auto() is replaced with 1);
- SECOND = auto(), -2 will work (auto is replaced with 2, so 2, -2 is used to create the SECOND enum member;
- THREE = [auto(), -3] will *not* work (<auto instance>, -3 is used to create the THREE enum member)

Changed in version 3.11.1: In prior versions, auto() had to be the only thing on the assignment line to work properly.

_generate_next_value_ can be overridden to customize the values used by *auto*.

> **Note**
>
> in 3.13 the default _generate_next_value_ will always return the highest member value incremented by 1, and will fail if any member is an incompatible type.

@enum.**property**

A decorator similar to the built-in *property*, but specifically for enumerations. It allows member attributes to have the same names as members themselves.

> **Note**
>
> the *property* and the member must be defined in separate classes; for example, the *value* and *name* attributes are defined in the *Enum* class, and *Enum* subclasses can define members with the names value and name.

Added in version 3.11.

@enum.**unique**

A class decorator specifically for enumerations. It searches an enumeration's *__members__*, gathering any aliases it finds; if any are found *ValueError* is raised with the details:

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...     FOUR = 3
...
Traceback (most recent call last):
...
ValueError: duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```

`@enum.`**`verify`**

A `class` decorator specifically for enumerations. Members from *EnumCheck* are used to specify which constraints should be checked on the decorated enumeration.

Added in version 3.11.

`@enum.`**`member`**

A decorator for use in enums: its target will become a member.

Added in version 3.11.

`@enum.`**`nonmember`**

A decorator for use in enums: its target will not become a member.

Added in version 3.11.

`@enum.`**`global_enum`**

A decorator to change the *str()* and *repr()* of an enum to show its members as belonging to the module instead of its class. Should only be used when the enum members are exported to the module global namespace (see *re.RegexFlag* for an example).

Added in version 3.11.

`enum.`**`show_flag_values`**`(`*value*`)`

Return a list of all power-of-two integers contained in a flag *value*.

Added in version 3.11.

8.14.4 Notes

IntEnum, *StrEnum*, and *IntFlag*

These three enum types are designed to be drop-in replacements for existing integer- and string-based values; as such, they have extra limitations:

- `__str__` uses the value and not the name of the enum member
- `__format__`, because it uses `__str__`, will also use the value of the enum member instead of its name

If you do not need/want those limitations, you can either create your own base class by mixing in the `int` or `str` type yourself:

```
>>> from enum import Enum
>>> class MyIntEnum(int, Enum):
...     pass
```

or you can reassign the appropriate *str()*, etc., in your enum:

```
>>> from enum import Enum, IntEnum
>>> class MyIntEnum(IntEnum):
...     __str__ = Enum.__str__
```

8.15 graphlib — Functionality to operate with graph-like structures

Source code: Lib/graphlib.py

---

class graphlib.**TopologicalSorter**(*graph=None*)

Provides functionality to topologically sort a graph of *hashable* nodes.

A topological order is a linear ordering of the vertices in a graph such that for every directed edge u -> v from vertex u to vertex v, vertex u comes before vertex v in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this example, a topological ordering is just a valid sequence for the tasks. A complete topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph.

If the optional *graph* argument is provided it must be a dictionary representing a directed acyclic graph where the keys are nodes and the values are iterables of all predecessors of that node in the graph (the nodes that have edges that point to the value in the key). Additional nodes can be added to the graph using the *add()* method.

In the general case, the steps required to perform the sorting of a given graph are as follows:

- Create an instance of the *TopologicalSorter* with an optional initial graph.
- Add additional nodes to the graph.
- Call *prepare()* on the graph.
- While *is_active()* is `True`, iterate over the nodes returned by *get_ready()* and process them. Call *done()* on each node as it finishes processing.

In case just an immediate sorting of the nodes in the graph is required and no parallelism is involved, the convenience method *TopologicalSorter.static_order()* can be used directly:

```
>>> graph = {"D": {"B", "C"}, "C": {"A"}, "B": {"A"}}
>>> ts = TopologicalSorter(graph)
>>> tuple(ts.static_order())
('A', 'C', 'B', 'D')
```

The class is designed to easily support parallel processing of the nodes as they become ready. For instance:

```
topological_sorter = TopologicalSorter()

# Add nodes to 'topological_sorter'...

topological_sorter.prepare()
while topological_sorter.is_active():
    for node in topological_sorter.get_ready():
        # Worker threads or processes take nodes to work on off the
        # 'task_queue' queue.
        task_queue.put(node)
```

(continues on next page)

(continued from previous page)

```
    # When the work for a node is done, workers put the node in
    # 'finalized_tasks_queue' so we can get more nodes to work on.
    # The definition of 'is_active()' guarantees that, at this point, at
    # least one node has been placed on 'task_queue' that hasn't yet
    # been passed to 'done()', so this blocking 'get()' must (eventually)
    # succeed.  After calling 'done()', we loop back to call 'get_ready()'
    # again, so put newly freed nodes on 'task_queue' as soon as
    # logically possible.
    node = finalized_tasks_queue.get()
    topological_sorter.done(node)
```

add(*node*, **predecessors*)

Add a new node and its predecessors to the graph. Both the *node* and all elements in *predecessors* must be *hashable*.

If called multiple times with the same node argument, the set of dependencies will be the union of all dependencies passed in.

It is possible to add a node with no dependencies (*predecessors* is not provided) or to provide a dependency twice. If a node that has not been provided before is included among *predecessors* it will be automatically added to the graph with no predecessors of its own.

Raises *ValueError* if called after *prepare()*.

prepare()

Mark the graph as finished and check for cycles in the graph. If any cycle is detected, *CycleError* will be raised, but *get_ready()* can still be used to obtain as many nodes as possible until cycles block more progress. After a call to this function, the graph cannot be modified, and therefore no more nodes can be added using *add()*.

is_active()

Returns `True` if more progress can be made and `False` otherwise. Progress can be made if cycles do not block the resolution and either there are still nodes ready that haven't yet been returned by *TopologicalSorter.get_ready()* or the number of nodes marked *TopologicalSorter.done()* is less than the number that have been returned by *TopologicalSorter.get_ready()*.

The `__bool__()` method of this class defers to this function, so instead of:

```
if ts.is_active():
    ...
```

it is possible to simply do:

```
if ts:
    ...
```

Raises *ValueError* if called without calling *prepare()* previously.

done(**nodes*)

Marks a set of nodes returned by *TopologicalSorter.get_ready()* as processed, unblocking any successor of each node in *nodes* for being returned in the future by a call to *TopologicalSorter.get_ready()*.

Raises *ValueError* if any node in *nodes* has already been marked as processed by a previous call to this method or if a node was not added to the graph by using *TopologicalSorter.add()*, if called without calling *prepare()* or if node has not yet been returned by *get_ready()*.

get_ready()

Returns a `tuple` with all the nodes that are ready. Initially it returns all nodes with no predecessors, and once those are marked as processed by calling *TopologicalSorter.done()*, further calls will return all new nodes that have all their predecessors already processed. Once no more progress can be made, empty tuples are returned.

Raises *ValueError* if called without calling *prepare()* previously.

static_order()

Returns an iterator object which will iterate over nodes in a topological order. When using this method, *prepare()* and *done()* should not be called. This method is equivalent to:

```
def static_order(self):
    self.prepare()
    while self.is_active():
        node_group = self.get_ready()
        yield from node_group
        self.done(*node_group)
```

The particular order that is returned may depend on the specific order in which the items were inserted in the graph. For example:

```
>>> ts = TopologicalSorter()
>>> ts.add(3, 2, 1)
>>> ts.add(1, 0)
>>> print([*ts.static_order()])
[2, 0, 1, 3]

>>> ts2 = TopologicalSorter()
>>> ts2.add(1, 0)
>>> ts2.add(3, 2, 1)
>>> print([*ts2.static_order()])
[0, 2, 1, 3]
```

This is due to the fact that "0" and "2" are in the same level in the graph (they would have been returned in the same call to *get_ready()*) and the order between them is determined by the order of insertion.

If any cycle is detected, *CycleError* will be raised.

Added in version 3.9.

8.15.1 Exceptions

The *graphlib* module defines the following exception classes:

exception graphlib.**CycleError**

Subclass of *ValueError* raised by *TopologicalSorter.prepare()* if cycles exist in the working graph. If multiple cycles exist, only one undefined choice among them will be reported and included in the exception.

The detected cycle can be accessed via the second element in the *args* attribute of the exception instance and consists in a list of nodes, such that each node is, in the graph, an immediate predecessor of the next node in the list. In the reported list, the first and the last node will be the same, to make it clear that it is cyclic.

CHAPTER

NINE

NUMERIC AND MATHEMATICAL MODULES

The modules described in this chapter provide numeric and math-related functions and data types. The *numbers* module defines an abstract hierarchy of numeric types. The *math* and *cmath* modules contain various mathematical functions for floating-point and complex numbers. The *decimal* module supports exact representations of decimal numbers, using arbitrary precision arithmetic.

The following modules are documented in this chapter:

9.1 numbers — Numeric abstract base classes

Source code: Lib/numbers.py

The numbers module (**PEP 3141**) defines a hierarchy of numeric *abstract base classes* which progressively define more operations. None of the types defined in this module are intended to be instantiated.

class numbers.**Number**

The root of the numeric hierarchy. If you just want to check if an argument *x* is a number, without caring what kind, use `isinstance(x, Number)`.

9.1.1 The numeric tower

class numbers.**Complex**

Subclasses of this type describe complex numbers and include the operations that work on the built-in *complex* type. These are: conversions to *complex* and *bool*, *real*, *imag*, +, −, *, /, **, *abs()*, *conjugate()*, ==, and !=. All except − and != are abstract.

real

Abstract. Retrieves the real component of this number.

imag

Abstract. Retrieves the imaginary component of this number.

abstractmethod conjugate()

Abstract. Returns the complex conjugate. For example, `(1+3j).conjugate() == (1-3j)`.

class numbers.**Real**

To *Complex*, Real adds the operations that work on real numbers.

In short, those are: a conversion to *float*, *math.trunc()*, *round()*, *math.floor()*, *math.ceil()*, *divmod()*, //, %, <, <=, >, and >=.

Real also provides defaults for *complex()*, *real*, *imag*, and *conjugate()*.

class numbers.**Rational**

Subtypes *Real* and adds *numerator* and *denominator* properties. It also provides a default for *float()*.

The *numerator* and *denominator* values should be instances of *Integral* and should be in lowest terms with *denominator* positive.

numerator

Abstract.

denominator

Abstract.

class numbers.**Integral**

Subtypes *Rational* and adds a conversion to *int*. Provides defaults for *float()*, *numerator*, and *denominator*. Adds abstract methods for *pow()* with modulus and bit-string operations: <<, >>, &, ^, |, ~.

9.1.2 Notes for type implementers

Implementers should be careful to make equal numbers equal and hash them to the same values. This may be subtle if there are two different extensions of the real numbers. For example, *fractions.Fraction* implements *hash()* as follows:

```
def __hash__(self):
    if self.denominator == 1:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple's hash to avoid a high collision rate on
        # simple fractions.
        return hash((self.numerator, self.denominator))
```

Adding More Numeric ABCs

There are, of course, more possible ABCs for numbers, and this would be a poor hierarchy if it precluded the possibility of adding those. You can add `MyFoo` between *Complex* and *Real* with:

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

Implementing the arithmetic operations

We want to implement the arithmetic operations so that mixed-mode operations either call an implementation whose author knew about the types of both arguments, or convert both to the nearest built in type and do the operation there. For subtypes of *Integral*, this means that `__add__()` and `__radd__()` should be defined as:

```
class MyIntegral(Integral):

    def __add__(self, other):
```

(continues on next page)

(continued from previous page)

```
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(self, other)
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

There are 5 different cases for a mixed-type operation on subclasses of *Complex*. I'll refer to all of the above code that doesn't refer to `MyIntegral` and `OtherTypeIKnowAbout` as "boilerplate". `a` will be an instance of `A`, which is a subtype of *Complex* (`a : A <: Complex`), and `b : B <: Complex`. I'll consider `a + b`:

1. If `A` defines an `__add__()` which accepts `b`, all is well.
2. If `A` falls back to the boilerplate code, and it were to return a value from `__add__()`, we'd miss the possibility that `B` defines a more intelligent `__radd__()`, so the boilerplate should return *NotImplemented* from `__add__()`. (Or `A` may not implement `__add__()` at all.)
3. Then `B`'s `__radd__()` gets a chance. If it accepts `a`, all is well.
4. If it falls back to the boilerplate, there are no more possible methods to try, so this is where the default implementation should live.
5. If `B <: A`, Python tries `B.__radd__` before `A.__add__`. This is ok, because it was implemented with knowledge of `A`, so it can handle those instances before delegating to *Complex*.

If `A <: Complex` and `B <: Real` without sharing any other knowledge, then the appropriate shared operation is the one involving the built in *complex*, and both `__radd__()` s land there, so `a+b == b+a`.

Because most of the operations on any given type will be very similar, it can be useful to define a helper function which generates the forward and reverse instances of any given operator. For example, *fractions.Fraction* uses:

```
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__
```

(continues on next page)

(continued from previous page)

```
    def reverse(b, a):
        if isinstance(a, Rational):
            # Includes ints.
            return monomorphic_operator(a, b)
        elif isinstance(a, Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, Complex):
            return fallback_operator(complex(a), complex(b))
        else:
            return NotImplemented
    reverse.__name__ = '__r' + fallback_operator.__name__ + '__'
    reverse.__doc__ = monomorphic_operator.__doc__

    return forward, reverse

def _add(a, b):
    """a + b"""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)

# ...
```

9.2 `math` — Mathematical functions

This module provides access to the mathematical functions defined by the C standard.

These functions cannot be used with complex numbers; use the functions of the same name from the *cmath* module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don't is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.

9.2.1 Number-theoretic and representation functions

`math.`**`ceil`**`(`*x*`)`

Return the ceiling of *x*, the smallest integer greater than or equal to *x*. If *x* is not a float, delegates to `x.__ceil__`, which should return an *Integral* value.

`math.`**`comb`**`(`*n*, *k*`)`

Return the number of ways to choose *k* items from *n* items without repetition and without order.

Evaluates to `n! / (k! * (n - k)!)` when `k <= n` and evaluates to zero when `k > n`.

Also called the binomial coefficient because it is equivalent to the coefficient of k-th term in polynomial expansion of $(1 + x)^n$.

Raises *TypeError* if either of the arguments are not integers. Raises *ValueError* if either of the arguments are negative.

Added in version 3.8.

math.**copysign**(*x*, *y*)

Return a float with the magnitude (absolute value) of *x* but the sign of *y*. On platforms that support signed zeros, `copysign(1.0, -0.0)` returns *-1.0*.

math.**fabs**(*x*)

Return the absolute value of *x*.

math.**factorial**(*n*)

Return *n* factorial as an integer. Raises *ValueError* if *n* is not integral or is negative.

Changed in version 3.10: Floats with integral values (like `5.0`) are no longer accepted.

math.**floor**(*x*)

Return the floor of *x*, the largest integer less than or equal to *x*. If *x* is not a float, delegates to `x.__floor__`, which should return an *Integral* value.

math.**fma**(*x*, *y*, *z*)

Fused multiply-add operation. Return `(x * y) + z`, computed as though with infinite precision and range followed by a single round to the `float` format. This operation often provides better accuracy than the direct expression `(x * y) + z`.

This function follows the specification of the fusedMultiplyAdd operation described in the IEEE 754 standard. The standard leaves one case implementation-defined, namely the result of `fma(0, inf, nan)` and `fma(inf, 0, nan)`. In these cases, `math.fma` returns a NaN, and does not raise any exception.

Added in version 3.13.

math.**fmod**(*x*, *y*)

Return `fmod(x, y)`, as defined by the platform C library. Note that the Python expression `x % y` may not return the same result. The intent of the C standard is that `fmod(x, y)` be exactly (mathematically; to infinite precision) equal to `x - n*y` for some integer *n* such that the result has the same sign as *x* and magnitude less than `abs(y)`. Python's `x % y` returns a result with the sign of *y* instead, and may not be exactly computable for float arguments. For example, `fmod(-1e-100, 1e100)` is `-1e-100`, but the result of Python's `-1e-100 % 1e100` is `1e100-1e-100`, which cannot be represented exactly as a float, and rounds to the surprising `1e100`. For this reason, function *fmod()* is generally preferred when working with floats, while Python's `x % y` is preferred when working with integers.

math.**frexp**(*x*)

Return the mantissa and exponent of *x* as the pair `(m, e)`. *m* is a float and *e* is an integer such that `x == m * 2**e` exactly. If *x* is zero, returns `(0.0, 0)`, otherwise `0.5 <= abs(m) < 1`. This is used to "pick apart" the internal representation of a float in a portable way.

math.**fsum**(*iterable*)

Return an accurate floating-point sum of values in the iterable. Avoids loss of precision by tracking multiple intermediate partial sums.

The algorithm's accuracy depends on IEEE-754 arithmetic guarantees and the typical case where the rounding mode is half-even. On some non-Windows builds, the underlying C library uses extended precision addition and may occasionally double-round an intermediate sum causing it to be off in its least significant bit.

For further discussion and two alternative approaches, see the ASPN cookbook recipes for accurate floating-point summation.

math.**gcd**(**integers*)

Return the greatest common divisor of the specified integer arguments. If any of the arguments is nonzero, then the returned value is the largest positive integer that is a divisor of all arguments. If all arguments are zero, then the returned value is `0`. `gcd()` without arguments returns `0`.

Added in version 3.5.

Changed in version 3.9: Added support for an arbitrary number of arguments. Formerly, only two arguments were supported.

math.**isclose**(*a*, *b*, ***, *rel_tol=1e-09*, *abs_tol=0.0*)

Return `True` if the values *a* and *b* are close to each other and `False` otherwise.

Whether or not two values are considered close is determined according to given absolute and relative tolerances.

rel_tol is the relative tolerance – it is the maximum allowed difference between *a* and *b*, relative to the larger absolute value of *a* or *b*. For example, to set a tolerance of 5%, pass `rel_tol=0.05`. The default tolerance is `1e-09`, which assures that the two values are the same within about 9 decimal digits. *rel_tol* must be greater than zero.

abs_tol is the minimum absolute tolerance – useful for comparisons near zero. *abs_tol* must be at least zero.

If no errors occur, the result will be: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`.

The IEEE 754 special values of `NaN`, `inf`, and `-inf` will be handled according to IEEE rules. Specifically, `NaN` is not considered close to any other value, including `NaN`. `inf` and `-inf` are only considered close to themselves.

Added in version 3.5.

> **See also**
>
> **PEP 485** – A function for testing approximate equality

math.**isfinite**(*x*)

Return `True` if *x* is neither an infinity nor a NaN, and `False` otherwise. (Note that `0.0` *is* considered finite.)

Added in version 3.2.

math.**isinf**(*x*)

Return `True` if *x* is a positive or negative infinity, and `False` otherwise.

math.**isnan**(*x*)

Return `True` if *x* is a NaN (not a number), and `False` otherwise.

math.**isqrt**(*n*)

Return the integer square root of the nonnegative integer *n*. This is the floor of the exact square root of *n*, or equivalently the greatest integer *a* such that $a^2 \le n$.

For some applications, it may be more convenient to have the least integer *a* such that $n \le a^2$, or in other words the ceiling of the exact square root of *n*. For positive *n*, this can be computed using `a = 1 + isqrt(n - 1)`.

Added in version 3.8.

math.**lcm**(**integers*)

Return the least common multiple of the specified integer arguments. If all arguments are nonzero, then the returned value is the smallest positive integer that is a multiple of all arguments. If any of the arguments is zero, then the returned value is `0`. `lcm()` without arguments returns `1`.

Added in version 3.9.

math.**ldexp**(*x*, *i*)

Return `x * (2**i)`. This is essentially the inverse of function *frexp()*.

math.**modf**(*x*)

Return the fractional and integer parts of *x*. Both results carry the sign of *x* and are floats.

math.**nextafter**(*x*, *y*, *steps=1*)

Return the floating-point value *steps* steps after *x* towards *y*.

If *x* is equal to *y*, return *y*, unless *steps* is zero.

Examples:

- `math.nextafter(x, math.inf)` goes up: towards positive infinity.
- `math.nextafter(x, -math.inf)` goes down: towards minus infinity.
- `math.nextafter(x, 0.0)` goes towards zero.
- `math.nextafter(x, math.copysign(math.inf, x))` goes away from zero.

See also *math.ulp()*.

Added in version 3.9.

Changed in version 3.12: Added the *steps* argument.

math.**perm**(*n*, *k=None*)

Return the number of ways to choose *k* items from *n* items without repetition and with order.

Evaluates to `n! / (n - k)!` when `k <= n` and evaluates to zero when `k > n`.

If *k* is not specified or is `None`, then *k* defaults to *n* and the function returns `n!`.

Raises *TypeError* if either of the arguments are not integers. Raises *ValueError* if either of the arguments are negative.

Added in version 3.8.

math.**prod**(*iterable*, *, *start=1*)

Calculate the product of all the elements in the input *iterable*. The default *start* value for the product is `1`.

When the iterable is empty, return the start value. This function is intended specifically for use with numeric values and may reject non-numeric types.

Added in version 3.8.

math.**remainder**(*x*, *y*)

Return the IEEE 754-style remainder of *x* with respect to *y*. For finite *x* and finite nonzero *y*, this is the difference `x - n*y`, where `n` is the closest integer to the exact value of the quotient `x / y`. If `x / y` is exactly halfway between two consecutive integers, the nearest *even* integer is used for `n`. The remainder `r = remainder(x, y)` thus always satisfies `abs(r) <= 0.5 * abs(y)`.

Special cases follow IEEE 754: in particular, `remainder(x, math.inf)` is *x* for any finite *x*, and `remainder(x, 0)` and `remainder(math.inf, x)` raise *ValueError* for any non-NaN *x*. If the result of the remainder operation is zero, that zero will have the same sign as *x*.

On platforms using IEEE 754 binary floating point, the result of this operation is always exactly representable: no rounding error is introduced.

Added in version 3.7.

math.**sumprod**(*p*, *q*)

Return the sum of products of values from two iterables *p* and *q*.

Raises *ValueError* if the inputs do not have the same length.

Roughly equivalent to:

```
sum(itertools.starmap(operator.mul, zip(p, q, strict=True)))
```

For float and mixed int/float inputs, the intermediate products and sums are computed with extended precision.

Added in version 3.12.

math.**trunc**(*x*)

Return *x* with the fractional part removed, leaving the integer part. This rounds toward 0: `trunc()` is equivalent to *floor()* for positive *x*, and equivalent to *ceil()* for negative *x*. If *x* is not a float, delegates to `x.__trunc__`, which should return an *Integral* value.

math.**ulp**(*x*)

Return the value of the least significant bit of the float *x*:

- If *x* is a NaN (not a number), return *x*.
- If *x* is negative, return `ulp(-x)`.
- If *x* is a positive infinity, return *x*.
- If *x* is equal to zero, return the smallest positive *denormalized* representable float (smaller than the minimum positive *normalized* float, *sys.float_info.min*).
- If *x* is equal to the largest positive representable float, return the value of the least significant bit of *x*, such that the first float smaller than *x* is `x - ulp(x)`.
- Otherwise (*x* is a positive finite number), return the value of the least significant bit of *x*, such that the first float bigger than *x* is `x + ulp(x)`.

ULP stands for "Unit in the Last Place".

See also *math.nextafter()* and *sys.float_info.epsilon*.

Added in version 3.9.

Note that *frexp()* and *modf()* have a different call/return pattern than their C equivalents: they take a single argument and return a pair of values, rather than returning their second return value through an 'output parameter' (there is no such thing in Python).

For the *ceil()*, *floor()*, and *modf()* functions, note that *all* floating-point numbers of sufficiently large magnitude are exact integers. Python floats typically carry no more than 53 bits of precision (the same as the platform C double type), in which case any float *x* with `abs(x) >= 2**52` necessarily has no fractional bits.

9.2.2 Power and logarithmic functions

math.**cbrt**(*x*)

Return the cube root of *x*.

Added in version 3.11.

math.**exp**(*x*)

Return *e* raised to the power *x*, where *e* = 2.718281… is the base of natural logarithms. This is usually more accurate than `math.e ** x` or `pow(math.e, x)`.

math.**exp2**(*x*)

Return *2* raised to the power *x*.

Added in version 3.11.

math.**expm1**(*x*)

Return *e* raised to the power *x*, minus 1. Here *e* is the base of natural logarithms. For small floats *x*, the subtraction in `exp(x) - 1` can result in a significant loss of precision; the *expm1()* function provides a way to compute this quantity to full precision:

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

Added in version 3.2.

math.**log**(*x*[, *base*])

With one argument, return the natural logarithm of *x* (to base *e*).

With two arguments, return the logarithm of *x* to the given *base*, calculated as `log(x)/log(base)`.

math.**log1p**(*x*)

Return the natural logarithm of *1+x* (base *e*). The result is calculated in a way which is accurate for *x* near zero.

math.**log2**(*x*)

Return the base-2 logarithm of *x*. This is usually more accurate than `log(x, 2)`.

Added in version 3.3.

See also

int.bit_length() returns the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros.

math.**log10**(*x*)

Return the base-10 logarithm of *x*. This is usually more accurate than `log(x, 10)`.

math.**pow**(*x*, *y*)

Return `x` raised to the power `y`. Exceptional cases follow the IEEE 754 standard as far as possible. In particular, `pow(1.0, x)` and `pow(x, 0.0)` always return `1.0`, even when `x` is a zero or a NaN. If both `x` and `y` are finite, `x` is negative, and `y` is not an integer then `pow(x, y)` is undefined, and raises *ValueError*.

Unlike the built-in `**` operator, *math.pow()* converts both its arguments to type *float*. Use `**` or the built-in *pow()* function for computing exact integer powers.

Changed in version 3.11: The special cases `pow(0.0, -inf)` and `pow(-0.0, -inf)` were changed to return `inf` instead of raising *ValueError*, for consistency with IEEE 754.

math.**sqrt**(*x*)

Return the square root of *x*.

9.2.3 Trigonometric functions

`math.acos(x)`

Return the arc cosine of *x*, in radians. The result is between `0` and `pi`.

`math.asin(x)`

Return the arc sine of *x*, in radians. The result is between `-pi/2` and `pi/2`.

`math.atan(x)`

Return the arc tangent of *x*, in radians. The result is between `-pi/2` and `pi/2`.

`math.atan2(y, x)`

Return `atan(y / x)`, in radians. The result is between `-pi` and `pi`. The vector in the plane from the origin to point `(x, y)` makes this angle with the positive X axis. The point of *atan2()* is that the signs of both inputs are known to it, so it can compute the correct quadrant for the angle. For example, `atan(1)` and `atan2(1, 1)` are both `pi/4`, but `atan2(-1, -1)` is `-3*pi/4`.

`math.cos(x)`

Return the cosine of *x* radians.

`math.dist(p, q)`

Return the Euclidean distance between two points *p* and *q*, each given as a sequence (or iterable) of coordinates. The two points must have the same dimension.

Roughly equivalent to:

```
sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q)))
```

Added in version 3.8.

`math.hypot(*coordinates)`

Return the Euclidean norm, `sqrt(sum(x**2 for x in coordinates))`. This is the length of the vector from the origin to the point given by the coordinates.

For a two dimensional point `(x, y)`, this is equivalent to computing the hypotenuse of a right triangle using the Pythagorean theorem, `sqrt(x*x + y*y)`.

Changed in version 3.8: Added support for n-dimensional points. Formerly, only the two dimensional case was supported.

Changed in version 3.10: Improved the algorithm's accuracy so that the maximum error is under 1 ulp (unit in the last place). More typically, the result is almost always correctly rounded to within 1/2 ulp.

`math.sin(x)`

Return the sine of *x* radians.

`math.tan(x)`

Return the tangent of *x* radians.

9.2.4 Angular conversion

`math.`**`degrees`**`(x)`

Convert angle *x* from radians to degrees.

`math.`**`radians`**`(x)`

Convert angle *x* from degrees to radians.

9.2.5 Hyperbolic functions

Hyperbolic functions are analogs of trigonometric functions that are based on hyperbolas instead of circles.

`math.`**`acosh`**`(x)`

Return the inverse hyperbolic cosine of *x*.

`math.`**`asinh`**`(x)`

Return the inverse hyperbolic sine of *x*.

`math.`**`atanh`**`(x)`

Return the inverse hyperbolic tangent of *x*.

`math.`**`cosh`**`(x)`

Return the hyperbolic cosine of *x*.

`math.`**`sinh`**`(x)`

Return the hyperbolic sine of *x*.

`math.`**`tanh`**`(x)`

Return the hyperbolic tangent of *x*.

9.2.6 Special functions

`math.`**`erf`**`(x)`

Return the error function at *x*.

The *`erf()`* function can be used to compute traditional statistical functions such as the cumulative standard normal distribution:

```
def phi(x):
    'Cumulative distribution function for the standard normal distribution'
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

Added in version 3.2.

`math.`**`erfc`**`(x)`

Return the complementary error function at *x*. The complementary error function is defined as `1.0 - erf(x)`. It is used for large values of *x* where a subtraction from one would cause a loss of significance.

Added in version 3.2.

`math.`**`gamma`**`(x)`

Return the Gamma function at *x*.

Added in version 3.2.

math.**lgamma**(*x*)

Return the natural logarithm of the absolute value of the Gamma function at *x*.

Added in version 3.2.

9.2.7 Constants

math.**pi**

The mathematical constant $\pi = 3.141592\ldots$, to available precision.

math.**e**

The mathematical constant $e = 2.718281\ldots$, to available precision.

math.**tau**

The mathematical constant $\tau = 6.283185\ldots$, to available precision. Tau is a circle constant equal to 2π, the ratio of a circle's circumference to its radius. To learn more about Tau, check out Vi Hart's video Pi is (still) Wrong, and start celebrating Tau day by eating twice as much pie!

Added in version 3.6.

math.**inf**

A floating-point positive infinity. (For negative infinity, use `-math.inf`.) Equivalent to the output of `float('inf')`.

Added in version 3.5.

math.**nan**

A floating-point "not a number" (NaN) value. Equivalent to the output of `float('nan')`. Due to the requirements of the IEEE-754 standard, `math.nan` and `float('nan')` are not considered to equal to any other numeric value, including themselves. To check whether a number is a NaN, use the *`isnan()`* function to test for NaNs instead of `is` or ==. Example:

```
>>> import math
>>> math.nan == math.nan
False
>>> float('nan') == float('nan')
False
>>> math.isnan(math.nan)
True
>>> math.isnan(float('nan'))
True
```

Added in version 3.5.

Changed in version 3.11: It is now always available.

CPython implementation detail: The *`math`* module consists mostly of thin wrappers around the platform C math library functions. Behavior in exceptional cases follows Annex F of the C99 standard where appropriate. The current implementation will raise *`ValueError`* for invalid operations like `sqrt(-1.0)` or `log(0.0)` (where C99 Annex F recommends signaling invalid operation or divide-by-zero), and *`OverflowError`* for results that overflow (for example, `exp(1000.0)`). A NaN will not be returned from any of the functions above unless one or more of the input arguments was a NaN; in that case, most functions will return a NaN, but (again following C99 Annex F) there are some exceptions to this rule, for example `pow(float('nan'), 0.0)` or `hypot(float('nan'), float('inf'))`.

Note that Python makes no effort to distinguish signaling NaNs from quiet NaNs, and behavior for signaling NaNs remains unspecified. Typical behavior is to treat all NaNs as though they were quiet.

See also

Module *cmath*
Complex number versions of many of these functions.

9.3 cmath — Mathematical functions for complex numbers

This module provides access to mathematical functions for complex numbers. The functions in this module accept integers, floating-point numbers or complex numbers as arguments. They will also accept any Python object that has either a `__complex__()` or a `__float__()` method: these methods are used to convert the object to a complex or floating-point number, respectively, and the function is then applied to the result of the conversion.

Note

For functions involving branch cuts, we have the problem of deciding how to define those functions on the cut itself. Following Kahan's "Branch cuts for complex elementary functions" paper, as well as Annex G of C99 and later C standards, we use the sign of zero to distinguish one side of the branch cut from the other: for a branch cut along (a portion of) the real axis we look at the sign of the imaginary part, while for a branch cut along the imaginary axis we look at the sign of the real part.

For example, the *cmath.sqrt()* function has a branch cut along the negative real axis. An argument of `complex(-2.0, -0.0)` is treated as though it lies *below* the branch cut, and so gives a result on the negative imaginary axis:

```
>>> cmath.sqrt(complex(-2.0, -0.0))
-1.4142135623730951j
```

But an argument of `complex(-2.0, 0.0)` is treated as though it lies above the branch cut:

```
>>> cmath.sqrt(complex(-2.0, 0.0))
1.4142135623730951j
```

9.3.1 Conversions to and from polar coordinates

A Python complex number `z` is stored internally using *rectangular* or *Cartesian* coordinates. It is completely determined by its *real part* `z.real` and its *imaginary part* `z.imag`.

Polar coordinates give an alternative way to represent a complex number. In polar coordinates, a complex number z is defined by the modulus r and the phase angle *phi*. The modulus r is the distance from z to the origin, while the phase *phi* is the counterclockwise angle, measured in radians, from the positive x-axis to the line segment that joins the origin to z.

The following functions can be used to convert from the native rectangular coordinates to polar coordinates and back.

`cmath.phase(x)`

Return the phase of x (also known as the *argument* of x), as a float. `phase(x)` is equivalent to `math.atan2(x.imag, x.real)`. The result lies in the range $[-\pi, \pi]$, and the branch cut for this operation lies along the negative real axis. The sign of the result is the same as the sign of `x.imag`, even when `x.imag` is zero:

```
>>> phase(complex(-1.0, 0.0))
3.141592653589793
>>> phase(complex(-1.0, -0.0))
-3.141592653589793
```

Note

The modulus (absolute value) of a complex number *x* can be computed using the built-in `abs()` function. There is no separate `cmath` module function for this operation.

`cmath.`**`polar`**(*x*)

Return the representation of *x* in polar coordinates. Returns a pair `(r, phi)` where *r* is the modulus of *x* and phi is the phase of *x*. `polar(x)` is equivalent to `(abs(x), phase(x))`.

`cmath.`**`rect`**(*r*, *phi*)

Return the complex number *x* with polar coordinates *r* and *phi*. Equivalent to `complex(r * math.cos(phi), r * math.sin(phi))`.

9.3.2 Power and logarithmic functions

`cmath.`**`exp`**(*x*)

Return *e* raised to the power *x*, where *e* is the base of natural logarithms.

`cmath.`**`log`**(*x*[, *base*])

Returns the logarithm of *x* to the given *base*. If the *base* is not specified, returns the natural logarithm of *x*. There is one branch cut, from 0 along the negative real axis to -∞.

`cmath.`**`log10`**(*x*)

Return the base-10 logarithm of *x*. This has the same branch cut as `log()`.

`cmath.`**`sqrt`**(*x*)

Return the square root of *x*. This has the same branch cut as `log()`.

9.3.3 Trigonometric functions

`cmath.`**`acos`**(*x*)

Return the arc cosine of *x*. There are two branch cuts: One extends right from 1 along the real axis to ∞. The other extends left from -1 along the real axis to -∞.

`cmath.`**`asin`**(*x*)

Return the arc sine of *x*. This has the same branch cuts as `acos()`.

`cmath.`**`atan`**(*x*)

Return the arc tangent of *x*. There are two branch cuts: One extends from `1j` along the imaginary axis to ∞`j`. The other extends from −`1j` along the imaginary axis to −∞`j`.

`cmath.`**`cos`**(*x*)

Return the cosine of *x*.

`cmath.`**`sin`**(*x*)

Return the sine of *x*.

cmath.**tan**(*x*)

Return the tangent of *x*.

9.3.4 Hyperbolic functions

cmath.**acosh**(*x*)

Return the inverse hyperbolic cosine of *x*. There is one branch cut, extending left from 1 along the real axis to -∞.

cmath.**asinh**(*x*)

Return the inverse hyperbolic sine of *x*. There are two branch cuts: One extends from `1j` along the imaginary axis to ∞`j`. The other extends from `-1j` along the imaginary axis to −∞`j`.

cmath.**atanh**(*x*)

Return the inverse hyperbolic tangent of *x*. There are two branch cuts: One extends from `1` along the real axis to ∞. The other extends from `-1` along the real axis to −∞.

cmath.**cosh**(*x*)

Return the hyperbolic cosine of *x*.

cmath.**sinh**(*x*)

Return the hyperbolic sine of *x*.

cmath.**tanh**(*x*)

Return the hyperbolic tangent of *x*.

9.3.5 Classification functions

cmath.**isfinite**(*x*)

Return `True` if both the real and imaginary parts of *x* are finite, and `False` otherwise.

Added in version 3.2.

cmath.**isinf**(*x*)

Return `True` if either the real or the imaginary part of *x* is an infinity, and `False` otherwise.

cmath.**isnan**(*x*)

Return `True` if either the real or the imaginary part of *x* is a NaN, and `False` otherwise.

cmath.**isclose**(*a*, *b*, ***, *rel_tol=1e-09*, *abs_tol=0.0*)

Return `True` if the values *a* and *b* are close to each other and `False` otherwise.

Whether or not two values are considered close is determined according to given absolute and relative tolerances.

rel_tol is the relative tolerance – it is the maximum allowed difference between *a* and *b*, relative to the larger absolute value of *a* or *b*. For example, to set a tolerance of 5%, pass `rel_tol=0.05`. The default tolerance is `1e-09`, which assures that the two values are the same within about 9 decimal digits. *rel_tol* must be greater than zero.

abs_tol is the minimum absolute tolerance – useful for comparisons near zero. *abs_tol* must be at least zero.

If no errors occur, the result will be: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`.

The IEEE 754 special values of `NaN`, `inf`, and `-inf` will be handled according to IEEE rules. Specifically, `NaN` is not considered close to any other value, including `NaN`. `inf` and `-inf` are only considered close to themselves.

Added in version 3.5.

> **See also**
>
> **PEP 485** – A function for testing approximate equality

9.3.6 Constants

`cmath.`**`pi`**

The mathematical constant π, as a float.

`cmath.`**`e`**

The mathematical constant e, as a float.

`cmath.`**`tau`**

The mathematical constant τ, as a float.

Added in version 3.6.

`cmath.`**`inf`**

Floating-point positive infinity. Equivalent to `float('inf')`.

Added in version 3.6.

`cmath.`**`infj`**

Complex number with zero real part and positive infinity imaginary part. Equivalent to `complex(0.0, float('inf'))`.

Added in version 3.6.

`cmath.`**`nan`**

A floating-point "not a number" (NaN) value. Equivalent to `float('nan')`.

Added in version 3.6.

`cmath.`**`nanj`**

Complex number with zero real part and NaN imaginary part. Equivalent to `complex(0.0, float('nan'))`.

Added in version 3.6.

Note that the selection of functions is similar, but not identical, to that in module *math*. The reason for having two modules is that some users aren't interested in complex numbers, and perhaps don't even know what they are. They would rather have `math.sqrt(-1)` raise an exception than return a complex number. Also note that the functions defined in *cmath* always return a complex number, even if the answer can be expressed as a real number (in which case the complex number has an imaginary part of zero).

A note on branch cuts: They are curves along which the given function fails to be continuous. They are a necessary feature of many complex functions. It is assumed that if you need to compute with complex functions, you will understand about branch cuts. Consult almost any (not too elementary) book on complex variables for enlightenment. For information of the proper choice of branch cuts for numerical purposes, a good reference should be the following:

> **See also**
>
> Kahan, W: Branch cuts for complex elementary functions; or, Much ado about nothing's sign bit. In Iserles, A., and Powell, M. (eds.), The state of the art in numerical analysis. Clarendon Press (1987) pp165–211.

9.4 `decimal` — Decimal fixed-point and floating-point arithmetic

Source code: Lib/decimal.py

The *decimal* module provides support for fast correctly rounded decimal floating-point arithmetic. It offers several advantages over the *float* datatype:

- Decimal "is based on a floating-point model which was designed with people in mind, and necessarily has a paramount guiding principle – computers must provide an arithmetic that works in the same way as the arithmetic that people learn at school." – excerpt from the decimal arithmetic specification.
- Decimal numbers can be represented exactly. In contrast, numbers like `1.1` and `2.2` do not have exact representations in binary floating point. End users typically would not expect `1.1 + 2.2` to display as `3.3000000000000003` as it does with binary floating point.
- The exactness carries over into arithmetic. In decimal floating point, `0.1 + 0.1 + 0.1 - 0.3` is exactly equal to zero. In binary floating point, the result is `5.5511151231257827e-017`. While near to zero, the differences prevent reliable equality testing and differences can accumulate. For this reason, decimal is preferred in accounting applications which have strict equality invariants.
- The decimal module incorporates a notion of significant places so that `1.30 + 1.20` is `2.50`. The trailing zero is kept to indicate significance. This is the customary presentation for monetary applications. For multiplication, the "schoolbook" approach uses all the figures in the multiplicands. For instance, `1.3 * 1.2` gives `1.56` while `1.30 * 1.20` gives `1.5600`.
- Unlike hardware based binary floating point, the decimal module has a user alterable precision (defaulting to 28 places) which can be as large as needed for a given problem:

```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

- Both binary and decimal floating point are implemented in terms of published standards. While the built-in float type exposes only a modest portion of its capabilities, the decimal module exposes all required parts of the standard. When needed, the programmer has full control over rounding and signal handling. This includes an option to enforce exact arithmetic by using exceptions to block any inexact operations.
- The decimal module was designed to support "without prejudice, both exact unrounded decimal arithmetic (sometimes called fixed-point arithmetic) and rounded floating-point arithmetic." – excerpt from the decimal arithmetic specification.

The module design is centered around three concepts: the decimal number, the context for arithmetic, and signals.

A decimal number is immutable. It has a sign, coefficient digits, and an exponent. To preserve significance, the coefficient digits do not truncate trailing zeros. Decimals also include special values such as `Infinity`, `-Infinity`, and `NaN`. The standard also differentiates `-0` from `+0`.

The context for arithmetic is an environment specifying precision, rounding rules, limits on exponents, flags indicating the results of operations, and trap enablers which determine whether signals are treated as exceptions. Rounding options include *ROUND_CEILING*, *ROUND_DOWN*, *ROUND_FLOOR*, *ROUND_HALF_DOWN*, *ROUND_HALF_EVEN*, *ROUND_HALF_UP*, *ROUND_UP*, and *ROUND_05UP*.

Signals are groups of exceptional conditions arising during the course of computation. Depending on the needs of the application, signals may be ignored, considered as informational, or treated as exceptions. The signals in the decimal module are: *Clamped*, *InvalidOperation*, *DivisionByZero*, *Inexact*, *Rounded*, *Subnormal*, *Overflow*, *Underflow* and *FloatOperation*.

For each signal there is a flag and a trap enabler. When a signal is encountered, its flag is set to one, then, if the trap enabler is set to one, an exception is raised. Flags are sticky, so the user needs to reset them before monitoring a calculation.

See also

- IBM's General Decimal Arithmetic Specification, The General Decimal Arithmetic Specification.

9.4.1 Quick-start Tutorial

The usual start to using decimals is importing the module, viewing the current context with *getcontext()* and, if necessary, setting new values for precision, rounding, or enabled traps:

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[Overflow, DivisionByZero,
        InvalidOperation])

>>> getcontext().prec = 7       # Set a new precision
```

Decimal instances can be constructed from integers, strings, floats, or tuples. Construction from an integer or a float performs an exact conversion of the value of that integer or float. Decimal numbers include special values such as `NaN` which stands for "Not a number", positive and negative `Infinity`, and `-0`:

```
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000000124344978758017532527446746826171875')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.4142135623730951')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.4142135623730950488016887242')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

If the *FloatOperation* signal is trapped, accidental mixing of decimals and floats in constructors or ordering comparisons raises an exception:

```
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
>>> Decimal(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(continues on next page)

(continued from previous page)

```
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') == 3.5
True
```

Added in version 3.3.

The significance of a new Decimal is determined solely by the number of digits input. Context precision and rounding only come into play during arithmetic operations.

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

If the internal limits of the C version are exceeded, constructing a decimal raises *InvalidOperation*:

```
>>> Decimal("1e9999999999999999999")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.InvalidOperation: [<class 'decimal.InvalidOperation'>]
```

Changed in version 3.3.

Decimals interact well with much of the rest of Python. Here is a small decimal floating-point flying circus:

```
>>> data = list(map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split()))
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
 Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)
Decimal('1.3')
>>> int(a)
1
>>> a * 5
Decimal('6.70')
>>> a * b
```

(continues on next page)

(continued from previous page)

```
Decimal('2.5058')
>>> c % a
Decimal('0.77')
```

And some mathematical functions are also available to Decimal:

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.414213562373095048801688724')
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal('10').ln()
Decimal('2.302585092994045684017991455')
>>> Decimal('10').log10()
Decimal('1')
```

The *quantize()* method rounds a number to a fixed exponent. This method is useful for monetary applications that often round results to a fixed number of places:

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

As shown above, the *getcontext()* function accesses the current context and allows the settings to be changed. This approach meets the needs of most applications.

For more advanced work, it may be useful to create alternate contexts using the Context() constructor. To make an alternate active, use the *setcontext()* function.

In accordance with the standard, the *decimal* module provides two ready to use standard contexts, *BasicContext* and *ExtendedContext*. The former is especially useful for debugging because many of the traps are enabled:

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

Contexts also have signal flags for monitoring exceptional conditions encountered during computations. The flags remain set until explicitly cleared, so it is best to clear the flags before each set of monitored computations by using the *clear_flags()* method.

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[Inexact, Rounded], traps=[])
```

The *flags* entry shows that the rational approximation to pi was rounded (digits beyond the context precision were thrown away) and that the result is inexact (some of the discarded digits were non-zero).

Individual traps are set using the dictionary in the `traps` attribute of a context:

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

Most programs adjust the current context only once, at the beginning of the program. And, in many applications, data is converted to *Decimal* with a single cast inside a loop. With context set and decimals created, the bulk of the program manipulates the data no differently than with other Python numeric types.

9.4.2 Decimal objects

class decimal.**Decimal**(*value='0'*, *context=None*)

Construct a new *Decimal* object based from *value*.

value can be an integer, string, tuple, *float*, or another *Decimal* object. If no *value* is given, returns `Decimal('0')`. If *value* is a string, it should conform to the decimal numeric string syntax after leading and trailing whitespace characters, as well as underscores throughout, are removed:

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
indicator      ::=  'e' | 'E'
digits         ::=  digit [digit]...
decimal-part   ::=  digits '.' [digits] | ['.'] digits
exponent-part  ::=  indicator [sign] digits
infinity       ::=  'Infinity' | 'Inf'
nan            ::=  'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | [sign] nan
```

Other Unicode decimal digits are also permitted where `digit` appears above. These include decimal digits from various other alphabets (for example, Arabic-Indic and Devanāgarī digits) along with the fullwidth digits `'\uff10'` through `'\uff19'`.

If *value* is a *tuple*, it should have three components, a sign (`0` for positive or `1` for negative), a *tuple* of digits, and an integer exponent. For example, `Decimal((0, (1, 4, 1, 4), -3))` returns `Decimal('1.414')`.

If *value* is a *float*, the binary floating-point value is losslessly converted to its exact decimal equivalent. This conversion can often require 53 or more digits of precision. For example, `Decimal(float('1.1'))` converts

to `Decimal('1.1000000000000000888178419700125232338905334472656 25')`.

The *context* precision does not affect how many digits are stored. That is determined exclusively by the number of digits in *value*. For example, `Decimal('3.00000')` records all five zeros even if the context precision is only three.

The purpose of the *context* argument is determining what to do if *value* is a malformed string. If the context traps *InvalidOperation*, an exception is raised; otherwise, the constructor returns a new Decimal with the value of `NaN`.

Once constructed, *Decimal* objects are immutable.

Changed in version 3.2: The argument to the constructor is now permitted to be a *float* instance.

Changed in version 3.3: *float* arguments raise an exception if the *FloatOperation* trap is set. By default the trap is off.

Changed in version 3.6: Underscores are allowed for grouping, as with integral and floating-point literals in code.

Decimal floating-point objects share many properties with the other built-in numeric types such as *float* and *int*. All of the usual math operations and special methods apply. Likewise, decimal objects can be copied, pickled, printed, used as dictionary keys, used as set elements, compared, sorted, and coerced to another type (such as *float* or *int*).

There are some small differences between arithmetic on Decimal objects and arithmetic on integers and floats. When the remainder operator `%` is applied to Decimal objects, the sign of the result is the sign of the *dividend* rather than the sign of the divisor:

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

The integer division operator `//` behaves analogously, returning the integer part of the true quotient (truncating towards zero) rather than its floor, so as to preserve the usual identity `x == (x // y) * y + x % y`:

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

The `%` and `//` operators implement the `remainder` and `divide-integer` operations (respectively) as described in the specification.

Decimal objects cannot generally be combined with floats or instances of *fractions.Fraction* in arithmetic operations: an attempt to add a *Decimal* to a *float*, for example, will raise a *TypeError*. However, it is possible to use Python's comparison operators to compare a *Decimal* instance `x` with another number `y`. This avoids confusing results when doing equality comparisons between numbers of different types.

Changed in version 3.2: Mixed-type comparisons between *Decimal* instances and other numeric types are now fully supported.

In addition to the standard numeric properties, decimal floating-point objects also have a number of specialized methods:

adjusted()

Return the adjusted exponent after shifting out the coefficient's rightmost digits until only the lead digit remains: `Decimal('321e+5').adjusted()` returns seven. Used for determining the position of the most significant digit with respect to the decimal point.

as_integer_ratio()

Return a pair `(n, d)` of integers that represent the given *Decimal* instance as a fraction, in lowest terms and with a positive denominator:

```
>>> Decimal('-3.14').as_integer_ratio()
(-157, 50)
```

The conversion is exact. Raise OverflowError on infinities and ValueError on NaNs.

Added in version 3.6.

as_tuple()

Return a *named tuple* representation of the number: `DecimalTuple(sign, digits, exponent)`.

canonical()

Return the canonical encoding of the argument. Currently, the encoding of a *Decimal* instance is always canonical, so this operation returns its argument unchanged.

compare(*other*, *context=None*)

Compare the values of two Decimal instances. *compare()* returns a Decimal instance, and if either operand is a NaN then the result is a NaN:

```
a or b is a NaN  ==> Decimal('NaN')
a < b            ==> Decimal('-1')
a == b           ==> Decimal('0')
a > b            ==> Decimal('1')
```

compare_signal(*other*, *context=None*)

This operation is identical to the *compare()* method, except that all NaNs signal. That is, if neither operand is a signaling NaN then any quiet NaN operand is treated as though it were a signaling NaN.

compare_total(*other*, *context=None*)

Compare two operands using their abstract representation rather than their numerical value. Similar to the *compare()* method, but the result gives a total ordering on *Decimal* instances. Two *Decimal* instances with the same numeric value but different representations compare unequal in this ordering:

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

Quiet and signaling NaNs are also included in the total ordering. The result of this function is `Decimal('0')` if both operands have the same representation, `Decimal('-1')` if the first operand is lower in the total order than the second, and `Decimal('1')` if the first operand is higher in the total order than the second operand. See the specification for details of the total order.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

compare_total_mag(*other*, *context=None*)

Compare two operands using their abstract representation rather than their value as in *compare_total()*, but ignoring the sign of each operand. `x.compare_total_mag(y)` is equivalent to `x.copy_abs().compare_total(y.copy_abs())`.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

conjugate()

Just returns self, this method is only to comply with the Decimal Specification.

copy_abs()

Return the absolute value of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

copy_negate()

Return the negation of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

copy_sign(*other*, *context=None*)

Return a copy of the first operand with the sign set to be the same as the sign of the second operand. For example:

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

exp(*context=None*)

Return the value of the (natural) exponential function `e**x` at the given number. The result is correctly rounded using the *ROUND_HALF_EVEN* rounding mode.

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

classmethod from_float(*f*)

Alternative constructor that only accepts instances of *float* or *int*.

Note `Decimal.from_float(0.1)` is not the same as `Decimal('0.1')`. Since 0.1 is not exactly representable in binary floating point, the value is stored as the nearest representable value which is `0x1.999999999999ap-4`. That equivalent value in decimal is `0.1000000000000000055511151231257827021181583404541015625`.

Note

From Python 3.2 onwards, a *Decimal* instance can also be constructed directly from a *float*.

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

Added in version 3.1.

fma(*other*, *third*, *context=None*)

Fused multiply-add. Return self*other+third with no rounding of the intermediate product self*other.

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

is_canonical()

Return *True* if the argument is canonical and *False* otherwise. Currently, a *Decimal* instance is always canonical, so this operation always returns *True*.

is_finite()

Return *True* if the argument is a finite number, and *False* if the argument is an infinity or a NaN.

is_infinite()

Return *True* if the argument is either positive or negative infinity and *False* otherwise.

is_nan()

Return *True* if the argument is a (quiet or signaling) NaN and *False* otherwise.

is_normal(*context=None*)

Return *True* if the argument is a *normal* finite number. Return *False* if the argument is zero, subnormal, infinite or a NaN.

is_qnan()

Return *True* if the argument is a quiet NaN, and *False* otherwise.

is_signed()

Return *True* if the argument has a negative sign and *False* otherwise. Note that zeros and NaNs can both carry signs.

is_snan()

Return *True* if the argument is a signaling NaN and *False* otherwise.

is_subnormal(*context=None*)

Return *True* if the argument is subnormal, and *False* otherwise.

is_zero()

Return *True* if the argument is a (positive or negative) zero and *False* otherwise.

ln(*context=None*)

Return the natural (base e) logarithm of the operand. The result is correctly rounded using the *ROUND_HALF_EVEN* rounding mode.

log10(*context=None*)

Return the base ten logarithm of the operand. The result is correctly rounded using the *ROUND_HALF_EVEN* rounding mode.

logb(*context=None*)

For a nonzero number, return the adjusted exponent of its operand as a *Decimal* instance. If the operand is a zero then `Decimal('-Infinity')` is returned and the *DivisionByZero* flag is raised. If the operand is an infinity then `Decimal('Infinity')` is returned.

logical_and(*other*, *context=None*)

logical_and() is a logical operation which takes two *logical operands* (see *Logical operands*). The result is the digit-wise `and` of the two operands.

logical_invert(*context=None*)

logical_invert() is a logical operation. The result is the digit-wise inversion of the operand.

logical_or(*other*, *context=None*)

logical_or() is a logical operation which takes two *logical operands* (see *Logical operands*). The result is the digit-wise `or` of the two operands.

logical_xor(*other*, *context=None*)

logical_xor() is a logical operation which takes two *logical operands* (see *Logical operands*). The result is the digit-wise exclusive or of the two operands.

max(*other*, *context=None*)

Like `max(self, other)` except that the context rounding rule is applied before returning and that `NaN` values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

max_mag(*other*, *context=None*)

Similar to the *max()* method, but the comparison is done using the absolute values of the operands.

min(*other*, *context=None*)

Like `min(self, other)` except that the context rounding rule is applied before returning and that `NaN` values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

min_mag(*other*, *context=None*)

Similar to the *min()* method, but the comparison is done using the absolute values of the operands.

next_minus(*context=None*)

Return the largest number representable in the given context (or in the current thread's context if no context is given) that is smaller than the given operand.

next_plus(*context=None*)

Return the smallest number representable in the given context (or in the current thread's context if no context is given) that is larger than the given operand.

next_toward(*other*, *context=None*)

If the two operands are unequal, return the number closest to the first operand in the direction of the second operand. If both operands are numerically equal, return a copy of the first operand with the sign set to be the same as the sign of the second operand.

normalize(*context=None*)

Used for producing canonical values of an equivalence class within either the current context or the specified context.

This has the same semantics as the unary plus operation, except that if the final result is finite it is reduced to its simplest form, with all trailing zeros removed and its sign preserved. That is, while the coefficient is non-zero and a multiple of ten the coefficient is divided by ten and the exponent is incremented by 1. Otherwise (the coefficient is zero) the exponent is set to 0. In all cases the sign is unchanged.

For example, `Decimal('32.100')` and `Decimal('0.321000e+2')` both normalize to the equivalent value `Decimal('32.1')`.

Note that rounding is applied *before* reducing to simplest form.

In the latest versions of the specification, this operation is also known as `reduce`.

number_class(*context=None*)

Return a string describing the *class* of the operand. The returned value is one of the following ten strings.

- `"-Infinity"`, indicating that the operand is negative infinity.
- `"-Normal"`, indicating that the operand is a negative normal number.
- `"-Subnormal"`, indicating that the operand is negative and subnormal.
- `"-Zero"`, indicating that the operand is a negative zero.
- `"+Zero"`, indicating that the operand is a positive zero.
- `"+Subnormal"`, indicating that the operand is positive and subnormal.

- `"+Normal"`, indicating that the operand is a positive normal number.
- `"+Infinity"`, indicating that the operand is positive infinity.
- `"NaN"`, indicating that the operand is a quiet NaN (Not a Number).
- `"sNaN"`, indicating that the operand is a signaling NaN.

quantize(*exp*, *rounding=None*, *context=None*)

Return a value equal to the first operand after rounding and having the exponent of the second operand.

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

Unlike other operations, if the length of the coefficient after the quantize operation would be greater than precision, then an *InvalidOperation* is signaled. This guarantees that, unless there is an error condition, the quantized exponent is always equal to that of the right-hand operand.

Also unlike other operations, quantize never signals Underflow, even if the result is subnormal and inexact.

If the exponent of the second operand is larger than that of the first then rounding may be necessary. In this case, the rounding mode is determined by the `rounding` argument if given, else by the given `context` argument; if neither argument is given the rounding mode of the current thread's context is used.

An error is returned whenever the resulting exponent is greater than `Emax` or less than *Etiny()*.

radix()

Return `Decimal(10)`, the radix (base) in which the *Decimal* class does all its arithmetic. Included for compatibility with the specification.

remainder_near(*other*, *context=None*)

Return the remainder from dividing *self* by *other*. This differs from `self % other` in that the sign of the remainder is chosen so as to minimize its absolute value. More precisely, the return value is `self - n * other` where `n` is the integer nearest to the exact value of `self / other`, and if two integers are equally near then the even one is chosen.

If the result is zero then its sign will be the sign of *self*.

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate(*other*, *context=None*)

Return the result of rotating the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to rotate. If the second operand is positive then rotation is to the left; otherwise rotation is to the right. The coefficient of the first operand is padded on the left with zeros to length precision if necessary. The sign and exponent of the first operand are unchanged.

same_quantum(*other*, *context=None*)

Test whether self and other have the same exponent or whether both are `NaN`.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

scaleb(*other*, *context=None*)

Return the first operand with exponent adjusted by the second. Equivalently, return the first operand multiplied by `10**other`. The second operand must be an integer.

shift(*other*, *context=None*)

Return the result of shifting the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to shift. If the second operand is positive then the shift is to the left; otherwise the shift is to the right. Digits shifted into the coefficient are zeros. The sign and exponent of the first operand are unchanged.

sqrt(*context=None*)

Return the square root of the argument to full precision.

to_eng_string(*context=None*)

Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

For example, this converts `Decimal('123E+1')` to `Decimal('1.23E+3')`.

to_integral(*rounding=None*, *context=None*)

Identical to the *to_integral_value()* method. The `to_integral` name has been kept for compatibility with older versions.

to_integral_exact(*rounding=None*, *context=None*)

Round to the nearest integer, signaling *Inexact* or *Rounded* as appropriate if rounding occurs. The rounding mode is determined by the `rounding` parameter if given, else by the given `context`. If neither parameter is given then the rounding mode of the current context is used.

to_integral_value(*rounding=None*, *context=None*)

Round to the nearest integer without signaling *Inexact* or *Rounded*. If given, applies *rounding*; otherwise, uses the rounding method in either the supplied *context* or the current context.

Decimal numbers can be rounded using the *round()* function:

round(number)

round(number, ndigits)

If *ndigits* is not given or `None`, returns the nearest *int* to *number*, rounding ties to even, and ignoring the rounding mode of the *Decimal* context. Raises *OverflowError* if *number* is an infinity or *ValueError* if it is a (quiet or signaling) NaN.

If *ndigits* is an *int*, the context's rounding mode is respected and a *Decimal* representing *number* rounded to the nearest multiple of `Decimal('1E-ndigits')` is returned; in this case, `round(number, ndigits)` is equivalent to `self.quantize(Decimal('1E-ndigits'))`. Returns `Decimal('NaN')` if *number* is a quiet NaN. Raises *InvalidOperation* if *number* is an infinity, a signaling NaN, or if the length of the coefficient after the quantize operation would be greater than the current context's precision. In other words, for the non-corner cases:

- if *ndigits* is positive, return *number* rounded to *ndigits* decimal places;
- if *ndigits* is zero, return *number* rounded to the nearest integer;
- if *ndigits* is negative, return *number* rounded to the nearest multiple of `10**abs(ndigits)`.

For example:

```
>>> from decimal import Decimal, getcontext, ROUND_DOWN
>>> getcontext().rounding = ROUND_DOWN
>>> round(Decimal('3.75'))      # context rounding ignored
4
>>> round(Decimal('3.5'))       # round-ties-to-even
4
>>> round(Decimal('3.75'), 0)   # uses the context rounding
Decimal('3')
>>> round(Decimal('3.75'), 1)
Decimal('3.7')
>>> round(Decimal('3.75'), -1)
Decimal('0E+1')
```

Logical operands

The *logical_and()*, *logical_invert()*, *logical_or()*, and *logical_xor()* methods expect their arguments to be *logical operands*. A *logical operand* is a *Decimal* instance whose exponent and sign are both zero, and whose digits are all either 0 or 1.

9.4.3 Context objects

Contexts are environments for arithmetic operations. They govern precision, set rules for rounding, determine which signals are treated as exceptions, and limit the range for exponents.

Each thread has its own current context which is accessed or changed using the *getcontext()* and *setcontext()* functions:

decimal.**getcontext**()

Return the current context for the active thread.

decimal.**setcontext**(*c*)

Set the current context for the active thread to *c*.

You can also use the with statement and the *localcontext()* function to temporarily change the active context.

decimal.**localcontext**(*ctx=None*, ***kwargs*)

Return a context manager that will set the current context for the active thread to a copy of *ctx* on entry to the with-statement and restore the previous context when exiting the with-statement. If no context is specified, a copy of the current context is used. The *kwargs* argument is used to set the attributes of the new context.

For example, the following code sets the current decimal precision to 42 places, performs a calculation, and then automatically restores the previous context:

```
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42   # Perform a high precision calculation
    s = calculate_something()
s = +s  # Round the final result back to the default precision
```

Using keyword arguments, the code would be the following:

```
from decimal import localcontext

with localcontext(prec=42) as ctx:
```

(continues on next page)

(continued from previous page)

```
    s = calculate_something()
s = +s
```

Raises *TypeError* if *kwargs* supplies an attribute that *Context* doesn't support. Raises either *TypeError* or *ValueError* if *kwargs* supplies an invalid value for an attribute.

Changed in version 3.11: *localcontext()* now supports setting context attributes through the use of keyword arguments.

New contexts can also be created using the *Context* constructor described below. In addition, the module provides three pre-made contexts:

class decimal.**BasicContext**

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to *ROUND_HALF_UP*. All flags are cleared. All traps are enabled (treated as exceptions) except *Inexact*, *Rounded*, and *Subnormal*.

Because many of the traps are enabled, this context is useful for debugging.

class decimal.**ExtendedContext**

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to *ROUND_HALF_EVEN*. All flags are cleared. No traps are enabled (so that exceptions are not raised during computations).

Because the traps are disabled, this context is useful for applications that prefer to have result value of `NaN` or `Infinity` instead of raising exceptions. This allows an application to complete a run in the presence of conditions that would otherwise halt the program.

class decimal.**DefaultContext**

This context is used by the *Context* constructor as a prototype for new contexts. Changing a field (such a precision) has the effect of changing the default for new contexts created by the *Context* constructor.

This context is most useful in multi-threaded environments. Changing one of the fields before threads are started has the effect of setting system-wide defaults. Changing the fields after threads have started is not recommended as it would require thread synchronization to prevent race conditions.

In single threaded environments, it is preferable to not use this context at all. Instead, simply create contexts explicitly as described below.

The default values are `Context.prec=28`, `Context.rounding=`*ROUND_HALF_EVEN*, and enabled traps for *Overflow*, *InvalidOperation*, and *DivisionByZero*.

In addition to the three supplied contexts, new contexts can be created with the *Context* constructor.

class decimal.**Context**(*prec=None*, *rounding=None*, *Emin=None*, *Emax=None*, *capitals=None*, *clamp=None*, *flags=None*, *traps=None*)

Creates a new context. If a field is not specified or is *None*, the default values are copied from the *DefaultContext*. If the *flags* field is not specified or is *None*, all flags are cleared.

prec is an integer in the range [`1`, *MAX_PREC*] that sets the precision for arithmetic operations in the context.

The *rounding* option is one of the constants listed in the section *Rounding Modes*.

The *traps* and *flags* fields list any signals to be set. Generally, new contexts should only set traps and leave the flags clear.

The *Emin* and *Emax* fields are integers specifying the outer limits allowable for exponents. *Emin* must be in the range [*MIN_EMIN*, `0`], *Emax* in the range [`0`, *MAX_EMAX*].

The *capitals* field is either `0` or `1` (the default). If set to `1`, exponents are printed with a capital `E`; otherwise, a lowercase `e` is used: `Decimal('6.02e+23')`.

The *clamp* field is either `0` (the default) or `1`. If set to `1`, the exponent `e` of a *Decimal* instance representable in this context is strictly limited to the range `Emin - prec + 1 <= e <= Emax - prec + 1`. If *clamp* is `0` then a weaker condition holds: the adjusted exponent of the *Decimal* instance is at most `Emax`. When *clamp* is `1`, a large normal number will, where possible, have its exponent reduced and a corresponding number of zeros added to its coefficient, in order to fit the exponent constraints; this preserves the value of the number but loses information about significant trailing zeros. For example:

```
>>> Context(prec=6, Emax=999, clamp=1).create_decimal('1.23e999')
Decimal('1.23000E+999')
```

A *clamp* value of `1` allows compatibility with the fixed-width decimal interchange formats specified in IEEE 754.

The *Context* class defines several general purpose methods as well as a large number of methods for doing arithmetic directly in a given context. In addition, for each of the *Decimal* methods described above (with the exception of the *adjusted()* and *as_tuple()* methods) there is a corresponding *Context* method. For example, for a *Context* instance `C` and *Decimal* instance `x`, `C.exp(x)` is equivalent to `x.exp(context=C)`. Each *Context* method accepts a Python integer (an instance of *int*) anywhere that a Decimal instance is accepted.

clear_flags()

Resets all of the flags to `0`.

clear_traps()

Resets all of the traps to `0`.

Added in version 3.3.

copy()

Return a duplicate of the context.

copy_decimal(*num*)

Return a copy of the Decimal instance num.

create_decimal(*num*)

Creates a new Decimal instance from *num* but using *self* as context. Unlike the *Decimal* constructor, the context precision, rounding method, flags, and traps are applied to the conversion.

This is useful because constants are often given to a greater precision than is needed by the application. Another benefit is that rounding immediately eliminates unintended effects from digits beyond the current precision. In the following example, using unrounded inputs means that adding zero to a sum can change the result:

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

This method implements the to-number operation of the IBM specification. If the argument is a string, no leading or trailing whitespace or underscores are permitted.

create_decimal_from_float(*f*)

Creates a new Decimal instance from a float *f* but rounding using *self* as the context. Unlike the *Decimal.from_float()* class method, the context precision, rounding method, flags, and traps are applied to the conversion.

```
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
    ...
decimal.Inexact: None
```

Added in version 3.1.

Etiny()

Returns a value equal to `Emin - prec + 1` which is the minimum exponent value for subnormal results. When underflow occurs, the exponent is set to *Etiny*.

Etop()

Returns a value equal to `Emax - prec + 1`.

The usual approach to working with decimals is to create *Decimal* instances and then apply arithmetic operations which take place within the current context for the active thread. An alternative approach is to use context methods for calculating within a specific context. The methods are similar to those for the *Decimal* class and are only briefly recounted here.

abs(*x*)

Returns the absolute value of *x*.

add(*x*, *y*)

Return the sum of *x* and *y*.

canonical(*x*)

Returns the same Decimal object *x*.

compare(*x*, *y*)

Compares *x* and *y* numerically.

compare_signal(*x*, *y*)

Compares the values of the two operands numerically.

compare_total(*x*, *y*)

Compares two operands using their abstract representation.

compare_total_mag(*x*, *y*)

Compares two operands using their abstract representation, ignoring sign.

copy_abs(*x*)

Returns a copy of *x* with the sign set to 0.

copy_negate(*x*)

Returns a copy of *x* with the sign inverted.

copy_sign(*x*, *y*)

Copies the sign from *y* to *x*.

divide(*x*, *y*)

Return *x* divided by *y*.

divide_int(*x*, *y*)

Return *x* divided by *y*, truncated to an integer.

divmod(*x*, *y*)

Divides two numbers and returns the integer part of the result.

exp(*x*)

Returns `e ** x`.

fma(*x*, *y*, *z*)

Returns *x* multiplied by *y*, plus *z*.

is_canonical(*x*)

Returns `True` if *x* is canonical; otherwise returns `False`.

is_finite(*x*)

Returns `True` if *x* is finite; otherwise returns `False`.

is_infinite(*x*)

Returns `True` if *x* is infinite; otherwise returns `False`.

is_nan(*x*)

Returns `True` if *x* is a qNaN or sNaN; otherwise returns `False`.

is_normal(*x*)

Returns `True` if *x* is a normal number; otherwise returns `False`.

is_qnan(*x*)

Returns `True` if *x* is a quiet NaN; otherwise returns `False`.

is_signed(*x*)

Returns `True` if *x* is negative; otherwise returns `False`.

is_snan(*x*)

Returns `True` if *x* is a signaling NaN; otherwise returns `False`.

is_subnormal(*x*)

Returns `True` if *x* is subnormal; otherwise returns `False`.

is_zero(*x*)

Returns `True` if *x* is a zero; otherwise returns `False`.

ln(*x*)

Returns the natural (base e) logarithm of *x*.

log10(*x*)

Returns the base 10 logarithm of *x*.

logb(*x*)

Returns the exponent of the magnitude of the operand's MSD.

logical_and(*x*, *y*)

Applies the logical operation *and* between each operand's digits.

logical_invert(*x*)

Invert all the digits in *x*.

logical_or(*x*, *y*)

Applies the logical operation *or* between each operand's digits.

logical_xor(*x*, *y*)

Applies the logical operation *xor* between each operand's digits.

max(*x*, *y*)

Compares two values numerically and returns the maximum.

max_mag(*x*, *y*)

Compares the values numerically with their sign ignored.

min(*x*, *y*)

Compares two values numerically and returns the minimum.

min_mag(*x*, *y*)

Compares the values numerically with their sign ignored.

minus(*x*)

Minus corresponds to the unary prefix minus operator in Python.

multiply(*x*, *y*)

Return the product of *x* and *y*.

next_minus(*x*)

Returns the largest representable number smaller than *x*.

next_plus(*x*)

Returns the smallest representable number larger than *x*.

next_toward(*x*, *y*)

Returns the number closest to *x*, in direction towards *y*.

normalize(*x*)

Reduces *x* to its simplest form.

number_class(*x*)

Returns an indication of the class of *x*.

plus(*x*)

Plus corresponds to the unary prefix plus operator in Python. This operation applies the context precision and rounding, so it is *not* an identity operation.

power(*x*, *y*, *modulo=None*)

Return `x` to the power of `y`, reduced modulo `modulo` if given.

With two arguments, compute `x**y`. If `x` is negative then `y` must be integral. The result will be inexact unless `y` is integral and the result is finite and can be expressed exactly in 'precision' digits. The rounding mode of the context is used. Results are always correctly rounded in the Python version.

`Decimal(0) ** Decimal(0)` results in `InvalidOperation`, and if `InvalidOperation` is not trapped, then results in `Decimal('NaN')`.

Changed in version 3.3: The C module computes *power()* in terms of the correctly rounded *exp()* and *ln()* functions. The result is well-defined but only "almost always correctly rounded".

With three arguments, compute `(x**y) % modulo`. For the three argument form, the following restrictions on the arguments hold:

- all three arguments must be integral
- `y` must be nonnegative
- at least one of `x` or `y` must be nonzero

- `modulo` must be nonzero and have at most 'precision' digits

The value resulting from `Context.power(x, y, modulo)` is equal to the value that would be obtained by computing `(x**y) % modulo` with unbounded precision, but is computed more efficiently. The exponent of the result is zero, regardless of the exponents of `x`, `y` and `modulo`. The result is always exact.

quantize(*x*, *y*)

Returns a value equal to *x* (rounded), having the exponent of *y*.

radix()

Just returns 10, as this is Decimal, :)

remainder(*x*, *y*)

Returns the remainder from integer division.

The sign of the result, if non-zero, is the same as that of the original dividend.

remainder_near(*x*, *y*)

Returns `x - y * n`, where *n* is the integer nearest the exact value of `x / y` (if the result is 0 then its sign will be the sign of *x*).

rotate(*x*, *y*)

Returns a rotated copy of *x*, *y* times.

same_quantum(*x*, *y*)

Returns `True` if the two operands have the same exponent.

scaleb(*x*, *y*)

Returns the first operand after adding the second value its exp.

shift(*x*, *y*)

Returns a shifted copy of *x*, *y* times.

sqrt(*x*)

Square root of a non-negative number to context precision.

subtract(*x*, *y*)

Return the difference between *x* and *y*.

to_eng_string(*x*)

Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

to_integral_exact(*x*)

Rounds to an integer.

to_sci_string(*x*)

Converts a number to a string using scientific notation.

9.4.4 Constants

The constants in this section are only relevant for the C module. They are also included in the pure Python version for compatibility.

	32-bit	64-bit
decimal.**MAX_PREC**	425000000	999999999999999999
decimal.**MAX_EMAX**	425000000	999999999999999999
decimal.**MIN_EMIN**	-425000000	-999999999999999999
decimal.**MIN_ETINY**	-849999999	-1999999999999999997

decimal.**HAVE_THREADS**

The value is `True`. Deprecated, because Python now always has threads.

Deprecated since version 3.9.

decimal.**HAVE_CONTEXTVAR**

The default value is `True`. If Python is `configured using the --without-decimal-contextvar option`, the C version uses a thread-local rather than a coroutine-local context and the value is `False`. This is slightly faster in some nested context scenarios.

Added in version 3.8.3.

9.4.5 Rounding modes

decimal.**ROUND_CEILING**

Round towards `Infinity`.

decimal.**ROUND_DOWN**

Round towards zero.

decimal.**ROUND_FLOOR**

Round towards `-Infinity`.

decimal.**ROUND_HALF_DOWN**

Round to nearest with ties going towards zero.

decimal.**ROUND_HALF_EVEN**

Round to nearest with ties going to nearest even integer.

decimal.**ROUND_HALF_UP**

Round to nearest with ties going away from zero.

decimal.**ROUND_UP**

Round away from zero.

decimal.**ROUND_05UP**

Round away from zero if last digit after rounding towards zero would have been 0 or 5; otherwise round towards zero.

9.4.6 Signals

Signals represent conditions that arise during computation. Each corresponds to one context flag and one context trap enabler.

The context flag is set whenever the condition is encountered. After the computation, flags may be checked for informational purposes (for instance, to determine whether a computation was exact). After checking the flags, be sure to clear all flags before starting the next computation.

If the context's trap enabler is set for the signal, then the condition causes a Python exception to be raised. For example, if the *DivisionByZero* trap is set, then a *DivisionByZero* exception is raised upon encountering the condition.

class decimal.**Clamped**

Altered an exponent to fit representation constraints.

Typically, clamping occurs when an exponent falls outside the context's `Emin` and `Emax` limits. If possible, the exponent is reduced to fit by adding zeros to the coefficient.

class decimal.**DecimalException**

Base class for other signals and a subclass of *ArithmeticError*.

class decimal.**DivisionByZero**

Signals the division of a non-infinite number by zero.

Can occur with division, modulo division, or when raising a number to a negative power. If this signal is not trapped, returns `Infinity` or `-Infinity` with the sign determined by the inputs to the calculation.

class decimal.**Inexact**

Indicates that rounding occurred and the result is not exact.

Signals when non-zero digits were discarded during rounding. The rounded result is returned. The signal flag or trap is used to detect when results are inexact.

class decimal.**InvalidOperation**

An invalid operation was performed.

Indicates that an operation was requested that does not make sense. If not trapped, returns `NaN`. Possible causes include:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

class decimal.**Overflow**

Numerical overflow.

Indicates the exponent is larger than `Context.Emax` after rounding has occurred. If not trapped, the result depends on the rounding mode, either pulling inward to the largest representable finite number or rounding outward to `Infinity`. In either case, *Inexact* and *Rounded* are also signaled.

class decimal.**Rounded**

Rounding occurred though possibly no information was lost.

Signaled whenever rounding discards digits; even if those digits are zero (such as rounding `5.00` to `5.0`). If not trapped, returns the result unchanged. This signal is used to detect loss of significant digits.

class decimal.**Subnormal**

Exponent was lower than `Emin` prior to rounding.

Occurs when an operation result is subnormal (the exponent is too small). If not trapped, returns the result unchanged.

class decimal.**Underflow**

Numerical underflow with result rounded to zero.

Occurs when a subnormal result is pushed to zero by rounding. *Inexact* and *Subnormal* are also signaled.

class decimal.**FloatOperation**

Enable stricter semantics for mixing floats and Decimals.

If the signal is not trapped (default), mixing floats and Decimals is permitted in the *Decimal* constructor, *create_decimal()* and all comparison operators. Both conversion and comparisons are exact. Any occurrence of a mixed operation is silently recorded by setting *FloatOperation* in the context flags. Explicit conversions with *from_float()* or *create_decimal_from_float()* do not set the flag.

Otherwise (the signal is trapped), only equality comparisons and explicit conversions are silent. All other mixed operations raise *FloatOperation*.

The following table summarizes the hierarchy of signals:

```
exceptions.ArithmeticError(exceptions.Exception)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
        Inexact
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
        FloatOperation(DecimalException, exceptions.TypeError)
```

9.4.7 Floating-Point Notes

Mitigating round-off error with increased precision

The use of decimal floating point eliminates decimal representation error (making it possible to represent `0.1` exactly); however, some operations can still incur round-off error when non-zero digits exceed the fixed precision.

The effects of round-off error can be amplified by the addition or subtraction of nearly offsetting quantities resulting in loss of significance. Knuth provides two instructive examples where rounded floating-point arithmetic with insufficient precision causes the breakdown of the associative and distributive properties of addition:

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8
```

(continues on next page)

(continued from previous page)

```
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')
```

The *decimal* module makes it possible to restore the identities by expanding the precision sufficiently to avoid loss of significance:

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
>>> u * (v+w)
Decimal('0.0060000')
```

Special values

The number system for the *decimal* module provides special values including `NaN`, `sNaN`, `-Infinity`, `Infinity`, and two zeros, `+0` and `-0`.

Infinities can be constructed directly with: `Decimal('Infinity')`. Also, they can arise from dividing by zero when the *DivisionByZero* signal is not trapped. Likewise, when the *Overflow* signal is not trapped, infinity can result from rounding beyond the limits of the largest representable number.

The infinities are signed (affine) and can be used in arithmetic operations where they get treated as very large, indeterminate numbers. For instance, adding a constant to infinity gives another infinite result.

Some operations are indeterminate and return `NaN`, or if the *InvalidOperation* signal is trapped, raise an exception. For example, `0/0` returns `NaN` which means "not a number". This variety of `NaN` is quiet and, once created, will flow through other computations always resulting in another `NaN`. This behavior can be useful for a series of computations that occasionally have missing inputs — it allows the calculation to proceed while flagging specific results as invalid.

A variant is `sNaN` which signals rather than remaining quiet after every operation. This is a useful return value when an invalid result needs to interrupt a calculation for special handling.

The behavior of Python's comparison operators can be a little surprising where a `NaN` is involved. A test for equality where one of the operands is a quiet or signaling `NaN` always returns *False* (even when doing `Decimal('NaN')==Decimal('NaN')`), while a test for inequality always returns *True*. An attempt to compare two Decimals using any of the <, <=, > or >= operators will raise the *InvalidOperation* signal if either operand is a `NaN`, and return *False* if this signal is not trapped. Note that the General Decimal Arithmetic specification does not specify the behavior of direct comparisons; these rules for comparisons involving a `NaN` were taken from the IEEE 854 standard (see Table 3 in section 5.7). To ensure strict standards-compliance, use the *compare()* and *compare_signal()* methods instead.

The signed zeros can result from calculations that underflow. They keep the sign that would have resulted if the calculation had been carried out to greater precision. Since their magnitude is zero, both positive and negative zeros are treated as equal and their sign is informational.

In addition to the two signed zeros which are distinct yet equal, there are various representations of zero with differing precisions yet equivalent in value. This takes a bit of getting used to. For an eye accustomed to normalized floating-point representations, it is not immediately obvious that the following calculation returns a value equal to zero:

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000026')
```

9.4.8 Working with threads

The *getcontext()* function accesses a different *Context* object for each thread. Having separate thread contexts means that threads may make changes (such as `getcontext().prec=10`) without interfering with other threads.

Likewise, the *setcontext()* function automatically assigns its target to the current thread.

If *setcontext()* has not been called before *getcontext()*, then *getcontext()* will automatically create a new context for use in the current thread.

The new context is copied from a prototype context called *DefaultContext*. To control the defaults so that each thread will use the same values throughout the application, directly modify the *DefaultContext* object. This should be done *before* any threads are started so that there won't be a race condition between threads calling *getcontext()*. For example:

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
 . . .
```

9.4.9 Recipes

Here are a few recipes that serve as utility functions and that demonstrate ways to work with the *Decimal* class:

```
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
    """Convert Decimal to a money formatted string.

    places:  required number of places after the decimal point
    curr:    optional currency symbol before the sign (may be blank)
    sep:     optional grouping separator (comma, period, space, or blank)
    dp:      decimal point indicator (comma or period)
             only specify as blank when places is zero
    pos:     optional sign for positive numbers: '+', space or blank
    neg:     optional sign for negative numbers: '-', '(', space or blank
    trailneg:optional trailing minus indicator:  '-', ')', space or blank
```

(continues on next page)

(continued from previous page)

```
    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '-$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
    '1.234.568-'
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
    '<0.02>'

    """
    q = Decimal(10) ** -places      # 2 places --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = list(map(str, digits))
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
    for i in range(places):
        build(next() if digits else '0')
    if places:
        build(dp)
    if not digits:
        build('0')
    i = 0
    while digits:
        build(next())
        i += 1
        if i == 3 and digits:
            i = 0
            build(sep)
    build(curr)
    build(neg if sign else pos)
    return ''.join(reversed(result))

def pi():
    """Compute Pi to the current precision.

    >>> print(pi())
    3.141592653589793238462643383

    """
    getcontext().prec += 2  # extra digits for intermediate steps
    three = Decimal(3)      # substitute "three=3.0" for regular floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
    return +s               # unary plus applies the new precision

def exp(x):
```

(continues on next page)

(continued from previous page)

```
    """Return e raised to the power of x.  Result type matches input type.

    >>> print(exp(Decimal(1)))
    2.718281828459045235360287471
    >>> print(exp(Decimal(2)))
    7.389056098930650227230427461
    >>> print(exp(2.0))
    7.38905609893
    >>> print(exp(2+0j))
    (7.38905609893+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1
        fact *= i
        num *= x
        s += num / fact
    getcontext().prec -= 2
    return +s

def cos(x):
    """Return the cosine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(cos(Decimal('0.5')))
    0.8775825618903727161162815826
    >>> print(cos(0.5))
    0.87758256189
    >>> print(cos(0.5+0j))
    (0.87758256189+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
    """Return the sine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(sin(Decimal('0.5')))
    0.4794255386042030002732879352
```

(continues on next page)

(continued from previous page)

```
    >>> print(sin(0.5))
    0.479425538604
    >>> print(sin(0.5+0j))
    (0.479425538604+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s
```

9.4.10 Decimal FAQ

Q. It is cumbersome to type `decimal.Decimal('1234.5')`. Is there a way to minimize typing when using the interactive interpreter?

A. Some users abbreviate the constructor to just a single letter:

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q. In a fixed-point application with two decimal places, some inputs have many places and need to be rounded. Others are not supposed to have excess digits and need to be validated. What methods should be used?

A. The *quantize()* method rounds to a fixed number of decimal places. If the *Inexact* trap is set, it is also useful for validation:

```
>>> TWOPLACES = Decimal(10) ** -2       # same as Decimal('0.01')
```

```
>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')
```

```
>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')
```

```
>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
   ...
Inexact: None
```

Q. Once I have valid two place inputs, how do I maintain that invariant throughout an application?

A. Some operations like addition, subtraction, and multiplication by an integer will automatically preserve fixed point. Others operations, like division and non-integer multiplication, will change the number of decimal places and need to be followed-up with a *quantize()* step:

```
>>> a = Decimal('102.72')           # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                           # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                          # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES)     # Must quantize non-integer multiplication
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES)     # And quantize division
Decimal('0.03')
```

In developing fixed-point applications, it is convenient to define functions to handle the *quantize()* step:

```
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
...
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)
```

```
>>> mul(a, b)                       # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

Q. There are many ways to express the same value. The numbers `200`, `200.000`, `2E2`, and `.02E+4` all have the same value at various precisions. Is there a way to transform them to a single recognizable canonical value?

A. The *normalize()* method maps all equivalent values to a single representative:

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q. When does rounding occur in a computation?

A. It occurs *after* the computation. The philosophy of the decimal specification is that numbers are considered exact and are created independent of the current context. They can even have greater precision than current context. Computations process with those exact inputs and then rounding (or other context operations) is applied to the *result* of the computation:

```
>>> getcontext().prec = 5
>>> pi = Decimal('3.1415926535')    # More than 5 digits
>>> pi                              # All digits are retained
Decimal('3.1415926535')
>>> pi + 0                          # Rounded after an addition
Decimal('3.1416')
>>> pi - Decimal('0.00005')         # Subtract unrounded numbers, then round
Decimal('3.1415')
>>> pi + 0 - Decimal('0.00005').    # Intermediate values are rounded
Decimal('3.1416')
```

Q. Some decimal values always print with exponential notation. Is there a way to get a non-exponential representation?

A. For some values, exponential notation is the only way to express the number of significant places in the coefficient. For example, expressing `5.0E+3` as `5000` keeps the value constant but cannot show the original's two-place significance.

If an application does not care about tracking significance, it is easy to remove the exponent and trailing zeroes, losing significance, but keeping the value unchanged:

```
>>> def remove_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```

```
>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')
```

Q. Is there a way to convert a regular float to a *Decimal*?

A. Yes, any binary floating-point number can be exactly expressed as a Decimal though an exact conversion may take more precision than intuition would suggest:

```
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

Q. Within a complex calculation, how can I make sure that I haven't gotten a spurious result because of insufficient precision or rounding anomalies.

A. The decimal module makes it easy to test results. A best practice is to re-run calculations using greater precision and with various rounding modes. Widely differing results indicate insufficient precision, rounding mode issues, ill-conditioned inputs, or a numerically unstable algorithm.

Q. I noticed that context precision is applied to the results of operations but not to the inputs. Is there anything to watch out for when mixing values of different precisions?

A. Yes. The principle is that all values are considered to be exact and so is the arithmetic on those values. Only the results are rounded. The advantage for inputs is that "what you type is what you get". A disadvantage is that the results can look odd if you forget that the inputs haven't been rounded:

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

The solution is either to increase precision or to force rounding of inputs using the unary plus operation:

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')       # unary plus triggers rounding
Decimal('1.23')
```

Alternatively, inputs can be rounded upon creation using the *Context.create_decimal()* method:

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

Q. Is the CPython implementation fast for large numbers?

A. Yes. In the CPython and PyPy3 implementations, the C/CFFI versions of the decimal module integrate the high speed libmpdec library for arbitrary precision correctly rounded decimal floating-point arithmetic[1]. `libmpdec` uses Karatsuba multiplication for medium-sized numbers and the Number Theoretic Transform for very large numbers.

The context must be adapted for exact arbitrary precision arithmetic. `Emin` and `Emax` should always be set to the maximum values, `clamp` should always be 0 (the default). Setting `prec` requires some care.

The easiest approach for trying out bignum arithmetic is to use the maximum value for `prec` as well[2]:

[1] Added in version 3.3.

[2] Changed in version 3.9: This approach now works for all exact results except for non-integer powers.

```
>>> setcontext(Context(prec=MAX_PREC, Emax=MAX_EMAX, Emin=MIN_EMIN))
>>> x = Decimal(2) ** 256
>>> x / 128
Decimal('904625697166532776746648320380374280103671755200316906558262375061821325312')
```

For inexact results, *MAX_PREC* is far too large on 64-bit platforms and the available memory will be insufficient:

```
>>> Decimal(1) / 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError
```

On systems with overallocation (e.g. Linux), a more sophisticated approach is to adjust `prec` to the amount of available RAM. Suppose that you have 8GB of RAM and expect 10 simultaneous operands using a maximum of 500MB each:

```
>>> import sys
>>>
>>> # Maximum number of digits for a single operand using 500MB in 8-byte words
>>> # with 19 digits per word (4-byte and 9 digits for the 32-bit build):
>>> maxdigits = 19 * ((500 * 1024**2) // 8)
>>>
>>> # Check that this works:
>>> c = Context(prec=maxdigits, Emax=MAX_EMAX, Emin=MIN_EMIN)
>>> c.traps[Inexact] = True
>>> setcontext(c)
>>>
>>> # Fill the available precision with nines:
>>> x = Decimal(0).logical_invert() * 9
>>> sys.getsizeof(x)
524288112
>>> x + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  decimal.Inexact: [<class 'decimal.Inexact'>]
```

In general (and especially on systems without overallocation), it is recommended to estimate even tighter bounds and set the *Inexact* trap if all calculations are expected to be exact.

9.5 `fractions` — Rational numbers

Source code: Lib/fractions.py

The *fractions* module provides support for rational number arithmetic.

A Fraction instance can be constructed from a pair of integers, from another rational number, or from a string.

class fractions.**Fraction**(*numerator=0*, *denominator=1*)

class fractions.**Fraction**(*other_fraction*)

class fractions.**Fraction**(*float*)

class fractions.**Fraction**(*decimal*)

class fractions.**Fraction**(*string*)

The first version requires that *numerator* and *denominator* are instances of *numbers.Rational* and returns a

new *Fraction* instance with value `numerator/denominator`. If *denominator* is `0`, it raises a *ZeroDivisionError*. The second version requires that *other_fraction* is an instance of *numbers.Rational* and returns a *Fraction* instance with the same value. The next two versions accept either a *float* or a *decimal.Decimal* instance, and return a *Fraction* instance with exactly the same value. Note that due to the usual issues with binary floating point (see tut-fp-issues), the argument to `Fraction(1.1)` is not exactly equal to 11/10, and so `Fraction(1.1)` does *not* return `Fraction(11, 10)` as one might expect. (But see the documentation for the *limit_denominator()* method below.) The last version of the constructor expects a string or unicode instance. The usual form for this instance is:

```
[sign] numerator ['/' denominator]
```

where the optional `sign` may be either '+' or '-' and `numerator` and `denominator` (if present) are strings of decimal digits (underscores may be used to delimit digits as with integral literals in code). In addition, any string that represents a finite value and is accepted by the *float* constructor is also accepted by the *Fraction* constructor. In either form the input string may also have leading and/or trailing whitespace. Here are some examples:

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)
```

The *Fraction* class inherits from the abstract base class *numbers.Rational*, and implements all of the methods and operations from that class. *Fraction* instances are *hashable*, and should be treated as immutable. In addition, *Fraction* has the following properties and methods:

Changed in version 3.2: The *Fraction* constructor now accepts *float* and *decimal.Decimal* instances.

Changed in version 3.9: The *math.gcd()* function is now used to normalize the *numerator* and *denominator*. *math.gcd()* always returns an *int* type. Previously, the GCD type depended on *numerator* and *denominator*.

Changed in version 3.11: Underscores are now permitted when creating a *Fraction* instance from a string, following **PEP 515** rules.

Changed in version 3.11: *Fraction* implements `__int__` now to satisfy `typing.SupportsInt` instance checks.

Changed in version 3.12: Space is allowed around the slash for string inputs: `Fraction('2 / 3')`.

Changed in version 3.12: *Fraction* instances now support float-style formatting, with presentation types `"e"`,

"E", "f", "F", "g", "G" and "%"".

Changed in version 3.13: Formatting of *Fraction* instances without a presentation type now supports fill, alignment, sign handling, minimum width and grouping.

numerator

Numerator of the Fraction in lowest term.

denominator

Denominator of the Fraction in lowest term.

as_integer_ratio()

Return a tuple of two integers, whose ratio is equal to the original Fraction. The ratio is in lowest terms and has a positive denominator.

Added in version 3.8.

is_integer()

Return True if the Fraction is an integer.

Added in version 3.12.

classmethod from_float(*flt*)

Alternative constructor which only accepts instances of *float* or *numbers.Integral*. Beware that Fraction.from_float(0.3) is not the same value as Fraction(3, 10).

> **Note**
>
> From Python 3.2 onwards, you can also construct a *Fraction* instance directly from a *float*.

classmethod from_decimal(*dec*)

Alternative constructor which only accepts instances of *decimal.Decimal* or *numbers.Integral*.

> **Note**
>
> From Python 3.2 onwards, you can also construct a *Fraction* instance directly from a *decimal.Decimal* instance.

limit_denominator(*max_denominator=1000000*)

Finds and returns the closest *Fraction* to self that has denominator at most max_denominator. This method is useful for finding rational approximations to a given floating-point number:

```
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

or for recovering a rational number that's represented as a float:

```
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)
```

__floor__()

Returns the greatest *int* <= `self`. This method can also be accessed through the *math.floor()* function:

```
>>> from math import floor
>>> floor(Fraction(355, 113))
3
```

__ceil__()

Returns the least *int* >= `self`. This method can also be accessed through the *math.ceil()* function.

__round__()

__round__(*ndigits*)

The first version returns the nearest *int* to `self`, rounding half to even. The second version rounds `self` to the nearest multiple of `Fraction(1, 10**ndigits)` (logically, if `ndigits` is negative), again rounding half toward even. This method can also be accessed through the *round()* function.

__format__(*format_spec*, */*)

Provides support for formatting of *Fraction* instances via the *str.format()* method, the *format()* built-in function, or Formatted string literals.

If the `format_spec` format specification string does not end with one of the presentation types `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` or `'%'` then formatting follows the general rules for fill, alignment, sign handling, minimum width, and grouping as described in the *format specification mini-language*. The "alternate form" flag `'#'` is supported: if present, it forces the output string to always include an explicit denominator, even when the value being formatted is an exact integer. The zero-fill flag `'0'` is not supported.

If the `format_spec` format specification string ends with one of the presentation types `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` or `'%'` then formatting follows the rules outlined for the *float* type in the *Format Specification Mini-Language* section.

Here are some examples:

```
>>> from fractions import Fraction
>>> format(Fraction(103993, 33102), '_')
'103_993/33_102'
>>> format(Fraction(1, 7), '.^+10')
'...+1/7...'
>>> format(Fraction(3, 1), '')
'3'
>>> format(Fraction(3, 1), '#')
'3/1'
>>> format(Fraction(1, 7), '.40g')
'0.1428571428571428571428571428571428571429'
>>> format(Fraction('1234567.855'), '_.2f')
'1_234_567.86'
>>> f"{Fraction(355, 113):*>20.6e}"
'********3.141593e+00'
>>> old_price, new_price = 499, 672
>>> "{:.2%} price increase".format(Fraction(new_price, old_price) - 1)
'34.67% price increase'
```

See also

Module *numbers*
The abstract base classes making up the numeric tower.

9.6 random — Generate pseudo-random numbers

Source code: Lib/random.py

This module implements pseudo-random number generators for various distributions.

For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

On the real line, there are functions to compute uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. For generating distributions of angles, the von Mises distribution is available.

Almost all module functions depend on the basic function *random()*, which generates a random float uniformly in the half-open range `0.0 <= X < 1.0`. Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of 2**19937-1. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

The functions supplied by this module are actually bound methods of a hidden instance of the *random.Random* class. You can instantiate your own instances of *Random* to get generators that don't share state.

Class *Random* can also be subclassed if you want to use a different basic generator of your own devising: see the documentation on that class for more details.

The *random* module also provides the *SystemRandom* class which uses the system function *os.urandom()* to generate random numbers from sources provided by the operating system.

Warning

The pseudo-random generators of this module should not be used for security purposes. For security or cryptographic uses, see the *secrets* module.

See also

M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3–30 1998.

Complementary-Multiply-with-Carry recipe for a compatible alternative random number generator with a long period and comparatively simple update operations.

Note

The global random number generator and instances of *Random* are thread-safe. However, in the free-threaded build, concurrent calls to the global generator or to the same instance of *Random* may encounter contention and poor performance. Consider using separate instances of *Random* per thread instead.

9.6.1 Bookkeeping functions

random.**seed**(*a=None*, *version=2*)

Initialize the random number generator.

If *a* is omitted or `None`, the current system time is used. If randomness sources are provided by the operating system, they are used instead of the system time (see the *os.urandom()* function for details on availability).

If *a* is an int, it is used directly.

With version 2 (the default), a *str*, *bytes*, or *bytearray* object gets converted to an *int* and all of its bits are used.

With version 1 (provided for reproducing random sequences from older versions of Python), the algorithm for *str* and *bytes* generates a narrower range of seeds.

Changed in version 3.2: Moved to the version 2 scheme which uses all of the bits in a string seed.

Changed in version 3.11: The *seed* must be one of the following types: `None`, *int*, *float*, *str*, *bytes*, or *bytearray*.

random.**getstate**()

Return an object capturing the current internal state of the generator. This object can be passed to *setstate()* to restore the state.

random.**setstate**(*state*)

state should have been obtained from a previous call to *getstate()*, and *setstate()* restores the internal state of the generator to what it was at the time *getstate()* was called.

9.6.2 Functions for bytes

random.**randbytes**(*n*)

Generate *n* random bytes.

This method should not be used for generating security tokens. Use *secrets.token_bytes()* instead.

Added in version 3.9.

9.6.3 Functions for integers

random.**randrange**(*stop*)

random.**randrange**(*start*, *stop*[, *step*])

Return a randomly selected element from `range(start, stop, step)`.

This is roughly equivalent to `choice(range(start, stop, step))` but supports arbitrarily large ranges and is optimized for common cases.

The positional argument pattern matches the *range()* function.

Keyword arguments should not be used because they can be interpreted in unexpected ways. For example `randrange(start=100)` is interpreted as `randrange(0, 100, 1)`.

Changed in version 3.2: *randrange()* is more sophisticated about producing equally distributed values. Formerly it used a style like `int(random()*n)` which could produce slightly uneven distributions.

Changed in version 3.12: Automatic conversion of non-integer types is no longer supported. Calls such as `randrange(10.0)` and `randrange(Fraction(10, 1))` now raise a *TypeError*.

random.**randint**(*a*, *b*)

Return a random integer *N* such that `a <= N <= b`. Alias for `randrange(a, b+1)`.

random.**getrandbits**(*k*)

Returns a non-negative Python integer with *k* random bits. This method is supplied with the Mersenne Twister generator and some other generators may also provide it as an optional part of the API. When available, *getrandbits()* enables *randrange()* to handle arbitrarily large ranges.

Changed in version 3.9: This method now accepts zero for *k*.

9.6.4 Functions for sequences

random.**choice**(*seq*)

Return a random element from the non-empty sequence *seq*. If *seq* is empty, raises *IndexError*.

random.**choices**(*population*, *weights=None*, ***, *cum_weights=None*, *k=1*)

Return a *k* sized list of elements chosen from the *population* with replacement. If the *population* is empty, raises *IndexError*.

If a *weights* sequence is specified, selections are made according to the relative weights. Alternatively, if a *cum_weights* sequence is given, the selections are made according to the cumulative weights (perhaps computed using *itertools.accumulate()*). For example, the relative weights `[10, 5, 30, 5]` are equivalent to the cumulative weights `[10, 15, 45, 50]`. Internally, the relative weights are converted to cumulative weights before making selections, so supplying the cumulative weights saves work.

If neither *weights* nor *cum_weights* are specified, selections are made with equal probability. If a weights sequence is supplied, it must be the same length as the *population* sequence. It is a *TypeError* to specify both *weights* and *cum_weights*.

The *weights* or *cum_weights* can use any numeric type that interoperates with the *float* values returned by *random()* (that includes integers, floats, and fractions but excludes decimals). Weights are assumed to be non-negative and finite. A *ValueError* is raised if all weights are zero.

For a given seed, the *choices()* function with equal weighting typically produces a different sequence than repeated calls to *choice()*. The algorithm used by *choices()* uses floating-point arithmetic for internal consistency and speed. The algorithm used by *choice()* defaults to integer arithmetic with repeated selections to avoid small biases from round-off error.

Added in version 3.6.

Changed in version 3.9: Raises a *ValueError* if all weights are zero.

random.**shuffle**(*x*)

Shuffle the sequence *x* in place.

To shuffle an immutable sequence and return a new shuffled list, use `sample(x, k=len(x))` instead.

Note that even for small `len(x)`, the total number of permutations of *x* can quickly grow larger than the period of most random number generators. This implies that most permutations of a long sequence can never be generated. For example, a sequence of length 2080 is the largest that can fit within the period of the Mersenne Twister random number generator.

Changed in version 3.11: Removed the optional parameter *random*.

random.**sample**(*population*, *k*, ***, *counts=None*)

Return a *k* length list of unique elements chosen from the population sequence. Used for random sampling without replacement.

Returns a new list containing elements from the population while leaving the original population unchanged. The resulting list is in selection order so that all sub-slices will also be valid random samples. This allows raffle winners (the sample) to be partitioned into grand prize and second place winners (the subslices).

Members of the population need not be *hashable* or unique. If the population contains repeats, then each occurrence is a possible selection in the sample.

Repeated elements can be specified one at a time or with the optional keyword-only *counts* parameter. For example, `sample(['red', 'blue'], counts=[4, 2], k=5)` is equivalent to `sample(['red', 'red', 'red', 'red', 'blue', 'blue'], k=5)`.

To choose a sample from a range of integers, use a *range()* object as an argument. This is especially fast and space efficient for sampling from a large population: `sample(range(10000000), k=60)`.

If the sample size is larger than the population size, a *ValueError* is raised.

Changed in version 3.9: Added the *counts* parameter.

Changed in version 3.11: The *population* must be a sequence. Automatic conversion of sets to lists is no longer supported.

9.6.5 Discrete distributions

The following function generates a discrete distribution.

`random.`**`binomialvariate`**(*n=1*, *p=0.5*)

Binomial distribution. Return the number of successes for *n* independent trials with the probability of success in each trial being *p*:

Mathematically equivalent to:

```
sum(random() < p for i in range(n))
```

The number of trials *n* should be a non-negative integer. The probability of success *p* should be between `0.0 <= p <= 1.0`. The result is an integer in the range `0 <= X <= n`.

Added in version 3.12.

9.6.6 Real-valued distributions

The following functions generate specific real-valued distributions. Function parameters are named after the corresponding variables in the distribution's equation, as used in common mathematical practice; most of these equations can be found in any statistics text.

`random.`**`random`**()

Return the next random floating-point number in the range `0.0 <= X < 1.0`

`random.`**`uniform`**(*a*, *b*)

Return a random floating-point number *N* such that `a <= N <= b` for `a <= b` and `b <= N <= a` for `b < a`.

The end-point value `b` may or may not be included in the range depending on floating-point rounding in the expression `a + (b-a) * random()`.

`random.`**`triangular`**(*low*, *high*, *mode*)

Return a random floating-point number *N* such that `low <= N <= high` and with the specified *mode* between those bounds. The *low* and *high* bounds default to zero and one. The *mode* argument defaults to the midpoint between the bounds, giving a symmetric distribution.

random.**betavariate**(*alpha*, *beta*)

Beta distribution. Conditions on the parameters are `alpha > 0` and `beta > 0`. Returned values range between 0 and 1.

random.**expovariate**(*lambd=1.0*)

Exponential distribution. *lambd* is 1.0 divided by the desired mean. It should be nonzero. (The parameter would be called "lambda", but that is a reserved word in Python.) Returned values range from 0 to positive infinity if *lambd* is positive, and from negative infinity to 0 if *lambd* is negative.

Changed in version 3.12: Added the default value for `lambd`.

random.**gammavariate**(*alpha*, *beta*)

Gamma distribution. (*Not* the gamma function!) The shape and scale parameters, *alpha* and *beta*, must have positive values. (Calling conventions vary and some sources define 'beta' as the inverse of the scale).

The probability distribution function is:

```
            x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

random.**gauss**(*mu=0.0*, *sigma=1.0*)

Normal distribution, also called the Gaussian distribution. *mu* is the mean, and *sigma* is the standard deviation. This is slightly faster than the *normalvariate()* function defined below.

Multithreading note: When two threads call this function simultaneously, it is possible that they will receive the same return value. This can be avoided in three ways. 1) Have each thread use a different instance of the random number generator. 2) Put locks around all calls. 3) Use the slower, but thread-safe *normalvariate()* function instead.

Changed in version 3.11: *mu* and *sigma* now have default arguments.

random.**lognormvariate**(*mu*, *sigma*)

Log normal distribution. If you take the natural logarithm of this distribution, you'll get a normal distribution with mean *mu* and standard deviation *sigma*. *mu* can have any value, and *sigma* must be greater than zero.

random.**normalvariate**(*mu=0.0*, *sigma=1.0*)

Normal distribution. *mu* is the mean, and *sigma* is the standard deviation.

Changed in version 3.11: *mu* and *sigma* now have default arguments.

random.**vonmisesvariate**(*mu*, *kappa*)

mu is the mean angle, expressed in radians between 0 and 2**pi*, and *kappa* is the concentration parameter, which must be greater than or equal to zero. If *kappa* is equal to zero, this distribution reduces to a uniform random angle over the range 0 to 2**pi*.

random.**paretovariate**(*alpha*)

Pareto distribution. *alpha* is the shape parameter.

random.**weibullvariate**(*alpha*, *beta*)

Weibull distribution. *alpha* is the scale parameter and *beta* is the shape parameter.

9.6.7 Alternative Generator

class random.**Random**([*seed*])

Class that implements the default pseudo-random number generator used by the *random* module.

Changed in version 3.11: Formerly the *seed* could be any hashable object. Now it is limited to: None, *int*, *float*, *str*, *bytes*, or *bytearray*.

Subclasses of Random should override the following methods if they wish to make use of a different basic generator:

seed(*a=None*, *version=2*)

Override this method in subclasses to customise the *seed()* behaviour of Random instances.

getstate()

Override this method in subclasses to customise the *getstate()* behaviour of Random instances.

setstate(*state*)

Override this method in subclasses to customise the *setstate()* behaviour of Random instances.

random()

Override this method in subclasses to customise the *random()* behaviour of Random instances.

Optionally, a custom generator subclass can also supply the following method:

getrandbits(*k*)

Override this method in subclasses to customise the *getrandbits()* behaviour of Random instances.

class random.**SystemRandom**([*seed*])

Class that uses the *os.urandom()* function for generating random numbers from sources provided by the operating system. Not available on all systems. Does not rely on software state, and sequences are not reproducible. Accordingly, the *seed()* method has no effect and is ignored. The *getstate()* and *setstate()* methods raise *NotImplementedError* if called.

9.6.8 Notes on Reproducibility

Sometimes it is useful to be able to reproduce the sequences given by a pseudo-random number generator. By reusing a seed value, the same sequence should be reproducible from run to run as long as multiple threads are not running.

Most of the random module's algorithms and seeding functions are subject to change across Python versions, but two aspects are guaranteed not to change:

- If a new seeding method is added, then a backward compatible seeder will be offered.
- The generator's *random()* method will continue to produce the same sequence when the compatible seeder is given the same seed.

9.6.9 Examples

Basic examples:

```
>>> random()                            # Random float:  0.0 <= x < 1.0
0.37444887175646646

>>> uniform(2.5, 10.0)                  # Random float:  2.5 <= x <= 10.0
3.1800146073117523

>>> expovariate(1 / 5)                  # Interval between arrivals averaging 5 seconds
5.148957571865031

>>> randrange(10)                       # Integer from 0 to 9 inclusive
7

>>> randrange(0, 101, 2)                # Even integer from 0 to 100 inclusive
26

>>> choice(['win', 'lose', 'draw'])     # Single random element from a sequence
'draw'

>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                       # Shuffle a list
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4) # Four samples without replacement
[40, 10, 50, 30]
```

Simulations:

```
>>> # Six roulette wheel spins (weighted sampling with replacement)
>>> choices(['red', 'black', 'green'], [18, 18, 2], k=6)
['red', 'green', 'black', 'black', 'red', 'black']

>>> # Deal 20 cards without replacement from a deck
>>> # of 52 playing cards, and determine the proportion of cards
>>> # with a ten-value:  ten, jack, queen, or king.
>>> deal = sample(['tens', 'low cards'], counts=[16, 36], k=20)
>>> deal.count('tens') / 20
0.15

>>> # Estimate the probability of getting 5 or more heads from 7 spins
>>> # of a biased coin that settles on heads 60% of the time.
>>> sum(binomialvariate(n=7, p=0.6) >= 5 for i in range(10_000)) / 10_000
0.4169

>>> # Probability of the median of 5 samples being in middle two quartiles
>>> def trial():
...     return 2_500 <= sorted(choices(range(10_000), k=5))[2] < 7_500
...
>>> sum(trial() for i in range(10_000)) / 10_000
0.7958
```

Example of statistical bootstrapping using resampling with replacement to estimate a confidence interval for the mean of a sample:

```
# https://www.thoughtco.com/example-of-bootstrapping-3126155
from statistics import fmean as mean
from random import choices

data = [41, 50, 29, 37, 81, 30, 73, 63, 20, 35, 68, 22, 60, 31, 95]
means = sorted(mean(choices(data, k=len(data))) for i in range(100))
print(f'The sample mean of {mean(data):.1f} has a 90% confidence '
      f'interval from {means[5]:.1f} to {means[94]:.1f}')
```

Example of a resampling permutation test to determine the statistical significance or p-value of an observed difference between the effects of a drug versus a placebo:

```
# Example from "Statistics is Easy" by Dennis Shasha and Manda Wilson
from statistics import fmean as mean
from random import shuffle

drug = [54, 73, 53, 70, 73, 68, 52, 65, 65]
placebo = [54, 51, 58, 44, 55, 52, 42, 47, 58, 46]
observed_diff = mean(drug) - mean(placebo)

n = 10_000
count = 0
combined = drug + placebo
for i in range(n):
    shuffle(combined)
    new_diff = mean(combined[:len(drug)]) - mean(combined[len(drug):])
    count += (new_diff >= observed_diff)

print(f'{n} label reshufflings produced only {count} instances with a difference')
print(f'at least as extreme as the observed difference of {observed_diff:.1f}.')
print(f'The one-sided p-value of {count / n:.4f} leads us to reject the null')
print(f'hypothesis that there is no difference between the drug and the placebo.')
```

Simulation of arrival times and service deliveries for a multiserver queue:

```
from heapq import heapify, heapreplace
from random import expovariate, gauss
from statistics import mean, quantiles

average_arrival_interval = 5.6
average_service_time = 15.0
stdev_service_time = 3.5
num_servers = 3

waits = []
arrival_time = 0.0
servers = [0.0] * num_servers  # time when each server becomes available
heapify(servers)
for i in range(1_000_000):
    arrival_time += expovariate(1.0 / average_arrival_interval)
    next_server_available = servers[0]
    wait = max(0.0, next_server_available - arrival_time)
    waits.append(wait)
    service_duration = max(0.0, gauss(average_service_time, stdev_service_time))
    service_completed = arrival_time + wait + service_duration
    heapreplace(servers, service_completed)
```

(continues on next page)

(continued from previous page)

```
print(f'Mean wait: {mean(waits):.1f}   Max wait: {max(waits):.1f}')
print('Quartiles:', [round(q, 1) for q in quantiles(waits)])
```

See also

Statistics for Hackers a video tutorial by Jake Vanderplas on statistical analysis using just a few fundamental concepts including simulation, sampling, shuffling, and cross-validation.

Economics Simulation a simulation of a marketplace by Peter Norvig that shows effective use of many of the tools and distributions provided by this module (gauss, uniform, sample, betavariate, choice, triangular, and randrange).

A Concrete Introduction to Probability (using Python) a tutorial by Peter Norvig covering the basics of probability theory, how to write simulations, and how to perform data analysis using Python.

9.6.10 Recipes

These recipes show how to efficiently make random selections from the combinatoric iterators in the *itertools* module:

```
def random_product(*args, repeat=1):
    "Random selection from itertools.product(*args, **kwds)"
    pools = [tuple(pool) for pool in args] * repeat
    return tuple(map(random.choice, pools))

def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable, r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
    "Choose r elements with replacement.  Order the result to match the iterable."
    # Result will be in set(itertools.combinations_with_replacement(iterable, r)).
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.choices(range(n), k=r))
    return tuple(pool[i] for i in indices)
```

The default *random()* returns multiples of 2^{-53} in the range $0.0 \le x < 1.0$. All such numbers are evenly spaced and are exactly representable as Python floats. However, many other representable floats in that interval are not possible selections. For example, `0.0595486140802560 9` isn't an integer multiple of 2^{-53}.

The following recipe takes a different approach. All floats in the interval are possible selections. The mantissa comes from a uniform distribution of integers in the range $2^{52} \le mantissa < 2^{53}$. The exponent comes from a geometric distribution where exponents smaller than *-53* occur half as often as the next larger exponent.

```
from random import Random
from math import ldexp
```

(continues on next page)

(continued from previous page)

```
class FullRandom(Random):

    def random(self):
        mantissa = 0x10_0000_0000_0000 | self.getrandbits(52)
        exponent = -53
        x = 0
        while not x:
            x = self.getrandbits(32)
            exponent += x.bit_length() - 32
        return ldexp(mantissa, exponent)
```

All *real valued distributions* in the class will use the new method:

```
>>> fr = FullRandom()
>>> fr.random()
0.05954861408025609
>>> fr.expovariate(0.25)
8.87925541791544
```

The recipe is conceptually equivalent to an algorithm that chooses from all the multiples of 2^{-1074} in the range $0.0 \le x < 1.0$. All such numbers are evenly spaced, but most have to be rounded down to the nearest representable Python float. (The value 2^{-1074} is the smallest positive unnormalized float and is equal to `math.ulp(0.0)`.)

See also

Generating Pseudo-random Floating-Point Values a paper by Allen B. Downey describing ways to generate more fine-grained floats than normally generated by *random()*.

9.6.11 Command-line usage

Added in version 3.13.

The `random` module can be executed from the command line.

```
python -m random [-h] [-c CHOICE [CHOICE ...] | -i N | -f N] [input ...]
```

The following options are accepted:

-h, --help

Show the help message and exit.

-c CHOICE [CHOICE ...]

--choice CHOICE [CHOICE ...]

Print a random choice, using *choice()*.

-i <N>

--integer <N>

Print a random integer between 1 and N inclusive, using *randint()*.

-f <N>

--float <N>

Print a random floating-point number between 1 and N inclusive, using *uniform()*.

If no options are given, the output depends on the input:

- String or multiple: same as *--choice*.
- Integer: same as *--integer*.
- Float: same as *--float*.

9.6.12 Command-line example

Here are some examples of the `random` command-line interface:

```
$ # Choose one at random
$ python -m random egg bacon sausage spam "Lobster Thermidor aux crevettes with a␣
↪Mornay sauce"
Lobster Thermidor aux crevettes with a Mornay sauce

$ # Random integer
$ python -m random 6
6

$ # Random floating-point number
$ python -m random 1.8
1.7080016272295635

$ # With explicit arguments
$ python  -m random --choice egg bacon sausage spam "Lobster Thermidor aux crevettes␣
↪with a Mornay sauce"
egg

$ python -m random --integer 6
3

$ python -m random --float 1.8
1.5666339105010318

$ python -m random --integer 6
5

$ python -m random --float 6
3.1942323316565915
```

9.7 `statistics` — Mathematical statistics functions

Added in version 3.4.

Source code: Lib/statistics.py

This module provides functions for calculating mathematical statistics of numeric (*Real*-valued) data.

The module is not intended to be a competitor to third-party libraries such as NumPy, SciPy, or proprietary full-featured statistics packages aimed at professional statisticians such as Minitab, SAS and Matlab. It is aimed at the level of graphing and scientific calculators.

Unless explicitly noted, these functions support *int*, *float*, *Decimal* and *Fraction*. Behaviour with other types (whether in the numeric tower or not) is currently unsupported. Collections with a mix of types are also undefined and implementation-dependent. If your input data consists of mixed types, you may be able to use *map()* to ensure a consistent result, for example: `map(float, input_data)`.

Some datasets use `NaN` (not a number) values to represent missing data. Since NaNs have unusual comparison semantics, they cause surprising or undefined behaviors in the statistics functions that sort data or that count occurrences. The functions affected are `median()`, `median_low()`, `median_high()`, `median_grouped()`, `mode()`, `multimode()`, and `quantiles()`. The `NaN` values should be stripped before calling these functions:

```
>>> from statistics import median
>>> from math import isnan
>>> from itertools import filterfalse

>>> data = [20.7, float('NaN'),19.2, 18.3, float('NaN'), 14.4]
>>> sorted(data)  # This has surprising behavior
[20.7, nan, 14.4, 18.3, 19.2, nan]
>>> median(data)  # This result is unexpected
16.35

>>> sum(map(isnan, data))     # Number of missing values
2
>>> clean = list(filterfalse(isnan, data))  # Strip NaN values
>>> clean
[20.7, 19.2, 18.3, 14.4]
>>> sorted(clean)  # Sorting now works as expected
[14.4, 18.3, 19.2, 20.7]
>>> median(clean)       # This result is now well defined
18.75
```

9.7.1 Averages and measures of central location

These functions calculate an average or typical value from a population or sample.

mean()	Arithmetic mean (“average”) of data.
fmean()	Fast, floating-point arithmetic mean, with optional weighting.
geometric_mean()	Geometric mean of data.
harmonic_mean()	Harmonic mean of data.
kde()	Estimate the probability density distribution of the data.
kde_random()	Random sampling from the PDF generated by kde().
median()	Median (middle value) of data.
median_low()	Low median of data.
median_high()	High median of data.
median_grouped()	Median (50th percentile) of grouped data.
mode()	Single mode (most common value) of discrete or nominal data.
multimode()	List of modes (most common values) of discrete or nominal data.
quantiles()	Divide data into intervals with equal probability.

9.7.2 Measures of spread

These functions calculate a measure of how much the population or sample tends to deviate from the typical or average values.

pstdev()	Population standard deviation of data.
pvariance()	Population variance of data.
stdev()	Sample standard deviation of data.
variance()	Sample variance of data.

9.7.3 Statistics for relations between two inputs

These functions calculate statistics regarding relations between two inputs.

covariance()	Sample covariance for two variables.
correlation()	Pearson and Spearman's correlation coefficients.
linear_regression()	Slope and intercept for simple linear regression.

9.7.4 Function details

Note: The functions do not require the data given to them to be sorted. However, for reading convenience, most of the examples show sorted sequences.

`statistics.`**`mean`**(*data*)

Return the sample arithmetic mean of *data* which can be a sequence or iterable.

The arithmetic mean is the sum of the data divided by the number of data points. It is commonly called "the average", although it is only one of many different mathematical averages. It is a measure of the central location of the data.

If *data* is empty, *StatisticsError* will be raised.

Some examples of use:

```
>>> mean([1, 2, 3, 4, 4])
2.8
>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625

>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)

>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
Decimal('0.5625')
```

Note

The mean is strongly affected by outliers and is not necessarily a typical example of the data points. For a more robust, although less efficient, measure of central tendency, see *median()*.

The sample mean gives an unbiased estimate of the true population mean, so that when taken on average over all the possible samples, `mean(sample)` converges on the true mean of the entire population. If *data* represents the entire population rather than a sample, then `mean(data)` is equivalent to calculating the true population mean μ.

`statistics.`**`fmean`**(*data*, *weights=None*)

Convert *data* to floats and compute the arithmetic mean.

This runs faster than the *`mean()`* function and it always returns a *`float`*. The *data* may be a sequence or iterable. If the input dataset is empty, raises a *`StatisticsError`*.

```
>>> fmean([3.5, 4.0, 5.25])
4.25
```

Optional weighting is supported. For example, a professor assigns a grade for a course by weighting quizzes at 20%, homework at 20%, a midterm exam at 30%, and a final exam at 30%:

```
>>> grades = [85, 92, 83, 91]
>>> weights = [0.20, 0.20, 0.30, 0.30]
>>> fmean(grades, weights)
87.6
```

If *weights* is supplied, it must be the same length as the *data* or a *`ValueError`* will be raised.

Added in version 3.8.

Changed in version 3.11: Added support for *weights*.

`statistics.`**`geometric_mean`**(*data*)

Convert *data* to floats and compute the geometric mean.

The geometric mean indicates the central tendency or typical value of the *data* using the product of the values (as opposed to the arithmetic mean which uses their sum).

Raises a *`StatisticsError`* if the input dataset is empty, if it contains a zero, or if it contains a negative value. The *data* may be a sequence or iterable.

No special efforts are made to achieve exact results. (However, this may change in the future.)

```
>>> round(geometric_mean([54, 24, 36]), 1)
36.0
```

Added in version 3.8.

`statistics.`**`harmonic_mean`**(*data*, *weights=None*)

Return the harmonic mean of *data*, a sequence or iterable of real-valued numbers. If *weights* is omitted or `None`, then equal weighting is assumed.

The harmonic mean is the reciprocal of the arithmetic *`mean()`* of the reciprocals of the data. For example, the harmonic mean of three values *a*, *b* and *c* will be equivalent to `3/(1/a + 1/b + 1/c)`. If one of the values is zero, the result will be zero.

The harmonic mean is a type of average, a measure of the central location of the data. It is often appropriate when averaging ratios or rates, for example speeds.

Suppose a car travels 10 km at 40 km/hr, then another 10 km at 60 km/hr. What is the average speed?

```
>>> harmonic_mean([40, 60])
48.0
```

Suppose a car travels 40 km/hr for 5 km, and when traffic clears, speeds-up to 60 km/hr for the remaining 30 km of the journey. What is the average speed?

```
>>> harmonic_mean([40, 60], weights=[5, 30])
56.0
```

StatisticsError is raised if *data* is empty, any element is less than zero, or if the weighted sum isn't positive.

The current algorithm has an early-out when it encounters a zero in the input. This means that the subsequent inputs are not tested for validity. (This behavior may change in the future.)

Added in version 3.6.

Changed in version 3.10: Added support for *weights*.

statistics.**kde**(*data*, *h*, *kernel='normal'*, ***, *cumulative=False*)

Kernel Density Estimation (KDE): Create a continuous probability density function or cumulative distribution function from discrete samples.

The basic idea is to smooth the data using a kernel function. to help draw inferences about a population from a sample.

The degree of smoothing is controlled by the scaling parameter *h* which is called the bandwidth. Smaller values emphasize local features while larger values give smoother results.

The *kernel* determines the relative weights of the sample data points. Generally, the choice of kernel shape does not matter as much as the more influential bandwidth smoothing parameter.

Kernels that give some weight to every sample point include *normal* (*gauss*), *logistic*, and *sigmoid*.

Kernels that only give weight to sample points within the bandwidth include *rectangular* (*uniform*), *triangular*, *parabolic* (*epanechnikov*), *quartic* (*biweight*), *triweight*, and *cosine*.

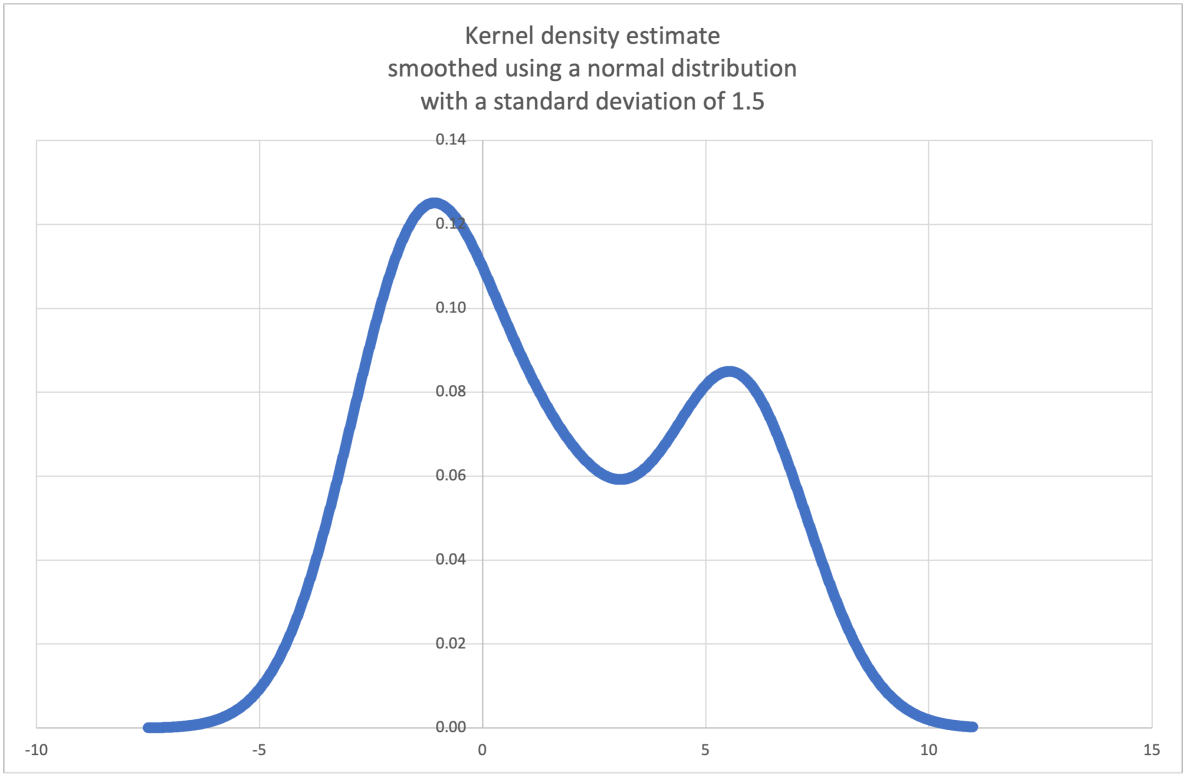
If *cumulative* is true, will return a cumulative distribution function.

A *StatisticsError* will be raised if the *data* sequence is empty.

Wikipedia has an example where we can use *kde()* to generate and plot a probability density function estimated from a small sample:

```
>>> sample = [-2.1, -1.3, -0.4, 1.9, 5.1, 6.2]
>>> f_hat = kde(sample, h=1.5)
>>> xarr = [i/100 for i in range(-750, 1100)]
>>> yarr = [f_hat(x) for x in xarr]
```

The points in `xarr` and `yarr` can be used to make a PDF plot:



Added in version 3.13.

`statistics.`**`kde_random`**(*data*, *h*, *kernel='normal'*, ***, *seed=None*)

Return a function that makes a random selection from the estimated probability density function produced by `kde(data, h, kernel)`.

Providing a *seed* allows reproducible selections. In the future, the values may change slightly as more accurate kernel inverse CDF estimates are implemented. The seed may be an integer, float, str, or bytes.

A *`StatisticsError`* will be raised if the *data* sequence is empty.

Continuing the example for *`kde()`*, we can use *`kde_random()`* to generate new random selections from an estimated probability density function:

```
>>> data = [-2.1, -1.3, -0.4, 1.9, 5.1, 6.2]
>>> rand = kde_random(data, h=1.5, seed=8675309)
>>> new_selections = [rand() for i in range(10)]
>>> [round(x, 1) for x in new_selections]
[0.7, 6.2, 1.2, 6.9, 7.0, 1.8, 2.5, -0.5, -1.8, 5.6]
```

Added in version 3.13.

`statistics.`**`median`**(*data*)

Return the median (middle value) of numeric data, using the common "mean of middle two" method. If *data* is empty, *`StatisticsError`* is raised. *data* can be a sequence or iterable.

The median is a robust measure of central location and is less affected by the presence of outliers. When the number of data points is odd, the middle data point is returned:

```
>>> median([1, 3, 5])
3
```

When the number of data points is even, the median is interpolated by taking the average of the two middle values:

```
>>> median([1, 3, 5, 7])
4.0
```

This is suited for when your data is discrete, and you don't mind that the median may not be an actual data point.

If the data is ordinal (supports order operations) but not numeric (doesn't support addition), consider using *median_low()* or *median_high()* instead.

statistics.**median_low**(*data*)

Return the low median of numeric data. If *data* is empty, *StatisticsError* is raised. *data* can be a sequence or iterable.

The low median is always a member of the data set. When the number of data points is odd, the middle value is returned. When it is even, the smaller of the two middle values is returned.

```
>>> median_low([1, 3, 5])
3
>>> median_low([1, 3, 5, 7])
3
```

Use the low median when your data are discrete and you prefer the median to be an actual data point rather than interpolated.

statistics.**median_high**(*data*)

Return the high median of data. If *data* is empty, *StatisticsError* is raised. *data* can be a sequence or iterable.

The high median is always a member of the data set. When the number of data points is odd, the middle value is returned. When it is even, the larger of the two middle values is returned.

```
>>> median_high([1, 3, 5])
3
>>> median_high([1, 3, 5, 7])
5
```

Use the high median when your data are discrete and you prefer the median to be an actual data point rather than interpolated.

statistics.**median_grouped**(*data*, *interval=1.0*)

Estimates the median for numeric data that has been grouped or binned around the midpoints of consecutive, fixed-width intervals.

The *data* can be any iterable of numeric data with each value being exactly the midpoint of a bin. At least one value must be present.

The *interval* is the width of each bin.

For example, demographic information may have been summarized into consecutive ten-year age groups with each group being represented by the 5-year midpoints of the intervals:

```
>>> from collections import Counter
>>> demographics = Counter({
...    25: 172,   # 20 to 30 years old
...    35: 484,   # 30 to 40 years old
...    45: 387,   # 40 to 50 years old
...    55:  22,   # 50 to 60 years old
...    65:   6,   # 60 to 70 years old
... })
...
```

The 50th percentile (median) is the 536th person out of the 1071 member cohort. That person is in the 30 to 40 year old age group.

The regular *median()* function would assume that everyone in the tricenarian age group was exactly 35 years old. A more tenable assumption is that the 484 members of that age group are evenly distributed between 30 and 40. For that, we use *median_grouped()*:

```
>>> data = list(demographics.elements())
>>> median(data)
35
>>> round(median_grouped(data, interval=10), 1)
37.5
```

The caller is responsible for making sure the data points are separated by exact multiples of *interval*. This is essential for getting a correct result. The function does not check this precondition.

Inputs may be any numeric type that can be coerced to a float during the interpolation step.

statistics.**mode**(*data*)

Return the single most common data point from discrete or nominal *data*. The mode (when it exists) is the most typical value and serves as a measure of central location.

If there are multiple modes with the same frequency, returns the first one encountered in the *data*. If the smallest or largest of those is desired instead, use `min(multimode(data))` or `max(multimode(data))`. If the input *data* is empty, *StatisticsError* is raised.

`mode` assumes discrete data and returns a single value. This is the standard treatment of the mode as commonly taught in schools:

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

The mode is unique in that it is the only statistic in this package that also applies to nominal (non-numeric) data:

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

Only hashable inputs are supported. To handle type *set*, consider casting to *frozenset*. To handle type *list*, consider casting to *tuple*. For mixed or nested inputs, consider using this slower quadratic algorithm that only depends on equality tests: `max(data, key=data.count)`.

Changed in version 3.8: Now handles multimodal datasets by returning the first mode encountered. Formerly, it raised *StatisticsError* when more than one mode was found.

statistics.**multimode**(*data*)

Return a list of the most frequently occurring values in the order they were first encountered in the *data*. Will return more than one result if there are multiple modes or an empty list if the *data* is empty:

```
>>> multimode('aabbbbccddddeeffffgg')
['b', 'd', 'f']
>>> multimode('')
[]
```

Added in version 3.8.

statistics.**pstdev**(*data*, *mu=None*)

Return the population standard deviation (the square root of the population variance). See *pvariance()* for arguments and other details.

```
>>> pstdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
0.986893273527251
```

statistics.**pvariance**(*data*, *mu=None*)

Return the population variance of *data*, a non-empty sequence or iterable of real-valued numbers. Variance, or second moment about the mean, is a measure of the variability (spread or dispersion) of data. A large variance indicates that the data is spread out; a small variance indicates it is clustered closely around the mean.

If the optional second argument *mu* is given, it should be the *population* mean of the *data*. It can also be used to compute the second moment around a point that is not the mean. If it is missing or `None` (the default), the arithmetic mean is automatically calculated.

Use this function to calculate the variance from the entire population. To estimate the variance from a sample, the *variance()* function is usually a better choice.

Raises *StatisticsError* if *data* is empty.

Examples:

```
>>> data = [0.0, 0.25, 0.25, 1.25, 1.5, 1.75, 2.75, 3.25]
>>> pvariance(data)
1.25
```

If you have already calculated the mean of your data, you can pass it as the optional second argument *mu* to avoid recalculation:

```
>>> mu = mean(data)
>>> pvariance(data, mu)
1.25
```

Decimals and Fractions are supported:

```
>>> from decimal import Decimal as D
>>> pvariance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('24.815')

>>> from fractions import Fraction as F
>>> pvariance([F(1, 4), F(5, 4), F(1, 2)])
Fraction(13, 72)
```

Note

When called with the entire population, this gives the population variance σ^2. When called on a sample instead, this is the biased sample variance s^2, also known as variance with N degrees of freedom.

If you somehow know the true population mean μ, you may use this function to calculate the variance of a sample, giving the known population mean as the second argument. Provided the data points are a random sample of the population, the result will be an unbiased estimate of the population variance.

statistics.**stdev**(*data*, *xbar=None*)

Return the sample standard deviation (the square root of the sample variance). See *variance()* for arguments and other details.

```
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
```

statistics.**variance**(*data*, *xbar=None*)

Return the sample variance of *data*, an iterable of at least two real-valued numbers. Variance, or second moment about the mean, is a measure of the variability (spread or dispersion) of data. A large variance indicates that the data is spread out; a small variance indicates it is clustered closely around the mean.

If the optional second argument *xbar* is given, it should be the *sample* mean of *data*. If it is missing or `None` (the default), the mean is automatically calculated.

Use this function when your data is a sample from a population. To calculate the variance from the entire population, see *pvariance()*.

Raises *StatisticsError* if *data* has fewer than two values.

Examples:

```
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> variance(data)
1.3720238095238095
```

If you have already calculated the sample mean of your data, you can pass it as the optional second argument *xbar* to avoid recalculation:

```
>>> m = mean(data)
>>> variance(data, m)
1.3720238095238095
```

This function does not attempt to verify that you have passed the actual mean as *xbar*. Using arbitrary values for *xbar* can lead to invalid or impossible results.

Decimal and Fraction values are supported:

```
>>> from decimal import Decimal as D
>>> variance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('31.01875')

>>> from fractions import Fraction as F
>>> variance([F(1, 6), F(1, 2), F(5, 3)])
Fraction(67, 108)
```

Note

This is the sample variance s^2 with Bessel's correction, also known as variance with N-1 degrees of freedom. Provided that the data points are representative (e.g. independent and identically distributed), the result should be an unbiased estimate of the true population variance.

If you somehow know the actual population mean μ you should pass it to the *pvariance()* function as the *mu* parameter to get the variance of a sample.

statistics.**quantiles**(*data*, *, *n=4*, *method='exclusive'*)

Divide *data* into *n* continuous intervals with equal probability. Returns a list of `n - 1` cut points separating the intervals.

Set *n* to 4 for quartiles (the default). Set *n* to 10 for deciles. Set *n* to 100 for percentiles which gives the 99 cuts points that separate *data* into 100 equal sized groups. Raises *StatisticsError* if *n* is not least 1.

The *data* can be any iterable containing sample data. For meaningful results, the number of data points in *data* should be larger than *n*. Raises *StatisticsError* if there is not at least one data point.

The cut points are linearly interpolated from the two nearest data points. For example, if a cut point falls one-third of the distance between two sample values, `100` and `112`, the cut-point will evaluate to `104`.

The *method* for computing quantiles can be varied depending on whether the *data* includes or excludes the lowest and highest possible values from the population.

The default *method* is "exclusive" and is used for data sampled from a population that can have more extreme values than found in the samples. The portion of the population falling below the *i-th* of *m* sorted data points is computed as `i / (m + 1)`. Given nine sample values, the method sorts them and assigns the following percentiles: 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%.

Setting the *method* to "inclusive" is used for describing population data or for samples that are known to include the most extreme values from the population. The minimum value in *data* is treated as the 0th percentile and the maximum value is treated as the 100th percentile. The portion of the population falling below the *i-th* of *m* sorted data points is computed as `(i - 1) / (m - 1)`. Given 11 sample values, the method sorts them and assigns the following percentiles: 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%.

```
# Decile cut points for empirically sampled data
>>> data = [105, 129, 87, 86, 111, 111, 89, 81, 108, 92, 110,
...         100, 75, 105, 103, 109, 76, 119, 99, 91, 103, 129,
...         106, 101, 84, 111, 74, 87, 86, 103, 103, 106, 86,
...         111, 75, 87, 102, 121, 111, 88, 89, 101, 106, 95,
...         103, 107, 101, 81, 109, 104]
>>> [round(q, 1) for q in quantiles(data, n=10)]
[81.0, 86.2, 89.0, 99.4, 102.5, 103.6, 106.0, 109.8, 111.0]
```

Added in version 3.8.

Changed in version 3.13: No longer raises an exception for an input with only a single data point. This allows quantile estimates to be built up one sample point at a time becoming gradually more refined with each new data point.

`statistics.`**`covariance`**(*x*, *y*, */*)

Return the sample covariance of two inputs *x* and *y*. Covariance is a measure of the joint variability of two inputs.

Both inputs must be of the same length (no less than two), otherwise *`StatisticsError`* is raised.

Examples:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> y = [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> covariance(x, y)
0.75
>>> z = [9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> covariance(x, z)
-7.5
>>> covariance(z, x)
-7.5
```

Added in version 3.10.

`statistics.`**`correlation`**(*x*, *y*, */*, *, *method='linear'*)

Return the Pearson's correlation coefficient for two inputs. Pearson's correlation coefficient *r* takes values between -1 and +1. It measures the strength and direction of a linear relationship.

If *method* is "ranked", computes Spearman's rank correlation coefficient for two inputs. The data is replaced by ranks. Ties are averaged so that equal values receive the same rank. The resulting coefficient measures the strength of a monotonic relationship.

Spearman's correlation coefficient is appropriate for ordinal data or for continuous data that doesn't meet the linear proportion requirement for Pearson's correlation coefficient.

Both inputs must be of the same length (no less than two), and need not to be constant, otherwise *StatisticsError* is raised.

Example with Kepler's laws of planetary motion:

```
>>> # Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, and  Neptune
>>> orbital_period = [88, 225, 365, 687, 4331, 10_756, 30_687, 60_190]    # days
>>> dist_from_sun = [58, 108, 150, 228, 778, 1_400, 2_900, 4_500] # million km

>>> # Show that a perfect monotonic relationship exists
>>> correlation(orbital_period, dist_from_sun, method='ranked')
1.0

>>> # Observe that a linear relationship is imperfect
>>> round(correlation(orbital_period, dist_from_sun), 4)
0.9882

>>> # Demonstrate Kepler's third law: There is a linear correlation
>>> # between the square of the orbital period and the cube of the
>>> # distance from the sun.
>>> period_squared = [p * p for p in orbital_period]
>>> dist_cubed = [d * d * d for d in dist_from_sun]
>>> round(correlation(period_squared, dist_cubed), 4)
1.0
```

Added in version 3.10.

Changed in version 3.12: Added support for Spearman's rank correlation coefficient.

statistics.**linear_regression**(*x*, *y*, */*, ***, *proportional=False*)

Return the slope and intercept of simple linear regression parameters estimated using ordinary least squares. Simple linear regression describes the relationship between an independent variable *x* and a dependent variable *y* in terms of this linear function:

*y = slope * x + intercept + noise*

where `slope` and `intercept` are the regression parameters that are estimated, and `noise` represents the variability of the data that was not explained by the linear regression (it is equal to the difference between predicted and actual values of the dependent variable).

Both inputs must be of the same length (no less than two), and the independent variable *x* cannot be constant; otherwise a *StatisticsError* is raised.

For example, we can use the release dates of the Monty Python films to predict the cumulative number of Monty Python films that would have been produced by 2019 assuming that they had kept the pace.

```
>>> year = [1971, 1975, 1979, 1982, 1983]
>>> films_total = [1, 2, 3, 4, 5]
>>> slope, intercept = linear_regression(year, films_total)
>>> round(slope * 2019 + intercept)
16
```

If *proportional* is true, the independent variable *x* and the dependent variable *y* are assumed to be directly proportional. The data is fit to a line passing through the origin. Since the *intercept* will always be 0.0, the underlying linear function simplifies to:

*y = slope * x + noise*

Continuing the example from *correlation()*, we look to see how well a model based on major planets can predict the orbital distances for dwarf planets:

```
>>> model = linear_regression(period_squared, dist_cubed, proportional=True)
>>> slope = model.slope

>>> # Dwarf planets:   Pluto,  Eris,    Makemake, Haumea, Ceres
>>> orbital_periods = [90_560, 204_199, 111_845, 103_410, 1_680]  # days
>>> predicted_dist = [math.cbrt(slope * (p * p)) for p in orbital_periods]
>>> list(map(round, predicted_dist))
[5912, 10166, 6806, 6459, 414]

>>> [5_906, 10_152, 6_796, 6_450, 414]  # actual distance in million km
[5906, 10152, 6796, 6450, 414]
```

Added in version 3.10.

Changed in version 3.11: Added support for *proportional*.

9.7.5 Exceptions

A single exception is defined:

exception statistics.**StatisticsError**

Subclass of *ValueError* for statistics-related exceptions.

9.7.6 `NormalDist` objects

NormalDist is a tool for creating and manipulating normal distributions of a random variable. It is a class that treats the mean and standard deviation of data measurements as a single entity.

Normal distributions arise from the Central Limit Theorem and have a wide range of applications in statistics.

class statistics.**NormalDist**(*mu=0.0*, *sigma=1.0*)

Returns a new *NormalDist* object where *mu* represents the arithmetic mean and *sigma* represents the standard deviation.

If *sigma* is negative, raises *StatisticsError*.

mean

A read-only property for the arithmetic mean of a normal distribution.

median

A read-only property for the median of a normal distribution.

mode

A read-only property for the mode of a normal distribution.

stdev

A read-only property for the standard deviation of a normal distribution.

variance

A read-only property for the variance of a normal distribution. Equal to the square of the standard deviation.

classmethod from_samples(*data*)

Makes a normal distribution instance with *mu* and *sigma* parameters estimated from the *data* using `fmean()` and `stdev()`.

The *data* can be any *iterable* and should consist of values that can be converted to type `float`. If *data* does not contain at least two elements, raises `StatisticsError` because it takes at least one point to estimate a central value and at least two points to estimate dispersion.

samples(*n*, *, *seed=None*)

Generates *n* random samples for a given mean and standard deviation. Returns a `list` of `float` values.

If *seed* is given, creates a new instance of the underlying random number generator. This is useful for creating reproducible results, even in a multi-threading context.

Changed in version 3.13.

Switched to a faster algorithm. To reproduce samples from previous versions, use `random.seed()` and `random.gauss()`.

pdf(*x*)

Using a probability density function (pdf), compute the relative likelihood that a random variable *X* will be near the given value *x*. Mathematically, it is the limit of the ratio `P(x <= X < x+dx) / dx` as *dx* approaches zero.

The relative likelihood is computed as the probability of a sample occurring in a narrow range divided by the width of the range (hence the word "density"). Since the likelihood is relative to other points, its value can be greater than `1.0`.

cdf(*x*)

Using a cumulative distribution function (cdf), compute the probability that a random variable *X* will be less than or equal to *x*. Mathematically, it is written `P(X <= x)`.

inv_cdf(*p*)

Compute the inverse cumulative distribution function, also known as the quantile function or the percent-point function. Mathematically, it is written `x : P(X <= x) = p`.

Finds the value *x* of the random variable *X* such that the probability of the variable being less than or equal to that value equals the given probability *p*.

overlap(*other*)

Measures the agreement between two normal probability distributions. Returns a value between 0.0 and 1.0 giving the overlapping area for the two probability density functions.

quantiles(*n=4*)

Divide the normal distribution into *n* continuous intervals with equal probability. Returns a list of (n - 1) cut points separating the intervals.

Set *n* to 4 for quartiles (the default). Set *n* to 10 for deciles. Set *n* to 100 for percentiles which gives the 99 cuts points that separate the normal distribution into 100 equal sized groups.

zscore(*x*)

Compute the Standard Score describing *x* in terms of the number of standard deviations above or below the mean of the normal distribution: `(x - mean) / stdev`.

Added in version 3.9.

Instances of `NormalDist` support addition, subtraction, multiplication and division by a constant. These operations are used for translation and scaling. For example:

```
>>> temperature_february = NormalDist(5, 2.5)             # Celsius
>>> temperature_february * (9/5) + 32                     # Fahrenheit
NormalDist(mu=41.0, sigma=4.5)
```

Dividing a constant by an instance of *NormalDist* is not supported because the result wouldn't be normally distributed.

Since normal distributions arise from additive effects of independent variables, it is possible to add and subtract two independent normally distributed random variables represented as instances of *NormalDist*. For example:

```
>>> birth_weights = NormalDist.from_samples([2.5, 3.1, 2.1, 2.4, 2.7, 3.5])
>>> drug_effects = NormalDist(0.4, 0.15)
>>> combined = birth_weights + drug_effects
>>> round(combined.mean, 1)
3.1
>>> round(combined.stdev, 1)
0.5
```

Added in version 3.8.

9.7.7 Examples and Recipes

Classic probability problems

NormalDist readily solves classic probability problems.

For example, given historical data for SAT exams showing that scores are normally distributed with a mean of 1060 and a standard deviation of 195, determine the percentage of students with test scores between 1100 and 1200, after rounding to the nearest whole number:

```
>>> sat = NormalDist(1060, 195)
>>> fraction = sat.cdf(1200 + 0.5) - sat.cdf(1100 - 0.5)
>>> round(fraction * 100.0, 1)
18.4
```

Find the quartiles and deciles for the SAT scores:

```
>>> list(map(round, sat.quantiles()))
[928, 1060, 1192]
>>> list(map(round, sat.quantiles(n=10)))
[810, 896, 958, 1011, 1060, 1109, 1162, 1224, 1310]
```

Monte Carlo inputs for simulations

To estimate the distribution for a model that isn't easy to solve analytically, *NormalDist* can generate input samples for a Monte Carlo simulation:

```
>>> def model(x, y, z):
...     return (3*x + 7*x*y - 5*y) / (11 * z)
...
>>> n = 100_000
>>> X = NormalDist(10, 2.5).samples(n, seed=3652260728)
>>> Y = NormalDist(15, 1.75).samples(n, seed=4582495471)
>>> Z = NormalDist(50, 1.25).samples(n, seed=6582483453)
```

(continues on next page)

(continued from previous page)

```
>>> quantiles(map(model, X, Y, Z))
[1.4591308524824727, 1.8035946855390597, 2.175091447274739]
```

Approximating binomial distributions

Normal distributions can be used to approximate Binomial distributions when the sample size is large and when the probability of a successful trial is near 50%.

For example, an open source conference has 750 attendees and two rooms with a 500 person capacity. There is a talk about Python and another about Ruby. In previous conferences, 65% of the attendees preferred to listen to Python talks. Assuming the population preferences haven't changed, what is the probability that the Python room will stay within its capacity limits?

```
>>> n = 750             # Sample size
>>> p = 0.65            # Preference for Python
>>> q = 1.0 - p         # Preference for Ruby
>>> k = 500             # Room capacity

>>> # Approximation using the cumulative normal distribution
>>> from math import sqrt
>>> round(NormalDist(mu=n*p, sigma=sqrt(n*p*q)).cdf(k + 0.5), 4)
0.8402

>>> # Exact solution using the cumulative binomial distribution
>>> from math import comb, fsum
>>> round(fsum(comb(n, r) * p**r * q**(n-r) for r in range(k+1)), 4)
0.8402

>>> # Approximation using a simulation
>>> from random import seed, binomialvariate
>>> seed(8675309)
>>> mean(binomialvariate(n, p) <= k for i in range(10_000))
0.8406
```

Naive bayesian classifier

Normal distributions commonly arise in machine learning problems.

Wikipedia has a nice example of a Naive Bayesian Classifier. The challenge is to predict a person's gender from measurements of normally distributed features including height, weight, and foot size.

We're given a training dataset with measurements for eight people. The measurements are assumed to be normally distributed, so we summarize the data with *NormalDist*:

```
>>> height_male = NormalDist.from_samples([6, 5.92, 5.58, 5.92])
>>> height_female = NormalDist.from_samples([5, 5.5, 5.42, 5.75])
>>> weight_male = NormalDist.from_samples([180, 190, 170, 165])
>>> weight_female = NormalDist.from_samples([100, 150, 130, 150])
>>> foot_size_male = NormalDist.from_samples([12, 11, 12, 10])
>>> foot_size_female = NormalDist.from_samples([6, 8, 7, 9])
```

Next, we encounter a new person whose feature measurements are known but whose gender is unknown:

```
>>> ht = 6.0         # height
>>> wt = 130         # weight
>>> fs = 8           # foot size
```

Starting with a 50% prior probability of being male or female, we compute the posterior as the prior times the product of likelihoods for the feature measurements given the gender:

```
>>> prior_male = 0.5
>>> prior_female = 0.5
>>> posterior_male = (prior_male * height_male.pdf(ht) *
...                   weight_male.pdf(wt) * foot_size_male.pdf(fs))

>>> posterior_female = (prior_female * height_female.pdf(ht) *
...                     weight_female.pdf(wt) * foot_size_female.pdf(fs))
```

The final prediction goes to the largest posterior. This is known as the maximum a posteriori or MAP:

```
>>> 'male' if posterior_male > posterior_female else 'female'
'female'
```

CHAPTER

TEN

FUNCTIONAL PROGRAMMING MODULES

The modules described in this chapter provide functions and classes that support a functional programming style, and general operations on callables.

The following modules are documented in this chapter:

10.1 `itertools` — Functions creating iterators for efficient looping

This module implements a number of *iterator* building blocks inspired by constructs from APL, Haskell, and SML. Each has been recast in a form suitable for Python.

The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an "iterator algebra" making it possible to construct specialized tools succinctly and efficiently in pure Python.

For instance, SML provides a tabulation tool: `tabulate(f)` which produces a sequence `f(0), f(1), ...`. The same effect can be achieved in Python by combining *`map()`* and *`count()`* to form `map(f, count())`.

These tools and their built-in counterparts also work well with the high-speed functions in the *`operator`* module. For example, the multiplication operator can be mapped across two vectors to form an efficient dot-product: `sum(starmap(operator.mul, zip(vec1, vec2, strict=True)))`.

Infinite iterators:

Iterator	Arguments	Results	Example
`count()`	[start[, step]]	start, start+step, start+2*step, …	`count(10) → 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, … plast, p0, p1, …	`cycle('ABCD') → A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, … endlessly or up to n times	`repeat(10, 3) → 10 10 10`

Iterators terminating on the shortest input sequence:

Iterator	Arguments	Results	Example
accumulate()	p [,func]	p0, p0+p1, p0+p1+p2, …	`accumulate([1,2,3,4,5]) → 1 3 6 10 15`
batched()	p, n	(p0, p1, …, p_n-1), …	`batched('ABCDEFG', n=3) → ABC DEF G`
chain()	p, q, …	p0, p1, … plast, q0, q1, …	`chain('ABC', 'DEF') → A B C D E F`
chain.from_iterable	iterable	p0, p1, … plast, q0, q1, …	`chain.from_iterable(['ABC', 'DEF']) → A B C D E F`
compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), …	`compress('ABCDEF', [1,0,1,0,1,1]) → A C E F`
dropwhile()	predicate, seq	seq[n], seq[n+1], starting when predicate fails	`dropwhile(lambda x: x<5, [1,4,6,3,8]) → 6 3 8`
filterfalse()	predicate, seq	elements of seq where predicate(elem) fails	`filterfalse(lambda x: x<5, [1,4,6,3,8]) → 6 8`
groupby()	iterable[, key]	sub-iterators grouped by value of key(v)	
islice()	seq, [start,] stop [, step]	elements from seq[start:stop:step]	`islice('ABCDEFG', 2, None) → C D E F G`
pairwise()	iterable	(p[0], p[1]), (p[1], p[2])	`pairwise('ABCDEFG') → AB BC CD DE EF FG`
starmap()	func, seq	func(*seq[0]), func(*seq[1]), …	`starmap(pow, [(2,5), (3,2), (10,3)]) → 32 9 1000`
takewhile()	predicate, seq	seq[0], seq[1], until predicate fails	`takewhile(lambda x: x<5, [1,4,6,3,8]) → 1 4`
tee()	it, n	it1, it2, … itn splits one iterator into n	
zip_longest()	p, q, …	(p[0], q[0]), (p[1], q[1]), …	`zip_longest('ABCD', 'xy', fillvalue='-') → Ax By C- D-`

Combinatoric iterators:

Iterator	Arguments	Results
product()	p, q, … [repeat=1]	cartesian product, equivalent to a nested for-loop
permutations()	p[, r]	r-length tuples, all possible orderings, no repeated elements
combinations()	p, r	r-length tuples, in sorted order, no repeated elements
combinations_with_replacement()	p, r	r-length tuples, in sorted order, with repeated elements

Examples	Results
`product('ABCD', repeat=2)`	`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`	`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`	`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD 2)`	`AA AB AC AD BB BC BD CC CD DD`

10.1.1 Itertool Functions

The following module functions all construct and return iterators. Some provide streams of infinite length, so they should only be accessed by functions or loops that truncate the stream.

itertools.**accumulate**(*iterable*[, *function*, *, *initial=None*])

Make an iterator that returns accumulated sums or accumulated results from other binary functions.

The *function* defaults to addition. The *function* should accept two arguments, an accumulated total and a value from the *iterable*.

If an *initial* value is provided, the accumulation will start with that value and the output will have one more element than the input iterable.

Roughly equivalent to:

```
def accumulate(iterable, function=operator.add, *, initial=None):
    'Return running totals'
    # accumulate([1,2,3,4,5]) → 1 3 6 10 15
    # accumulate([1,2,3,4,5], initial=100) → 100 101 103 106 110 115
    # accumulate([1,2,3,4,5], operator.mul) → 1 2 6 24 120

    iterator = iter(iterable)
    total = initial
    if initial is None:
        try:
            total = next(iterator)
        except StopIteration:
            return

    yield total
    for element in iterator:
        total = function(total, element)
        yield total
```

The *function* argument can be set to *min()* for a running minimum, *max()* for a running maximum, or *operator.mul()* for a running product. Amortization tables can be built by accumulating interest and applying payments:

```
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, max))                # running maximum
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]
>>> list(accumulate(data, operator.mul))       # running product
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]

# Amortize a 5% loan of 1000 with 10 annual payments of 90
>>> update = lambda balance, payment: round(balance * 1.05) - payment
>>> list(accumulate(repeat(90, 10), update, initial=1_000))
[1000, 960, 918, 874, 828, 779, 728, 674, 618, 559, 497]
```

See *functools.reduce()* for a similar function that returns only the final accumulated value.

Added in version 3.2.

Changed in version 3.3: Added the optional *function* parameter.

Changed in version 3.8: Added the optional *initial* parameter.

itertools.**batched**(*iterable*, *n*, *, *strict=False*)

Batch data from the *iterable* into tuples of length *n*. The last batch may be shorter than *n*.

If *strict* is true, will raise a *ValueError* if the final batch is shorter than *n*.

Loops over the input iterable and accumulates data into tuples up to size *n*. The input is consumed lazily, just enough to fill a batch. The result is yielded as soon as the batch is full or when the input iterable is exhausted:

```
>>> flattened_data = ['roses', 'red', 'violets', 'blue', 'sugar', 'sweet']
>>> unflattened = list(batched(flattened_data, 2))
>>> unflattened
[('roses', 'red'), ('violets', 'blue'), ('sugar', 'sweet')]
```

Roughly equivalent to:

```
def batched(iterable, n, *, strict=False):
    # batched('ABCDEFG', 3) → ABC DEF G
    if n < 1:
        raise ValueError('n must be at least one')
    iterator = iter(iterable)
    while batch := tuple(islice(iterator, n)):
        if strict and len(batch) != n:
            raise ValueError('batched(): incomplete batch')
        yield batch
```

Added in version 3.12.

Changed in version 3.13: Added the *strict* option.

itertools.**chain**(**iterables*)

Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. Used for treating consecutive sequences as a single sequence. Roughly equivalent to:

```
def chain(*iterables):
    # chain('ABC', 'DEF') → A B C D E F
    for iterable in iterables:
        yield from iterable
```

classmethod chain.**from_iterable**(*iterable*)

Alternate constructor for *chain()*. Gets chained inputs from a single iterable argument that is evaluated lazily. Roughly equivalent to:

```
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) → A B C D E F
    for iterable in iterables:
        yield from iterable
```

itertools.**combinations**(*iterable*, *r*)

Return *r* length subsequences of elements from the input *iterable*.

The output is a subsequence of *product()* keeping only entries that are subsequences of the *iterable*. The length of the output is given by *math.comb()* which computes `n! / r! / (n - r)!` when `0 ≤ r ≤ n` or zero when `r > n`.

The combination tuples are emitted in lexicographic order according to the order of the input *iterable*. If the input *iterable* is sorted, the output tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. If the input elements are unique, there will be no repeated values within each combination.

Roughly equivalent to:

```
def combinations(iterable, r):
    # combinations('ABCD', 2) → AB AC AD BC BD CD
    # combinations(range(4), 3) → 012 013 023 123

    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = list(range(r))

    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

itertools.**combinations_with_replacement**(*iterable*, *r*)

Return *r* length subsequences of elements from the input *iterable* allowing individual elements to be repeated more than once.

The output is a subsequence of *product()* that keeps only entries that are subsequences (with possible repeated elements) of the *iterable*. The number of subsequence returned is `(n + r - 1)! / r! / (n - 1)!` when `n > 0`.

The combination tuples are emitted in lexicographic order according to the order of the input *iterable*. if the input *iterable* is sorted, the output tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. If the input elements are unique, the generated combinations will also be unique.

Roughly equivalent to:

```
def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) → AA AB AC BB BC CC

    pool = tuple(iterable)
    n = len(pool)
    if not n and r:
        return
    indices = [0] * r

    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return
        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)
```

Added in version 3.1.

itertools.**compress**(*data*, *selectors*)

Make an iterator that returns elements from *data* where the corresponding element in *selectors* is true. Stops when either the *data* or *selectors* iterables have been exhausted. Roughly equivalent to:

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) → A C E F
    return (datum for datum, selector in zip(data, selectors) if selector)
```

Added in version 3.1.

itertools.**count**(*start=0*, *step=1*)

Make an iterator that returns evenly spaced values beginning with *start*. Can be used with `map()` to generate consecutive data points or with `zip()` to add sequence numbers. Roughly equivalent to:

```
def count(start=0, step=1):
    # count(10) → 10 11 12 13 14 ...
    # count(2.5, 0.5) → 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

When counting with floating-point numbers, better accuracy can sometimes be achieved by substituting multiplicative code such as: `(start + step * i for i in count())`.

Changed in version 3.1: Added *step* argument and allowed non-integer arguments.

itertools.**cycle**(*iterable*)

Make an iterator returning elements from the *iterable* and saving a copy of each. When the iterable is exhausted, return elements from the saved copy. Repeats indefinitely. Roughly equivalent to:

```
def cycle(iterable):
    # cycle('ABCD') → A B C D A B C D A B C D ...
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
            yield element
```

This itertool may require significant auxiliary storage (depending on the length of the iterable).

itertools.**dropwhile**(*predicate*, *iterable*)

Make an iterator that drops elements from the *iterable* while the *predicate* is true and afterwards returns every element. Roughly equivalent to:

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,3,8]) → 6 3 8

    iterator = iter(iterable)
    for x in iterator:
        if not predicate(x):
            yield x
            break

    for x in iterator:
        yield x
```

Note this does not produce *any* output until the predicate first becomes false, so this itertool may have a lengthy start-up time.

itertools.**filterfalse**(*predicate*, *iterable*)

Make an iterator that filters elements from the *iterable* returning only those for which the *predicate* returns a false value. If *predicate* is `None`, returns the items that are false. Roughly equivalent to:

```
def filterfalse(predicate, iterable):
    # filterfalse(lambda x: x<5, [1,4,6,3,8]) → 6 8
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate(x):
            yield x
```

itertools.**groupby**(*iterable*, *key=None*)

Make an iterator that returns consecutive keys and groups from the *iterable*. The *key* is a function computing a key value for each element. If not specified or is `None`, *key* defaults to an identity function and returns the element unchanged. Generally, the iterable needs to already be sorted on the same key function.

The operation of *groupby()* is similar to the `uniq` filter in Unix. It generates a break or new group every time the value of the key function changes (which is why it is usually necessary to have sorted the data using the same key function). That behavior differs from SQL's GROUP BY which aggregates common elements regardless of their input order.

The returned group is itself an iterator that shares the underlying iterable with *groupby()*. Because the source is shared, when the *groupby()* object is advanced, the previous group is no longer visible. So, if that data is needed later, it should be stored as a list:

```
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))      # Store group iterator as a list
    uniquekeys.append(k)
```

groupby() is roughly equivalent to:

```
def groupby(iterable, key=None):
    # [k for k, g in groupby('AAAABBBCCDAABBB')] → A B C D A B
    # [list(g) for k, g in groupby('AAAABBBCCD')] → AAAA BBB CC D

    keyfunc = (lambda x: x) if key is None else key
    iterator = iter(iterable)
    exhausted = False

    def _grouper(target_key):
        nonlocal curr_value, curr_key, exhausted
        yield curr_value
        for curr_value in iterator:
            curr_key = keyfunc(curr_value)
            if curr_key != target_key:
                return
            yield curr_value
        exhausted = True

    try:
        curr_value = next(iterator)
```

(continues on next page)

(continued from previous page)

```
    except StopIteration:
        return
    curr_key = keyfunc(curr_value)

    while not exhausted:
        target_key = curr_key
        curr_group = _grouper(target_key)
        yield curr_key, curr_group
        if curr_key == target_key:
            for _ in curr_group:
                pass
```

itertools.**islice**(*iterable*, *stop*)

itertools.**islice**(*iterable*, *start*, *stop*[, *step*])

Make an iterator that returns selected elements from the iterable. Works like sequence slicing but does not support negative values for *start*, *stop*, or *step*.

If *start* is zero or `None`, iteration starts at zero. Otherwise, elements from the iterable are skipped until *start* is reached.

If *stop* is `None`, iteration continues until the iterator is exhausted, if at all. Otherwise, it stops at the specified position.

If *step* is `None`, the step defaults to one. Elements are returned consecutively unless *step* is set higher than one which results in items being skipped.

Roughly equivalent to:

```
def islice(iterable, *args):
    # islice('ABCDEFG', 2) → A B
    # islice('ABCDEFG', 2, 4) → C D
    # islice('ABCDEFG', 2, None) → C D E F G
    # islice('ABCDEFG', 0, None, 2) → A C E G

    s = slice(*args)
    start = 0 if s.start is None else s.start
    stop = s.stop
    step = 1 if s.step is None else s.step
    if start < 0 or (stop is not None and stop < 0) or step <= 0:
        raise ValueError

    indices = count() if stop is None else range(max(start, stop))
    next_i = start
    for i, element in zip(indices, iterable):
        if i == next_i:
            yield element
            next_i += step
```

itertools.**pairwise**(*iterable*)

Return successive overlapping pairs taken from the input *iterable*.

The number of 2-tuples in the output iterator will be one fewer than the number of inputs. It will be empty if the input iterable has fewer than two values.

Roughly equivalent to:

```
def pairwise(iterable):
    # pairwise('ABCDEFG') → AB BC CD DE EF FG
    iterator = iter(iterable)
    a = next(iterator, None)
    for b in iterator:
        yield a, b
        a = b
```

Added in version 3.10.

itertools.**permutations**(*iterable*, *r=None*)

Return successive *r* length permutations of elements from the *iterable*.

If *r* is not specified or is `None`, then *r* defaults to the length of the *iterable* and all possible full-length permutations are generated.

The output is a subsequence of *`product()`* where entries with repeated elements have been filtered out. The length of the output is given by *`math.perm()`* which computes `n! / (n - r)!` when `0 ≤ r ≤ n` or zero when `r > n`.

The permutation tuples are emitted in lexicographic order according to the order of the input *iterable*. If the input *iterable* is sorted, the output tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. If the input elements are unique, there will be no repeated values within a permutation.

Roughly equivalent to:

```
def permutations(iterable, r=None):
    # permutations('ABCD', 2) → AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) → 012 021 102 120 201 210

    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return

    indices = list(range(n))
    cycles = list(range(n, n-r, -1))
    yield tuple(pool[i] for i in indices[:r])

    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

itertools.**product**(**iterables*, *repeat=1*)

Cartesian product of input iterables.

Roughly equivalent to nested for-loops in a generator expression. For example, `product(A, B)` returns the same as `((x,y) for x in A for y in B)`.

The nested loops cycle like an odometer with the rightmost element advancing on every iteration. This pattern creates a lexicographic ordering so that if the input's iterables are sorted, the product tuples are emitted in sorted order.

To compute the product of an iterable with itself, specify the number of repetitions with the optional *repeat* keyword argument. For example, `product(A, repeat=4)` means the same as `product(A, A, A, A)`.

This function is roughly equivalent to the following code, except that the actual implementation does not build up intermediate results in memory:

```
def product(*iterables, repeat=1):
    # product('ABCD', 'xy') → Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) → 000 001 010 011 100 101 110 111

    pools = [tuple(pool) for pool in iterables] * repeat

    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]

    for prod in result:
        yield tuple(prod)
```

Before *product()* runs, it completely consumes the input iterables, keeping pools of values in memory to generate the products. Accordingly, it is only useful with finite inputs.

`itertools.`**`repeat`**(*object*[, *times*])

Make an iterator that returns *object* over and over again. Runs indefinitely unless the *times* argument is specified.

Roughly equivalent to:

```
def repeat(object, times=None):
    # repeat(10, 3) → 10 10 10
    if times is None:
        while True:
            yield object
    else:
        for i in range(times):
            yield object
```

A common use for *repeat* is to supply a stream of constant values to *map* or *zip*:

```
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`itertools.`**`starmap`**(*function*, *iterable*)

Make an iterator that computes the *function* using arguments obtained from the *iterable*. Used instead of *map()* when argument parameters have already been "pre-zipped" into tuples.

The difference between *map()* and *starmap()* parallels the distinction between `function(a,b)` and `function(*c)`. Roughly equivalent to:

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) → 32 9 1000
    for args in iterable:
        yield function(*args)
```

itertools.**takewhile**(*predicate*, *iterable*)

Make an iterator that returns elements from the *iterable* as long as the *predicate* is true. Roughly equivalent to:

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,3,8]) → 1 4
    for x in iterable:
        if not predicate(x):
            break
        yield x
```

Note, the element that first fails the predicate condition is consumed from the input iterator and there is no way to access it. This could be an issue if an application wants to further consume the input iterator after *takewhile* has been run to exhaustion. To work around this problem, consider using more-iterools before_and_after() instead.

itertools.**tee**(*iterable*, *n=2*)

Return *n* independent iterators from a single iterable.

Roughly equivalent to:

```
def tee(iterable, n=2):
    iterator = iter(iterable)
    shared_link = [None, None]
    return tuple(_tee(iterator, shared_link) for _ in range(n))

def _tee(iterator, link):
    try:
        while True:
            if link[1] is None:
                link[0] = next(iterator)
                link[1] = [None, None]
            value, link = link
            yield value
    except StopIteration:
        return
```

Once a *tee()* has been created, the original *iterable* should not be used anywhere else; otherwise, the *iterable* could get advanced without the tee objects being informed.

tee iterators are not threadsafe. A *RuntimeError* may be raised when simultaneously using iterators returned by the same *tee()* call, even if the original *iterable* is threadsafe.

This itertool may require significant auxiliary storage (depending on how much temporary data needs to be stored). In general, if one iterator uses most or all of the data before another iterator starts, it is faster to use *list()* instead of *tee()*.

itertools.**zip_longest**(**iterables*, *fillvalue=None*)

Make an iterator that aggregates elements from each of the *iterables*.

If the iterables are of uneven length, missing values are filled-in with *fillvalue*. If not specified, *fillvalue* defaults to None.

Iteration continues until the longest iterable is exhausted.

Roughly equivalent to:

```
def zip_longest(*iterables, fillvalue=None):
    # zip_longest('ABCD', 'xy', fillvalue='-') → Ax By C- D-

    iterators = list(map(iter, iterables))
```

(continues on next page)

(continued from previous page)

```
    num_active = len(iterators)
    if not num_active:
        return

    while True:
        values = []
        for i, iterator in enumerate(iterators):
            try:
                value = next(iterator)
            except StopIteration:
                num_active -= 1
                if not num_active:
                    return
                iterators[i] = repeat(fillvalue)
                value = fillvalue
            values.append(value)
        yield tuple(values)
```

If one of the iterables is potentially infinite, then the *zip_longest()* function should be wrapped with something that limits the number of calls (for example *islice()* or *takewhile()*).

10.1.2 Itertools Recipes

This section shows recipes for creating an extended toolset using the existing itertools as building blocks.

The primary purpose of the itertools recipes is educational. The recipes show various ways of thinking about individual tools — for example, that `chain.from_iterable` is related to the concept of flattening. The recipes also give ideas about ways that the tools can be combined — for example, how `starmap()` and `repeat()` can work together. The recipes also show patterns for using itertools with the *operator* and *collections* modules as well as with the built-in itertools such as `map()`, `filter()`, `reversed()`, and `enumerate()`.

A secondary purpose of the recipes is to serve as an incubator. The `accumulate()`, `compress()`, and `pairwise()` itertools started out as recipes. Currently, the `sliding_window()`, `iter_index()`, and `sieve()` recipes are being tested to see whether they prove their worth.

Substantially all of these recipes and many, many others can be installed from the more-itertools project found on the Python Package Index:

```
python -m pip install more-itertools
```

Many of the recipes offer the same high performance as the underlying toolset. Superior memory performance is kept by processing elements one at a time rather than bringing the whole iterable into memory all at once. Code volume is kept small by linking the tools together in a functional style. High speed is retained by preferring "vectorized" building blocks over the use of for-loops and *generators* which incur interpreter overhead.

```
import collections
import contextlib
import functools
import math
import operator
import random

def take(n, iterable):
    "Return first n items of the iterable as a list."
    return list(islice(iterable, n))
```

(continues on next page)

(continued from previous page)

```
def prepend(value, iterable):
    "Prepend a single value in front of an iterable."
    # prepend(1, [2, 3, 4]) → 1 2 3 4
    return chain([value], iterable)

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return map(function, count(start))

def repeatfunc(func, times=None, *args):
    "Repeat calls to func with specified arguments."
    if times is None:
        return starmap(func, repeat(args))
    return starmap(func, repeat(args, times))

def flatten(list_of_lists):
    "Flatten one level of nesting."
    return chain.from_iterable(list_of_lists)

def ncycles(iterable, n):
    "Returns the sequence elements n times."
    return chain.from_iterable(repeat(tuple(iterable), n))

def tail(n, iterable):
    "Return an iterator over the last n items."
    # tail(3, 'ABCDEFG') → E F G
    return iter(collections.deque(iterable, maxlen=n))

def consume(iterator, n=None):
    "Advance the iterator n-steps ahead. If n is None, consume entirely."
    # Use functions that consume iterators at C speed.
    if n is None:
        collections.deque(iterator, maxlen=0)
    else:
        next(islice(iterator, n, n), None)

def nth(iterable, n, default=None):
    "Returns the nth item or a default value."
    return next(islice(iterable, n, None), default)

def quantify(iterable, predicate=bool):
    "Given a predicate that returns True or False, count the True results."
    return sum(map(predicate, iterable))

def first_true(iterable, default=False, predicate=None):
    "Returns the first true value or the *default* if there is no true value."
    # first_true([a,b,c], x) → a or b or c or x
    # first_true([a,b], x, f) → a if f(a) else b if f(b) else x
    return next(filter(predicate, iterable), default)

def all_equal(iterable, key=None):
    "Returns True if all the elements are equal to each other."
    # all_equal('4□□□□', key=int) → True
    return len(take(2, groupby(iterable, key))) <= 1

def unique_justseen(iterable, key=None):
```

(continues on next page)

(continued from previous page)

```
    "Yield unique elements, preserving order. Remember only the element just seen."
    # unique_justseen('AAAABBBCCDAABBB') → A B C D A B
    # unique_justseen('ABBcCAD', str.casefold) → A B c A D
    if key is None:
        return map(operator.itemgetter(0), groupby(iterable))
    return map(next, map(operator.itemgetter(1), groupby(iterable, key)))

def unique_everseen(iterable, key=None):
    "Yield unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') → A B C D
    # unique_everseen('ABBcCAD', str.casefold) → A B c D
    seen = set()
    if key is None:
        for element in filterfalse(seen.__contains__, iterable):
            seen.add(element)
            yield element
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen.add(k)
                yield element

def unique(iterable, key=None, reverse=False):
   "Yield unique elements in sorted order. Supports unhashable inputs."
   # unique([[1, 2], [3, 4], [1, 2]]) → [1, 2] [3, 4]
   return unique_justseen(sorted(iterable, key=key, reverse=reverse), key=key)

def sliding_window(iterable, n):
    "Collect data into overlapping fixed-length chunks or blocks."
    # sliding_window('ABCDEFG', 4) → ABCD BCDE CDEF DEFG
    iterator = iter(iterable)
    window = collections.deque(islice(iterator, n - 1), maxlen=n)
    for x in iterator:
        window.append(x)
        yield tuple(window)

def grouper(iterable, n, *, incomplete='fill', fillvalue=None):
    "Collect data into non-overlapping fixed-length chunks or blocks."
    # grouper('ABCDEFG', 3, fillvalue='x') → ABC DEF Gxx
    # grouper('ABCDEFG', 3, incomplete='strict') → ABC DEF ValueError
    # grouper('ABCDEFG', 3, incomplete='ignore') → ABC DEF
    iterators = [iter(iterable)] * n
    match incomplete:
        case 'fill':
            return zip_longest(*iterators, fillvalue=fillvalue)
        case 'strict':
            return zip(*iterators, strict=True)
        case 'ignore':
            return zip(*iterators)
        case _:
            raise ValueError('Expected fill, strict, or ignore')

def roundrobin(*iterables):
    "Visit input iterables in a cycle until each is exhausted."
    # roundrobin('ABC', 'D', 'EF') → A D E B F C
    # Algorithm credited to George Sakkis
```

(continues on next page)

(continued from previous page)

```
    iterators = map(iter, iterables)
    for num_active in range(len(iterables), 0, -1):
        iterators = cycle(islice(iterators, num_active))
        yield from map(next, iterators)

def partition(predicate, iterable):
    """Partition entries into false entries and true entries.

    If *predicate* is slow, consider wrapping it with functools.lru_cache().
    """
    # partition(is_odd, range(10)) → 0 2 4 6 8   and  1 3 5 7 9
    t1, t2 = tee(iterable)
    return filterfalse(predicate, t1), filter(predicate, t2)

def subslices(seq):
    "Return all contiguous non-empty subslices of a sequence."
    # subslices('ABCD') → A AB ABC ABCD B BC BCD C CD D
    slices = starmap(slice, combinations(range(len(seq) + 1), 2))
    return map(operator.getitem, repeat(seq), slices)

def iter_index(iterable, value, start=0, stop=None):
    "Return indices where a value occurs in a sequence or iterable."
    # iter_index('AABCADEAF', 'A') → 0 1 4 7
    seq_index = getattr(iterable, 'index', None)
    if seq_index is None:
        iterator = islice(iterable, start, stop)
        for i, element in enumerate(iterator, start):
            if element is value or element == value:
                yield i
    else:
        stop = len(iterable) if stop is None else stop
        i = start
        with contextlib.suppress(ValueError):
            while True:
                yield (i := seq_index(value, i, stop))
                i += 1

def iter_except(func, exception, first=None):
    "Convert a call-until-exception interface to an iterator interface."
    # iter_except(d.popitem, KeyError) → non-blocking dictionary iterator
    with contextlib.suppress(exception):
        if first is not None:
            yield first()
        while True:
            yield func()
```

The following recipes have a more mathematical flavor:

```
def powerset(iterable):
    "powerset([1,2,3]) → () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

def sum_of_squares(iterable):
    "Add up the squares of the input values."
    # sum_of_squares([10, 20, 30]) → 1400
    return math.sumprod(*tee(iterable))
```

(continues on next page)

(continued from previous page)

```
def reshape(matrix, cols):
    "Reshape a 2-D matrix to have a given number of columns."
    # reshape([(0, 1), (2, 3), (4, 5)], 3) → (0, 1, 2), (3, 4, 5)
    return batched(chain.from_iterable(matrix), cols, strict=True)

def transpose(matrix):
    "Swap the rows and columns of a 2-D matrix."
    # transpose([(1, 2, 3), (11, 22, 33)]) → (1, 11) (2, 22) (3, 33)
    return zip(*matrix, strict=True)

def matmul(m1, m2):
    "Multiply two matrices."
    # matmul([(7, 5), (3, 5)], [(2, 5), (7, 9)]) → (49, 80), (41, 60)
    n = len(m2[0])
    return batched(starmap(math.sumprod, product(m1, transpose(m2))), n)

def convolve(signal, kernel):
    """Discrete linear convolution of two iterables.
    Equivalent to polynomial multiplication.

    Convolutions are mathematically commutative; however, the inputs are
    evaluated differently.  The signal is consumed lazily and can be
    infinite. The kernel is fully consumed before the calculations begin.

    Article:  https://betterexplained.com/articles/intuitive-convolution/
    Video:    https://www.youtube.com/watch?v=KuXjwB4LzSA
    """
    # convolve([1, -1, -20], [1, -3]) → 1 -4 -17 60
    # convolve(data, [0.25, 0.25, 0.25, 0.25]) → Moving average (blur)
    # convolve(data, [1/2, 0, -1/2]) → 1st derivative estimate
    # convolve(data, [1, -2, 1]) → 2nd derivative estimate
    kernel = tuple(kernel)[::-1]
    n = len(kernel)
    padded_signal = chain(repeat(0, n-1), signal, repeat(0, n-1))
    windowed_signal = sliding_window(padded_signal, n)
    return map(math.sumprod, repeat(kernel), windowed_signal)

def polynomial_from_roots(roots):
    """Compute a polynomial's coefficients from its roots.

       (x - 5) (x + 4) (x - 3)  expands to:   x³ -4x² -17x + 60
    """
    # polynomial_from_roots([5, -4, 3]) → [1, -4, -17, 60]
    factors = zip(repeat(1), map(operator.neg, roots))
    return list(functools.reduce(convolve, factors, [1]))

def polynomial_eval(coefficients, x):
    """Evaluate a polynomial at a specific value.

    Computes with better numeric stability than Horner's method.
    """
    # Evaluate x³ -4x² -17x + 60 at x = 5
    # polynomial_eval([1, -4, -17, 60], x=5) → 0
    n = len(coefficients)
    if not n:
        return type(x)(0)
```

(continues on next page)

(continued from previous page)

```
    powers = map(pow, repeat(x), reversed(range(n)))
    return math.sumprod(coefficients, powers)

def polynomial_derivative(coefficients):
    """Compute the first derivative of a polynomial.

       f(x)  =  x³ -4x² -17x + 60
       f'(x) = 3x² -8x  -17
    """
    # polynomial_derivative([1, -4, -17, 60]) → [3, -8, -17]
    n = len(coefficients)
    powers = reversed(range(1, n))
    return list(map(operator.mul, coefficients, powers))

def sieve(n):
    "Primes less than n."
    # sieve(30) → 2 3 5 7 11 13 17 19 23 29
    if n > 2:
        yield 2
    data = bytearray((0, 1)) * (n // 2)
    for p in iter_index(data, 1, start=3, stop=math.isqrt(n) + 1):
        data[p*p : n : p+p] = bytes(len(range(p*p, n, p+p)))
    yield from iter_index(data, 1, start=3)

def factor(n):
    "Prime factors of n."
    # factor(99) → 3 3 11
    # factor(1_000_000_000_000_007) → 47 59 360620266859
    # factor(1_000_000_000_000_403) → 1000000000000403
    for prime in sieve(math.isqrt(n) + 1):
        while not n % prime:
            yield prime
            n //= prime
            if n == 1:
                return
    if n > 1:
        yield n

def totient(n):
    "Count of natural numbers up to n that are coprime to n."
    # https://mathworld.wolfram.com/TotientFunction.html
    # totient(12) → 4 because len([1, 5, 7, 11]) == 4
    for prime in set(factor(n)):
        n -= n // prime
    return n
```

10.2 `functools` — Higher-order functions and operations on callable objects

Source code: Lib/functools.py

The *functools* module is for higher-order functions: functions that act on or return other functions. In general, any callable object can be treated as a function for the purposes of this module.

The *functools* module defines the following functions:

`@functools.cache(user_function)`

Simple lightweight unbounded function cache. Sometimes called "memoize".

Returns the same as `lru_cache(maxsize=None)`, creating a thin wrapper around a dictionary lookup for the function arguments. Because it never needs to evict old values, this is smaller and faster than *lru_cache()* with a size limit.

For example:

```
@cache
def factorial(n):
    return n * factorial(n-1) if n else 1

>>> factorial(10)      # no previously cached result, makes 11 recursive calls
3628800
>>> factorial(5)       # just looks up cached value result
120
>>> factorial(12)      # makes two new recursive calls, the other 10 are cached
479001600
```

The cache is threadsafe so that the wrapped function can be used in multiple threads. This means that the underlying data structure will remain coherent during concurrent updates.

It is possible for the wrapped function to be called more than once if another thread makes an additional call before the initial call has been completed and cached.

Added in version 3.9.

`@functools.cached_property(func)`

Transform a method of a class into a property whose value is computed once and then cached as a normal attribute for the life of the instance. Similar to *property()*, with the addition of caching. Useful for expensive computed properties of instances that are otherwise effectively immutable.

Example:

```
class DataSet:

    def __init__(self, sequence_of_numbers):
        self._data = tuple(sequence_of_numbers)

    @cached_property
    def stdev(self):
        return statistics.stdev(self._data)
```

The mechanics of *cached_property()* are somewhat different from *property()*. A regular property blocks attribute writes unless a setter is defined. In contrast, a *cached_property* allows writes.

The *cached_property* decorator only runs on lookups and only when an attribute of the same name doesn't exist. When it does run, the *cached_property* writes to the attribute with the same name. Subsequent attribute reads and writes take precedence over the *cached_property* method and it works like a normal attribute.

The cached value can be cleared by deleting the attribute. This allows the *cached_property* method to run again.

The *cached_property* does not prevent a possible race condition in multi-threaded usage. The getter function could run more than once on the same instance, with the latest run setting the cached value. If the cached property is idempotent or otherwise not harmful to run more than once on an instance, this is fine. If synchronization is needed, implement the necessary locking inside the decorated getter function or around the cached property access.

Note, this decorator interferes with the operation of **PEP 412** key-sharing dictionaries. This means that instance dictionaries can take more space than usual.

Also, this decorator requires that the `__dict__` attribute on each instance be a mutable mapping. This means it will not work with some types, such as metaclasses (since the `__dict__` attributes on type instances are read-only proxies for the class namespace), and those that specify `__slots__` without including `__dict__` as one of the defined slots (as such classes don't provide a `__dict__` attribute at all).

If a mutable mapping is not available or if space-efficient key sharing is desired, an effect similar to *cached_property()* can also be achieved by stacking *property()* on top of *lru_cache()*. See faq-cache-method-calls for more details on how this differs from *cached_property()*.

Added in version 3.8.

Changed in version 3.12: Prior to Python 3.12, `cached_property` included an undocumented lock to ensure that in multi-threaded usage the getter function was guaranteed to run only once per instance. However, the lock was per-property, not per-instance, which could result in unacceptably high lock contention. In Python 3.12+ this locking is removed.

`functools.`**`cmp_to_key`**(*func*)

Transform an old-style comparison function to a *key function*. Used with tools that accept key functions (such as *sorted()*, *min()*, *max()*, *heapq.nlargest()*, *heapq.nsmallest()*, *itertools.groupby()*). This function is primarily used as a transition tool for programs being converted from Python 2 which supported the use of comparison functions.

A comparison function is any callable that accepts two arguments, compares them, and returns a negative number for less-than, zero for equality, or a positive number for greater-than. A key function is a callable that accepts one argument and returns another value to be used as the sort key.

Example:

```
sorted(iterable, key=cmp_to_key(locale.strcoll))  # locale-aware sort order
```

For sorting examples and a brief sorting tutorial, see sortinghowto.

Added in version 3.2.

`@functools.`**`lru_cache`**(*user_function*)

`@functools.`**`lru_cache`**(*maxsize=128*, *typed=False*)

Decorator to wrap a function with a memoizing callable that saves up to the *maxsize* most recent calls. It can save time when an expensive or I/O bound function is periodically called with the same arguments.

The cache is threadsafe so that the wrapped function can be used in multiple threads. This means that the underlying data structure will remain coherent during concurrent updates.

It is possible for the wrapped function to be called more than once if another thread makes an additional call before the initial call has been completed and cached.

Since a dictionary is used to cache results, the positional and keyword arguments to the function must be *hashable*.

Distinct argument patterns may be considered to be distinct calls with separate cache entries. For example, `f(a=1, b=2)` and `f(b=2, a=1)` differ in their keyword argument order and may have two separate cache entries.

If *user_function* is specified, it must be a callable. This allows the *lru_cache* decorator to be applied directly to a user function, leaving the *maxsize* at its default value of 128:

```
@lru_cache
def count_vowels(sentence):
    return sum(sentence.count(vowel) for vowel in 'AEIOUaeiou')
```

If *maxsize* is set to `None`, the LRU feature is disabled and the cache can grow without bound.

If *typed* is set to true, function arguments of different types will be cached separately. If *typed* is false, the implementation will usually regard them as equivalent calls and only cache a single result. (Some types such as *str* and *int* may be cached separately even when *typed* is false.)

Note, type specificity applies only to the function's immediate arguments rather than their contents. The scalar arguments, `Decimal(42)` and `Fraction(42)` are be treated as distinct calls with distinct results. In contrast, the tuple arguments `('answer', Decimal(42))` and `('answer', Fraction(42))` are treated as equivalent.

The wrapped function is instrumented with a `cache_parameters()` function that returns a new *dict* showing the values for *maxsize* and *typed*. This is for information purposes only. Mutating the values has no effect.

To help measure the effectiveness of the cache and tune the *maxsize* parameter, the wrapped function is instrumented with a `cache_info()` function that returns a *named tuple* showing *hits*, *misses*, *maxsize* and *currsize*.

The decorator also provides a `cache_clear()` function for clearing or invalidating the cache.

The original underlying function is accessible through the `__wrapped__` attribute. This is useful for introspection, for bypassing the cache, or for rewrapping the function with a different cache.

The cache keeps references to the arguments and return values until they age out of the cache or until the cache is cleared.

If a method is cached, the `self` instance argument is included in the cache. See faq-cache-method-calls

An LRU (least recently used) cache works best when the most recent calls are the best predictors of upcoming calls (for example, the most popular articles on a news server tend to change each day). The cache's size limit assures that the cache does not grow without bound on long-running processes such as web servers.

In general, the LRU cache should only be used when you want to reuse previously computed values. Accordingly, it doesn't make sense to cache functions with side-effects, functions that need to create distinct mutable objects on each call (such as generators and async functions), or impure functions such as time() or random().

Example of an LRU cache for static web content:

```
@lru_cache(maxsize=32)
def get_pep(num):
    'Retrieve text of a Python Enhancement Proposal'
    resource = f'https://peps.python.org/pep-{num:04d}'
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...     pep = get_pep(n)
...     print(n, len(pep))
```

(continues on next page)

(continued from previous page)

```
>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)
```

Example of efficiently computing Fibonacci numbers using a cache to implement a dynamic programming technique:

```
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)
```

Added in version 3.2.

Changed in version 3.3: Added the *typed* option.

Changed in version 3.8: Added the *user_function* option.

Changed in version 3.9: Added the function `cache_parameters()`

`@functools.total_ordering`

Given a class defining one or more rich comparison ordering methods, this class decorator supplies the rest. This simplifies the effort involved in specifying all of the possible rich comparison operations:

The class must define one of `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()`. In addition, the class should supply an `__eq__()` method.

For example:

```
@total_ordering
class Student:
    def _is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname"))
    def __eq__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
```

Note

While this decorator makes it easy to create well behaved totally ordered types, it *does* come at the cost of slower execution and more complex stack traces for the derived comparison methods. If performance benchmarking indicates this is a bottleneck for a given application, implementing all six rich comparison methods instead is likely to provide an easy speed boost.

> **Note**
>
> This decorator makes no attempt to override methods that have been declared in the class *or its superclasses*. Meaning that if a superclass defines a comparison operator, *total_ordering* will not implement it again, even if the original method is abstract.

Added in version 3.2.

Changed in version 3.4: Returning `NotImplemented` from the underlying comparison function for unrecognised types is now supported.

`functools.`**`partial`**(*func*, */*, **args*, ***keywords*)

Return a new *partial object* which when called will behave like *func* called with the positional arguments *args* and keyword arguments *keywords*. If more arguments are supplied to the call, they are appended to *args*. If additional keyword arguments are supplied, they extend and override *keywords*. Roughly equivalent to:

```
def partial(func, /, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = {**keywords, **fkeywords}
        return func(*args, *fargs, **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

The *partial()* is used for partial function application which "freezes" some portion of a function's arguments and/or keywords resulting in a new object with a simplified signature. For example, *partial()* can be used to create a callable that behaves like the *int()* function where the *base* argument defaults to two:

```
>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

class `functools.`**`partialmethod`**(*func*, */*, **args*, ***keywords*)

Return a new *partialmethod* descriptor which behaves like *partial* except that it is designed to be used as a method definition rather than being directly callable.

func must be a *descriptor* or a callable (objects which are both, like normal functions, are handled as descriptors).

When *func* is a descriptor (such as a normal Python function, *classmethod()*, *staticmethod()*, `abstractmethod()` or another instance of *partialmethod*), calls to `__get__` are delegated to the underlying descriptor, and an appropriate *partial object* returned as the result.

When *func* is a non-descriptor callable, an appropriate bound method is created dynamically. This behaves like a normal Python function when used as a method: the *self* argument will be inserted as the first positional argument, even before the *args* and *keywords* supplied to the *partialmethod* constructor.

Example:

```
>>> class Cell:
...     def __init__(self):
...         self._alive = False
...     @property
...     def alive(self):
...         return self._alive
```

(continues on next page)

(continued from previous page)

```
...     def set_state(self, state):
...         self._alive = bool(state)
...     set_alive = partialmethod(set_state, True)
...     set_dead = partialmethod(set_state, False)
...
>>> c = Cell()
>>> c.alive
False
>>> c.set_alive()
>>> c.alive
True
```

Added in version 3.4.

`functools.`**`reduce`**(*function*, *iterable*, [*initial*,]/)

Apply *function* of two arguments cumulatively to the items of *iterable*, from left to right, so as to reduce the iterable to a single value. For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates `((((1+2)+3)+4)+5)`. The left argument, *x*, is the accumulated value and the right argument, *y*, is the update value from the *iterable*. If the optional *initial* is present, it is placed before the items of the iterable in the calculation, and serves as a default when the iterable is empty. If *initial* is not given and *iterable* contains only one item, the first item is returned.

Roughly equivalent to:

```
initial_missing = object()

def reduce(function, iterable, initial=initial_missing, /):
    it = iter(iterable)
    if initial is initial_missing:
        value = next(it)
    else:
        value = initial
    for element in it:
        value = function(value, element)
    return value
```

See *`itertools.accumulate()`* for an iterator that yields all intermediate values.

`@functools.`**`singledispatch`**

Transform a function into a *single-dispatch generic function*.

To define a generic function, decorate it with the `@singledispatch` decorator. When defining a function using `@singledispatch`, note that the dispatch happens on the type of the first argument:

```
>>> from functools import singledispatch
>>> @singledispatch
... def fun(arg, verbose=False):
...     if verbose:
...         print("Let me just say,", end=" ")
...     print(arg)
```

To add overloaded implementations to the function, use the `register()` attribute of the generic function, which can be used as a decorator. For functions annotated with types, the decorator will infer the type of the first argument automatically:

```
>>> @fun.register
... def _(arg: int, verbose=False):
```

(continues on next page)

(continued from previous page)

```
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> @fun.register
... def _(arg: list, verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

types.UnionType and *typing.Union* can also be used:

```
>>> @fun.register
... def _(arg: int | float, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> from typing import Union
>>> @fun.register
... def _(arg: Union[list, set], verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
...
```

For code which doesn't use type annotations, the appropriate type argument can be passed explicitly to the decorator itself:

```
>>> @fun.register(complex)
... def _(arg, verbose=False):
...     if verbose:
...         print("Better than complicated.", end=" ")
...     print(arg.real, arg.imag)
...
```

To enable registering *lambdas* and pre-existing functions, the `register()` attribute can also be used in a functional form:

```
>>> def nothing(arg, verbose=False):
...     print("Nothing.")
...
>>> fun.register(type(None), nothing)
```

The `register()` attribute returns the undecorated function. This enables decorator stacking, *pickling*, and the creation of unit tests for each variant independently:

```
>>> @fun.register(float)
... @fun.register(Decimal)
... def fun_num(arg, verbose=False):
...     if verbose:
...         print("Half of your number:", end=" ")
...     print(arg / 2)
...
```

(continues on next page)

(continued from previous page)

```
>>> fun_num is fun
False
```

When called, the generic function dispatches on the type of the first argument:

```
>>> fun("Hello, world.")
Hello, world.
>>> fun("test.", verbose=True)
Let me just say, test.
>>> fun(42, verbose=True)
Strength in numbers, eh? 42
>>> fun(['spam', 'spam', 'eggs', 'spam'], verbose=True)
Enumerate this:
0 spam
1 spam
2 eggs
3 spam
>>> fun(None)
Nothing.
>>> fun(1.23)
0.615
```

Where there is no registered implementation for a specific type, its method resolution order is used to find a more generic implementation. The original function decorated with `@singledispatch` is registered for the base *`object`* type, which means it is used if no better implementation is found.

If an implementation is registered to an *abstract base class*, virtual subclasses of the base class will be dispatched to that implementation:

```
>>> from collections.abc import Mapping
>>> @fun.register
... def _(arg: Mapping, verbose=False):
...     if verbose:
...         print("Keys & Values")
...     for key, value in arg.items():
...         print(key, "=>", value)
...
>>> fun({"a": "b"})
a => b
```

To check which implementation the generic function will choose for a given type, use the `dispatch()` attribute:

```
>>> fun.dispatch(float)
<function fun_num at 0x1035a2840>
>>> fun.dispatch(dict)    # note: default implementation
<function fun at 0x103fe0000>
```

To access all registered implementations, use the read-only `registry` attribute:

```
>>> fun.registry.keys()
dict_keys([<class 'NoneType'>, <class 'int'>, <class 'object'>,
           <class 'decimal.Decimal'>, <class 'list'>,
           <class 'float'>])
>>> fun.registry[float]
<function fun_num at 0x1035a2840>
>>> fun.registry[object]
<function fun at 0x103fe0000>
```

Added in version 3.4.

Changed in version 3.7: The `register()` attribute now supports using type annotations.

Changed in version 3.11: The `register()` attribute now supports *types.UnionType* and *typing.Union* as type annotations.

class functools.**singledispatchmethod**(*func*)

Transform a method into a *single-dispatch generic function*.

To define a generic method, decorate it with the `@singledispatchmethod` decorator. When defining a function using `@singledispatchmethod`, note that the dispatch happens on the type of the first non-*self* or non-*cls* argument:

```
class Negator:
    @singledispatchmethod
    def neg(self, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    def _(self, arg: int):
        return -arg

    @neg.register
    def _(self, arg: bool):
        return not arg
```

`@singledispatchmethod` supports nesting with other decorators such as *@classmethod*. Note that to allow for `dispatcher.register`, `singledispatchmethod` must be the *outer most* decorator. Here is the `Negator` class with the `neg` methods bound to the class, rather than an instance of the class:

```
class Negator:
    @singledispatchmethod
    @classmethod
    def neg(cls, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    @classmethod
    def _(cls, arg: int):
        return -arg

    @neg.register
    @classmethod
    def _(cls, arg: bool):
        return not arg
```

The same pattern can be used for other similar decorators: *@staticmethod*, *@abstractmethod*, and others.

Added in version 3.8.

functools.**update_wrapper**(*wrapper*, *wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

Update a *wrapper* function to look like the *wrapped* function. The optional arguments are tuples to specify which attributes of the original function are assigned directly to the matching attributes on the wrapper function and which attributes of the wrapper function are updated with the corresponding attributes from the original function. The default values for these arguments are the module level constants `WRAPPER_ASSIGNMENTS` (which assigns to the wrapper function's `__module__`, `__name__`, `__qualname__`, `__annotations__`,

__type_params__, and __doc__, the documentation string) and WRAPPER_UPDATES (which updates the wrapper function's __dict__, i.e. the instance dictionary).

To allow access to the original function for introspection and other purposes (e.g. bypassing a caching decorator such as *lru_cache()*), this function automatically adds a __wrapped__ attribute to the wrapper that refers to the function being wrapped.

The main intended use for this function is in *decorator* functions which wrap the decorated function and return the wrapper. If the wrapper function is not updated, the metadata of the returned function will reflect the wrapper definition rather than the original function definition, which is typically less than helpful.

update_wrapper() may be used with callables other than functions. Any attributes named in *assigned* or *updated* that are missing from the object being wrapped are ignored (i.e. this function will not attempt to set them on the wrapper function). *AttributeError* is still raised if the wrapper function itself is missing any attributes named in *updated*.

Changed in version 3.2: The __wrapped__ attribute is now automatically added. The __annotations__ attribute is now copied by default. Missing attributes no longer trigger an *AttributeError*.

Changed in version 3.4: The __wrapped__ attribute now always refers to the wrapped function, even if that function defined a __wrapped__ attribute. (see bpo-17482)

Changed in version 3.12: The __type_params__ attribute is now copied by default.

@functools.**wraps**(*wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

This is a convenience function for invoking *update_wrapper()* as a function decorator when defining a wrapper function. It is equivalent to partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated). For example:

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print('Calling decorated function')
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print('Called example function')
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

Without the use of this decorator factory, the name of the example function would have been 'wrapper', and the docstring of the original example() would have been lost.

10.2.1 `partial` Objects

partial objects are callable objects created by *partial()*. They have three read-only attributes:

partial.**func**

A callable object or function. Calls to the *partial* object will be forwarded to *func* with new arguments and keywords.

partial.**args**

The leftmost positional arguments that will be prepended to the positional arguments provided to a *partial* object call.

partial.**keywords**

The keyword arguments that will be supplied when the *partial* object is called.

partial objects are like `function` objects in that they are callable, weak referenceable, and can have attributes. There are some important differences. For instance, the *__name__* and `__doc__` attributes are not created automatically. Also, *partial* objects defined in classes behave like static methods and do not transform into bound methods during instance attribute look-up.

10.3 `operator` — Standard operators as functions

Source code: Lib/operator.py

The *operator* module exports a set of efficient functions corresponding to the intrinsic operators of Python. For example, `operator.add(x, y)` is equivalent to the expression `x+y`. Many function names are those used for special methods, without the double underscores. For backward compatibility, many of these have a variant with the double underscores kept. The variants without the double underscores are preferred for clarity.

The functions fall into categories that perform object comparisons, logical operations, mathematical operations and sequence operations.

The object comparison functions are useful for all objects, and are named after the rich comparison operators they support:

operator.**lt**(*a*, *b*)
operator.**le**(*a*, *b*)
operator.**eq**(*a*, *b*)
operator.**ne**(*a*, *b*)
operator.**ge**(*a*, *b*)
operator.**gt**(*a*, *b*)
operator.**__lt__**(*a*, *b*)
operator.**__le__**(*a*, *b*)
operator.**__eq__**(*a*, *b*)
operator.**__ne__**(*a*, *b*)
operator.**__ge__**(*a*, *b*)
operator.**__gt__**(*a*, *b*)

Perform "rich comparisons" between *a* and *b*. Specifically, `lt(a, b)` is equivalent to `a < b`, `le(a, b)` is equivalent to `a <= b`, `eq(a, b)` is equivalent to `a == b`, `ne(a, b)` is equivalent to `a != b`, `gt(a, b)` is equivalent to `a > b` and `ge(a, b)` is equivalent to `a >= b`. Note that these functions can return any value, which may or may not be interpretable as a Boolean value. See comparisons for more information about rich comparisons.

The logical operations are also generally applicable to all objects, and support truth tests, identity tests, and boolean operations:

operator.**not_**(*obj*)

operator.**__not__**(*obj*)

Return the outcome of `not` *obj*. (Note that there is no `__not__()` method for object instances; only the interpreter core defines this operation. The result is affected by the `__bool__()` and `__len__()` methods.)

operator.**truth**(*obj*)

Return `True` if *obj* is true, and `False` otherwise. This is equivalent to using the `bool` constructor.

operator.**is_**(*a*, *b*)

Return `a is b`. Tests object identity.

operator.**is_not**(*a*, *b*)

Return `a is not b`. Tests object identity.

The mathematical and bitwise operations are the most numerous:

operator.**abs**(*obj*)

operator.**__abs__**(*obj*)

Return the absolute value of *obj*.

operator.**add**(*a*, *b*)

operator.**__add__**(*a*, *b*)

Return `a + b`, for *a* and *b* numbers.

operator.**and_**(*a*, *b*)

operator.**__and__**(*a*, *b*)

Return the bitwise and of *a* and *b*.

operator.**floordiv**(*a*, *b*)

operator.**__floordiv__**(*a*, *b*)

Return `a // b`.

operator.**index**(*a*)

operator.**__index__**(*a*)

Return *a* converted to an integer. Equivalent to `a.__index__()`.

Changed in version 3.10: The result always has exact type `int`. Previously, the result could have been an instance of a subclass of `int`.

operator.**inv**(*obj*)

operator.**invert**(*obj*)

operator.**__inv__**(*obj*)

operator.**__invert__**(*obj*)

Return the bitwise inverse of the number *obj*. This is equivalent to `~obj`.

operator.**lshift**(*a*, *b*)

operator.**__lshift__**(*a*, *b*)

Return *a* shifted left by *b*.

operator.**mod**(*a*, *b*)

operator.**__mod__**(*a*, *b*)
Return `a % b`.

operator.**mul**(*a*, *b*)
operator.**__mul__**(*a*, *b*)
Return `a * b`, for *a* and *b* numbers.

operator.**matmul**(*a*, *b*)
operator.**__matmul__**(*a*, *b*)
Return `a @ b`.

Added in version 3.5.

operator.**neg**(*obj*)
operator.**__neg__**(*obj*)
Return *obj* negated (`-obj`).

operator.**or_**(*a*, *b*)
operator.**__or__**(*a*, *b*)
Return the bitwise or of *a* and *b*.

operator.**pos**(*obj*)
operator.**__pos__**(*obj*)
Return *obj* positive (`+obj`).

operator.**pow**(*a*, *b*)
operator.**__pow__**(*a*, *b*)
Return `a ** b`, for *a* and *b* numbers.

operator.**rshift**(*a*, *b*)
operator.**__rshift__**(*a*, *b*)
Return *a* shifted right by *b*.

operator.**sub**(*a*, *b*)
operator.**__sub__**(*a*, *b*)
Return `a - b`.

operator.**truediv**(*a*, *b*)
operator.**__truediv__**(*a*, *b*)
Return `a / b` where 2/3 is .66 rather than 0. This is also known as "true" division.

operator.**xor**(*a*, *b*)
operator.**__xor__**(*a*, *b*)
Return the bitwise exclusive or of *a* and *b*.

Operations which work with sequences (some of them with mappings too) include:

operator.**concat**(*a*, *b*)
operator.**__concat__**(*a*, *b*)
Return `a + b` for *a* and *b* sequences.

operator.**contains**(*a*, *b*)
operator.**__contains__**(*a*, *b*)
Return the outcome of the test `b in a`. Note the reversed operands.

operator.**countOf**(*a*, *b*)

Return the number of occurrences of *b* in *a*.

operator.**delitem**(*a*, *b*)

operator.**__delitem__**(*a*, *b*)

Remove the value of *a* at index *b*.

operator.**getitem**(*a*, *b*)

operator.**__getitem__**(*a*, *b*)

Return the value of *a* at index *b*.

operator.**indexOf**(*a*, *b*)

Return the index of the first of occurrence of *b* in *a*.

operator.**setitem**(*a*, *b*, *c*)

operator.**__setitem__**(*a*, *b*, *c*)

Set the value of *a* at index *b* to *c*.

operator.**length_hint**(*obj*, *default=0*)

Return an estimated length for the object *obj*. First try to return its actual length, then an estimate using `object.__length_hint__()`, and finally return the default value.

Added in version 3.4.

The following operation works with callables:

operator.**call**(*obj*, */*, **args*, ***kwargs*)

operator.**__call__**(*obj*, */*, **args*, ***kwargs*)

Return `obj(*args, **kwargs)`.

Added in version 3.11.

The *operator* module also defines tools for generalized attribute and item lookups. These are useful for making fast field extractors as arguments for *map()*, *sorted()*, *itertools.groupby()*, or other functions that expect a function argument.

operator.**attrgetter**(*attr*)

operator.**attrgetter**(**attrs*)

Return a callable object that fetches *attr* from its operand. If more than one attribute is requested, returns a tuple of attributes. The attribute names can also contain dots. For example:

- After `f = attrgetter('name')`, the call `f(b)` returns `b.name`.
- After `f = attrgetter('name', 'date')`, the call `f(b)` returns `(b.name, b.date)`.
- After `f = attrgetter('name.first', 'name.last')`, the call `f(b)` returns `(b.name.first, b.name.last)`.

Equivalent to:

```
def attrgetter(*items):
    if any(not isinstance(item, str) for item in items):
        raise TypeError('attribute name must be a string')
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
```

(continues on next page)

(continued from previous page)

```
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in items)
    return g

def resolve_attr(obj, attr):
    for name in attr.split("."):
        obj = getattr(obj, name)
    return obj
```

operator.**itemgetter**(*item*)

operator.**itemgetter**(**items*)

Return a callable object that fetches *item* from its operand using the operand's `__getitem__()` method. If multiple items are specified, returns a tuple of lookup values. For example:

- After `f = itemgetter(2)`, the call `f(r)` returns `r[2]`.
- After `g = itemgetter(2, 5, 3)`, the call `g(r)` returns `(r[2], r[5], r[3])`.

Equivalent to:

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

The items can be any type accepted by the operand's `__getitem__()` method. Dictionaries accept any *hashable* value. Lists, tuples, and strings accept an index or a slice:

```
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1, 3, 5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2, None))('ABCDEFG')
'CDEFG'
>>> soldier = dict(rank='captain', name='dotterbart')
>>> itemgetter('rank')(soldier)
'captain'
```

Example of using *`itemgetter()`* to retrieve specific fields from a tuple record:

```
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> list(map(getcount, inventory))
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

operator.**methodcaller**(*name*, */*, **args*, ***kwargs*)

Return a callable object that calls the method *name* on its operand. If additional arguments and/or keyword arguments are given, they will be given to the method as well. For example:

- After `f = methodcaller('name')`, the call `f(b)` returns `b.name()`.

- After `f = methodcaller('name', 'foo', bar=1)`, the call `f(b)` returns `b.name('foo', bar=1)`.

Equivalent to:

```
def methodcaller(name, /, *args, **kwargs):
    def caller(obj):
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

10.3.1 Mapping Operators to Functions

This table shows how abstract operations correspond to operator symbols in the Python syntax and the functions in the *operator* module.

Operation	Syntax	Function
Addition	`a + b`	`add(a, b)`
Concatenation	`seq1 + seq2`	`concat(seq1, seq2)`
Containment Test	`obj in seq`	`contains(seq, obj)`
Division	`a / b`	`truediv(a, b)`
Division	`a // b`	`floordiv(a, b)`
Bitwise And	`a & b`	`and_(a, b)`
Bitwise Exclusive Or	`a ^ b`	`xor(a, b)`
Bitwise Inversion	`~ a`	`invert(a)`
Bitwise Or	`a | b`	`or_(a, b)`
Exponentiation	`a ** b`	`pow(a, b)`
Identity	`a is b`	`is_(a, b)`
Identity	`a is not b`	`is_not(a, b)`
Indexed Assignment	`obj[k] = v`	`setitem(obj, k, v)`
Indexed Deletion	`del obj[k]`	`delitem(obj, k)`
Indexing	`obj[k]`	`getitem(obj, k)`
Left Shift	`a << b`	`lshift(a, b)`
Modulo	`a % b`	`mod(a, b)`
Multiplication	`a * b`	`mul(a, b)`
Matrix Multiplication	`a @ b`	`matmul(a, b)`
Negation (Arithmetic)	`- a`	`neg(a)`
Negation (Logical)	`not a`	`not_(a)`
Positive	`+ a`	`pos(a)`
Right Shift	`a >> b`	`rshift(a, b)`
Slice Assignment	`seq[i:j] = values`	`setitem(seq, slice(i, j), values)`
Slice Deletion	`del seq[i:j]`	`delitem(seq, slice(i, j))`
Slicing	`seq[i:j]`	`getitem(seq, slice(i, j))`
String Formatting	`s % obj`	`mod(s, obj)`
Subtraction	`a - b`	`sub(a, b)`
Truth Test	`obj`	`truth(obj)`
Ordering	`a < b`	`lt(a, b)`
Ordering	`a <= b`	`le(a, b)`
Equality	`a == b`	`eq(a, b)`
Difference	`a != b`	`ne(a, b)`
Ordering	`a >= b`	`ge(a, b)`
Ordering	`a > b`	`gt(a, b)`

10.3.2 In-place Operators

Many operations have an "in-place" version. Listed below are functions providing a more primitive access to in-place operators than the usual syntax does; for example, the *statement* `x += y` is equivalent to `x = operator.iadd(x, y)`. Another way to put it is to say that `z = operator.iadd(x, y)` is equivalent to the compound statement `z = x; z += y`.

In those examples, note that when an in-place method is called, the computation and assignment are performed in two separate steps. The in-place functions listed below only do the first step, calling the in-place method. The second step, assignment, is not handled.

For immutable targets such as strings, numbers, and tuples, the updated value is computed, but not assigned back to the input variable:

```
>>> a = 'hello'
>>> iadd(a, ' world')
'hello world'
>>> a
'hello'
```

For mutable targets such as lists and dictionaries, the in-place method will perform the update, so no subsequent assignment is necessary:

```
>>> s = ['h', 'e', 'l', 'l', 'o']
>>> iadd(s, [' ', 'w', 'o', 'r', 'l', 'd'])
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> s
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

operator.**iadd**(*a*, *b*)

operator.**__iadd__**(*a*, *b*)

`a = iadd(a, b)` is equivalent to `a += b`.

operator.**iand**(*a*, *b*)

operator.**__iand__**(*a*, *b*)

`a = iand(a, b)` is equivalent to `a &= b`.

operator.**iconcat**(*a*, *b*)

operator.**__iconcat__**(*a*, *b*)

`a = iconcat(a, b)` is equivalent to `a += b` for *a* and *b* sequences.

operator.**ifloordiv**(*a*, *b*)

operator.**__ifloordiv__**(*a*, *b*)

`a = ifloordiv(a, b)` is equivalent to `a //= b`.

operator.**ilshift**(*a*, *b*)

operator.**__ilshift__**(*a*, *b*)

`a = ilshift(a, b)` is equivalent to `a <<= b`.

operator.**imod**(*a*, *b*)

operator.**__imod__**(*a*, *b*)

`a = imod(a, b)` is equivalent to `a %= b`.

operator.**imul**(*a*, *b*)

operator.**__imul__**(*a*, *b*)

`a = imul(a, b)` is equivalent to `a *= b`.

operator.**imatmul**(*a*, *b*)

operator.**__imatmul__**(*a*, *b*)

`a = imatmul(a, b)` is equivalent to `a @= b`.

Added in version 3.5.

operator.**ior**(*a*, *b*)

operator.**__ior__**(*a*, *b*)

`a = ior(a, b)` is equivalent to `a |= b`.

operator.**ipow**(*a*, *b*)

operator.**__ipow__**(*a*, *b*)

`a = ipow(a, b)` is equivalent to `a **= b`.

operator.**irshift**(*a*, *b*)

operator.**__irshift__**(*a*, *b*)

`a = irshift(a, b)` is equivalent to `a >>= b`.

operator.**isub**(*a*, *b*)

operator.**__isub__**(*a*, *b*)

`a = isub(a, b)` is equivalent to `a -= b`.

operator.**itruediv**(*a*, *b*)

operator.**__itruediv__**(*a*, *b*)

`a = itruediv(a, b)` is equivalent to `a /= b`.

operator.**ixor**(*a*, *b*)

operator.**__ixor__**(*a*, *b*)

`a = ixor(a, b)` is equivalent to `a ^= b`.

CHAPTER

ELEVEN

FILE AND DIRECTORY ACCESS

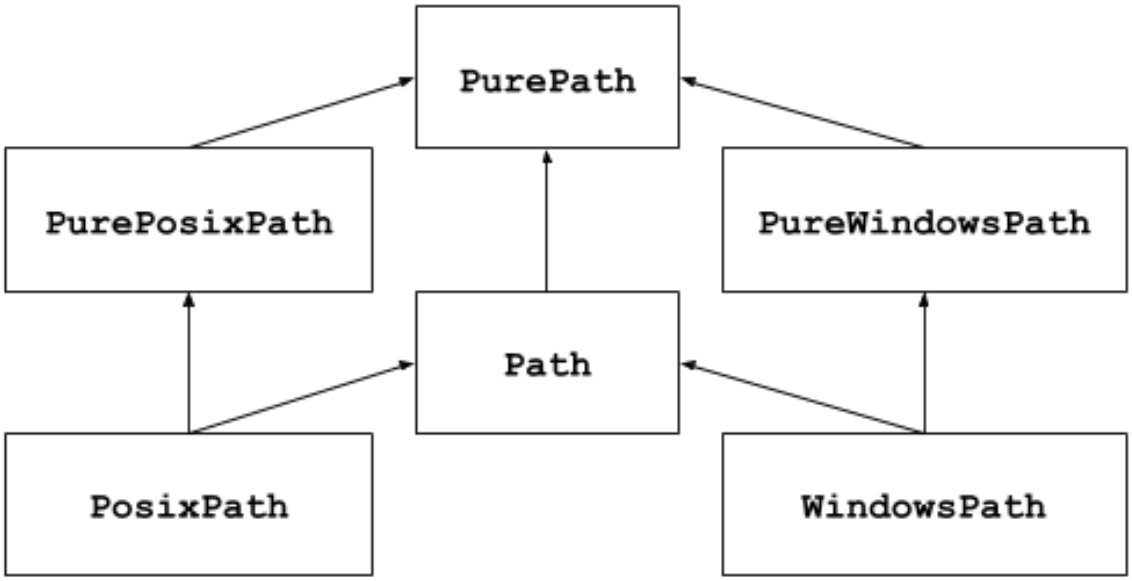
The modules described in this chapter deal with disk files and directories. For example, there are modules for reading the properties of files, manipulating paths in a portable way, and creating temporary files. The full list of modules in this chapter is:

11.1 `pathlib` — Object-oriented filesystem paths

Added in version 3.4.

Source code: Lib/pathlib/

This module offers classes representing filesystem paths with semantics appropriate for different operating systems. Path classes are divided between *pure paths*, which provide purely computational operations without I/O, and *concrete paths*, which inherit from pure paths but also provide I/O operations.



If you've never used this module before or just aren't sure which class is right for your task, `Path` is most likely what you need. It instantiates a *concrete path* for the platform the code is running on.

Pure paths are useful in some special cases; for example:

1. If you want to manipulate Windows paths on a Unix machine (or vice versa). You cannot instantiate a `WindowsPath` when running on Unix, but you can instantiate `PureWindowsPath`.
2. You want to make sure that your code only manipulates paths without actually accessing the OS. In this case, instantiating one of the pure classes may be useful since those simply don't have any OS-accessing operations.

See also

PEP 428: The pathlib module – object-oriented filesystem paths.

See also

For low-level path manipulation on strings, you can also use the `os.path` module.

11.1.1 Basic use

Importing the main class:

```
>>> from pathlib import Path
```

Listing subdirectories:

```
>>> p = Path('.')
>>> [x for x in p.iterdir() if x.is_dir()]
[PosixPath('.hg'), PosixPath('docs'), PosixPath('dist'),
 PosixPath('__pycache__'), PosixPath('build')]
```

Listing Python source files in this directory tree:

```
>>> list(p.glob('**/*.py'))
[PosixPath('test_pathlib.py'), PosixPath('setup.py'),
 PosixPath('pathlib.py'), PosixPath('docs/conf.py'),
 PosixPath('build/lib/pathlib.py')]
```

Navigating inside a directory tree:

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
>>> q.resolve()
PosixPath('/etc/rc.d/init.d/halt')
```

Querying path properties:

```
>>> q.exists()
True
>>> q.is_dir()
False
```

Opening a file:

```
>>> with q.open() as f: f.readline()
...
'#!/bin/bash\n'
```

11.1.2 Exceptions

exception pathlib.**UnsupportedOperation**

An exception inheriting *NotImplementedError* that is raised when an unsupported operation is called on a path object.

Added in version 3.13.

11.1.3 Pure paths

Pure path objects provide path-handling operations which don't actually access a filesystem. There are three ways to access these classes, which we also call *flavours*:

class pathlib.**PurePath**(**pathsegments*)

A generic class that represents the system's path flavour (instantiating it creates either a *PurePosixPath* or a *PureWindowsPath*):

```
>>> PurePath('setup.py')      # Running on a Unix machine
PurePosixPath('setup.py')
```

Each element of *pathsegments* can be either a string representing a path segment, or an object implementing the *os.PathLike* interface where the *__fspath__()* method returns a string, such as another path object:

```
>>> PurePath('foo', 'some/path', 'bar')
PurePosixPath('foo/some/path/bar')
>>> PurePath(Path('foo'), Path('bar'))
PurePosixPath('foo/bar')
```

When *pathsegments* is empty, the current directory is assumed:

```
>>> PurePath()
PurePosixPath('.')
```

If a segment is an absolute path, all previous segments are ignored (like *os.path.join()*):

```
>>> PurePath('/etc', '/usr', 'lib64')
PurePosixPath('/usr/lib64')
>>> PureWindowsPath('c:/Windows', 'd:bar')
PureWindowsPath('d:bar')
```

On Windows, the drive is not reset when a rooted relative path segment (e.g., `r'\foo'`) is encountered:

```
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

Spurious slashes and single dots are collapsed, but double dots (`'..'`) and leading double slashes (`'//'`) are not, since this would change the meaning of a path for various reasons (e.g. symbolic links, UNC paths):

```
>>> PurePath('foo//bar')
PurePosixPath('foo/bar')
>>> PurePath('//foo/bar')
PurePosixPath('//foo/bar')
>>> PurePath('foo/./bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/../bar')
PurePosixPath('foo/../bar')
```

(a naïve approach would make `PurePosixPath('foo/../bar')` equivalent to `PurePosixPath('bar')`, which is wrong if `foo` is a symbolic link to another directory)

Pure path objects implement the *os.PathLike* interface, allowing them to be used anywhere the interface is accepted.

Changed in version 3.6: Added support for the *os.PathLike* interface.

class pathlib.**PurePosixPath**(**pathsegments*)

A subclass of *PurePath*, this path flavour represents non-Windows filesystem paths:

```
>>> PurePosixPath('/etc')
PurePosixPath('/etc')
```

pathsegments is specified similarly to *PurePath*.

class pathlib.**PureWindowsPath**(**pathsegments*)

A subclass of *PurePath*, this path flavour represents Windows filesystem paths, including UNC paths:

```
>>> PureWindowsPath('c:/Program Files/')
PureWindowsPath('c:/Program Files')
>>> PureWindowsPath('//server/share/file')
PureWindowsPath('//server/share/file')
```

pathsegments is specified similarly to *PurePath*.

Regardless of the system you're running on, you can instantiate all of these classes, since they don't provide any operation that does system calls.

General properties

Paths are immutable and *hashable*. Paths of a same flavour are comparable and orderable. These properties respect the flavour's case-folding semantics:

```
>>> PurePosixPath('foo') == PurePosixPath('FOO')
False
>>> PureWindowsPath('foo') == PureWindowsPath('FOO')
True
>>> PureWindowsPath('FOO') in { PureWindowsPath('foo') }
True
>>> PureWindowsPath('C:') < PureWindowsPath('d:')
True
```

Paths of a different flavour compare unequal and cannot be ordered:

```
>>> PureWindowsPath('foo') == PurePosixPath('foo')
False
>>> PureWindowsPath('foo') < PurePosixPath('foo')
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'PureWindowsPath' and 'PurePosixPath
↪'
```

Operators

The slash operator helps create child paths, like *os.path.join()*. If the argument is an absolute path, the previous path is ignored. On Windows, the drive is not reset when the argument is a rooted relative path (e.g., `r'\foo'`):

```
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
>>> p / '/an_absolute_path'
PurePosixPath('/an_absolute_path')
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

A path object can be used anywhere an object implementing *os.PathLike* is accepted:

```
>>> import os
>>> p = PurePath('/etc')
>>> os.fspath(p)
'/etc'
```

The string representation of a path is the raw filesystem path itself (in native form, e.g. with backslashes under Windows), which you can pass to any function taking a file path as a string:

```
>>> p = PurePath('/etc')
>>> str(p)
'/etc'
>>> p = PureWindowsPath('c:/Program Files')
>>> str(p)
'c:\\Program Files'
```

Similarly, calling *bytes* on a path gives the raw filesystem path as a bytes object, as encoded by *os.fsencode()*:

```
>>> bytes(p)
b'/etc'
```

Note

Calling *bytes* is only recommended under Unix. Under Windows, the unicode form is the canonical representation of filesystem paths.

Accessing individual parts

To access the individual "parts" (components) of a path, use the following property:

`PurePath.`**`parts`**

A tuple giving access to the path's various components:

```
>>> p = PurePath('/usr/bin/python3')
>>> p.parts
('/', 'usr', 'bin', 'python3')

>>> p = PureWindowsPath('c:/Program Files/PSF')
>>> p.parts
('c:\\', 'Program Files', 'PSF')
```

(note how the drive and local root are regrouped in a single part)

Methods and properties

Pure paths provide the following methods and properties:

`PurePath.`**`parser`**

The implementation of the *`os.path`* module used for low-level path parsing and joining: either `posixpath` or `ntpath`.

Added in version 3.13.

`PurePath.`**`drive`**

A string representing the drive letter or name, if any:

```
>>> PureWindowsPath('c:/Program Files/').drive
'c:'
>>> PureWindowsPath('/Program Files/').drive
''
>>> PurePosixPath('/etc').drive
''
```

UNC shares are also considered drives:

```
>>> PureWindowsPath('//host/share/foo.txt').drive
'\\\\host\\share'
```

`PurePath.`**`root`**

A string representing the (local or global) root, if any:

```
>>> PureWindowsPath('c:/Program Files/').root
'\\'
>>> PureWindowsPath('c:Program Files/').root
''
>>> PurePosixPath('/etc').root
'/'
```

UNC shares always have a root:

```
>>> PureWindowsPath('//host/share').root
'\\'
```

If the path starts with more than two successive slashes, *PurePosixPath* collapses them:

```
>>> PurePosixPath('//etc').root
'//'
>>> PurePosixPath('///etc').root
'/'
>>> PurePosixPath('////etc').root
'/'
```

Note

This behavior conforms to *The Open Group Base Specifications Issue 6*, paragraph 4.11 Pathname Resolution:

"A pathname that begins with two successive slashes may be interpreted in an implementation-defined manner, although more than two leading slashes shall be treated as a single slash."

PurePath.**anchor**

The concatenation of the drive and root:

```
>>> PureWindowsPath('c:/Program Files/').anchor
'c:\\'
>>> PureWindowsPath('c:Program Files/').anchor
'c:'
>>> PurePosixPath('/etc').anchor
'/'
>>> PureWindowsPath('//host/share').anchor
'\\\\host\\share\\'
```

PurePath.**parents**

An immutable sequence providing access to the logical ancestors of the path:

```
>>> p = PureWindowsPath('c:/foo/bar/setup.py')
>>> p.parents[0]
PureWindowsPath('c:/foo/bar')
>>> p.parents[1]
PureWindowsPath('c:/foo')
>>> p.parents[2]
PureWindowsPath('c:/')
```

Changed in version 3.10: The parents sequence now supports *slices* and negative index values.

PurePath.**parent**

The logical parent of the path:

```
>>> p = PurePosixPath('/a/b/c/d')
>>> p.parent
PurePosixPath('/a/b/c')
```

You cannot go past an anchor, or empty path:

```
>>> p = PurePosixPath('/')
>>> p.parent
PurePosixPath('/')
>>> p = PurePosixPath('.')
>>> p.parent
PurePosixPath('.')
```

> **Note**
>
> This is a purely lexical operation, hence the following behaviour:
>
> ```
> >>> p = PurePosixPath('foo/..')
> >>> p.parent
> PurePosixPath('foo')
> ```
>
> If you want to walk an arbitrary filesystem path upwards, it is recommended to first call *Path.resolve()* so as to resolve symlinks and eliminate `".."` components.

`PurePath.`**`name`**

A string representing the final path component, excluding the drive and root, if any:

```
>>> PurePosixPath('my/library/setup.py').name
'setup.py'
```

UNC drive names are not considered:

```
>>> PureWindowsPath('//some/share/setup.py').name
'setup.py'
>>> PureWindowsPath('//some/share').name
''
```

`PurePath.`**`suffix`**

The last dot-separated portion of the final component, if any:

```
>>> PurePosixPath('my/library/setup.py').suffix
'.py'
>>> PurePosixPath('my/library.tar.gz').suffix
'.gz'
>>> PurePosixPath('my/library').suffix
''
```

This is commonly called the file extension.

`PurePath.`**`suffixes`**

A list of the path's suffixes, often called file extensions:

```
>>> PurePosixPath('my/library.tar.gar').suffixes
['.tar', '.gar']
>>> PurePosixPath('my/library.tar.gz').suffixes
['.tar', '.gz']
>>> PurePosixPath('my/library').suffixes
[]
```

`PurePath.`**`stem`**

The final path component, without its suffix:

```
>>> PurePosixPath('my/library.tar.gz').stem
'library.tar'
>>> PurePosixPath('my/library.tar').stem
'library'
>>> PurePosixPath('my/library').stem
'library'
```

PurePath.**as_posix**()

Return a string representation of the path with forward slashes (/):

```
>>> p = PureWindowsPath('c:\\windows')
>>> str(p)
'c:\\windows'
>>> p.as_posix()
'c:/windows'
```

PurePath.**is_absolute**()

Return whether the path is absolute or not. A path is considered absolute if it has both a root and (if the flavour allows) a drive:

```
>>> PurePosixPath('/a/b').is_absolute()
True
>>> PurePosixPath('a/b').is_absolute()
False

>>> PureWindowsPath('c:/a/b').is_absolute()
True
>>> PureWindowsPath('/a/b').is_absolute()
False
>>> PureWindowsPath('c:').is_absolute()
False
>>> PureWindowsPath('//some/share').is_absolute()
True
```

PurePath.**is_relative_to**(*other*)

Return whether or not this path is relative to the *other* path.

```
>>> p = PurePath('/etc/passwd')
>>> p.is_relative_to('/etc')
True
>>> p.is_relative_to('/usr')
False
```

This method is string-based; it neither accesses the filesystem nor treats "`..`" segments specially. The following code is equivalent:

```
>>> u = PurePath('/usr')
>>> u == p or u in p.parents
False
```

Added in version 3.9.

Deprecated since version 3.12, will be removed in version 3.14: Passing additional arguments is deprecated; if supplied, they are joined with *other*.

PurePath.**is_reserved**()

With *PureWindowsPath*, return `True` if the path is considered reserved under Windows, `False` otherwise. With *PurePosixPath*, `False` is always returned.

Changed in version 3.13: Windows path names that contain a colon, or end with a dot or a space, are considered reserved. UNC paths may be reserved.

Deprecated since version 3.13, will be removed in version 3.15: This method is deprecated; use *os.path.isreserved()* to detect reserved paths on Windows.

PurePath.**joinpath**(**pathsegments*)

Calling this method is equivalent to combining the path with each of the given *pathsegments* in turn:

```
>>> PurePosixPath('/etc').joinpath('passwd')
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath(PurePosixPath('passwd'))
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath('init.d', 'apache2')
PurePosixPath('/etc/init.d/apache2')
>>> PureWindowsPath('c:').joinpath('/Program Files')
PureWindowsPath('c:/Program Files')
```

PurePath.**full_match**(*pattern*, *, *case_sensitive=None*)

Match this path against the provided glob-style pattern. Return `True` if matching is successful, `False` otherwise. For example:

```
>>> PurePath('a/b.py').full_match('a/*.py')
True
>>> PurePath('a/b.py').full_match('*.py')
False
>>> PurePath('/a/b/c.py').full_match('/a/**')
True
>>> PurePath('/a/b/c.py').full_match('**/*.py')
True
```

See also

Pattern language documentation.

As with other methods, case-sensitivity follows platform defaults:

```
>>> PurePosixPath('b.py').full_match('*.PY')
False
>>> PureWindowsPath('b.py').full_match('*.PY')
True
```

Set *case_sensitive* to `True` or `False` to override this behaviour.

Added in version 3.13.

PurePath.**match**(*pattern*, *, *case_sensitive=None*)

Match this path against the provided non-recursive glob-style pattern. Return `True` if matching is successful, `False` otherwise.

This method is similar to *full_match()*, but empty patterns aren't allowed (*ValueError* is raised), the recursive wildcard "`**`" isn't supported (it acts like non-recursive "`*`"), and if a relative pattern is provided, then matching is done from the right:

```
>>> PurePath('a/b.py').match('*.py')
True
>>> PurePath('/a/b/c.py').match('b/*.py')
True
>>> PurePath('/a/b/c.py').match('a/*.py')
False
```

Changed in version 3.12: The *pattern* parameter accepts a *path-like object*.

Changed in version 3.12: The *case_sensitive* parameter was added.

PurePath.**relative_to**(*other*, *walk_up=False*)

Compute a version of this path relative to the path represented by *other*. If it's impossible, *ValueError* is raised:

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.relative_to('/')
PurePosixPath('etc/passwd')
>>> p.relative_to('/etc')
PurePosixPath('passwd')
>>> p.relative_to('/usr')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 941, in relative_to
    raise ValueError(error_message.format(str(self), str(formatted)))
ValueError: '/etc/passwd' is not in the subpath of '/usr' OR one path is relative␣
↪and the other is absolute.
```

When *walk_up* is false (the default), the path must start with *other*. When the argument is true, `..` entries may be added to form the relative path. In all other cases, such as the paths referencing different drives, *ValueError* is raised.:

```
>>> p.relative_to('/usr', walk_up=True)
PurePosixPath('../etc/passwd')
>>> p.relative_to('foo', walk_up=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 941, in relative_to
    raise ValueError(error_message.format(str(self), str(formatted)))
ValueError: '/etc/passwd' is not on the same drive as 'foo' OR one path is␣
↪relative and the other is absolute.
```

Warning

This function is part of *PurePath* and works with strings. It does not check or access the underlying file structure. This can impact the *walk_up* option as it assumes that no symlinks are present in the path; call *resolve()* first if necessary to resolve symlinks.

Changed in version 3.12: The *walk_up* parameter was added (old behavior is the same as `walk_up=False`).

Deprecated since version 3.12, will be removed in version 3.14: Passing additional positional arguments is deprecated; if supplied, they are joined with *other*.

PurePath.**with_name**(*name*)

Return a new path with the *name* changed. If the original path doesn't have a name, ValueError is raised:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_name('setup.py')
PureWindowsPath('c:/Downloads/setup.py')
>>> p = PureWindowsPath('c:/')
>>> p.with_name('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 751, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

PurePath.**with_stem**(*stem*)

Return a new path with the *stem* changed. If the original path doesn't have a name, ValueError is raised:

```
>>> p = PureWindowsPath('c:/Downloads/draft.txt')
>>> p.with_stem('final')
PureWindowsPath('c:/Downloads/final.txt')
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_stem('lib')
PureWindowsPath('c:/Downloads/lib.gz')
>>> p = PureWindowsPath('c:/')
>>> p.with_stem('')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 861, in with_stem
    return self.with_name(stem + self.suffix)
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 851, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

Added in version 3.9.

PurePath.**with_suffix**(*suffix*)

Return a new path with the *suffix* changed. If the original path doesn't have a suffix, the new *suffix* is appended instead. If the *suffix* is an empty string, the original suffix is removed:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_suffix('.bz2')
PureWindowsPath('c:/Downloads/pathlib.tar.bz2')
>>> p = PureWindowsPath('README')
>>> p.with_suffix('.txt')
PureWindowsPath('README.txt')
>>> p = PureWindowsPath('README.txt')
>>> p.with_suffix('')
PureWindowsPath('README')
```

PurePath.**with_segments**(**pathsegments*)

Create a new path object of the same type by combining the given *pathsegments*. This method is called whenever a derivative path is created, such as from *parent* and *relative_to()*. Subclasses may override this method to pass information to derivative paths, for example:

```
from pathlib import PurePosixPath

class MyPath(PurePosixPath):
    def __init__(self, *pathsegments, session_id):
        super().__init__(*pathsegments)
        self.session_id = session_id

    def with_segments(self, *pathsegments):
        return type(self)(*pathsegments, session_id=self.session_id)

etc = MyPath('/etc', session_id=42)
hosts = etc / 'hosts'
print(hosts.session_id)  # 42
```

Added in version 3.12.

11.1.4 Concrete paths

Concrete paths are subclasses of the pure path classes. In addition to operations provided by the latter, they also provide methods to do system calls on path objects. There are three ways to instantiate concrete paths:

class pathlib.**Path**(**pathsegments*)

A subclass of *PurePath*, this class represents concrete paths of the system's path flavour (instantiating it creates either a *PosixPath* or a *WindowsPath*):

```
>>> Path('setup.py')
PosixPath('setup.py')
```

pathsegments is specified similarly to *PurePath*.

class pathlib.**PosixPath**(**pathsegments*)

A subclass of *Path* and *PurePosixPath*, this class represents concrete non-Windows filesystem paths:

```
>>> PosixPath('/etc')
PosixPath('/etc')
```

pathsegments is specified similarly to *PurePath*.

Changed in version 3.13: Raises *UnsupportedOperation* on Windows. In previous versions, *NotImplementedError* was raised instead.

class pathlib.**WindowsPath**(**pathsegments*)

A subclass of *Path* and *PureWindowsPath*, this class represents concrete Windows filesystem paths:

```
>>> WindowsPath('c:/Program Files/')
WindowsPath('c:/Program Files')
```

pathsegments is specified similarly to *PurePath*.

Changed in version 3.13: Raises *UnsupportedOperation* on non-Windows platforms. In previous versions, *NotImplementedError* was raised instead.

You can only instantiate the class flavour that corresponds to your system (allowing system calls on non-compatible path flavours could lead to bugs or failures in your application):

```
>>> import os
>>> os.name
'posix'
>>> Path('setup.py')
PosixPath('setup.py')
>>> PosixPath('setup.py')
PosixPath('setup.py')
>>> WindowsPath('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 798, in __new__
    % (cls.__name__,))
UnsupportedOperation: cannot instantiate 'WindowsPath' on your system
```

Some concrete path methods can raise an *OSError* if a system call fails (for example because the path doesn't exist).

Parsing and generating URIs

Concrete path objects can be created from, and represented as, 'file' URIs conforming to **RFC 8089**.

> **Note**
>
> File URIs are not portable across machines with different *filesystem encodings*.

classmethod Path.**from_uri**(*uri*)

Return a new path object from parsing a 'file' URI. For example:

```
>>> p = Path.from_uri('file:///etc/hosts')
PosixPath('/etc/hosts')
```

On Windows, DOS device and UNC paths may be parsed from URIs:

```
>>> p = Path.from_uri('file:///c:/windows')
WindowsPath('c:/windows')
>>> p = Path.from_uri('file://server/share')
WindowsPath('//server/share')
```

Several variant forms are supported:

```
>>> p = Path.from_uri('file:////server/share')
WindowsPath('//server/share')
>>> p = Path.from_uri('file://///server/share')
WindowsPath('//server/share')
>>> p = Path.from_uri('file:c:/windows')
WindowsPath('c:/windows')
>>> p = Path.from_uri('file:/c|/windows')
WindowsPath('c:/windows')
```

ValueError is raised if the URI does not start with `file:`, or the parsed path isn't absolute.

Added in version 3.13.

Path.**as_uri**()

Represent the path as a 'file' URI. *ValueError* is raised if the path isn't absolute.

```
>>> p = PosixPath('/etc/passwd')
>>> p.as_uri()
'file:///etc/passwd'
>>> p = WindowsPath('c:/Windows')
>>> p.as_uri()
'file:///c:/Windows'
```

For historical reasons, this method is also available from *PurePath* objects. However, its use of *os.fsencode()* makes it strictly impure.

Expanding and resolving paths

classmethod Path.**home**()

Return a new path object representing the user's home directory (as returned by *os.path.expanduser()* with ~ construct). If the home directory can't be resolved, *RuntimeError* is raised.

```
>>> Path.home()
PosixPath('/home/antoine')
```

Added in version 3.5.

Path.**expanduser**()

Return a new path with expanded ~ and ~user constructs, as returned by *os.path.expanduser()*. If a home directory can't be resolved, *RuntimeError* is raised.

```
>>> p = PosixPath('~/films/Monty Python')
>>> p.expanduser()
PosixPath('/home/eric/films/Monty Python')
```

Added in version 3.5.

classmethod Path.**cwd**()

Return a new path object representing the current directory (as returned by *os.getcwd()*):

```
>>> Path.cwd()
PosixPath('/home/antoine/pathlib')
```

Path.**absolute**()

Make the path absolute, without normalization or resolving symlinks. Returns a new path object:

```
>>> p = Path('tests')
>>> p
PosixPath('tests')
>>> p.absolute()
PosixPath('/home/antoine/pathlib/tests')
```

Path.**resolve**(*strict=False*)

Make the path absolute, resolving any symlinks. A new path object is returned:

```
>>> p = Path()
>>> p
PosixPath('.')
>>> p.resolve()
PosixPath('/home/antoine/pathlib')
```

"`..`" components are also eliminated (this is the only method to do so):

```
>>> p = Path('docs/../setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
```

If a path doesn't exist or a symlink loop is encountered, and *strict* is `True`, *OSError* is raised. If *strict* is `False`, the path is resolved as far as possible and any remainder is appended without checking whether it exists.

Changed in version 3.6: The *strict* parameter was added (pre-3.6 behavior is strict).

Changed in version 3.13: Symlink loops are treated like other errors: *OSError* is raised in strict mode, and no exception is raised in non-strict mode. In previous versions, *RuntimeError* is raised no matter the value of *strict*.

Path.**readlink**()

Return the path to which the symbolic link points (as returned by *os.readlink()*):

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.readlink()
PosixPath('setup.py')
```

Added in version 3.9.

Changed in version 3.13: Raises *UnsupportedOperation* if *os.readlink()* is not available. In previous versions, *NotImplementedError* was raised.

Querying file type and status

Changed in version 3.8: *exists()*, *is_dir()*, *is_file()*, *is_mount()*, *is_symlink()*, *is_block_device()*, *is_char_device()*, *is_fifo()*, *is_socket()* now return `False` instead of raising an exception for paths that contain characters unrepresentable at the OS level.

Path.**stat**(*, *follow_symlinks=True*)

Return an *os.stat_result* object containing information about this path, like *os.stat()*. The result is looked up at each call to this method.

This method normally follows symlinks; to stat a symlink add the argument `follow_symlinks=False`, or use *lstat()*.

```
>>> p = Path('setup.py')
>>> p.stat().st_size
956
>>> p.stat().st_mtime
1327883547.852554
```

Changed in version 3.10: The *follow_symlinks* parameter was added.

Path.**lstat**()

Like *Path.stat()* but, if the path points to a symbolic link, return the symbolic link's information rather than its target's.

Path.**exists**(*, *follow_symlinks=True*)

Return `True` if the path points to an existing file or directory.

This method normally follows symlinks; to check if a symlink exists, add the argument `follow_symlinks=False`.

```
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

Changed in version 3.12: The *follow_symlinks* parameter was added.

Path.**is_file**(*, *follow_symlinks=True*)

Return `True` if the path points to a regular file, `False` if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

This method normally follows symlinks; to exclude symlinks, add the argument `follow_symlinks=False`.

Changed in version 3.13: The *follow_symlinks* parameter was added.

Path.**is_dir**(*, *follow_symlinks=True*)

Return `True` if the path points to a directory, `False` if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

This method normally follows symlinks; to exclude symlinks to directories, add the argument `follow_symlinks=False`.

Changed in version 3.13: The *follow_symlinks* parameter was added.

Path.**is_symlink**()

Return `True` if the path points to a symbolic link, `False` otherwise.

`False` is also returned if the path doesn't exist; other errors (such as permission errors) are propagated.

Path.**is_junction**()

Return `True` if the path points to a junction, and `False` for any other type of file. Currently only Windows supports junctions.

Added in version 3.12.

Path.**is_mount**()

Return `True` if the path is a *mount point*: a point in a file system where a different file system has been mounted. On POSIX, the function checks whether *path*'s parent, `path/..`, is on a different device than *path*, or whether `path/..` and *path* point to the same i-node on the same device — this should detect mount points for all Unix and POSIX variants. On Windows, a mount point is considered to be a drive letter root (e.g. `c:\`), a UNC share (e.g. `\\server\share`), or a mounted filesystem directory.

Added in version 3.7.

Changed in version 3.12: Windows support was added.

Path.**is_socket**()

Return `True` if the path points to a Unix socket (or a symbolic link pointing to a Unix socket), `False` if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

Path.**is_fifo**()

Return `True` if the path points to a FIFO (or a symbolic link pointing to a FIFO), `False` if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

Path.**is_block_device**()

Return `True` if the path points to a block device (or a symbolic link pointing to a block device), `False` if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.`**`is_char_device`**`()`

Return `True` if the path points to a character device (or a symbolic link pointing to a character device), `False` if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.`**`samefile`**`(`*other_path*`)`

Return whether this path points to the same file as *other_path*, which can be either a Path object, or a string. The semantics are similar to *os.path.samefile()* and *os.path.samestat()*.

An *OSError* can be raised if either file cannot be accessed for some reason.

```
>>> p = Path('spam')
>>> q = Path('eggs')
>>> p.samefile(q)
False
>>> p.samefile('spam')
True
```

Added in version 3.5.

Reading and writing files

`Path.`**`open`**`(`*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*`)`

Open the file pointed to by the path, like the built-in *open()* function does:

```
>>> p = Path('setup.py')
>>> with p.open() as f:
...     f.readline()
...
'#!/usr/bin/env python3\n'
```

`Path.`**`read_text`**`(`*encoding=None*, *errors=None*, *newline=None*`)`

Return the decoded contents of the pointed-to file as a string:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

The file is opened and then closed. The optional parameters have the same meaning as in *open()*.

Added in version 3.5.

Changed in version 3.13: The *newline* parameter was added.

`Path.`**`read_bytes`**`()`

Return the binary contents of the pointed-to file as a bytes object:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
```

(continues on next page)

(continued from previous page)

```
>>> p.read_bytes()
b'Binary file contents'
```

Added in version 3.5.

`Path.`**`write_text`**(*data*, *encoding=None*, *errors=None*, *newline=None*)

Open the file pointed to in text mode, write *data* to it, and close the file:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

An existing file of the same name is overwritten. The optional parameters have the same meaning as in *open()*.

Added in version 3.5.

Changed in version 3.10: The *newline* parameter was added.

`Path.`**`write_bytes`**(*data*)

Open the file pointed to in bytes mode, write *data* to it, and close the file:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

An existing file of the same name is overwritten.

Added in version 3.5.

Reading directories

`Path.`**`iterdir`**()

When the path points to a directory, yield path objects of the directory contents:

```
>>> p = Path('docs')
>>> for child in p.iterdir(): child
...
PosixPath('docs/conf.py')
PosixPath('docs/_templates')
PosixPath('docs/make.bat')
PosixPath('docs/index.rst')
PosixPath('docs/_build')
PosixPath('docs/_static')
PosixPath('docs/Makefile')
```

The children are yielded in arbitrary order, and the special entries `'.'` and `'..'` are not included. If a file is removed from or added to the directory after creating the iterator, it is unspecified whether a path object for that file is included.

If the path is not a directory or otherwise inaccessible, *OSError* is raised.

`Path.`**`glob`**(*pattern*, *, *case_sensitive=None*, *recurse_symlinks=False*)

Glob the given relative *pattern* in the directory represented by this path, yielding all matching files (of any kind):

```
>>> sorted(Path('.').glob('*.py'))
[PosixPath('pathlib.py'), PosixPath('setup.py'), PosixPath('test_pathlib.py')]
>>> sorted(Path('.').glob('*/*.py'))
[PosixPath('docs/conf.py')]
>>> sorted(Path('.').glob('**/*.py'))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

See also

Pattern language documentation.

By default, or when the *case_sensitive* keyword-only argument is set to `None`, this method matches paths using platform-specific casing rules: typically, case-sensitive on POSIX, and case-insensitive on Windows. Set *case_sensitive* to `True` or `False` to override this behaviour.

By default, or when the *recurse_symlinks* keyword-only argument is set to `False`, this method follows symlinks except when expanding "`**`" wildcards. Set *recurse_symlinks* to `True` to always follow symlinks.

Raises an *auditing event* `pathlib.Path.glob` with arguments `self`, `pattern`.

Changed in version 3.12: The *case_sensitive* parameter was added.

Changed in version 3.13: The *recurse_symlinks* parameter was added.

Changed in version 3.13: The *pattern* parameter accepts a *path-like object*.

Changed in version 3.13: Any `OSError` exceptions raised from scanning the filesystem are suppressed. In previous versions, such exceptions are suppressed in many cases, but not all.

`Path.`**`rglob`**(*pattern*, *, *case_sensitive=None*, *recurse_symlinks=False*)

Glob the given relative *pattern* recursively. This is like calling `Path.glob()` with "`**/`" added in front of the *pattern*.

See also

Pattern language and `Path.glob()` documentation.

Raises an *auditing event* `pathlib.Path.rglob` with arguments `self`, `pattern`.

Changed in version 3.12: The *case_sensitive* parameter was added.

Changed in version 3.13: The *recurse_symlinks* parameter was added.

Changed in version 3.13: The *pattern* parameter accepts a *path-like object*.

`Path.`**`walk`**(*top_down=True*, *on_error=None*, *follow_symlinks=False*)

Generate the file names in a directory tree by walking the tree either top-down or bottom-up.

For each directory in the directory tree rooted at *self* (including *self* but excluding '.' and '..'), the method yields a 3-tuple of `(dirpath, dirnames, filenames)`.

dirpath is a `Path` to the directory currently being walked, *dirnames* is a list of strings for the names of subdirectories in *dirpath* (excluding `'.'` and `'..'`), and *filenames* is a list of strings for the names of the non-directory files in *dirpath*. To get a full path (which begins with *self*) to a file or directory in *dirpath*, do `dirpath / name`. Whether or not the lists are sorted is file system-dependent.

If the optional argument *top_down* is true (which is the default), the triple for a directory is generated before the triples for any of its subdirectories (directories are walked top-down). If *top_down* is false, the triple for a directory is generated after the triples for all of its subdirectories (directories are walked bottom-up). No matter the value of *top_down*, the list of subdirectories is retrieved before the triples for the directory and its subdirectories are walked.

When *top_down* is true, the caller can modify the *dirnames* list in-place (for example, using `del` or slice assignment), and `Path.walk()` will only recurse into the subdirectories whose names remain in *dirnames*. This can be used to prune the search, or to impose a specific order of visiting, or even to inform `Path.walk()` about directories the caller creates or renames before it resumes `Path.walk()` again. Modifying *dirnames* when *top_down* is false has no effect on the behavior of `Path.walk()` since the directories in *dirnames* have already been generated by the time *dirnames* is yielded to the caller.

By default, errors from `os.scandir()` are ignored. If the optional argument *on_error* is specified, it should be a callable; it will be called with one argument, an `OSError` instance. The callable can handle the error to continue the walk or re-raise it to stop the walk. Note that the filename is available as the `filename` attribute of the exception object.

By default, `Path.walk()` does not follow symbolic links, and instead adds them to the *filenames* list. Set *follow_symlinks* to true to resolve symlinks and place them in *dirnames* and *filenames* as appropriate for their targets, and consequently visit directories pointed to by symlinks (where supported).

Note

Be aware that setting *follow_symlinks* to true can lead to infinite recursion if a link points to a parent directory of itself. `Path.walk()` does not keep track of the directories it has already visited.

Note

`Path.walk()` assumes the directories it walks are not modified during execution. For example, if a directory from *dirnames* has been replaced with a symlink and *follow_symlinks* is false, `Path.walk()` will still try to descend into it. To prevent such behavior, remove directories from *dirnames* as appropriate.

Note

Unlike `os.walk()`, `Path.walk()` lists symlinks to directories in *filenames* if *follow_symlinks* is false.

This example displays the number of bytes used by all files in each directory, while ignoring `__pycache__` directories:

```
from pathlib import Path
for root, dirs, files in Path("cpython/Lib/concurrent").walk(on_error=print):
  print(
      root,
      "consumes",
      sum((root / file).stat().st_size for file in files),
      "bytes in",
      len(files),
```

(continues on next page)

(continued from previous page)

```
        "non-directory files"
    )
    if '__pycache__' in dirs:
        dirs.remove('__pycache__')
```

This next example is a simple implementation of *shutil.rmtree()*. Walking the tree bottom-up is essential as *rmdir()* doesn't allow deleting a directory before it is empty:

```
# Delete everything reachable from the directory "top".
# CAUTION:  This is dangerous! For example, if top == Path('/'),
# it could delete all of your files.
for root, dirs, files in top.walk(top_down=False):
    for name in files:
        (root / name).unlink()
    for name in dirs:
        (root / name).rmdir()
```

Added in version 3.12.

Creating files and directories

`Path.`**`touch`**(*mode=0o666*, *exist_ok=True*)

Create a file at this given path. If *mode* is given, it is combined with the process's `umask` value to determine the file mode and access flags. If the file already exists, the function succeeds when *exist_ok* is true (and its modification time is updated to the current time), otherwise *FileExistsError* is raised.

See also

The *open()*, *write_text()* and *write_bytes()* methods are often used to create files.

`Path.`**`mkdir`**(*mode=0o777*, *parents=False*, *exist_ok=False*)

Create a new directory at this given path. If *mode* is given, it is combined with the process's `umask` value to determine the file mode and access flags. If the path already exists, *FileExistsError* is raised.

If *parents* is true, any missing parents of this path are created as needed; they are created with the default permissions without taking *mode* into account (mimicking the POSIX `mkdir -p` command).

If *parents* is false (the default), a missing parent raises *FileNotFoundError*.

If *exist_ok* is false (the default), *FileExistsError* is raised if the target directory already exists.

If *exist_ok* is true, *FileExistsError* will not be raised unless the given path already exists in the file system and is not a directory (same behavior as the POSIX `mkdir -p` command).

Changed in version 3.5: The *exist_ok* parameter was added.

`Path.`**`symlink_to`**(*target*, *target_is_directory=False*)

Make this path a symbolic link pointing to *target*.

On Windows, a symlink represents either a file or a directory, and does not morph to the target dynamically. If the target is present, the type of the symlink will be created to match. Otherwise, the symlink will be created as a directory if *target_is_directory* is true or a file symlink (the default) otherwise. On non-Windows platforms, *target_is_directory* is ignored.

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
>>> p.stat().st_size
956
>>> p.lstat().st_size
8
```

Note

The order of arguments (link, target) is the reverse of *os.symlink()*'s.

Changed in version 3.13: Raises *UnsupportedOperation* if *os.symlink()* is not available. In previous versions, *NotImplementedError* was raised.

`Path.`**`hardlink_to`**`(`*target*`)`

Make this path a hard link to the same file as *target*.

Note

The order of arguments (link, target) is the reverse of *os.link()*'s.

Added in version 3.10.

Changed in version 3.13: Raises *UnsupportedOperation* if *os.link()* is not available. In previous versions, *NotImplementedError* was raised.

Renaming and deleting

`Path.`**`rename`**`(`*target*`)`

Rename this file or directory to the given *target*, and return a new `Path` instance pointing to *target*. On Unix, if *target* exists and is a file, it will be replaced silently if the user has permission. On Windows, if *target* exists, *FileExistsError* will be raised. *target* can be either a string or another path object:

```
>>> p = Path('foo')
>>> p.open('w').write('some text')
9
>>> target = Path('bar')
>>> p.rename(target)
PosixPath('bar')
>>> target.open().read()
'some text'
```

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, *not* the directory of the `Path` object.

It is implemented in terms of *os.rename()* and gives the same guarantees.

Changed in version 3.8: Added return value, return the new `Path` instance.

`Path.`**`replace`**`(`*target*`)`

Rename this file or directory to the given *target*, and return a new `Path` instance pointing to *target*. If *target* points to an existing file or empty directory, it will be unconditionally replaced.

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, *not* the directory of the `Path` object.

Changed in version 3.8: Added return value, return the new `Path` instance.

`Path.`**`unlink`**(*missing_ok=False*)

Remove this file or symbolic link. If the path points to a directory, use *`Path.rmdir()`* instead.

If *missing_ok* is false (the default), *`FileNotFoundError`* is raised if the path does not exist.

If *missing_ok* is true, *`FileNotFoundError`* exceptions will be ignored (same behavior as the POSIX `rm -f` command).

Changed in version 3.8: The *missing_ok* parameter was added.

`Path.`**`rmdir`**()

Remove this directory. The directory must be empty.

Permissions and ownership

`Path.`**`owner`**(*, *follow_symlinks=True*)

Return the name of the user owning the file. *`KeyError`* is raised if the file's user identifier (UID) isn't found in the system database.

This method normally follows symlinks; to get the owner of the symlink, add the argument `follow_symlinks=False`.

Changed in version 3.13: Raises *`UnsupportedOperation`* if the *`pwd`* module is not available. In earlier versions, *`NotImplementedError`* was raised.

Changed in version 3.13: The *follow_symlinks* parameter was added.

`Path.`**`group`**(*, *follow_symlinks=True*)

Return the name of the group owning the file. *`KeyError`* is raised if the file's group identifier (GID) isn't found in the system database.

This method normally follows symlinks; to get the group of the symlink, add the argument `follow_symlinks=False`.

Changed in version 3.13: Raises *`UnsupportedOperation`* if the *`grp`* module is not available. In earlier versions, *`NotImplementedError`* was raised.

Changed in version 3.13: The *follow_symlinks* parameter was added.

`Path.`**`chmod`**(*mode*, *, *follow_symlinks=True*)

Change the file mode and permissions, like *`os.chmod()`*.

This method normally follows symlinks. Some Unix flavours support changing permissions on the symlink itself; on these platforms you may add the argument `follow_symlinks=False`, or use *`lchmod()`*.

```
>>> p = Path('setup.py')
>>> p.stat().st_mode
33277
>>> p.chmod(0o444)
>>> p.stat().st_mode
33060
```

Changed in version 3.10: The *follow_symlinks* parameter was added.

Path.**lchmod**(*mode*)

Like *Path.chmod()* but, if the path points to a symbolic link, the symbolic link's mode is changed rather than its target's.

11.1.5 Pattern language

The following wildcards are supported in patterns for *full_match()*, *glob()* and *rglob()*:

**** (entire segment)**
Matches any number of file or directory segments, including zero.

*** (entire segment)**
Matches one file or directory segment.

*** (part of a segment)**
Matches any number of non-separator characters, including zero.

?
Matches one non-separator character.

[seq]
Matches one character in *seq*.

[!seq]
Matches one character not in *seq*.

For a literal match, wrap the meta-characters in brackets. For example, `"[?]"` matches the character `"?"`.

The "`**`" wildcard enables recursive globbing. A few examples:

Pattern	Meaning
"`**/*`"	Any path with at least one segment.
"`**/*.py`"	Any path with a final segment ending "`.py`".
"`assets/**`"	Any path starting with "`assets/`".
"`assets/**/*`"	Any path starting with "`assets/`", excluding "`assets/`" itself.

Note

Globbing with the "`**`" wildcard visits every directory in the tree. Large directory trees may take a long time to search.

Changed in version 3.13: Globbing with a pattern that ends with "`**`" returns both files and directories. In previous versions, only directories were returned.

In *Path.glob()* and *rglob()*, a trailing slash may be added to the pattern to match only directories.

Changed in version 3.11: Globbing with a pattern that ends with a pathname components separator (*sep* or *altsep*) returns only directories.

11.1.6 Comparison to the glob module

The patterns accepted and results generated by *Path.glob()* and *Path.rglob()* differ slightly from those by the *glob* module:

1. Files beginning with a dot are not special in pathlib. This is like passing `include_hidden=True` to *glob.glob()*.
2. "`**`" pattern components are always recursive in pathlib. This is like passing `recursive=True` to *glob.glob()*.
3. "`**`" pattern components do not follow symlinks by default in pathlib. This behaviour has no equivalent in *glob.glob()*, but you can pass `recurse_symlinks=True` to *Path.glob()* for compatible behaviour.
4. Like all *PurePath* and *Path* objects, the values returned from *Path.glob()* and *Path.rglob()* don't include trailing slashes.
5. The values returned from pathlib's `path.glob()` and `path.rglob()` include the *path* as a prefix, unlike the results of `glob.glob(root_dir=path)`.
6. The values returned from pathlib's `path.glob()` and `path.rglob()` may include *path* itself, for example when globbing "`**`", whereas the results of `glob.glob(root_dir=path)` never include an empty string that would correspond to *path*.

11.1.7 Comparison to the os and os.path modules

pathlib implements path operations using *PurePath* and *Path* objects, and so it's said to be *object-oriented*. On the other hand, the *os* and *os.path* modules supply functions that work with low-level `str` and `bytes` objects, which is a more *procedural* approach. Some users consider the object-oriented style to be more readable.

Many functions in *os* and *os.path* support `bytes` paths and *paths relative to directory descriptors*. These features aren't available in pathlib.

Python's `str` and `bytes` types, and portions of the *os* and *os.path* modules, are written in C and are very speedy. pathlib is written in pure Python and is often slower, but rarely slow enough to matter.

pathlib's path normalization is slightly more opinionated and consistent than *os.path*. For example, whereas *os.path.abspath()* eliminates "`..`" segments from a path, which may change its meaning if symlinks are involved, *Path.absolute()* preserves these segments for greater safety.

pathlib's path normalization may render it unsuitable for some applications:

1. pathlib normalizes `Path("my_folder/")` to `Path("my_folder")`, which changes a path's meaning when supplied to various operating system APIs and command-line utilities. Specifically, the absence of a trailing separator may allow the path to be resolved as either a file or directory, rather than a directory only.
2. pathlib normalizes `Path("./my_program")` to `Path("my_program")`, which changes a path's meaning when used as an executable search path, such as in a shell or when spawning a child process. Specifically, the absence of a separator in the path may force it to be looked up in `PATH` rather than the current directory.

As a consequence of these differences, pathlib is not a drop-in replacement for *os.path*.

Corresponding tools

Below is a table mapping various *os* functions to their corresponding *PurePath*/*Path* equivalent.

os and *os.path*	*pathlib*
os.path.dirname()	*PurePath.parent*
os.path.basename()	*PurePath.name*
os.path.splitext()	*PurePath.stem*, *PurePath.suffix*
os.path.join()	*PurePath.joinpath()*
os.path.isabs()	*PurePath.is_absolute()*
os.path.relpath()	*PurePath.relative_to()*[1]
os.path.expanduser()	*Path.expanduser()*[2]
os.path.realpath()	*Path.resolve()*
os.path.abspath()	*Path.absolute()*[3]
os.path.exists()	*Path.exists()*
os.path.isfile()	*Path.is_file()*
os.path.isdir()	*Path.is_dir()*
os.path.islink()	*Path.is_symlink()*
os.path.isjunction()	*Path.is_junction()*
os.path.ismount()	*Path.is_mount()*
os.path.samefile()	*Path.samefile()*
os.getcwd()	*Path.cwd()*
os.stat()	*Path.stat()*
os.lstat()	*Path.lstat()*
os.listdir()	*Path.iterdir()*
os.walk()	*Path.walk()*[4]
os.mkdir(), *os.makedirs()*	*Path.mkdir()*
os.link()	*Path.hardlink_to()*
os.symlink()	*Path.symlink_to()*
os.readlink()	*Path.readlink()*
os.rename()	*Path.rename()*
os.replace()	*Path.replace()*
os.remove(), *os.unlink()*	*Path.unlink()*
os.rmdir()	*Path.rmdir()*
os.chmod()	*Path.chmod()*
os.lchmod()	*Path.lchmod()*

[1] *os.path.relpath()* calls *abspath()* to make paths absolute and remove "`..`" parts, whereas *PurePath.relative_to()* is a lexical operation that raises *ValueError* when its inputs' anchors differ (e.g. if one path is absolute and the other relative.)

[2] *os.path.expanduser()* returns the path unchanged if the home directory can't be resolved, whereas *Path.expanduser()* raises *RuntimeError*.

[3] *os.path.abspath()* removes "`..`" components without resolving symlinks, which may change the meaning of the path, whereas *Path.absolute()* leaves any "`..`" components in the path.

[4] *os.walk()* always follows symlinks when categorizing paths into *dirnames* and *filenames*, whereas *Path.walk()* categorizes all symlinks into *filenames* when *follow_symlinks* is false (the default.)

11.2 os.path — Common pathname manipulations

Source code: Lib/genericpath.py, Lib/posixpath.py (for POSIX) and Lib/ntpath.py (for Windows).

This module implements some useful functions on pathnames. To read or write files see *open()*, and for accessing the filesystem see the *os* module. The path parameters can be passed as strings, or bytes, or any object implementing the *os.PathLike* protocol.

Unlike a Unix shell, Python does not do any *automatic* path expansions. Functions such as *expanduser()* and *expandvars()* can be invoked explicitly when an application desires shell-like path expansion. (See also the *glob* module.)

See also

The *pathlib* module offers high-level path objects.

Note

All of these functions accept either only bytes or only string objects as their parameters. The result is an object of the same type, if a path or file name is returned.

Note

Since different operating systems have different path name conventions, there are several versions of this module in the standard library. The *os.path* module is always the path module suitable for the operating system Python is running on, and therefore usable for local paths. However, you can also import and use the individual modules if you want to manipulate a path that is *always* in one of the different formats. They all have the same interface:

- `posixpath` for UNIX-style paths
- `ntpath` for Windows paths

Changed in version 3.8: *exists()*, *lexists()*, *isdir()*, *isfile()*, *islink()*, and *ismount()* now return `False` instead of raising an exception for paths that contain characters or bytes unrepresentable at the OS level.

`os.path.`**`abspath`**(*path*)

Return a normalized absolutized version of the pathname *path*. On most platforms, this is equivalent to calling the function *normpath()* as follows: `normpath(join(os.getcwd(), path))`.

Changed in version 3.6: Accepts a *path-like object*.

`os.path.`**`basename`**(*path*)

Return the base name of pathname *path*. This is the second element of the pair returned by passing *path* to the function *split()*. Note that the result of this function is different from the Unix **basename** program; where **basename** for `'/foo/bar/'` returns `'bar'`, the *basename()* function returns an empty string (`''`).

Changed in version 3.6: Accepts a *path-like object*.

`os.path.`**`commonpath`**(*paths*)

Return the longest common sub-path of each pathname in the iterable *paths*. Raise *ValueError* if *paths* contain both absolute and relative pathnames, the *paths* are on the different drives or if *paths* is empty. Unlike *commonprefix()*, this returns a valid path.

Added in version 3.5.

Changed in version 3.6: Accepts a sequence of *path-like objects*.

Changed in version 3.13: Any iterable can now be passed, rather than just sequences.

os.path.**commonprefix**(*list*)

Return the longest path prefix (taken character-by-character) that is a prefix of all paths in *list*. If *list* is empty, return the empty string (`''`).

Note

This function may return invalid paths because it works a character at a time. To obtain a valid path, see *commonpath()*.

```
>>> os.path.commonprefix(['/usr/lib', '/usr/local/lib'])
'/usr/l'

>>> os.path.commonpath(['/usr/lib', '/usr/local/lib'])
'/usr'
```

Changed in version 3.6: Accepts a *path-like object*.

os.path.**dirname**(*path*)

Return the directory name of pathname *path*. This is the first element of the pair returned by passing *path* to the function *split()*.

Changed in version 3.6: Accepts a *path-like object*.

os.path.**exists**(*path*)

Return `True` if *path* refers to an existing path or an open file descriptor. Returns `False` for broken symbolic links. On some platforms, this function may return `False` if permission is not granted to execute *os.stat()* on the requested file, even if the *path* physically exists.

Changed in version 3.3: *path* can now be an integer: `True` is returned if it is an open file descriptor, `False` otherwise.

Changed in version 3.6: Accepts a *path-like object*.

os.path.**lexists**(*path*)

Return `True` if *path* refers to an existing path, including broken symbolic links. Equivalent to *exists()* on platforms lacking *os.lstat()*.

Changed in version 3.6: Accepts a *path-like object*.

os.path.**expanduser**(*path*)

On Unix and Windows, return the argument with an initial component of ~ or `~user` replaced by that *user*'s home directory.

On Unix, an initial ~ is replaced by the environment variable `HOME` if it is set; otherwise the current user's home directory is looked up in the password directory through the built-in module *pwd*. An initial `~user` is looked up directly in the password directory.

On Windows, `USERPROFILE` will be used if set, otherwise a combination of `HOMEPATH` and `HOMEDRIVE` will be used. An initial `~user` is handled by checking that the last directory component of the current user's home directory matches `USERNAME`, and replacing it if so.

If the expansion fails or if the path does not begin with a tilde, the path is returned unchanged.

Changed in version 3.6: Accepts a *path-like object*.

Changed in version 3.8: No longer uses `HOME` on Windows.

`os.path.`**`expandvars`**(*path*)

Return the argument with environment variables expanded. Substrings of the form `$name` or `${name}` are replaced by the value of environment variable *name*. Malformed variable names and references to non-existing variables are left unchanged.

On Windows, `%name%` expansions are supported in addition to `$name` and `${name}`.

Changed in version 3.6: Accepts a *path-like object*.

`os.path.`**`getatime`**(*path*)

Return the time of last access of *path*. The return value is a floating-point number giving the number of seconds since the epoch (see the *time* module). Raise *OSError* if the file does not exist or is inaccessible.

`os.path.`**`getmtime`**(*path*)

Return the time of last modification of *path*. The return value is a floating-point number giving the number of seconds since the epoch (see the *time* module). Raise *OSError* if the file does not exist or is inaccessible.

Changed in version 3.6: Accepts a *path-like object*.

`os.path.`**`getctime`**(*path*)

Return the system's ctime which, on some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time for *path*. The return value is a number giving the number of seconds since the epoch (see the *time* module). Raise *OSError* if the file does not exist or is inaccessible.

Changed in version 3.6: Accepts a *path-like object*.

`os.path.`**`getsize`**(*path*)

Return the size, in bytes, of *path*. Raise *OSError* if the file does not exist or is inaccessible.

Changed in version 3.6: Accepts a *path-like object*.

`os.path.`**`isabs`**(*path*)

Return `True` if *path* is an absolute pathname. On Unix, that means it begins with a slash, on Windows that it begins with two (back)slashes, or a drive letter, colon, and (back)slash together.

Changed in version 3.6: Accepts a *path-like object*.

Changed in version 3.13: On Windows, returns `False` if the given path starts with exactly one (back)slash.

`os.path.`**`isfile`**(*path*)

Return `True` if *path* is an *existing* regular file. This follows symbolic links, so both *islink()* and *isfile()* can be true for the same path.

Changed in version 3.6: Accepts a *path-like object*.

`os.path.`**`isdir`**(*path*)

Return `True` if *path* is an *existing* directory. This follows symbolic links, so both *islink()* and *isdir()* can be true for the same path.

Changed in version 3.6: Accepts a *path-like object*.

`os.path.`**`isjunction`**(*path*)

Return `True` if *path* refers to an *existing* directory entry that is a junction. Always return `False` if junctions are not supported on the current platform.

Added in version 3.12.

os.path.**islink**(*path*)

Return `True` if *path* refers to an *existing* directory entry that is a symbolic link. Always `False` if symbolic links are not supported by the Python runtime.

Changed in version 3.6: Accepts a *path-like object*.

os.path.**ismount**(*path*)

Return `True` if pathname *path* is a *mount point*: a point in a file system where a different file system has been mounted. On POSIX, the function checks whether *path*'s parent, *path*`/..`, is on a different device than *path*, or whether *path*`/..` and *path* point to the same i-node on the same device — this should detect mount points for all Unix and POSIX variants. It is not able to reliably detect bind mounts on the same filesystem. On Windows, a drive letter root and a share UNC are always mount points, and for any other path `GetVolumePathName` is called to see if it is different from the input path.

Changed in version 3.4: Added support for detecting non-root mount points on Windows.

Changed in version 3.6: Accepts a *path-like object*.

os.path.**isdevdrive**(*path*)

Return `True` if pathname *path* is located on a Windows Dev Drive. A Dev Drive is optimized for developer scenarios, and offers faster performance for reading and writing files. It is recommended for use for source code, temporary build directories, package caches, and other IO-intensive operations.

May raise an error for an invalid path, for example, one without a recognizable drive, but returns `False` on platforms that do not support Dev Drives. See the Windows documentation for information on enabling and creating Dev Drives.

Added in version 3.12.

Changed in version 3.13: The function is now available on all platforms, and will always return `False` on those that have no support for Dev Drives

os.path.**isreserved**(*path*)

Return `True` if *path* is a reserved pathname on the current system.

On Windows, reserved filenames include those that end with a space or dot; those that contain colons (i.e. file streams such as "name:stream"), wildcard characters (i.e. `'*?"<>'`), pipe, or ASCII control characters; as well as DOS device names such as "NUL", "CON", "CONIN$", "CONOUT$", "AUX", "PRN", "COM1", and "LPT1".

> **Note**
>
> This function approximates rules for reserved paths on most Windows systems. These rules change over time in various Windows releases. This function may be updated in future Python releases as changes to the rules become broadly available.

Availability: Windows.

Added in version 3.13.

os.path.**join**(*path*, **paths*)

Join one or more path segments intelligently. The return value is the concatenation of *path* and all members of **paths*, with exactly one directory separator following each non-empty part, except the last. That is, the result will only end in a separator if the last part is either empty or ends in a separator. If a segment is an absolute path (which on Windows requires both a drive and a root), then all previous segments are ignored and joining continues from the absolute path segment.

On Windows, the drive is not reset when a rooted path segment (e.g., `r'\foo'`) is encountered. If a segment is on a different drive or is an absolute path, all previous segments are ignored and the drive is reset. Note that since

there is a current directory for each drive, `os.path.join("c:", "foo")` represents a path relative to the current directory on drive `C:` (`c:foo`), not `c:\foo`.

Changed in version 3.6: Accepts a *path-like object* for *path* and *paths*.

`os.path.`**`normcase`**(*path*)

Normalize the case of a pathname. On Windows, convert all characters in the pathname to lowercase, and also convert forward slashes to backward slashes. On other operating systems, return the path unchanged.

Changed in version 3.6: Accepts a *path-like object*.

`os.path.`**`normpath`**(*path*)

Normalize a pathname by collapsing redundant separators and up-level references so that `A//B`, `A/B/`, `A/./B` and `A/foo/../B` all become `A/B`. This string manipulation may change the meaning of a path that contains symbolic links. On Windows, it converts forward slashes to backward slashes. To normalize case, use *`normcase()`*.

> **Note**
>
> On POSIX systems, in accordance with IEEE Std 1003.1 2013 Edition; 4.13 Pathname Resolution, if a pathname begins with exactly two slashes, the first component following the leading characters may be interpreted in an implementation-defined manner, although more than two leading characters shall be treated as a single character.

Changed in version 3.6: Accepts a *path-like object*.

`os.path.`**`realpath`**(*path*, *, *strict=False*)

Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path (if they are supported by the operating system). On Windows, this function will also resolve MS-DOS (also called 8.3) style names such as `C:\\PROGRA~1` to `C:\\Program Files`.

If a path doesn't exist or a symlink loop is encountered, and *strict* is `True`, *`OSError`* is raised. If *strict* is `False` these errors are ignored, and so the result might be missing or otherwise inaccessible.

> **Note**
>
> This function emulates the operating system's procedure for making a path canonical, which differs slightly between Windows and UNIX with respect to how links and subsequent path components interact.
>
> Operating system APIs make paths canonical as needed, so it's not normally necessary to call this function.

Changed in version 3.6: Accepts a *path-like object*.

Changed in version 3.8: Symbolic links and junctions are now resolved on Windows.

Changed in version 3.10: The *strict* parameter was added.

`os.path.`**`relpath`**(*path*, *start=os.curdir*)

Return a relative filepath to *path* either from the current directory or from an optional *start* directory. This is a path computation: the filesystem is not accessed to confirm the existence or nature of *path* or *start*. On Windows, *`ValueError`* is raised when *path* and *start* are on different drives.

start defaults to *`os.curdir`*.

Changed in version 3.6: Accepts a *path-like object*.

os.path.**samefile**(*path1*, *path2*)

Return `True` if both pathname arguments refer to the same file or directory. This is determined by the device number and i-node number and raises an exception if an *os.stat()* call on either pathname fails.

Changed in version 3.2: Added Windows support.

Changed in version 3.4: Windows now uses the same implementation as all other platforms.

Changed in version 3.6: Accepts a *path-like object*.

os.path.**sameopenfile**(*fp1*, *fp2*)

Return `True` if the file descriptors *fp1* and *fp2* refer to the same file.

Changed in version 3.2: Added Windows support.

Changed in version 3.6: Accepts a *path-like object*.

os.path.**samestat**(*stat1*, *stat2*)

Return `True` if the stat tuples *stat1* and *stat2* refer to the same file. These structures may have been returned by *os.fstat()*, *os.lstat()*, or *os.stat()*. This function implements the underlying comparison used by *samefile()* and *sameopenfile()*.

Changed in version 3.4: Added Windows support.

Changed in version 3.6: Accepts a *path-like object*.

os.path.**split**(*path*)

Split the pathname *path* into a pair, `(head, tail)` where *tail* is the last pathname component and *head* is everything leading up to that. The *tail* part will never contain a slash; if *path* ends in a slash, *tail* will be empty. If there is no slash in *path*, *head* will be empty. If *path* is empty, both *head* and *tail* are empty. Trailing slashes are stripped from *head* unless it is the root (one or more slashes only). In all cases, `join(head, tail)` returns a path to the same location as *path* (but the strings may differ). Also see the functions *dirname()* and *basename()*.

Changed in version 3.6: Accepts a *path-like object*.

os.path.**splitdrive**(*path*)

Split the pathname *path* into a pair `(drive, tail)` where *drive* is either a mount point or the empty string. On systems which do not use drive specifications, *drive* will always be the empty string. In all cases, `drive + tail` will be the same as *path*.

On Windows, splits a pathname into drive/UNC sharepoint and relative path.

If the path contains a drive letter, drive will contain everything up to and including the colon:

```
>>> splitdrive("c:/dir")
("c:", "/dir")
```

If the path contains a UNC path, drive will contain the host name and share:

```
>>> splitdrive("//host/computer/dir")
("//host/computer", "/dir")
```

Changed in version 3.6: Accepts a *path-like object*.

os.path.**splitroot**(*path*)

Split the pathname *path* into a 3-item tuple `(drive, root, tail)` where *drive* is a device name or mount point, *root* is a string of separators after the drive, and *tail* is everything after the root. Any of these items may be the empty string. In all cases, `drive + root + tail` will be the same as *path*.

On POSIX systems, *drive* is always empty. The *root* may be empty (if *path* is relative), a single forward slash (if *path* is absolute), or two forward slashes (implementation-defined per IEEE Std 1003.1-2017; 4.13 Pathname Resolution.) For example:

```
>>> splitroot('/home/sam')
('', '/', 'home/sam')
>>> splitroot('//home/sam')
('', '//', 'home/sam')
>>> splitroot('///home/sam')
('', '/', '//home/sam')
```

On Windows, *drive* may be empty, a drive-letter name, a UNC share, or a device name. The *root* may be empty, a forward slash, or a backward slash. For example:

```
>>> splitroot('C:/Users/Sam')
('C:', '/', 'Users/Sam')
>>> splitroot('//Server/Share/Users/Sam')
('//Server/Share', '/', 'Users/Sam')
```

Added in version 3.12.

os.path.**splitext**(*path*)

Split the pathname *path* into a pair `(root, ext)` such that `root + ext == path`, and the extension, *ext*, is empty or begins with a period and contains at most one period.

If the path contains no extension, *ext* will be `''`:

```
>>> splitext('bar')
('bar', '')
```

If the path contains an extension, then *ext* will be set to this extension, including the leading period. Note that previous periods will be ignored:

```
>>> splitext('foo.bar.exe')
('foo.bar', '.exe')
>>> splitext('/foo/bar.exe')
('/foo/bar', '.exe')
```

Leading periods of the last component of the path are considered to be part of the root:

```
>>> splitext('.cshrc')
('.cshrc', '')
>>> splitext('/foo/....jpg')
('/foo/....jpg', '')
```

Changed in version 3.6: Accepts a *path-like object*.

os.path.**supports_unicode_filenames**

`True` if arbitrary Unicode strings can be used as file names (within limitations imposed by the file system).

11.3 `fileinput` — Iterate over lines from multiple input streams

Source code: Lib/fileinput.py

This module implements a helper class and functions to quickly write a loop over standard input or a list of files. If you just want to read or write one file see *open()*.

The typical use is:

```
import fileinput
for line in fileinput.input(encoding="utf-8"):
    process(line)
```

This iterates over the lines of all files listed in `sys.argv[1:]`, defaulting to `sys.stdin` if the list is empty. If a filename is `'-'`, it is also replaced by `sys.stdin` and the optional arguments *mode* and *openhook* are ignored. To specify an alternative list of filenames, pass it as the first argument to *input()*. A single file name is also allowed.

All files are opened in text mode by default, but you can override this by specifying the *mode* parameter in the call to *input()* or *FileInput*. If an I/O error occurs during opening or reading a file, *OSError* is raised.

Changed in version 3.3: *IOError* used to be raised; it is now an alias of *OSError*.

If `sys.stdin` is used more than once, the second and further use will return no lines, except perhaps for interactive use, or if it has been explicitly reset (e.g. using `sys.stdin.seek(0)`).

Empty files are opened and immediately closed; the only time their presence in the list of filenames is noticeable at all is when the last file opened is empty.

Lines are returned with any newlines intact, which means that the last line in a file may not have one.

You can control how files are opened by providing an opening hook via the *openhook* parameter to *fileinput.input()* or *FileInput()*. The hook must be a function that takes two arguments, *filename* and *mode*, and returns an accordingly opened file-like object. If *encoding* and/or *errors* are specified, they will be passed to the hook as additional keyword arguments. This module provides a *hook_compressed()* to support compressed files.

The following function is the primary interface of this module:

`fileinput.`**`input`**(*files=None*, *inplace=False*, *backup=''*, ***, *mode='r'*, *openhook=None*, *encoding=None*, *errors=None*)

Create an instance of the *FileInput* class. The instance will be used as global state for the functions of this module, and is also returned to use during iteration. The parameters to this function will be passed along to the constructor of the *FileInput* class.

The *FileInput* instance can be used as a context manager in the `with` statement. In this example, *input* is closed after the `with` statement is exited, even if an exception occurs:

```
with fileinput.input(files=('spam.txt', 'eggs.txt'), encoding="utf-8") as f:
    for line in f:
        process(line)
```

Changed in version 3.2: Can be used as a context manager.

Changed in version 3.8: The keyword parameters *mode* and *openhook* are now keyword-only.

Changed in version 3.10: The keyword-only parameter *encoding* and *errors* are added.

The following functions use the global state created by *fileinput.input()*; if there is no active state, *RuntimeError* is raised.

fileinput.**filename**()

Return the name of the file currently being read. Before the first line has been read, returns `None`.

fileinput.**fileno**()

Return the integer "file descriptor" for the current file. When no file is opened (before the first line and between files), returns `-1`.

fileinput.**lineno**()

Return the cumulative line number of the line that has just been read. Before the first line has been read, returns `0`. After the last line of the last file has been read, returns the line number of that line.

fileinput.**filelineno**()

Return the line number in the current file. Before the first line has been read, returns `0`. After the last line of the last file has been read, returns the line number of that line within the file.

fileinput.**isfirstline**()

Return `True` if the line just read is the first line of its file, otherwise return `False`.

fileinput.**isstdin**()

Return `True` if the last line was read from `sys.stdin`, otherwise return `False`.

fileinput.**nextfile**()

Close the current file so that the next iteration will read the first line from the next file (if any); lines not read from the file will not count towards the cumulative line count. The filename is not changed until after the first line of the next file has been read. Before the first line has been read, this function has no effect; it cannot be used to skip the first file. After the last line of the last file has been read, this function has no effect.

fileinput.**close**()

Close the sequence.

The class which implements the sequence behavior provided by the module is available for subclassing as well:

class fileinput.**FileInput**(*files=None*, *inplace=False*, *backup=''*, ***, *mode='r'*, *openhook=None*, *encoding=None*, *errors=None*)

Class *FileInput* is the implementation; its methods *filename()*, *fileno()*, *lineno()*, *filelineno()*, *isfirstline()*, *isstdin()*, *nextfile()* and *close()* correspond to the functions of the same name in the module. In addition it is *iterable* and has a *readline()* method which returns the next input line. The sequence must be accessed in strictly sequential order; random access and *readline()* cannot be mixed.

With *mode* you can specify which file mode will be passed to *open()*. It must be one of `'r'` and `'rb'`.

The *openhook*, when given, must be a function that takes two arguments, *filename* and *mode*, and returns an accordingly opened file-like object. You cannot use *inplace* and *openhook* together.

You can specify *encoding* and *errors* that is passed to *open()* or *openhook*.

A *FileInput* instance can be used as a context manager in the `with` statement. In this example, *input* is closed after the `with` statement is exited, even if an exception occurs:

```
with FileInput(files=('spam.txt', 'eggs.txt')) as input:
    process(input)
```

Changed in version 3.2: Can be used as a context manager.

Changed in version 3.8: The keyword parameter *mode* and *openhook* are now keyword-only.

Changed in version 3.10: The keyword-only parameter *encoding* and *errors* are added.

Changed in version 3.11: The `'rU'` and `'U'` modes and the `__getitem__()` method have been removed.

Optional in-place filtering: if the keyword argument `inplace=True` is passed to *fileinput.input()* or to the *FileInput* constructor, the file is moved to a backup file and standard output is directed to the input file (if a file of the same name as the backup file already exists, it will be replaced silently). This makes it possible to write a filter that rewrites its input file in place. If the *backup* parameter is given (typically as `backup='.<some extension>'`), it specifies the extension for the backup file, and the backup file remains around; by default, the extension is `'.bak'` and it is deleted when the output file is closed. In-place filtering is disabled when standard input is read.

The two following opening hooks are provided by this module:

`fileinput.`**`hook_compressed`**(*filename*, *mode*, *, *encoding=None*, *errors=None*)

Transparently opens files compressed with gzip and bzip2 (recognized by the extensions `'.gz'` and `'.bz2'`) using the *gzip* and *bz2* modules. If the filename extension is not `'.gz'` or `'.bz2'`, the file is opened normally (ie, using *open()* without any decompression).

The *encoding* and *errors* values are passed to *io.TextIOWrapper* for compressed files and open for normal files.

Usage example: `fi = fileinput.FileInput(openhook=fileinput.hook_compressed, encoding="utf-8")`

Changed in version 3.10: The keyword-only parameter *encoding* and *errors* are added.

`fileinput.`**`hook_encoded`**(*encoding*, *errors=None*)

Returns a hook which opens each file with *open()*, using the given *encoding* and *errors* to read the file.

Usage example: `fi = fileinput.FileInput(openhook=fileinput.hook_encoded("utf-8", "surrogateescape"))`

Changed in version 3.6: Added the optional *errors* parameter.

Deprecated since version 3.10: This function is deprecated since *fileinput.input()* and *FileInput* now have *encoding* and *errors* parameters.

11.4 `stat` — Interpreting `stat()` results

Source code: Lib/stat.py

The *stat* module defines constants and functions for interpreting the results of *os.stat()*, *os.fstat()* and *os.lstat()* (if they exist). For complete details about the `stat()`, `fstat()` and `lstat()` calls, consult the documentation for your system.

Changed in version 3.4: The stat module is backed by a C implementation.

The *stat* module defines the following functions to test for specific file types:

`stat.`**`S_ISDIR`**(*mode*)

Return non-zero if the mode is from a directory.

`stat.`**`S_ISCHR`**(*mode*)

Return non-zero if the mode is from a character special device file.

`stat.`**`S_ISBLK`**(*mode*)

Return non-zero if the mode is from a block special device file.

`stat.`**`S_ISREG`**(*mode*)

Return non-zero if the mode is from a regular file.

stat.**S_ISFIFO**(*mode*)
Return non-zero if the mode is from a FIFO (named pipe).

stat.**S_ISLNK**(*mode*)
Return non-zero if the mode is from a symbolic link.

stat.**S_ISSOCK**(*mode*)
Return non-zero if the mode is from a socket.

stat.**S_ISDOOR**(*mode*)
Return non-zero if the mode is from a door.

Added in version 3.4.

stat.**S_ISPORT**(*mode*)
Return non-zero if the mode is from an event port.

Added in version 3.4.

stat.**S_ISWHT**(*mode*)
Return non-zero if the mode is from a whiteout.

Added in version 3.4.

Two additional functions are defined for more general manipulation of the file's mode:

stat.**S_IMODE**(*mode*)
Return the portion of the file's mode that can be set by *os.chmod()*—that is, the file's permission bits, plus the sticky bit, set-group-id, and set-user-id bits (on systems that support them).

stat.**S_IFMT**(*mode*)
Return the portion of the file's mode that describes the file type (used by the `S_IS*()` functions above).

Normally, you would use the `os.path.is*()` functions for testing the type of a file; the functions here are useful when you are doing multiple tests of the same file and wish to avoid the overhead of the `stat()` system call for each test. These are also useful when checking for information about a file that isn't handled by *os.path*, like the tests for block and character devices.

Example:

```
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top,
       calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.lstat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print('Skipping %s' % pathname)
```

(continues on next page)

(continued from previous page)

```
def visitfile(file):
    print('visiting', file)

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

An additional utility function is provided to convert a file's mode in a human readable string:

stat.**filemode**(*mode*)

Convert a file's mode to a string of the form '-rwxrwxrwx'.

Added in version 3.3.

Changed in version 3.4: The function supports *S_IFDOOR*, *S_IFPORT* and *S_IFWHT*.

All the variables below are simply symbolic indexes into the 10-tuple returned by *os.stat()*, *os.fstat()* or *os.lstat()*.

stat.**ST_MODE**

Inode protection mode.

stat.**ST_INO**

Inode number.

stat.**ST_DEV**

Device inode resides on.

stat.**ST_NLINK**

Number of links to the inode.

stat.**ST_UID**

User id of the owner.

stat.**ST_GID**

Group id of the owner.

stat.**ST_SIZE**

Size in bytes of a plain file; amount of data waiting on some special files.

stat.**ST_ATIME**

Time of last access.

stat.**ST_MTIME**

Time of last modification.

stat.**ST_CTIME**

The "ctime" as reported by the operating system. On some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time (see platform documentation for details).

The interpretation of "file size" changes according to the file type. For plain files this is the size of the file in bytes. For FIFOs and sockets under most flavors of Unix (including Linux in particular), the "size" is the number of bytes waiting to be read at the time of the call to *os.stat()*, *os.fstat()*, or *os.lstat()*; this can sometimes be useful, especially for polling one of these special files after a non-blocking open. The meaning of the size field for other character and block devices varies more, depending on the implementation of the underlying system call.

The variables below define the flags used in the *ST_MODE* field.

Use of the functions above is more portable than use of the first set of flags:

stat.**S_IFSOCK**
Socket.

stat.**S_IFLNK**
Symbolic link.

stat.**S_IFREG**
Regular file.

stat.**S_IFBLK**
Block device.

stat.**S_IFDIR**
Directory.

stat.**S_IFCHR**
Character device.

stat.**S_IFIFO**
FIFO.

stat.**S_IFDOOR**
Door.

Added in version 3.4.

stat.**S_IFPORT**
Event port.

Added in version 3.4.

stat.**S_IFWHT**
Whiteout.

Added in version 3.4.

Note

S_IFDOOR, *S_IFPORT* or *S_IFWHT* are defined as 0 when the platform does not have support for the file types.

The following flags can also be used in the *mode* argument of *os.chmod()*:

stat.**S_ISUID**
Set UID bit.

stat.**S_ISGID**
Set-group-ID bit. This bit has several special uses. For a directory it indicates that BSD semantics is to be used for that directory: files created there inherit their group ID from the directory, not from the effective group ID of the creating process, and directories created there will also get the *S_ISGID* bit set. For a file that does not have the group execution bit (*S_IXGRP*) set, the set-group-ID bit indicates mandatory file/record locking (see also *S_ENFMT*).

stat.**S_ISVTX**
Sticky bit. When this bit is set on a directory it means that a file in that directory can be renamed or deleted only by the owner of the file, by the owner of the directory, or by a privileged process.

stat.**S_IRWXU**
Mask for file owner permissions.

stat.**S_IRUSR**
Owner has read permission.

stat.**S_IWUSR**
Owner has write permission.

stat.**S_IXUSR**
Owner has execute permission.

stat.**S_IRWXG**
Mask for group permissions.

stat.**S_IRGRP**
Group has read permission.

stat.**S_IWGRP**
Group has write permission.

stat.**S_IXGRP**
Group has execute permission.

stat.**S_IRWXO**
Mask for permissions for others (not in group).

stat.**S_IROTH**
Others have read permission.

stat.**S_IWOTH**
Others have write permission.

stat.**S_IXOTH**
Others have execute permission.

stat.**S_ENFMT**
System V file locking enforcement. This flag is shared with *S_ISGID*: file/record locking is enforced on files that do not have the group execution bit (*S_IXGRP*) set.

stat.**S_IREAD**
Unix V7 synonym for *S_IRUSR*.

stat.**S_IWRITE**
Unix V7 synonym for *S_IWUSR*.

stat.**S_IEXEC**
Unix V7 synonym for *S_IXUSR*.

The following flags can be used in the *flags* argument of *os.chflags()*:

stat.**UF_SETTABLE**
All user settable flags.

Added in version 3.13.

stat.**UF_NODUMP**
Do not dump the file.

stat.**UF_IMMUTABLE**
The file may not be changed.

stat.**UF_APPEND**
The file may only be appended to.

stat.**UF_OPAQUE**
The directory is opaque when viewed through a union stack.

stat.**UF_NOUNLINK**
The file may not be renamed or deleted.

stat.**UF_COMPRESSED**
The file is stored compressed (macOS 10.6+).

stat.**UF_TRACKED**
Used for handling document IDs (macOS)

Added in version 3.13.

stat.**UF_DATAVAULT**
The file needs an entitlement for reading or writing (macOS 10.13+)

Added in version 3.13.

stat.**UF_HIDDEN**
The file should not be displayed in a GUI (macOS 10.5+).

stat.**SF_SETTABLE**
All super-user changeable flags

Added in version 3.13.

stat.**SF_SUPPORTED**
All super-user supported flags

Availability: macOS

Added in version 3.13.

stat.**SF_SYNTHETIC**
All super-user read-only synthetic flags

Availability: macOS

Added in version 3.13.

stat.**SF_ARCHIVED**
The file may be archived.

stat.**SF_IMMUTABLE**
The file may not be changed.

stat.**SF_APPEND**
The file may only be appended to.

stat.**SF_RESTRICTED**
The file needs an entitlement to write to (macOS 10.13+)

Added in version 3.13.

stat.**SF_NOUNLINK**

The file may not be renamed or deleted.

stat.**SF_SNAPSHOT**

The file is a snapshot file.

stat.**SF_FIRMLINK**

The file is a firmlink (macOS 10.15+)

Added in version 3.13.

stat.**SF_DATALESS**

The file is a dataless object (macOS 10.15+)

Added in version 3.13.

See the *BSD or macOS systems man page *chflags(2)* for more information.

On Windows, the following file attribute constants are available for use when testing bits in the `st_file_attributes` member returned by *os.stat()*. See the Windows API documentation for more detail on the meaning of these constants.

stat.**FILE_ATTRIBUTE_ARCHIVE**

stat.**FILE_ATTRIBUTE_COMPRESSED**

stat.**FILE_ATTRIBUTE_DEVICE**

stat.**FILE_ATTRIBUTE_DIRECTORY**

stat.**FILE_ATTRIBUTE_ENCRYPTED**

stat.**FILE_ATTRIBUTE_HIDDEN**

stat.**FILE_ATTRIBUTE_INTEGRITY_STREAM**

stat.**FILE_ATTRIBUTE_NORMAL**

stat.**FILE_ATTRIBUTE_NOT_CONTENT_INDEXED**

stat.**FILE_ATTRIBUTE_NO_SCRUB_DATA**

stat.**FILE_ATTRIBUTE_OFFLINE**

stat.**FILE_ATTRIBUTE_READONLY**

stat.**FILE_ATTRIBUTE_REPARSE_POINT**

stat.**FILE_ATTRIBUTE_SPARSE_FILE**

stat.**FILE_ATTRIBUTE_SYSTEM**

stat.**FILE_ATTRIBUTE_TEMPORARY**

stat.**FILE_ATTRIBUTE_VIRTUAL**

Added in version 3.5.

On Windows, the following constants are available for comparing against the `st_reparse_tag` member returned by *os.lstat()*. These are well-known constants, but are not an exhaustive list.

stat.**IO_REPARSE_TAG_SYMLINK**

stat.**IO_REPARSE_TAG_MOUNT_POINT**

stat.**IO_REPARSE_TAG_APPEXECLINK**

Added in version 3.8.

11.5 filecmp — File and Directory Comparisons

Source code: Lib/filecmp.py

The *filecmp* module defines functions to compare files and directories, with various optional time/correctness trade-offs. For comparing files, see also the *difflib* module.

The *filecmp* module defines the following functions:

filecmp.**cmp**(*f1*, *f2*, *shallow=True*)

Compare the files named *f1* and *f2*, returning `True` if they seem equal, `False` otherwise.

If *shallow* is true and the *os.stat()* signatures (file type, size, and modification time) of both files are identical, the files are taken to be equal.

Otherwise, the files are treated as different if their sizes or contents differ.

Note that no external programs are called from this function, giving it portability and efficiency.

This function uses a cache for past comparisons and the results, with cache entries invalidated if the *os.stat()* information for the file changes. The entire cache may be cleared using *clear_cache()*.

filecmp.**cmpfiles**(*dir1*, *dir2*, *common*, *shallow=True*)

Compare the files in the two directories *dir1* and *dir2* whose names are given by *common*.

Returns three lists of file names: *match*, *mismatch*, *errors*. *match* contains the list of files that match, *mismatch* contains the names of those that don't, and *errors* lists the names of files which could not be compared. Files are listed in *errors* if they don't exist in one of the directories, the user lacks permission to read them or if the comparison could not be done for some other reason.

The *shallow* parameter has the same meaning and default value as for *filecmp.cmp()*.

For example, `cmpfiles('a', 'b', ['c', 'd/e'])` will compare `a/c` with `b/c` and `a/d/e` with `b/d/e`. `'c'` and `'d/e'` will each be in one of the three returned lists.

filecmp.**clear_cache**()

Clear the filecmp cache. This may be useful if a file is compared so quickly after it is modified that it is within the mtime resolution of the underlying filesystem.

Added in version 3.4.

11.5.1 The dircmp class

class filecmp.**dircmp**(*a*, *b*, *ignore=None*, *hide=None*, *, *shallow=True*)

Construct a new directory comparison object, to compare the directories *a* and *b*. *ignore* is a list of names to ignore, and defaults to *filecmp.DEFAULT_IGNORES*. *hide* is a list of names to hide, and defaults to `[os.curdir, os.pardir]`.

The *dircmp* class compares files by doing *shallow* comparisons as described for *filecmp.cmp()* by default using the *shallow* parameter.

Changed in version 3.13: Added the *shallow* parameter.

The *dircmp* class provides the following methods:

report()

Print (to *sys.stdout*) a comparison between *a* and *b*.

report_partial_closure()

Print a comparison between *a* and *b* and common immediate subdirectories.

report_full_closure()

Print a comparison between *a* and *b* and common subdirectories (recursively).

The *dircmp* class offers a number of interesting attributes that may be used to get various bits of information about the directory trees being compared.

Note that via __getattr__() hooks, all attributes are computed lazily, so there is no speed penalty if only those attributes which are lightweight to compute are used.

left

The directory *a*.

right

The directory *b*.

left_list

Files and subdirectories in *a*, filtered by *hide* and *ignore*.

right_list

Files and subdirectories in *b*, filtered by *hide* and *ignore*.

common

Files and subdirectories in both *a* and *b*.

left_only

Files and subdirectories only in *a*.

right_only

Files and subdirectories only in *b*.

common_dirs

Subdirectories in both *a* and *b*.

common_files

Files in both *a* and *b*.

common_funny

Names in both *a* and *b*, such that the type differs between the directories, or names for which *os.stat()* reports an error.

same_files

Files which are identical in both *a* and *b*, using the class's file comparison operator.

diff_files

Files which are in both *a* and *b*, whose contents differ according to the class's file comparison operator.

funny_files

Files which are in both *a* and *b*, but could not be compared.

subdirs

A dictionary mapping names in *common_dirs* to *dircmp* instances (or MyDirCmp instances if this instance is of type MyDirCmp, a subclass of *dircmp*).

Changed in version 3.10: Previously entries were always *dircmp* instances. Now entries are the same type as *self*, if *self* is a subclass of *dircmp*.

filecmp.**DEFAULT_IGNORES**

Added in version 3.4.

List of directories ignored by *dircmp* by default.

Here is a simplified example of using the `subdirs` attribute to search recursively through two directories to show common different files:

```
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
...         print("diff_file %s found in %s and %s" % (name, dcmp.left,
...               dcmp.right))
...     for sub_dcmp in dcmp.subdirs.values():
...         print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)
```

11.6 `tempfile` — Generate temporary files and directories

Source code: Lib/tempfile.py

This module creates temporary files and directories. It works on all supported platforms. *TemporaryFile*, *NamedTemporaryFile*, *TemporaryDirectory*, and *SpooledTemporaryFile* are high-level interfaces which provide automatic cleanup and can be used as *context managers*. *mkstemp()* and *mkdtemp()* are lower-level functions which require manual cleanup.

All the user-callable functions and constructors take additional arguments which allow direct control over the location and name of temporary files and directories. Files names used by this module include a string of random characters which allows those files to be securely created in shared temporary directories. To maintain backward compatibility, the argument order is somewhat odd; it is recommended to use keyword arguments for clarity.

The module defines the following user-callable items:

tempfile.**TemporaryFile**(*mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, ***, *errors=None*)

Return a *file-like object* that can be used as a temporary storage area. The file is created securely, using the same rules as *mkstemp()*. It will be destroyed as soon as it is closed (including an implicit close when the object is garbage collected). Under Unix, the directory entry for the file is either not created at all or is removed immediately after the file is created. Other platforms do not support this; your code should not rely on a temporary file created using this function having or not having a visible name in the file system.

The resulting object can be used as a *context manager* (see *Examples*). On completion of the context or destruction of the file object the temporary file will be removed from the filesystem.

The *mode* parameter defaults to `'w+b'` so that the file created can be read and written without being closed. Binary mode is used so that it behaves consistently on all platforms without regard for the data that is stored. *buffering*, *encoding*, *errors* and *newline* are interpreted as for *open()*.

The *dir*, *prefix* and *suffix* parameters have the same meaning and defaults as with *mkstemp()*.

The returned object is a true file object on POSIX platforms. On other platforms, it is a file-like object whose `file` attribute is the underlying true file object.

The *os.O_TMPFILE* flag is used if it is available and works (Linux-specific, requires Linux kernel 3.11 or later).

On platforms that are neither Posix nor Cygwin, TemporaryFile is an alias for NamedTemporaryFile.

Raises an *auditing event* `tempfile.mkstemp` with argument `fullpath`.

Changed in version 3.5: The `os.O_TMPFILE` flag is now used if available.

Changed in version 3.8: Added *errors* parameter.

tempfile.NamedTemporaryFile(*mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, *delete=True*, ***, *errors=None*, *delete_on_close=True*)

This function operates exactly as `TemporaryFile()` does, except the following differences:

- This function returns a file that is guaranteed to have a visible name in the file system.
- To manage the named file, it extends the parameters of `TemporaryFile()` with *delete* and *delete_on_close* parameters that determine whether and how the named file should be automatically deleted.

The returned object is always a *file-like object* whose `file` attribute is the underlying true file object. This file-like object can be used in a `with` statement, just like a normal file. The name of the temporary file can be retrieved from the `name` attribute of the returned file-like object. On Unix, unlike with the `TemporaryFile()`, the directory entry does not get unlinked immediately after the file creation.

If *delete* is true (the default) and *delete_on_close* is true (the default), the file is deleted as soon as it is closed. If *delete* is true and *delete_on_close* is false, the file is deleted on context manager exit only, or else when the *file-like object* is finalized. Deletion is not always guaranteed in this case (see `object.__del__()`). If *delete* is false, the value of *delete_on_close* is ignored.

Therefore to use the name of the temporary file to reopen the file after closing it, either make sure not to delete the file upon closure (set the *delete* parameter to be false) or, in case the temporary file is created in a `with` statement, set the *delete_on_close* parameter to be false. The latter approach is recommended as it provides assistance in automatic cleaning of the temporary file upon the context manager exit.

Opening the temporary file again by its name while it is still open works as follows:

- On POSIX the file can always be opened again.
- On Windows, make sure that at least one of the following conditions are fulfilled:
  - *delete* is false
  - additional open shares delete access (e.g. by calling `os.open()` with the flag `O_TEMPORARY`)
  - *delete* is true but *delete_on_close* is false. Note, that in this case the additional opens that do not share delete access (e.g. created via builtin `open()`) must be closed before exiting the context manager, else the `os.unlink()` call on context manager exit will fail with a `PermissionError`.

On Windows, if *delete_on_close* is false, and the file is created in a directory for which the user lacks delete access, then the `os.unlink()` call on exit of the context manager will fail with a `PermissionError`. This cannot happen when *delete_on_close* is true because delete access is requested by the open, which fails immediately if the requested access is not granted.

On POSIX (only), a process that is terminated abruptly with SIGKILL cannot automatically delete any NamedTemporaryFiles it created.

Raises an *auditing event* `tempfile.mkstemp` with argument `fullpath`.

Changed in version 3.8: Added *errors* parameter.

Changed in version 3.12: Added *delete_on_close* parameter.

class tempfile.**SpooledTemporaryFile**(*max_size=0*, *mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, ***, *errors=None*)

This class operates exactly as *TemporaryFile()* does, except that data is spooled in memory until the file size exceeds *max_size*, or until the file's *fileno()* method is called, at which point the contents are written to disk and operation proceeds as with *TemporaryFile()*.

rollover()

The resulting file has one additional method, `rollover()`, which causes the file to roll over to an on-disk file regardless of its size.

The returned object is a file-like object whose `_file` attribute is either an *io.BytesIO* or *io.TextIOWrapper* object (depending on whether binary or text *mode* was specified) or a true file object, depending on whether *rollover()* has been called. This file-like object can be used in a `with` statement, just like a normal file.

Changed in version 3.3: the truncate method now accepts a *size* argument.

Changed in version 3.8: Added *errors* parameter.

Changed in version 3.11: Fully implements the *io.BufferedIOBase* and *io.TextIOBase* abstract base classes (depending on whether binary or text *mode* was specified).

class tempfile.**TemporaryDirectory**(*suffix=None*, *prefix=None*, *dir=None*, *ignore_cleanup_errors=False*, *, *delete=True*)

This class securely creates a temporary directory using the same rules as *mkdtemp()*. The resulting object can be used as a *context manager* (see *Examples*). On completion of the context or destruction of the temporary directory object, the newly created temporary directory and all its contents are removed from the filesystem.

name

The directory name can be retrieved from the `name` attribute of the returned object. When the returned object is used as a *context manager*, the `name` will be assigned to the target of the `as` clause in the `with` statement, if there is one.

cleanup()

The directory can be explicitly cleaned up by calling the `cleanup()` method. If *ignore_cleanup_errors* is true, any unhandled exceptions during explicit or implicit cleanup (such as a *PermissionError* removing open files on Windows) will be ignored, and the remaining removable items deleted on a "best-effort" basis. Otherwise, errors will be raised in whatever context cleanup occurs (the `cleanup()` call, exiting the context manager, when the object is garbage-collected or during interpreter shutdown).

The *delete* parameter can be used to disable cleanup of the directory tree upon exiting the context. While it may seem unusual for a context manager to disable the action taken when exiting the context, it can be useful during debugging or when you need your cleanup behavior to be conditional based on other logic.

Raises an *auditing event* `tempfile.mkdtemp` with argument `fullpath`.

Added in version 3.2.

Changed in version 3.10: Added *ignore_cleanup_errors* parameter.

Changed in version 3.12: Added the *delete* parameter.

tempfile.**mkstemp**(*suffix=None*, *prefix=None*, *dir=None*, *text=False*)

Creates a temporary file in the most secure manner possible. There are no race conditions in the file's creation, assuming that the platform properly implements the *os.O_EXCL* flag for *os.open()*. The file is readable and writable only by the creating user ID. If the platform uses permission bits to indicate whether a file is executable, the file is executable by no one. The file descriptor is not inherited by child processes.

Unlike *TemporaryFile()*, the user of *mkstemp()* is responsible for deleting the temporary file when done with it.

If *suffix* is not `None`, the file name will end with that suffix, otherwise there will be no suffix. *mkstemp()* does not put a dot between the file name and the suffix; if you need one, put it at the beginning of *suffix*.

If *prefix* is not `None`, the file name will begin with that prefix; otherwise, a default prefix is used. The default is the return value of *gettempprefix()* or *gettempprefixb()*, as appropriate.

If *dir* is not `None`, the file will be created in that directory; otherwise, a default directory is used. The default directory is chosen from a platform-dependent list, but the user of the application can control the directory location by setting the *TMPDIR*, *TEMP* or *TMP* environment variables. There is thus no guarantee that the generated filename will have any nice properties, such as not requiring quoting when passed to external commands via `os.popen()`.

If any of *suffix*, *prefix*, and *dir* are not `None`, they must be the same type. If they are bytes, the returned name will be bytes instead of str. If you want to force a bytes return value with otherwise default behavior, pass `suffix=b''`.

If *text* is specified and true, the file is opened in text mode. Otherwise, (the default) the file is opened in binary mode.

mkstemp() returns a tuple containing an OS-level handle to an open file (as would be returned by *os.open()*) and the absolute pathname of that file, in that order.

Raises an *auditing event* `tempfile.mkstemp` with argument `fullpath`.

Changed in version 3.5: *suffix*, *prefix*, and *dir* may now be supplied in bytes in order to obtain a bytes return value. Prior to this, only str was allowed. *suffix* and *prefix* now accept and default to `None` to cause an appropriate default value to be used.

Changed in version 3.6: The *dir* parameter now accepts a *path-like object*.

`tempfile.`**`mkdtemp`**(*suffix=None*, *prefix=None*, *dir=None*)

Creates a temporary directory in the most secure manner possible. There are no race conditions in the directory's creation. The directory is readable, writable, and searchable only by the creating user ID.

The user of *mkdtemp()* is responsible for deleting the temporary directory and its contents when done with it.

The *prefix*, *suffix*, and *dir* arguments are the same as for *mkstemp()*.

mkdtemp() returns the absolute pathname of the new directory.

Raises an *auditing event* `tempfile.mkdtemp` with argument `fullpath`.

Changed in version 3.5: *suffix*, *prefix*, and *dir* may now be supplied in bytes in order to obtain a bytes return value. Prior to this, only str was allowed. *suffix* and *prefix* now accept and default to `None` to cause an appropriate default value to be used.

Changed in version 3.6: The *dir* parameter now accepts a *path-like object*.

Changed in version 3.12: *mkdtemp()* now always returns an absolute path, even if *dir* is relative.

`tempfile.`**`gettempdir`**()

Return the name of the directory used for temporary files. This defines the default value for the *dir* argument to all functions in this module.

Python searches a standard list of directories to find one which the calling user can create files in. The list is:

1. The directory named by the `TMPDIR` environment variable.
2. The directory named by the `TEMP` environment variable.
3. The directory named by the `TMP` environment variable.
4. A platform-specific location:
   - On Windows, the directories `C:\TEMP`, `C:\TMP`, `\TEMP`, and `\TMP`, in that order.
   - On all other platforms, the directories `/tmp`, `/var/tmp`, and `/usr/tmp`, in that order.

5. As a last resort, the current working directory.

The result of this search is cached, see the description of *tempdir* below.

Changed in version 3.10: Always returns a str. Previously it would return any *tempdir* value regardless of type so long as it was not None.

tempfile.**gettempdirb**()

Same as *gettempdir()* but the return value is in bytes.

Added in version 3.5.

tempfile.**gettempprefix**()

Return the filename prefix used to create temporary files. This does not contain the directory component.

tempfile.**gettempprefixb**()

Same as *gettempprefix()* but the return value is in bytes.

Added in version 3.5.

The module uses a global variable to store the name of the directory used for temporary files returned by *gettempdir()*. It can be set directly to override the selection process, but this is discouraged. All functions in this module take a *dir* argument which can be used to specify the directory. This is the recommended approach that does not surprise other unsuspecting code by changing global API behavior.

tempfile.**tempdir**

When set to a value other than None, this variable defines the default value for the *dir* argument to the functions defined in this module, including its type, bytes or str. It cannot be a *path-like object*.

If tempdir is None (the default) at any call to any of the above functions except *gettempprefix()* it is initialized following the algorithm described in *gettempdir()*.

Note

Beware that if you set tempdir to a bytes value, there is a nasty side effect: The global default return type of *mkstemp()* and *mkdtemp()* changes to bytes when no explicit prefix, suffix, or dir arguments of type str are supplied. Please do not write code expecting or depending on this. This awkward behavior is maintained for compatibility with the historical implementation.

11.6.1 Examples

Here are some examples of typical usage of the *tempfile* module:

```
>>> import tempfile

# create a temporary file and write some data to it
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# read data from file
>>> fp.seek(0)
>>> fp.read()
b'Hello world!'
# close the file, it will be removed
>>> fp.close()

# create a temporary file using a context manager
```

(continues on next page)

(continued from previous page)

```
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
b'Hello world!'
>>>
# file is now closed and removed

# create a temporary file using a context manager
# close the file, use the name to open the file again
>>> with tempfile.NamedTemporaryFile(delete_on_close=False) as fp:
...     fp.write(b'Hello world!')
...     fp.close()
... # the file is closed, but not removed
... # open the file again by using its name
...     with open(fp.name, mode='rb') as f:
...         f.read()
b'Hello world!'
>>>
# file is now removed

# create a temporary directory using the context manager
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...     print('created temporary directory', tmpdirname)
>>>
# directory and contents have been removed
```

11.6.2 Deprecated functions and variables

A historical way to create temporary files was to first generate a file name with the *mktemp()* function and then create a file using this name. Unfortunately this is not secure, because a different process may create a file with this name in the time between the call to *mktemp()* and the subsequent attempt to create the file by the first process. The solution is to combine the two steps and create the file immediately. This approach is used by *mkstemp()* and the other functions described above.

tempfile.**mktemp**(*suffix=''*, *prefix='tmp'*, *dir=None*)

Deprecated since version 2.3: Use *mkstemp()* instead.

Return an absolute pathname of a file that did not exist at the time the call is made. The *prefix*, *suffix*, and *dir* arguments are similar to those of *mkstemp()*, except that bytes file names, `suffix=None` and `prefix=None` are not supported.

Warning

Use of this function may introduce a security hole in your program. By the time you get around to doing anything with the file name it returns, someone else may have beaten you to the punch. *mktemp()* usage can be replaced easily with *NamedTemporaryFile()*, passing it the `delete=False` parameter:

```
>>> f = NamedTemporaryFile(delete=False)
>>> f.name
'/tmp/tmptjujjt'
>>> f.write(b"Hello World!\n")
13
>>> f.close()
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```

11.7 glob — Unix style pathname pattern expansion

Source code: Lib/glob.py

The *glob* module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell, although results are returned in arbitrary order. No tilde expansion is done, but `*`, `?`, and character ranges expressed with `[]` will be correctly matched. This is done by using the *os.scandir()* and *fnmatch.fnmatch()* functions in concert, and not by actually invoking a subshell.

Note that files beginning with a dot (`.`) can only be matched by patterns that also start with a dot, unlike *fnmatch.fnmatch()* or *pathlib.Path.glob()*. (For tilde and shell variable expansion, use *os.path.expanduser()* and *os.path.expandvars()*.)

For a literal match, wrap the meta-characters in brackets. For example, `'[?]'` matches the character `'?'`.

The *glob* module defines the following functions:

glob.**glob**(*pathname*, *, *root_dir=None*, *dir_fd=None*, *recursive=False*, *include_hidden=False*)

Return a possibly empty list of path names that match *pathname*, which must be a string containing a path specification. *pathname* can be either absolute (like `/usr/src/Python-1.5/Makefile`) or relative (like `../../Tools/*/*.gif`), and can contain shell-style wildcards. Broken symlinks are included in the results (as in the shell). Whether or not the results are sorted depends on the file system. If a file that satisfies conditions is removed or added during the call of this function, whether a path name for that file will be included is unspecified.

If *root_dir* is not `None`, it should be a *path-like object* specifying the root directory for searching. It has the same effect on *glob()* as changing the current directory before calling it. If *pathname* is relative, the result will contain paths relative to *root_dir*.

This function can support *paths relative to directory descriptors* with the *dir_fd* parameter.

If *recursive* is true, the pattern "`**`" will match any files and zero or more directories, subdirectories and symbolic links to directories. If the pattern is followed by an *os.sep* or *os.altsep* then files will not match.

If *include_hidden* is true, "`**`" pattern will match hidden directories.

Raises an *auditing event* `glob.glob` with arguments `pathname`, `recursive`.

Raises an *auditing event* `glob.glob/2` with arguments `pathname`, `recursive`, `root_dir`, `dir_fd`.

> **Note**
>
> Using the "`**`" pattern in large directory trees may consume an inordinate amount of time.

> **Note**
>
> This function may return duplicate path names if *pathname* contains multiple "`**`" patterns and *recursive* is true.

Changed in version 3.5: Support for recursive globs using "`**`".

Changed in version 3.10: Added the *root_dir* and *dir_fd* parameters.

Changed in version 3.11: Added the *include_hidden* parameter.

glob.**iglob**(*pathname*, ***, *root_dir=None*, *dir_fd=None*, *recursive=False*, *include_hidden=False*)

Return an *iterator* which yields the same values as `glob()` without actually storing them all simultaneously.

Raises an *auditing event* `glob.glob` with arguments `pathname`, `recursive`.

Raises an *auditing event* `glob.glob/2` with arguments `pathname`, `recursive`, `root_dir`, `dir_fd`.

Note

This function may return duplicate path names if *pathname* contains multiple "`**`" patterns and *recursive* is true.

Changed in version 3.5: Support for recursive globs using "`**`".

Changed in version 3.10: Added the *root_dir* and *dir_fd* parameters.

Changed in version 3.11: Added the *include_hidden* parameter.

glob.**escape**(*pathname*)

Escape all special characters (`'?'`, `'*'` and `'['`). This is useful if you want to match an arbitrary literal string that may have special characters in it. Special characters in drive/UNC sharepoints are not escaped, e.g. on Windows `escape('//?/c:/Quo vadis?.txt')` returns `'//?/c:/Quo vadis[?].txt'`.

Added in version 3.4.

glob.**translate**(*pathname*, ***, *recursive=False*, *include_hidden=False*, *seps=None*)

Convert the given path specification to a regular expression for use with `re.match()`. The path specification can contain shell-style wildcards.

For example:

```
>>> import glob, re
>>>
>>> regex = glob.translate('**/*.txt', recursive=True, include_hidden=True)
>>> regex
'(?s:(?:.+/)?[^/]*\\.txt)\\Z'
>>> reobj = re.compile(regex)
>>> reobj.match('foo/bar/baz.txt')
<re.Match object; span=(0, 15), match='foo/bar/baz.txt'>
```

Path separators and segments are meaningful to this function, unlike `fnmatch.translate()`. By default wildcards do not match path separators, and `*` pattern segments match precisely one path segment.

If *recursive* is true, the pattern segment "`**`" will match any number of path segments.

If *include_hidden* is true, wildcards can match path segments that start with a dot (`.`).

A sequence of path separators may be supplied to the *seps* argument. If not given, `os.sep` and `altsep` (if available) are used.

See also

`pathlib.PurePath.full_match()` and `pathlib.Path.glob()` methods, which call this function to implement pattern matching and globbing.

Added in version 3.13.

11.7.1 Examples

Consider a directory containing the following files: `1.gif`, `2.txt`, `card.gif` and a subdirectory `sub` which contains only the file `3.txt`. *glob()* will produce the following results. Notice how any leading components of the path are preserved.

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
>>> glob.glob('**/*.txt', recursive=True)
['2.txt', 'sub/3.txt']
>>> glob.glob('./**/', recursive=True)
['./', './sub/']
```

If the directory contains files starting with `.` they won't be matched by default. For example, consider a directory containing `card.gif` and `.card.gif`:

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

See also

The *fnmatch* module offers shell-style filename (not path) expansion.

See also

The *pathlib* module offers high-level path objects.

11.8 fnmatch — Unix filename pattern matching

Source code: Lib/fnmatch.py

This module provides support for Unix shell-style wildcards, which are *not* the same as regular expressions (which are documented in the *re* module). The special characters used in shell-style wildcards are:

Pattern	Meaning
`*`	matches everything
`?`	matches any single character
`[seq]`	matches any character in *seq*
`[!seq]`	matches any character not in *seq*

For a literal match, wrap the meta-characters in brackets. For example, `'[?]'` matches the character `'?'`.

Note that the filename separator (`'/'` on Unix) is *not* special to this module. See module *glob* for pathname expansion (*glob* uses *filter()* to match pathname segments). Similarly, filenames starting with a period are not special for this module, and are matched by the `*` and `?` patterns.

Also note that *functools.lru_cache()* with the *maxsize* of 32768 is used to cache the compiled regex patterns in the following functions: *fnmatch()*, *fnmatchcase()*, *filter()*.

`fnmatch.`**`fnmatch`**(*name*, *pat*)

Test whether the filename string *name* matches the pattern string *pat*, returning `True` or `False`. Both parameters are case-normalized using *os.path.normcase()*. *fnmatchcase()* can be used to perform a case-sensitive comparison, regardless of whether that's standard for the operating system.

This example will print all file names in the current directory with the extension `.txt`:

```
import fnmatch
import os

for file in os.listdir('.'):
    if fnmatch.fnmatch(file, '*.txt'):
        print(file)
```

`fnmatch.`**`fnmatchcase`**(*name*, *pat*)

Test whether the filename string *name* matches the pattern string *pat*, returning `True` or `False`; the comparison is case-sensitive and does not apply *os.path.normcase()*.

`fnmatch.`**`filter`**(*names*, *pat*)

Construct a list from those elements of the *iterable* *names* that match pattern *pat*. It is the same as `[n for n in names if fnmatch(n, pat)]`, but implemented more efficiently.

`fnmatch.`**`translate`**(*pat*)

Return the shell-style pattern *pat* converted to a regular expression for using with *re.match()*.

Example:

```
>>> import fnmatch, re
>>>
>>> regex = fnmatch.translate('*.txt')
>>> regex
'(?s:.*\\.txt)\\Z'
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<re.Match object; span=(0, 10), match='foobar.txt'>
```

See also

Module *glob*
Unix shell-style path expansion.

11.9 linecache — Random access to text lines

Source code: Lib/linecache.py

The *linecache* module allows one to get any line from a Python source file, while attempting to optimize internally, using a cache, the common case where many lines are read from a single file. This is used by the *traceback* module to retrieve source lines for inclusion in the formatted traceback.

The *tokenize.open()* function is used to open files. This function uses *tokenize.detect_encoding()* to get the encoding of the file; in the absence of an encoding token, the file encoding defaults to UTF-8.

The *linecache* module defines the following functions:

linecache.**getline**(*filename*, *lineno*, *module_globals=None*)

Get line *lineno* from file named *filename*. This function will never raise an exception — it will return `''` on errors (the terminating newline character will be included for lines that are found).

If a file named *filename* is not found, the function first checks for a **PEP 302** `__loader__` in *module_globals*. If there is such a loader and it defines a `get_source` method, then that determines the source lines (if `get_source()` returns `None`, then `''` is returned). Finally, if *filename* is a relative filename, it is looked up relative to the entries in the module search path, `sys.path`.

linecache.**clearcache**()

Clear the cache. Use this function if you no longer need lines from files previously read using *getline()*.

linecache.**checkcache**(*filename=None*)

Check the cache for validity. Use this function if files in the cache may have changed on disk, and you require the updated version. If *filename* is omitted, it will check all the entries in the cache.

linecache.**lazycache**(*filename*, *module_globals*)

Capture enough detail about a non-file-based module to permit getting its lines later via *getline()* even if *module_globals* is `None` in the later call. This avoids doing I/O until a line is actually needed, without having to carry the module globals around indefinitely.

Added in version 3.5.

Example:

```
>>> import linecache
>>> linecache.getline(linecache.__file__, 8)
'import sys\n'
```

11.10 shutil — High-level file operations

Source code: Lib/shutil.py

The *shutil* module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal. For operations on individual files, see also the *os* module.

Warning

Even the higher-level file copying functions (`shutil.copy()`, `shutil.copy2()`) cannot copy all file metadata.

On POSIX platforms, this means that file owner and group are lost as well as ACLs. On Mac OS, the resource fork and other metadata are not used. This means that resources will be lost and file type and creator codes will not be correct. On Windows, file owners, ACLs and alternate data streams are not copied.

11.10.1 Directory and files operations

`shutil.`**`copyfileobj`**(*fsrc*, *fdst*[, *length*])

Copy the contents of the *file-like object* *fsrc* to the file-like object *fdst*. The integer *length*, if given, is the buffer size. In particular, a negative *length* value means to copy the data without looping over the source data in chunks; by default the data is read in chunks to avoid uncontrolled memory consumption. Note that if the current file position of the *fsrc* object is not 0, only the contents from the current file position to the end of the file will be copied.

`shutil.`**`copyfile`**(*src*, *dst*, *, *follow_symlinks=True*)

Copy the contents (no metadata) of the file named *src* to a file named *dst* and return *dst* in the most efficient way possible. *src* and *dst* are *path-like objects* or path names given as strings.

dst must be the complete target file name; look at `copy()` for a copy that accepts a target directory path. If *src* and *dst* specify the same file, `SameFileError` is raised.

The destination location must be writable; otherwise, an `OSError` exception will be raised. If *dst* already exists, it will be replaced. Special files such as character or block devices and pipes cannot be copied with this function.

If *follow_symlinks* is false and *src* is a symbolic link, a new symbolic link will be created instead of copying the file *src* points to.

Raises an *auditing event* `shutil.copyfile` with arguments `src`, `dst`.

Changed in version 3.3: `IOError` used to be raised instead of `OSError`. Added *follow_symlinks* argument. Now returns *dst*.

Changed in version 3.4: Raise `SameFileError` instead of `Error`. Since the former is a subclass of the latter, this change is backward compatible.

Changed in version 3.8: Platform-specific fast-copy syscalls may be used internally in order to copy the file more efficiently. See *Platform-dependent efficient copy operations* section.

exception `shutil.`**`SameFileError`**

This exception is raised if source and destination in `copyfile()` are the same file.

Added in version 3.4.

`shutil.`**`copymode`**(*src*, *dst*, *, *follow_symlinks=True*)

Copy the permission bits from *src* to *dst*. The file contents, owner, and group are unaffected. *src* and *dst* are *path-like objects* or path names given as strings. If *follow_symlinks* is false, and both *src* and *dst* are symbolic links, `copymode()` will attempt to modify the mode of *dst* itself (rather than the file it points to). This functionality is not available on every platform; please see `copystat()` for more information. If `copymode()` cannot modify symbolic links on the local platform, and it is asked to do so, it will do nothing and return.

Raises an *auditing event* `shutil.copymode` with arguments `src`, `dst`.

Changed in version 3.3: Added *follow_symlinks* argument.

shutil.**copystat**(*src*, *dst*, *, *follow_symlinks=True*)

Copy the permission bits, last access time, last modification time, and flags from *src* to *dst*. On Linux, *copystat()* also copies the "extended attributes" where possible. The file contents, owner, and group are unaffected. *src* and *dst* are *path-like objects* or path names given as strings.

If *follow_symlinks* is false, and *src* and *dst* both refer to symbolic links, *copystat()* will operate on the symbolic links themselves rather than the files the symbolic links refer to—reading the information from the *src* symbolic link, and writing the information to the *dst* symbolic link.

> **Note**
>
> Not all platforms provide the ability to examine and modify symbolic links. Python itself can tell you what functionality is locally available.
>
> - If `os.chmod in os.supports_follow_symlinks` is `True`, *copystat()* can modify the permission bits of a symbolic link.
> - If `os.utime in os.supports_follow_symlinks` is `True`, *copystat()* can modify the last access and modification times of a symbolic link.
> - If `os.chflags in os.supports_follow_symlinks` is `True`, *copystat()* can modify the flags of a symbolic link. (`os.chflags` is not available on all platforms.)
>
> On platforms where some or all of this functionality is unavailable, when asked to modify a symbolic link, *copystat()* will copy everything it can. *copystat()* never returns failure.
>
> Please see *os.supports_follow_symlinks* for more information.

Raises an *auditing event* `shutil.copystat` with arguments `src`, `dst`.

Changed in version 3.3: Added *follow_symlinks* argument and support for Linux extended attributes.

shutil.**copy**(*src*, *dst*, *, *follow_symlinks=True*)

Copies the file *src* to the file or directory *dst*. *src* and *dst* should be *path-like objects* or strings. If *dst* specifies a directory, the file will be copied into *dst* using the base filename from *src*. If *dst* specifies a file that already exists, it will be replaced. Returns the path to the newly created file.

If *follow_symlinks* is false, and *src* is a symbolic link, *dst* will be created as a symbolic link. If *follow_symlinks* is true and *src* is a symbolic link, *dst* will be a copy of the file *src* refers to.

copy() copies the file data and the file's permission mode (see *os.chmod()*). Other metadata, like the file's creation and modification times, is not preserved. To preserve all file metadata from the original, use *copy2()* instead.

Raises an *auditing event* `shutil.copyfile` with arguments `src`, `dst`.

Raises an *auditing event* `shutil.copymode` with arguments `src`, `dst`.

Changed in version 3.3: Added *follow_symlinks* argument. Now returns path to the newly created file.

Changed in version 3.8: Platform-specific fast-copy syscalls may be used internally in order to copy the file more efficiently. See *Platform-dependent efficient copy operations* section.

shutil.**copy2**(*src*, *dst*, *, *follow_symlinks=True*)

Identical to *copy()* except that *copy2()* also attempts to preserve file metadata.

When *follow_symlinks* is false, and *src* is a symbolic link, *copy2()* attempts to copy all metadata from the *src* symbolic link to the newly created *dst* symbolic link. However, this functionality is not available on all platforms. On platforms where some or all of this functionality is unavailable, *copy2()* will preserve all the metadata it can; *copy2()* never raises an exception because it cannot preserve file metadata.

`copy2()` uses `copystat()` to copy the file metadata. Please see `copystat()` for more information about platform support for modifying symbolic link metadata.

Raises an *auditing event* `shutil.copyfile` with arguments `src`, `dst`.

Raises an *auditing event* `shutil.copystat` with arguments `src`, `dst`.

Changed in version 3.3: Added *follow_symlinks* argument, try to copy extended file system attributes too (currently Linux only). Now returns path to the newly created file.

Changed in version 3.8: Platform-specific fast-copy syscalls may be used internally in order to copy the file more efficiently. See *Platform-dependent efficient copy operations* section.

`shutil.`**`ignore_patterns`**(**patterns*)

This factory function creates a function that can be used as a callable for `copytree()`'s *ignore* argument, ignoring files and directories that match one of the glob-style *patterns* provided. See the example below.

`shutil.`**`copytree`**(*src*, *dst*, *symlinks=False*, *ignore=None*, *copy_function=copy2*, *ignore_dangling_symlinks=False*, *dirs_exist_ok=False*)

Recursively copy an entire directory tree rooted at *src* to a directory named *dst* and return the destination directory. All intermediate directories needed to contain *dst* will also be created by default.

Permissions and times of directories are copied with `copystat()`, individual files are copied using `copy2()`.

If *symlinks* is true, symbolic links in the source tree are represented as symbolic links in the new tree and the metadata of the original links will be copied as far as the platform allows; if false or omitted, the contents and metadata of the linked files are copied to the new tree.

When *symlinks* is false, if the file pointed to by the symlink doesn't exist, an exception will be added in the list of errors raised in an `Error` exception at the end of the copy process. You can set the optional *ignore_dangling_symlinks* flag to true if you want to silence this exception. Notice that this option has no effect on platforms that don't support `os.symlink()`.

If *ignore* is given, it must be a callable that will receive as its arguments the directory being visited by `copytree()`, and a list of its contents, as returned by `os.listdir()`. Since `copytree()` is called recursively, the *ignore* callable will be called once for each directory that is copied. The callable must return a sequence of directory and file names relative to the current directory (i.e. a subset of the items in its second argument); these names will then be ignored in the copy process. `ignore_patterns()` can be used to create such a callable that ignores names based on glob-style patterns.

If exception(s) occur, an `Error` is raised with a list of reasons.

If *copy_function* is given, it must be a callable that will be used to copy each file. It will be called with the source path and the destination path as arguments. By default, `copy2()` is used, but any function that supports the same signature (like `copy()`) can be used.

If *dirs_exist_ok* is false (the default) and *dst* already exists, a `FileExistsError` is raised. If *dirs_exist_ok* is true, the copying operation will continue if it encounters existing directories, and files within the *dst* tree will be overwritten by corresponding files from the *src* tree.

Raises an *auditing event* `shutil.copytree` with arguments `src`, `dst`.

Changed in version 3.2: Added the *copy_function* argument to be able to provide a custom copy function. Added the *ignore_dangling_symlinks* argument to silence dangling symlinks errors when *symlinks* is false.

Changed in version 3.3: Copy metadata when *symlinks* is false. Now returns *dst*.

Changed in version 3.8: Platform-specific fast-copy syscalls may be used internally in order to copy the file more efficiently. See *Platform-dependent efficient copy operations* section.

Changed in version 3.8: Added the *dirs_exist_ok* parameter.

shutil.**rmtree**(*path*, *ignore_errors=False*, *onerror=None*, *, *onexc=None*, *dir_fd=None*)

Delete an entire directory tree; *path* must point to a directory (but not a symbolic link to a directory). If *ignore_errors* is true, errors resulting from failed removals will be ignored; if false or omitted, such errors are handled by calling a handler specified by *onexc* or *onerror* or, if both are omitted, exceptions are propagated to the caller.

This function can support *paths relative to directory descriptors*.

> **Note**
>
> On platforms that support the necessary fd-based functions a symlink attack resistant version of `rmtree()` is used by default. On other platforms, the `rmtree()` implementation is susceptible to a symlink attack: given proper timing and circumstances, attackers can manipulate symlinks on the filesystem to delete files they wouldn't be able to access otherwise. Applications can use the `rmtree.avoids_symlink_attacks` function attribute to determine which case applies.

If *onexc* is provided, it must be a callable that accepts three parameters: *function*, *path*, and *excinfo*.

The first parameter, *function*, is the function which raised the exception; it depends on the platform and implementation. The second parameter, *path*, will be the path name passed to *function*. The third parameter, *excinfo*, is the exception that was raised. Exceptions raised by *onexc* will not be caught.

The deprecated *onerror* is similar to *onexc*, except that the third parameter it receives is the tuple returned from `sys.exc_info()`.

Raises an *auditing event* `shutil.rmtree` with arguments `path`, `dir_fd`.

Changed in version 3.3: Added a symlink attack resistant version that is used automatically if platform supports fd-based functions.

Changed in version 3.8: On Windows, will no longer delete the contents of a directory junction before removing the junction.

Changed in version 3.11: Added the *dir_fd* parameter.

Changed in version 3.12: Added the *onexc* parameter, deprecated *onerror*.

Changed in version 3.13: `rmtree()` now ignores `FileNotFoundError` exceptions for all but the top-level path. Exceptions other than `OSError` and subclasses of `OSError` are now always propagated to the caller.

rmtree.**avoids_symlink_attacks**

Indicates whether the current platform and implementation provides a symlink attack resistant version of `rmtree()`. Currently this is only true for platforms supporting fd-based directory access functions.

Added in version 3.3.

shutil.**move**(*src*, *dst*, *copy_function=copy2*)

Recursively move a file or directory (*src*) to another location and return the destination.

If *dst* is an existing directory or a symlink to a directory, then *src* is moved inside that directory. The destination path in that directory must not already exist.

If *dst* already exists but is not a directory, it may be overwritten depending on `os.rename()` semantics.

If the destination is on the current filesystem, then `os.rename()` is used. Otherwise, *src* is copied to the destination using *copy_function* and then removed. In case of symlinks, a new symlink pointing to the target of *src* will be created as the destination and *src* will be removed.

If *copy_function* is given, it must be a callable that takes two arguments, *src* and the destination, and will be used to copy *src* to the destination if `os.rename()` cannot be used. If the source is a directory, `copytree()` is called, passing it the *copy_function*. The default *copy_function* is `copy2()`. Using `copy()` as the *copy_function*

allows the move to succeed when it is not possible to also copy the metadata, at the expense of not copying any of the metadata.

Raises an *auditing event* `shutil.move` with arguments `src`, `dst`.

Changed in version 3.3: Added explicit symlink handling for foreign filesystems, thus adapting it to the behavior of GNU's **mv**. Now returns *dst*.

Changed in version 3.5: Added the *copy_function* keyword argument.

Changed in version 3.8: Platform-specific fast-copy syscalls may be used internally in order to copy the file more efficiently. See *Platform-dependent efficient copy operations* section.

Changed in version 3.9: Accepts a *path-like object* for both *src* and *dst*.

`shutil.`**`disk_usage`**(*path*)

Return disk usage statistics about the given path as a *named tuple* with the attributes *total*, *used* and *free*, which are the amount of total, used and free space, in bytes. *path* may be a file or a directory.

> **Note**
>
> On Unix filesystems, *path* must point to a path within a **mounted** filesystem partition. On those platforms, CPython doesn't attempt to retrieve disk usage information from non-mounted filesystems.

Added in version 3.3.

Changed in version 3.8: On Windows, *path* can now be a file or directory.

Availability: Unix, Windows.

`shutil.`**`chown`**(*path*, *user=None*, *group=None*, ***, *dir_fd=None*, *follow_symlinks=True*)

Change owner *user* and/or *group* of the given *path*.

user can be a system user name or a uid; the same applies to *group*. At least one argument is required.

See also `os.chown()`, the underlying function.

Raises an *auditing event* `shutil.chown` with arguments `path`, `user`, `group`.

Availability: Unix.

Added in version 3.3.

Changed in version 3.13: Added *dir_fd* and *follow_symlinks* parameters.

`shutil.`**`which`**(*cmd*, *mode=os.F_OK | os.X_OK*, *path=None*)

Return the path to an executable which would be run if the given *cmd* was called. If no *cmd* would be called, return `None`.

mode is a permission mask passed to `os.access()`, by default determining if the file exists and is executable.

path is a "`PATH` string" specifying the lookup directory list. When no *path* is specified, the results of `os.environ()` are used, returning either the "PATH" value or a fallback of `os.defpath`.

On Windows, the current directory is prepended to the *path* if *mode* does not include `os.X_OK`. When the *mode* does include `os.X_OK`, the Windows API `NeedCurrentDirectoryForExePathW` will be consulted to determine if the current directory should be prepended to *path*. To avoid consulting the current working directory for executables: set the environment variable `NoDefaultCurrentDirectoryInExePath`.

Also on Windows, the `PATHEXT` variable is used to resolve commands that may not already include an extension. For example, if you call `shutil.which("python")`, `which()` will search `PATHEXT` to know that it should look for `python.exe` within the *path* directories. For example, on Windows:

```
>>> shutil.which("python")
'C:\\Python33\\python.EXE'
```

This is also applied when *cmd* is a path that contains a directory component:

```
>> shutil.which("C:\\Python33\\python")
'C:\\Python33\\python.EXE'
```

Added in version 3.3.

Changed in version 3.8: The *bytes* type is now accepted. If *cmd* type is *bytes*, the result type is also *bytes*.

Changed in version 3.12: On Windows, the current directory is no longer prepended to the search path if *mode* includes `os.X_OK` and WinAPI `NeedCurrentDirectoryForExePathW(cmd)` is false, else the current directory is prepended even if it is already in the search path; `PATHEXT` is used now even when *cmd* includes a directory component or ends with an extension that is in `PATHEXT`; and filenames that have no extension can now be found.

Changed in version 3.12.1: On Windows, if *mode* includes `os.X_OK`, executables with an extension in `PATHEXT` will be preferred over executables without a matching extension. This brings behavior closer to that of Python 3.11.

exception shutil.**Error**

This exception collects exceptions that are raised during a multi-file operation. For *copytree()*, the exception argument is a list of 3-tuples (*srcname*, *dstname*, *exception*).

Platform-dependent efficient copy operations

Starting from Python 3.8, all functions involving a file copy (*copyfile()*, *copy()*, *copy2()*, *copytree()*, and *move()*) may use platform-specific "fast-copy" syscalls in order to copy the file more efficiently (see bpo-33671). "fast-copy" means that the copying operation occurs within the kernel, avoiding the use of userspace buffers in Python as in "`outfd.write(infd.read())`".

On macOS fcopyfile is used to copy the file content (not metadata).

On Linux *os.sendfile()* is used.

On Windows *shutil.copyfile()* uses a bigger default buffer size (1 MiB instead of 64 KiB) and a *memoryview()*-based variant of *shutil.copyfileobj()* is used.

If the fast-copy operation fails and no data was written in the destination file then shutil will silently fallback on using less efficient *copyfileobj()* function internally.

Changed in version 3.8.

copytree example

An example that uses the *ignore_patterns()* helper:

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

This will copy everything except `.pyc` files and files or directories whose name starts with `tmp`.

Another example that uses the *ignore* argument to add a logging call:

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return []   # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```

rmtree example

This example shows how to remove a directory tree on Windows where some of the files have their read-only bit set. It uses the onexc callback to clear the readonly bit and reattempt the remove. Any subsequent failure will propagate.

```
import os, stat
import shutil

def remove_readonly(func, path, _):
    "Clear the readonly bit and reattempt the removal"
    os.chmod(path, stat.S_IWRITE)
    func(path)

shutil.rmtree(directory, onexc=remove_readonly)
```

11.10.2 Archiving operations

Added in version 3.2.

Changed in version 3.5: Added support for the *xztar* format.

High-level utilities to create and read compressed and archived files are also provided. They rely on the `zipfile` and `tarfile` modules.

shutil.**make_archive**(*base_name*, *format*[, *root_dir*[, *base_dir*[, *verbose*[, *dry_run*[, *owner*[, *group*[, *logger*]]]]]]])

Create an archive file (such as zip or tar) and return its name.

base_name is the name of the file to create, including the path, minus any format-specific extension.

format is the archive format: one of "zip" (if the `zlib` module is available), "tar", "gztar" (if the `zlib` module is available), "bztar" (if the `bz2` module is available), or "xztar" (if the `lzma` module is available).

root_dir is a directory that will be the root directory of the archive, all paths in the archive will be relative to it; for example, we typically chdir into *root_dir* before creating the archive.

base_dir is the directory where we start archiving from; i.e. *base_dir* will be the common prefix of all files and directories in the archive. *base_dir* must be given relative to *root_dir*. See *Archiving example with base_dir* for how to use *base_dir* and *root_dir* together.

root_dir and *base_dir* both default to the current directory.

If *dry_run* is true, no archive is created, but the operations that would be executed are logged to *logger*.

owner and *group* are used when creating a tar archive. By default, uses the current owner and group.

logger must be an object compatible with **PEP 282**, usually an instance of `logging.Logger`.

The *verbose* argument is unused and deprecated.

Raises an *auditing event* `shutil.make_archive` with arguments `base_name`, `format`, `root_dir`, `base_dir`.

> **Note**
>
> This function is not thread-safe when custom archivers registered with *register_archive_format()* do not support the *root_dir* argument. In this case it temporarily changes the current working directory of the process to *root_dir* to perform archiving.

Changed in version 3.8: The modern pax (POSIX.1-2001) format is now used instead of the legacy GNU format for archives created with `format="tar"`.

Changed in version 3.10.6: This function is now made thread-safe during creation of standard `.zip` and tar archives.

`shutil.`**`get_archive_formats`**`()`

Return a list of supported formats for archiving. Each element of the returned sequence is a tuple `(name, description)`.

By default *shutil* provides these formats:

- *zip*: ZIP file (if the *zlib* module is available).
- *tar*: Uncompressed tar file. Uses POSIX.1-2001 pax format for new archives.
- *gztar*: gzip'ed tar-file (if the *zlib* module is available).
- *bztar*: bzip2'ed tar-file (if the *bz2* module is available).
- *xztar*: xz'ed tar-file (if the *lzma* module is available).

You can register new formats or provide your own archiver for any existing formats, by using *register_archive_format()*.

`shutil.`**`register_archive_format`**`(`*name*, *function*[, *extra_args*[, *description*]]`)`

Register an archiver for the format *name*.

function is the callable that will be used to unpack archives. The callable will receive the *base_name* of the file to create, followed by the *base_dir* (which defaults to *os.curdir*) to start archiving from. Further arguments are passed as keyword arguments: *owner*, *group*, *dry_run* and *logger* (as passed in *make_archive()*).

If *function* has the custom attribute `function.supports_root_dir` set to `True`, the *root_dir* argument is passed as a keyword argument. Otherwise the current working directory of the process is temporarily changed to *root_dir* before calling *function*. In this case *make_archive()* is not thread-safe.

If given, *extra_args* is a sequence of `(name, value)` pairs that will be used as extra keywords arguments when the archiver callable is used.

description is used by *get_archive_formats()* which returns the list of archivers. Defaults to an empty string.

Changed in version 3.12: Added support for functions supporting the *root_dir* argument.

`shutil.`**`unregister_archive_format`**`(`*name*`)`

Remove the archive format *name* from the list of supported formats.

`shutil.`**`unpack_archive`**`(`*filename*[, *extract_dir*[, *format*[, *filter*]]]`)`

Unpack an archive. *filename* is the full path of the archive.

extract_dir is the name of the target directory where the archive is unpacked. If not provided, the current working directory is used.

format is the archive format: one of "zip", "tar", "gztar", "bztar", or "xztar". Or any other format registered with *register_unpack_format()*. If not provided, *unpack_archive()* will use the archive file name extension and see if an unpacker was registered for that extension. In case none is found, a *ValueError* is raised.

The keyword-only *filter* argument is passed to the underlying unpacking function. For zip files, *filter* is not accepted. For tar files, it is recommended to set it to `'data'`, unless using features specific to tar and UNIX-like filesystems. (See *Extraction filters* for details.) The `'data'` filter will become the default for tar files in Python 3.14.

Raises an *auditing event* `shutil.unpack_archive` with arguments `filename`, `extract_dir`, `format`.

Warning

Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of the path specified in the *extract_dir* argument, e.g. members that have absolute filenames starting with "/" or filenames with two dots "..".

Changed in version 3.7: Accepts a *path-like object* for *filename* and *extract_dir*.

Changed in version 3.12: Added the *filter* argument.

shutil.**register_unpack_format**(*name*, *extensions*, *function*[, *extra_args*[, *description*]])

Registers an unpack format. *name* is the name of the format and *extensions* is a list of extensions corresponding to the format, like `.zip` for Zip files.

function is the callable that will be used to unpack archives. The callable will receive:

- the path of the archive, as a positional argument;
- the directory the archive must be extracted to, as a positional argument;
- possibly a *filter* keyword argument, if it was given to *unpack_archive()*;
- additional keyword arguments, specified by *extra_args* as a sequence of `(name, value)` tuples.

description can be provided to describe the format, and will be returned by the *get_unpack_formats()* function.

shutil.**unregister_unpack_format**(*name*)

Unregister an unpack format. *name* is the name of the format.

shutil.**get_unpack_formats**()

Return a list of all registered formats for unpacking. Each element of the returned sequence is a tuple `(name, extensions, description)`.

By default *shutil* provides these formats:

- *zip*: ZIP file (unpacking compressed files works only if the corresponding module is available).
- *tar*: uncompressed tar file.
- *gztar*: gzip'ed tar-file (if the *zlib* module is available).
- *bztar*: bzip2'ed tar-file (if the *bz2* module is available).
- *xztar*: xz'ed tar-file (if the *lzma* module is available).

You can register new formats or provide your own unpacker for any existing formats, by using *register_unpack_format()*.

Archiving example

In this example, we create a gzip'ed tar-file archive containing all files found in the `.ssh` directory of the user:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

The resulting archive contains:

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
drwx------ tarek/staff       0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff     609 2008-06-09 13:26:54 ./authorized_keys
-rwxr-xr-x tarek/staff      65 2008-06-09 13:26:54 ./config
-rwx------ tarek/staff     668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff     609 2008-06-09 13:26:54 ./id_dsa.pub
-rw------- tarek/staff    1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff     397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff   37192 2010-02-06 18:23:10 ./known_hosts
```

Archiving example with *base_dir*

In this example, similar to the *one above*, we show how to use *`make_archive()`*, but this time with the usage of *base_dir*. We now have the following directory structure:

```
$ tree tmp
tmp
└── root
    └── structure
        ├── content
        │   └── please_add.txt
        └── do_not_add.txt
```

In the final archive, `please_add.txt` should be included, but `do_not_add.txt` should not. Therefore we use the following:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> make_archive(
...     archive_name,
...     'tar',
...     root_dir='tmp/root',
...     base_dir='structure/content',
... )
'/Users/tarek/my_archive.tar'
```

Listing the files in the resulting archive gives us:

```
$ python -m tarfile -l /Users/tarek/myarchive.tar
structure/content/
structure/content/please_add.txt
```

11.10.3 Querying the size of the output terminal

shutil.**get_terminal_size**(*fallback=(columns, lines)*)

Get the size of the terminal window.

For each of the two dimensions, the environment variable, COLUMNS and LINES respectively, is checked. If the variable is defined and the value is a positive integer, it is used.

When COLUMNS or LINES is not defined, which is the common case, the terminal connected to *sys.__stdout__* is queried by invoking *os.get_terminal_size()*.

If the terminal size cannot be successfully queried, either because the system doesn't support querying, or because we are not connected to a terminal, the value given in fallback parameter is used. fallback defaults to (80, 24) which is the default size used by many terminal emulators.

The value returned is a named tuple of type *os.terminal_size*.

See also: The Single UNIX Specification, Version 2, Other Environment Variables.

Added in version 3.3.

Changed in version 3.11: The fallback values are also used if *os.get_terminal_size()* returns zeroes.

See also

Module *os*
Operating system interfaces, including functions to work with files at a lower level than Python *file objects*.

Module *io*
Python's built-in I/O library, including both abstract classes and some concrete classes such as file I/O.

Built-in function *open()*
The standard way to open files for reading and writing with Python.

CHAPTER

TWELVE

DATA PERSISTENCE

The modules described in this chapter support storing Python data in a persistent form on disk. The `pickle` and `marshal` modules can turn many Python data types into a stream of bytes and then recreate the objects from the bytes. The various DBM-related modules support a family of hash-based file formats that store a mapping of strings to other strings.

The list of modules described in this chapter is:

12.1 `pickle` — Python object serialization

Source code: Lib/pickle.py

The `pickle` module implements binary protocols for serializing and de-serializing a Python object structure. *"Pickling"* is the process whereby a Python object hierarchy is converted into a byte stream, and *"unpickling"* is the inverse operation, whereby a byte stream (from a *binary file* or *bytes-like object*) is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as "serialization", "marshalling,"[1] or "flattening"; however, to avoid confusion, the terms used here are "pickling" and "unpickling".

> **Warning**
>
> The `pickle` module **is not secure**. Only unpickle data you trust.
>
> It is possible to construct malicious pickle data which will **execute arbitrary code during unpickling**. Never unpickle data that could have come from an untrusted source, or that could have been tampered with.
>
> Consider signing data with `hmac` if you need to ensure that it has not been tampered with.
>
> Safer serialization formats such as `json` may be more appropriate if you are processing untrusted data. See *Comparison with json*.

[1] Don't confuse this with the `marshal` module

12.1.1 Relationship to other Python modules

Comparison with `marshal`

Python has a more primitive serialization module called *marshal*, but in general *pickle* should always be the preferred way to serialize Python objects. *marshal* exists primarily to support Python's `.pyc` files.

The *pickle* module differs from *marshal* in several significant ways:

- The *pickle* module keeps track of the objects it has already serialized, so that later references to the same object won't be serialized again. *marshal* doesn't do this.

  This has implications both for recursive objects and object sharing. Recursive objects are objects that contain references to themselves. These are not handled by marshal, and in fact, attempting to marshal recursive objects will crash your Python interpreter. Object sharing happens when there are multiple references to the same object in different places in the object hierarchy being serialized. *pickle* stores such objects only once, and ensures that all other references point to the master copy. Shared objects remain shared, which can be very important for mutable objects.

- *marshal* cannot be used to serialize user-defined classes and their instances. *pickle* can save and restore class instances transparently, however the class definition must be importable and live in the same module as when the object was stored.

- The *marshal* serialization format is not guaranteed to be portable across Python versions. Because its primary job in life is to support `.pyc` files, the Python implementers reserve the right to change the serialization format in non-backwards compatible ways should the need arise. The *pickle* serialization format is guaranteed to be backwards compatible across Python releases provided a compatible pickle protocol is chosen and pickling and unpickling code deals with Python 2 to Python 3 type differences if your data is crossing that unique breaking change language boundary.

Comparison with `json`

There are fundamental differences between the pickle protocols and JSON (JavaScript Object Notation):

- JSON is a text serialization format (it outputs unicode text, although most of the time it is then encoded to `utf-8`), while pickle is a binary serialization format;

- JSON is human-readable, while pickle is not;

- JSON is interoperable and widely used outside of the Python ecosystem, while pickle is Python-specific;

- JSON, by default, can only represent a subset of the Python built-in types, and no custom classes; pickle can represent an extremely large number of Python types (many of them automatically, by clever usage of Python's introspection facilities; complex cases can be tackled by implementing *specific object APIs*);

- Unlike pickle, deserializing untrusted JSON does not in itself create an arbitrary code execution vulnerability.

See also

The *json* module: a standard library module allowing JSON serialization and deserialization.

12.1.2 Data stream format

The data format used by *pickle* is Python-specific. This has the advantage that there are no restrictions imposed by external standards such as JSON (which can't represent pointer sharing); however it means that non-Python programs may not be able to reconstruct pickled Python objects.

By default, the *pickle* data format uses a relatively compact binary representation. If you need optimal size characteristics, you can efficiently *compress* pickled data.

The module *pickletools* contains tools for analyzing data streams generated by *pickle*. *pickletools* source code has extensive comments about opcodes used by pickle protocols.

There are currently 6 different protocols which can be used for pickling. The higher the protocol used, the more recent the version of Python needed to read the pickle produced.

- Protocol version 0 is the original "human-readable" protocol and is backwards compatible with earlier versions of Python.
- Protocol version 1 is an old binary format which is also compatible with earlier versions of Python.
- Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of *new-style classes*. Refer to **PEP 307** for information about improvements brought by protocol 2.
- Protocol version 3 was added in Python 3.0. It has explicit support for *bytes* objects and cannot be unpickled by Python 2.x. This was the default protocol in Python 3.0–3.7.
- Protocol version 4 was added in Python 3.4. It adds support for very large objects, pickling more kinds of objects, and some data format optimizations. It is the default protocol starting with Python 3.8. Refer to **PEP 3154** for information about improvements brought by protocol 4.
- Protocol version 5 was added in Python 3.8. It adds support for out-of-band data and speedup for in-band data. Refer to **PEP 574** for information about improvements brought by protocol 5.

> **Note**
>
> Serialization is a more primitive notion than persistence; although *pickle* reads and writes file objects, it does not handle the issue of naming persistent objects, nor the (even more complicated) issue of concurrent access to persistent objects. The *pickle* module can transform a complex object into a byte stream and it can transform the byte stream into an object with the same internal structure. Perhaps the most obvious thing to do with these byte streams is to write them onto a file, but it is also conceivable to send them across a network or store them in a database. The *shelve* module provides a simple interface to pickle and unpickle objects on DBM-style database files.

12.1.3 Module Interface

To serialize an object hierarchy, you simply call the *dumps()* function. Similarly, to de-serialize a data stream, you call the *loads()* function. However, if you want more control over serialization and de-serialization, you can create a *Pickler* or an *Unpickler* object, respectively.

The *pickle* module provides the following constants:

pickle.**HIGHEST_PROTOCOL**

An integer, the highest *protocol version* available. This value can be passed as a *protocol* value to functions *dump()* and *dumps()* as well as the *Pickler* constructor.

pickle.**DEFAULT_PROTOCOL**

An integer, the default *protocol version* used for pickling. May be less than *HIGHEST_PROTOCOL*. Currently the default protocol is 4, first introduced in Python 3.4 and incompatible with previous versions.

Changed in version 3.0: The default protocol is 3.

Changed in version 3.8: The default protocol is 4.

The *pickle* module provides the following functions to make the pickling process more convenient:

pickle.**dump**(*obj*, *file*, *protocol=None*, ***, *fix_imports=True*, *buffer_callback=None*)

Write the pickled representation of the object *obj* to the open *file object file*. This is equivalent to `Pickler(file, protocol).dump(obj)`.

Arguments *file*, *protocol*, *fix_imports* and *buffer_callback* have the same meaning as in the *Pickler* constructor.

Changed in version 3.8: The *buffer_callback* argument was added.

pickle.**dumps**(*obj*, *protocol=None*, ***, *fix_imports=True*, *buffer_callback=None*)

Return the pickled representation of the object *obj* as a *bytes* object, instead of writing it to a file.

Arguments *protocol*, *fix_imports* and *buffer_callback* have the same meaning as in the *Pickler* constructor.

Changed in version 3.8: The *buffer_callback* argument was added.

pickle.**load**(*file*, ***, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

Read the pickled representation of an object from the open *file object file* and return the reconstituted object hierarchy specified therein. This is equivalent to `Unpickler(file).load()`.

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled representation of the object are ignored.

Arguments *file*, *fix_imports*, *encoding*, *errors*, *strict* and *buffers* have the same meaning as in the *Unpickler* constructor.

Changed in version 3.8: The *buffers* argument was added.

pickle.**loads**(*data*, */*, ***, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

Return the reconstituted object hierarchy of the pickled representation *data* of an object. *data* must be a *bytes-like object*.

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled representation of the object are ignored.

Arguments *fix_imports*, *encoding*, *errors*, *strict* and *buffers* have the same meaning as in the *Unpickler* constructor.

Changed in version 3.8: The *buffers* argument was added.

The *pickle* module defines three exceptions:

exception pickle.**PickleError**

Common base class for the other pickling exceptions. It inherits from *Exception*.

exception pickle.**PicklingError**

Error raised when an unpicklable object is encountered by *Pickler*. It inherits from *PickleError*.

Refer to *What can be pickled and unpickled?* to learn what kinds of objects can be pickled.

exception pickle.**UnpicklingError**

Error raised when there is a problem unpickling an object, such as a data corruption or a security violation. It inherits from *PickleError*.

Note that other exceptions may also be raised during unpickling, including (but not necessarily limited to) AttributeError, EOFError, ImportError, and IndexError.

The *pickle* module exports three classes, *Pickler*, *Unpickler* and *PickleBuffer*:

class pickle.**Pickler**(*file*, *protocol=None*, ***, *fix_imports=True*, *buffer_callback=None*)

This takes a binary file for writing a pickle data stream.

The optional *protocol* argument, an integer, tells the pickler to use the given protocol; supported protocols are 0 to *HIGHEST_PROTOCOL*. If not specified, the default is *DEFAULT_PROTOCOL*. If a negative number is specified, *HIGHEST_PROTOCOL* is selected.

The *file* argument must have a write() method that accepts a single bytes argument. It can thus be an on-disk file opened for binary writing, an *io.BytesIO* instance, or any other custom object that meets this interface.

If *fix_imports* is true and *protocol* is less than 3, pickle will try to map the new Python 3 names to the old module names used in Python 2, so that the pickle data stream is readable with Python 2.

If *buffer_callback* is `None` (the default), buffer views are serialized into *file* as part of the pickle stream.

If *buffer_callback* is not `None`, then it can be called any number of times with a buffer view. If the callback returns a false value (such as `None`), the given buffer is *out-of-band*; otherwise the buffer is serialized in-band, i.e. inside the pickle stream.

It is an error if *buffer_callback* is not `None` and *protocol* is `None` or smaller than 5.

Changed in version 3.8: The *buffer_callback* argument was added.

dump(*obj*)

Write the pickled representation of *obj* to the open file object given in the constructor.

persistent_id(*obj*)

Do nothing by default. This exists so a subclass can override it.

If *persistent_id()* returns `None`, *obj* is pickled as usual. Any other value causes *Pickler* to emit the returned value as a persistent ID for *obj*. The meaning of this persistent ID should be defined by *Unpickler.persistent_load()*. Note that the value returned by *persistent_id()* cannot itself have a persistent ID.

See *Persistence of External Objects* for details and examples of uses.

Changed in version 3.13: Add the default implementation of this method in the C implementation of `Pickler`.

dispatch_table

A pickler object's dispatch table is a registry of *reduction functions* of the kind which can be declared using *copyreg.pickle()*. It is a mapping whose keys are classes and whose values are reduction functions. A reduction function takes a single argument of the associated class and should conform to the same interface as a *__reduce__()* method.

By default, a pickler object will not have a *dispatch_table* attribute, and it will instead use the global dispatch table managed by the *copyreg* module. However, to customize the pickling for a specific pickler object one can set the *dispatch_table* attribute to a dict-like object. Alternatively, if a subclass of *Pickler* has a *dispatch_table* attribute then this will be used as the default dispatch table for instances of that class.

See *Dispatch Tables* for usage examples.

Added in version 3.3.

reducer_override(*obj*)

Special reducer that can be defined in *Pickler* subclasses. This method has priority over any reducer in the *dispatch_table*. It should conform to the same interface as a *__reduce__()* method, and can optionally return *NotImplemented* to fallback on *dispatch_table*-registered reducers to pickle `obj`.

For a detailed example, see *Custom Reduction for Types, Functions, and Other Objects*.

Added in version 3.8.

fast

Deprecated. Enable fast mode if set to a true value. The fast mode disables the usage of memo, therefore speeding the pickling process by not generating superfluous PUT opcodes. It should not be used with self-referential objects, doing otherwise will cause `Pickler` to recurse infinitely.

Use `pickletools.optimize()` if you need more compact pickles.

class pickle.**Unpickler**(*file*, *, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

This takes a binary file for reading a pickle data stream.

The protocol version of the pickle is detected automatically, so no protocol argument is needed.

The argument *file* must have three methods, a read() method that takes an integer argument, a readinto() method that takes a buffer argument and a readline() method that requires no arguments, as in the `io.BufferedIOBase` interface. Thus *file* can be an on-disk file opened for binary reading, an `io.BytesIO` object, or any other custom object that meets this interface.

The optional arguments *fix_imports*, *encoding* and *errors* are used to control compatibility support for pickle stream generated by Python 2. If *fix_imports* is true, pickle will try to map the old Python 2 names to the new names used in Python 3. The *encoding* and *errors* tell pickle how to decode 8-bit string instances pickled by Python 2; these default to 'ASCII' and 'strict', respectively. The *encoding* can be 'bytes' to read these 8-bit string instances as bytes objects. Using `encoding='latin1'` is required for unpickling NumPy arrays and instances of `datetime`, `date` and `time` pickled by Python 2.

If *buffers* is `None` (the default), then all data necessary for deserialization must be contained in the pickle stream. This means that the *buffer_callback* argument was `None` when a `Pickler` was instantiated (or when `dump()` or `dumps()` was called).

If *buffers* is not `None`, it should be an iterable of buffer-enabled objects that is consumed each time the pickle stream references an *out-of-band* buffer view. Such buffers have been given in order to the *buffer_callback* of a Pickler object.

Changed in version 3.8: The *buffers* argument was added.

load()

Read the pickled representation of an object from the open file object given in the constructor, and return the reconstituted object hierarchy specified therein. Bytes past the pickled representation of the object are ignored.

persistent_load(*pid*)

Raise an `UnpicklingError` by default.

If defined, `persistent_load()` should return the object specified by the persistent ID *pid*. If an invalid persistent ID is encountered, an `UnpicklingError` should be raised.

See *Persistence of External Objects* for details and examples of uses.

Changed in version 3.13: Add the default implementation of this method in the C implementation of `Unpickler`.

find_class(*module*, *name*)

Import *module* if necessary and return the object called *name* from it, where the *module* and *name* arguments are `str` objects. Note, unlike its name suggests, `find_class()` is also used for finding functions.

Subclasses may override this to gain control over what type of objects and how they can be loaded, potentially reducing security risks. Refer to *Restricting Globals* for details.

Raises an *auditing event* `pickle.find_class` with arguments `module`, `name`.

class pickle.**PickleBuffer**(*buffer*)

A wrapper for a buffer representing picklable data. *buffer* must be a buffer-providing object, such as a *bytes-like object* or a N-dimensional array.

`PickleBuffer` is itself a buffer provider, therefore it is possible to pass it to other APIs expecting a buffer-providing object, such as `memoryview`.

`PickleBuffer` objects can only be serialized using pickle protocol 5 or higher. They are eligible for *out-of-band serialization*.

Added in version 3.8.

raw()

Return a `memoryview` of the memory area underlying this buffer. The returned object is a one-dimensional, C-contiguous memoryview with format `B` (unsigned bytes). `BufferError` is raised if the buffer is neither C- nor Fortran-contiguous.

release()

Release the underlying buffer exposed by the PickleBuffer object.

12.1.4 What can be pickled and unpickled?

The following types can be pickled:

- built-in constants (`None`, `True`, `False`, `Ellipsis`, and `NotImplemented`);
- integers, floating-point numbers, complex numbers;
- strings, bytes, bytearrays;
- tuples, lists, sets, and dictionaries containing only picklable objects;
- functions (built-in and user-defined) accessible from the top level of a module (using `def`, not `lambda`);
- classes accessible from the top level of a module;
- instances of such classes whose the result of calling `__getstate__()` is picklable (see section *Pickling Class Instances* for details).

Attempts to pickle unpicklable objects will raise the `PicklingError` exception; when this happens, an unspecified number of bytes may have already been written to the underlying file. Trying to pickle a highly recursive data structure may exceed the maximum recursion depth, a `RecursionError` will be raised in this case. You can carefully raise this limit with `sys.setrecursionlimit()`.

Note that functions (built-in and user-defined) are pickled by fully *qualified name*, not by value.[2] This means that only the function name is pickled, along with the name of the containing module and classes. Neither the function's code, nor any of its function attributes are pickled. Thus the defining module must be importable in the unpickling environment, and the module must contain the named object, otherwise an exception will be raised.[3]

Similarly, classes are pickled by fully qualified name, so the same restrictions in the unpickling environment apply. Note that none of the class's code or data is pickled, so in the following example the class attribute `attr` is not restored in the unpickling environment:

```
class Foo:
    attr = 'A class attribute'

picklestring = pickle.dumps(Foo)
```

[2] This is why `lambda` functions cannot be pickled: all `lambda` functions share the same name: `<lambda>`.
[3] The exception raised will likely be an `ImportError` or an `AttributeError` but it could be something else.

These restrictions are why picklable functions and classes must be defined at the top level of a module.

Similarly, when class instances are pickled, their class's code and data are not pickled along with them. Only the instance data are pickled. This is done on purpose, so you can fix bugs in a class or add methods to the class and still load objects that were created with an earlier version of the class. If you plan to have long-lived objects that will see many versions of a class, it may be worthwhile to put a version number in the objects so that suitable conversions can be made by the class's *__setstate__()* method.

12.1.5 Pickling Class Instances

In this section, we describe the general mechanisms available to you to define, customize, and control how class instances are pickled and unpickled.

In most cases, no additional code is needed to make instances picklable. By default, pickle will retrieve the class and the attributes of an instance via introspection. When a class instance is unpickled, its `__init__()` method is usually *not* invoked. The default behaviour first creates an uninitialized instance and then restores the saved attributes. The following code shows an implementation of this behaviour:

```
def save(obj):
    return (obj.__class__, obj.__dict__)

def restore(cls, attributes):
    obj = cls.__new__(cls)
    obj.__dict__.update(attributes)
    return obj
```

Classes can alter the default behaviour by providing one or several special methods:

`object.`**`__getnewargs_ex__`**`()`

In protocols 2 and newer, classes that implements the *__getnewargs_ex__()* method can dictate the values passed to the `__new__()` method upon unpickling. The method must return a pair `(args, kwargs)` where *args* is a tuple of positional arguments and *kwargs* a dictionary of named arguments for constructing the object. Those will be passed to the `__new__()` method upon unpickling.

You should implement this method if the `__new__()` method of your class requires keyword-only arguments. Otherwise, it is recommended for compatibility to implement *__getnewargs__()*.

Changed in version 3.6: *__getnewargs_ex__()* is now used in protocols 2 and 3.

`object.`**`__getnewargs__`**`()`

This method serves a similar purpose as *__getnewargs_ex__()*, but supports only positional arguments. It must return a tuple of arguments `args` which will be passed to the `__new__()` method upon unpickling.

__getnewargs__() will not be called if *__getnewargs_ex__()* is defined.

Changed in version 3.6: Before Python 3.6, *__getnewargs__()* was called instead of *__getnewargs_ex__()* in protocols 2 and 3.

`object.`**`__getstate__`**`()`

Classes can further influence how their instances are pickled by overriding the method *__getstate__()*. It is called and the returned object is pickled as the contents for the instance, instead of a default state. There are several cases:

- For a class that has no instance *__dict__* and no `__slots__`, the default state is `None`.
- For a class that has an instance *__dict__* and no `__slots__`, the default state is `self.__dict__`.

- For a class that has an instance *__dict__* and __slots__, the default state is a tuple consisting of two dictionaries: `self.__dict__`, and a dictionary mapping slot names to slot values. Only slots that have a value are included in the latter.
- For a class that has __slots__ and no instance *__dict__*, the default state is a tuple whose first item is `None` and whose second item is a dictionary mapping slot names to slot values described in the previous bullet.

Changed in version 3.11: Added the default implementation of the `__getstate__()` method in the *object* class.

`object.`**`__setstate__`**(*state*)

Upon unpickling, if the class defines *__setstate__()*, it is called with the unpickled state. In that case, there is no requirement for the state object to be a dictionary. Otherwise, the pickled state must be a dictionary and its items are assigned to the new instance's dictionary.

> **Note**
>
> If *__reduce__()* returns a state with value `None` at pickling, the *__setstate__()* method will not be called upon unpickling.

Refer to the section *Handling Stateful Objects* for more information about how to use the methods *__getstate__()* and *__setstate__()*.

> **Note**
>
> At unpickling time, some methods like `__getattr__()`, `__getattribute__()`, or `__setattr__()` may be called upon the instance. In case those methods rely on some internal invariant being true, the type should implement `__new__()` to establish such an invariant, as `__init__()` is not called when unpickling an instance.

As we shall see, pickle does not use directly the methods described above. In fact, these methods are part of the copy protocol which implements the *__reduce__()* special method. The copy protocol provides a unified interface for retrieving the data necessary for pickling and copying objects.[4]

Although powerful, implementing *__reduce__()* directly in your classes is error prone. For this reason, class designers should use the high-level interface (i.e., *__getnewargs_ex__()*, *__getstate__()* and *__setstate__()*) whenever possible. We will show, however, cases where using `__reduce__()` is the only option or leads to more efficient pickling or both.

`object.`**`__reduce__`**`()`

The interface is currently defined as follows. The *__reduce__()* method takes no argument and shall return either a string or preferably a tuple (the returned object is often referred to as the "reduce value").

If a string is returned, the string should be interpreted as the name of a global variable. It should be the object's local name relative to its module; the pickle module searches the module namespace to determine the object's module. This behaviour is typically useful for singletons.

When a tuple is returned, it must be between two and six items long. Optional items can either be omitted, or `None` can be provided as their value. The semantics of each item are in order:

- A callable object that will be called to create the initial version of the object.
- A tuple of arguments for the callable object. An empty tuple must be given if the callable does not accept any argument.

[4] The *copy* module uses this protocol for shallow and deep copying operations.

- Optionally, the object's state, which will be passed to the object's *__setstate__()* method as previously described. If the object has no such method then, the value must be a dictionary and it will be added to the object's *__dict__* attribute.
- Optionally, an iterator (and not a sequence) yielding successive items. These items will be appended to the object either using `obj.append(item)` or, in batch, using `obj.extend(list_of_items)`. This is primarily used for list subclasses, but may be used by other classes as long as they have *append and extend methods* with the appropriate signature. (Whether `append()` or `extend()` is used depends on which pickle protocol version is used as well as the number of items to append, so both must be supported.)
- Optionally, an iterator (not a sequence) yielding successive key-value pairs. These items will be stored to the object using `obj[key] = value`. This is primarily used for dictionary subclasses, but may be used by other classes as long as they implement `__setitem__()`.
- Optionally, a callable with a `(obj, state)` signature. This callable allows the user to programmatically control the state-updating behavior of a specific object, instead of using `obj`'s static *__setstate__()* method. If not `None`, this callable will have priority over `obj`'s *__setstate__()*.

  Added in version 3.8: The optional sixth tuple item, `(obj, state)`, was added.

`object.`**`__reduce_ex__`**(*protocol*)

Alternatively, a *__reduce_ex__()* method may be defined. The only difference is this method should take a single integer argument, the protocol version. When defined, pickle will prefer it over the *__reduce__()* method. In addition, *__reduce__()* automatically becomes a synonym for the extended version. The main use for this method is to provide backwards-compatible reduce values for older Python releases.

Persistence of External Objects

For the benefit of object persistence, the *pickle* module supports the notion of a reference to an object outside the pickled data stream. Such objects are referenced by a persistent ID, which should be either a string of alphanumeric characters (for protocol 0)[5] or just an arbitrary object (for any newer protocol).

The resolution of such persistent IDs is not defined by the *pickle* module; it will delegate this resolution to the user-defined methods on the pickler and unpickler, *persistent_id()* and *persistent_load()* respectively.

To pickle objects that have an external persistent ID, the pickler must have a custom *persistent_id()* method that takes an object as an argument and returns either `None` or the persistent ID for that object. When `None` is returned, the pickler simply pickles the object as normal. When a persistent ID string is returned, the pickler will pickle that object, along with a marker so that the unpickler will recognize it as a persistent ID.

To unpickle external objects, the unpickler must have a custom *persistent_load()* method that takes a persistent ID object and returns the referenced object.

Here is a comprehensive example presenting how persistent ID can be used to pickle external objects by reference.

```
# Simple example presenting how persistent ID can be used to pickle
# external objects by reference.

import pickle
import sqlite3
from collections import namedtuple

# Simple class representing a record in our database.
MemoRecord = namedtuple("MemoRecord", "key, task")
```

(continues on next page)

[5] The limitation on alphanumeric characters is due to the fact that persistent IDs in protocol 0 are delimited by the newline character. Therefore if any kind of newline characters occurs in persistent IDs, the resulting pickled data will become unreadable.

(continued from previous page)

```
class DBPickler(pickle.Pickler):

    def persistent_id(self, obj):
        # Instead of pickling MemoRecord as a regular class instance, we emit a
        # persistent ID.
        if isinstance(obj, MemoRecord):
            # Here, our persistent ID is simply a tuple, containing a tag and a
            # key, which refers to a specific record in the database.
            return ("MemoRecord", obj.key)
        else:
            # If obj does not have a persistent ID, return None. This means obj
            # needs to be pickled as usual.
            return None


class DBUnpickler(pickle.Unpickler):

    def __init__(self, file, connection):
        super().__init__(file)
        self.connection = connection

    def persistent_load(self, pid):
        # This method is invoked whenever a persistent ID is encountered.
        # Here, pid is the tuple returned by DBPickler.
        cursor = self.connection.cursor()
        type_tag, key_id = pid
        if type_tag == "MemoRecord":
            # Fetch the referenced record from the database and return it.
            cursor.execute("SELECT * FROM memos WHERE key=?", (str(key_id),))
            key, task = cursor.fetchone()
            return MemoRecord(key, task)
        else:
            # Always raises an error if you cannot return the correct object.
            # Otherwise, the unpickler will think None is the object referenced
            # by the persistent ID.
            raise pickle.UnpicklingError("unsupported persistent object")


def main():
    import io
    import pprint

    # Initialize and populate our database.
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE memos(key INTEGER PRIMARY KEY, task TEXT)")
    tasks = (
        'give food to fish',
        'prepare group meeting',
        'fight with a zebra',
        )
    for task in tasks:
        cursor.execute("INSERT INTO memos VALUES(NULL, ?)", (task,))

    # Fetch the records to be pickled.
    cursor.execute("SELECT * FROM memos")
    memos = [MemoRecord(key, task) for key, task in cursor]
```

(continues on next page)

(continued from previous page)

```
    # Save the records using our custom DBPickler.
    file = io.BytesIO()
    DBPickler(file).dump(memos)

    print("Pickled records:")
    pprint.pprint(memos)

    # Update a record, just for good measure.
    cursor.execute("UPDATE memos SET task='learn italian' WHERE key=1")

    # Load the records from the pickle data stream.
    file.seek(0)
    memos = DBUnpickler(file, conn).load()

    print("Unpickled records:")
    pprint.pprint(memos)


if __name__ == '__main__':
    main()
```

Dispatch Tables

If one wants to customize pickling of some classes without disturbing any other code which depends on pickling, then one can create a pickler with a private dispatch table.

The global dispatch table managed by the *copyreg* module is available as `copyreg.dispatch_table`. Therefore, one may choose to use a modified copy of `copyreg.dispatch_table` as a private dispatch table.

For example

```
f = io.BytesIO()
p = pickle.Pickler(f)
p.dispatch_table = copyreg.dispatch_table.copy()
p.dispatch_table[SomeClass] = reduce_SomeClass
```

creates an instance of *pickle.Pickler* with a private dispatch table which handles the `SomeClass` class specially. Alternatively, the code

```
class MyPickler(pickle.Pickler):
    dispatch_table = copyreg.dispatch_table.copy()
    dispatch_table[SomeClass] = reduce_SomeClass
f = io.BytesIO()
p = MyPickler(f)
```

does the same but all instances of `MyPickler` will by default share the private dispatch table. On the other hand, the code

```
copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)
```

modifies the global dispatch table shared by all users of the *copyreg* module.

Handling Stateful Objects

Here's an example that shows how to modify pickling behavior for a class. The `TextReader` class below opens a text file, and returns the line number and line contents each time its `readline()` method is called. If a `TextReader` instance is pickled, all attributes *except* the file object member are saved. When the instance is unpickled, the file is reopened, and reading resumes from the last location. The `__setstate__()` and `__getstate__()` methods are used to implement this behavior.

```
class TextReader:
    """Print and number lines in a text file."""

    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename)
        self.lineno = 0

    def readline(self):
        self.lineno += 1
        line = self.file.readline()
        if not line:
            return None
        if line.endswith('\n'):
            line = line[:-1]
        return "%i: %s" % (self.lineno, line)

    def __getstate__(self):
        # Copy the object's state from self.__dict__ which contains
        # all our instance attributes. Always use the dict.copy()
        # method to avoid modifying the original state.
        state = self.__dict__.copy()
        # Remove the unpicklable entries.
        del state['file']
        return state

    def __setstate__(self, state):
        # Restore instance attributes (i.e., filename and lineno).
        self.__dict__.update(state)
        # Restore the previously opened file's state. To do so, we need to
        # reopen it and read from it until the line count is restored.
        file = open(self.filename)
        for _ in range(self.lineno):
            file.readline()
        # Finally, save the file.
        self.file = file
```

A sample usage might be something like this:

```
>>> reader = TextReader("hello.txt")
>>> reader.readline()
'1: Hello world!'
>>> reader.readline()
'2: I am line number two.'
>>> new_reader = pickle.loads(pickle.dumps(reader))
>>> new_reader.readline()
'3: Goodbye!'
```

12.1.6 Custom Reduction for Types, Functions, and Other Objects

Added in version 3.8.

Sometimes, *dispatch_table* may not be flexible enough. In particular we may want to customize pickling based on another criterion than the object's type, or we may want to customize the pickling of functions and classes.

For those cases, it is possible to subclass from the *Pickler* class and implement a *reducer_override()* method. This method can return an arbitrary reduction tuple (see *__reduce__()*). It can alternatively return *NotImplemented* to fallback to the traditional behavior.

If both the *dispatch_table* and *reducer_override()* are defined, then *reducer_override()* method takes priority.

Note

For performance reasons, *reducer_override()* may not be called for the following objects: `None`, `True`, `False`, and exact instances of *int*, *float*, *bytes*, *str*, *dict*, *set*, *frozenset*, *list* and *tuple*.

Here is a simple example where we allow pickling and reconstructing a given class:

```
import io
import pickle

class MyClass:
    my_attribute = 1

class MyPickler(pickle.Pickler):
    def reducer_override(self, obj):
        """Custom reducer for MyClass."""
        if getattr(obj, "__name__", None) == "MyClass":
            return type, (obj.__name__, obj.__bases__,
                          {'my_attribute': obj.my_attribute})
        else:
            # For any other object, fallback to usual reduction
            return NotImplemented

f = io.BytesIO()
p = MyPickler(f)
p.dump(MyClass)

del MyClass

unpickled_class = pickle.loads(f.getvalue())

assert isinstance(unpickled_class, type)
assert unpickled_class.__name__ == "MyClass"
assert unpickled_class.my_attribute == 1
```

12.1.7 Out-of-band Buffers

Added in version 3.8.

In some contexts, the *pickle* module is used to transfer massive amounts of data. Therefore, it can be important to minimize the number of memory copies, to preserve performance and resource consumption. However, normal operation of the *pickle* module, as it transforms a graph-like structure of objects into a sequential stream of bytes, intrinsically involves copying data to and from the pickle stream.

This constraint can be eschewed if both the *provider* (the implementation of the object types to be transferred) and the *consumer* (the implementation of the communications system) support the out-of-band transfer facilities provided by pickle protocol 5 and higher.

Provider API

The large data objects to be pickled must implement a *__reduce_ex__()* method specialized for protocol 5 and higher, which returns a *PickleBuffer* instance (instead of e.g. a *bytes* object) for any large data.

A *PickleBuffer* object *signals* that the underlying buffer is eligible for out-of-band data transfer. Those objects remain compatible with normal usage of the *pickle* module. However, consumers can also opt-in to tell *pickle* that they will handle those buffers by themselves.

Consumer API

A communications system can enable custom handling of the *PickleBuffer* objects generated when serializing an object graph.

On the sending side, it needs to pass a *buffer_callback* argument to *Pickler* (or to the *dump()* or *dumps()* function), which will be called with each *PickleBuffer* generated while pickling the object graph. Buffers accumulated by the *buffer_callback* will not see their data copied into the pickle stream, only a cheap marker will be inserted.

On the receiving side, it needs to pass a *buffers* argument to *Unpickler* (or to the *load()* or *loads()* function), which is an iterable of the buffers which were passed to *buffer_callback*. That iterable should produce buffers in the same order as they were passed to *buffer_callback*. Those buffers will provide the data expected by the reconstructors of the objects whose pickling produced the original *PickleBuffer* objects.

Between the sending side and the receiving side, the communications system is free to implement its own transfer mechanism for out-of-band buffers. Potential optimizations include the use of shared memory or datatype-dependent compression.

Example

Here is a trivial example where we implement a *bytearray* subclass able to participate in out-of-band buffer pickling:

```
class ZeroCopyByteArray(bytearray):

    def __reduce_ex__(self, protocol):
        if protocol >= 5:
            return type(self)._reconstruct, (PickleBuffer(self),), None
        else:
            # PickleBuffer is forbidden with pickle protocols <= 4.
            return type(self)._reconstruct, (bytearray(self),)

    @classmethod
    def _reconstruct(cls, obj):
```

(continues on next page)

(continued from previous page)

```
        with memoryview(obj) as m:
            # Get a handle over the original buffer object
            obj = m.obj
            if type(obj) is cls:
                # Original buffer object is a ZeroCopyByteArray, return it
                # as-is.
                return obj
            else:
                return cls(obj)
```

The reconstructor (the `_reconstruct` class method) returns the buffer's providing object if it has the right type. This is an easy way to simulate zero-copy behaviour on this toy example.

On the consumer side, we can pickle those objects the usual way, which when unserialized will give us a copy of the original object:

```
b = ZeroCopyByteArray(b"abc")
data = pickle.dumps(b, protocol=5)
new_b = pickle.loads(data)
print(b == new_b)  # True
print(b is new_b)  # False: a copy was made
```

But if we pass a *buffer_callback* and then give back the accumulated buffers when unserializing, we are able to get back the original object:

```
b = ZeroCopyByteArray(b"abc")
buffers = []
data = pickle.dumps(b, protocol=5, buffer_callback=buffers.append)
new_b = pickle.loads(data, buffers=buffers)
print(b == new_b)  # True
print(b is new_b)  # True: no copy was made
```

This example is limited by the fact that *bytearray* allocates its own memory: you cannot create a *bytearray* instance that is backed by another object's memory. However, third-party datatypes such as NumPy arrays do not have this limitation, and allow use of zero-copy pickling (or making as few copies as possible) when transferring between distinct processes or systems.

See also

PEP 574 – Pickle protocol 5 with out-of-band data

12.1.8 Restricting Globals

By default, unpickling will import any class or function that it finds in the pickle data. For many applications, this behaviour is unacceptable as it permits the unpickler to import and invoke arbitrary code. Just consider what this hand-crafted pickle data stream does when loaded:

```
>>> import pickle
>>> pickle.loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
hello world
0
```

In this example, the unpickler imports the *os.system()* function and then apply the string argument "echo hello world". Although this example is inoffensive, it is not difficult to imagine one that could damage your system.

For this reason, you may want to control what gets unpickled by customizing *Unpickler.find_class()*. Unlike its name suggests, *Unpickler.find_class()* is called whenever a global (i.e., a class or a function) is requested. Thus it is possible to either completely forbid globals or restrict them to a safe subset.

Here is an example of an unpickler allowing only few safe classes from the *builtins* module to be loaded:

```
import builtins
import io
import pickle

safe_builtins = {
    'range',
    'complex',
    'set',
    'frozenset',
    'slice',
}

class RestrictedUnpickler(pickle.Unpickler):

    def find_class(self, module, name):
        # Only allow safe classes from builtins.
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # Forbid everything else.
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
                                     (module, name))

def restricted_loads(s):
    """Helper function analogous to pickle.loads()."""
    return RestrictedUnpickler(io.BytesIO(s)).load()
```

A sample usage of our unpickler working as intended:

```
>>> restricted_loads(pickle.dumps([1, 2, range(15)]))
[1, 2, range(0, 15)]
>>> restricted_loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'os.system' is forbidden
>>> restricted_loads(b'cbuiltins\neval\n'
...                  b'(S\'getattr(__import__("os"), "system")'
...                  b'("echo hello world")\'\ntR.')
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'builtins.eval' is forbidden
```

As our examples shows, you have to be careful with what you allow to be unpickled. Therefore if security is a concern, you may want to consider alternatives such as the marshalling API in *xmlrpc.client* or third-party solutions.

12.1.9 Performance

Recent versions of the pickle protocol (from protocol 2 and upwards) feature efficient binary encodings for several common features and built-in types. Also, the *pickle* module has a transparent optimizer written in C.

12.1.10 Examples

For the simplest code, use the *dump()* and *load()* functions.

```
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3+4j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

The following example reads the resulting pickled data.

```
import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)
```

See also

Module *copyreg*
Pickle interface constructor registration for extension types.

Module *pickletools*
Tools for working with and analyzing pickled data.

Module *shelve*
Indexed databases of objects; uses *pickle*.

Module *copy*
Shallow and deep object copying.

Module *marshal*
High-performance serialization of built-in types.

12.2 copyreg — Register pickle support functions

Source code: Lib/copyreg.py

The *copyreg* module offers a way to define functions used while pickling specific objects. The *pickle* and *copy* modules use those functions when pickling/copying those objects. The module provides configuration information about object constructors which are not classes. Such constructors may be factory functions or class instances.

copyreg.**constructor**(*object*)

Declares *object* to be a valid constructor. If *object* is not callable (and hence not valid as a constructor), raises *TypeError*.

copyreg.**pickle**(*type*, *function*, *constructor_ob=None*)

Declares that *function* should be used as a "reduction" function for objects of type *type*. *function* must return either a string or a tuple containing between two and six elements. See the *dispatch_table* for more details on the interface of *function*.

The *constructor_ob* parameter is a legacy feature and is now ignored, but if passed it must be a callable.

Note that the *dispatch_table* attribute of a pickler object or subclass of *pickle.Pickler* can also be used for declaring reduction functions.

12.2.1 Example

The example below would like to show how to register a pickle function and how it will be used:

```
>>> import copyreg, copy, pickle
>>> class C:
...     def __init__(self, a):
...         self.a = a
...
>>> def pickle_c(c):
...     print("pickling a C instance...")
...     return C, (c.a,)
...
>>> copyreg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...
```

12.3 shelve — Python object persistence

Source code: Lib/shelve.py

A "shelf" is a persistent, dictionary-like object. The difference with "dbm" databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects — anything that the *pickle* module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys are ordinary strings.

shelve.**open**(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

Open a persistent dictionary. The filename specified is the base filename for the underlying database. As a side-effect, an extension may be added to the filename and more than one file may be created. By default, the underlying database file is opened for reading and writing. The optional *flag* parameter has the same interpretation as the *flag* parameter of *dbm.open()*.

By default, pickles created with *pickle.DEFAULT_PROTOCOL* are used to serialize values. The version of the pickle protocol can be specified with the *protocol* parameter.

Because of Python semantics, a shelf cannot know when a mutable persistent-dictionary entry is modified. By default modified objects are written *only* when assigned to the shelf (see *Example*). If the optional *writeback* parameter is set to `True`, all entries accessed are also cached in memory, and written back on *sync()* and *close()*; this can make it handier to mutate mutable entries in the persistent dictionary, but, if many entries are accessed, it can consume vast amounts of memory for the cache, and it can make the close operation very slow since all accessed entries are written back (there is no way to determine which accessed entries are mutable, nor which ones were actually mutated).

Changed in version 3.10: *pickle.DEFAULT_PROTOCOL* is now used as the default pickle protocol.

Changed in version 3.11: Accepts *path-like object* for filename.

Note

Do not rely on the shelf being closed automatically; always call *close()* explicitly when you don't need it any more, or use *shelve.open()* as a context manager:

```
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

Warning

Because the *shelve* module is backed by *pickle*, it is insecure to load a shelf from an untrusted source. Like with pickle, loading a shelf can execute arbitrary code.

Shelf objects support most of methods and operations supported by dictionaries (except copying, constructors and operators `|` and `|=`). This eases the transition from dictionary based scripts to those requiring persistent storage.

Two additional methods are supported:

Shelf.**sync**()

Write back all entries in the cache if the shelf was opened with *writeback* set to *True*. Also empty the cache and synchronize the persistent dictionary on disk, if feasible. This is called automatically when the shelf is closed with *close()*.

Shelf.**close**()

Synchronize and close the persistent *dict* object. Operations on a closed shelf will fail with a *ValueError*.

See also

Persistent dictionary recipe with widely supported storage formats and having the speed of native dictionaries.

12.3.1 Restrictions

- The choice of which database package will be used (such as *dbm.ndbm* or *dbm.gnu*) depends on which interface is available. Therefore it is not safe to open the database directly using *dbm*. The database is also (unfortunately) subject to the limitations of *dbm*, if it is used — this means that (the pickled representation of) the objects stored in the database should be fairly small, and in rare cases key collisions may cause the database to refuse updates.
- The *shelve* module does not support *concurrent* read/write access to shelved objects. (Multiple simultaneous read accesses are safe.) When a program has a shelf open for writing, no other program should have it open for reading or writing. Unix file locking can be used to solve this, but this differs across Unix versions and requires knowledge about the database implementation used.
- On macOS *dbm.ndbm* can silently corrupt the database file on updates, which can cause hard crashes when trying to read from the database.

class shelve.**Shelf**(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)

A subclass of *collections.abc.MutableMapping* which stores pickled values in the *dict* object.

By default, pickles created with *pickle.DEFAULT_PROTOCOL* are used to serialize values. The version of the pickle protocol can be specified with the *protocol* parameter. See the *pickle* documentation for a discussion of the pickle protocols.

If the *writeback* parameter is `True`, the object will hold a cache of all entries accessed and write them back to the *dict* at sync and close times. This allows natural operations on mutable entries, but can consume much more memory and make sync and close take a long time.

The *keyencoding* parameter is the encoding used to encode keys before they are used with the underlying dict.

A *Shelf* object can also be used as a context manager, in which case it will be automatically closed when the `with` block ends.

Changed in version 3.2: Added the *keyencoding* parameter; previously, keys were always encoded in UTF-8.

Changed in version 3.4: Added context manager support.

Changed in version 3.10: *pickle.DEFAULT_PROTOCOL* is now used as the default pickle protocol.

class shelve.**BsdDbShelf**(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)

A subclass of *Shelf* which exposes `first()`, `next()`, `previous()`, `last()` and `set_location()` methods. These are available in the third-party `bsddb` module from pybsddb but not in other database modules. The *dict* object passed to the constructor must support those methods. This is generally accomplished by calling one of `bsddb.hashopen()`, `bsddb.btopen()` or `bsddb.rnopen()`. The optional *protocol*, *writeback*, and *keyencoding* parameters have the same interpretation as for the *Shelf* class.

class shelve.**DbfilenameShelf**(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

A subclass of *Shelf* which accepts a *filename* instead of a dict-like object. The underlying file will be opened using *dbm.open()*. By default, the file will be created and opened for both read and write. The optional *flag* parameter has the same interpretation as for the *open()* function. The optional *protocol* and *writeback* parameters have the same interpretation as for the *Shelf* class.

12.3.2 Example

To summarize the interface (`key` is a string, `data` is an arbitrary object):

```
import shelve

d = shelve.open(filename)  # open -- file may get suffix added by low-level
                           # library

d[key] = data              # store data at key (overwrites old data if
                           # using an existing key)
data = d[key]              # retrieve a COPY of data at key (raise KeyError
                           # if no such key)
del d[key]                 # delete data stored at key (raises KeyError
                           # if no such key)

flag = key in d            # true if the key exists
klist = list(d.keys())     # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = [0, 1, 2]        # this works as expected, but...
d['xx'].append(3)          # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']             # extracts the copy
temp.append(5)             # mutates the copy
d['xx'] = temp             # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()                  # close it
```

See also

Module *dbm*
Generic interface to `dbm`-style databases.

Module *pickle*
Object serialization used by *shelve*.

12.4 `marshal` — Internal Python object serialization

This module contains functions that can read and write Python values in a binary format. The format is specific to Python, but independent of machine architecture issues (e.g., you can write a Python value to a file on a PC, transport the file to a Mac, and read it back there). Details of the format are undocumented on purpose; it may change between Python versions (although it rarely does).[1]

[1] The name of this module stems from a bit of terminology used by the designers of Modula-3 (amongst others), who use the term "marshalling" for shipping of data around in a self-contained form. Strictly speaking, "to marshal" means to convert some data from internal to external form (in an RPC buffer for instance) and "unmarshalling" for the reverse process.

This is not a general "persistence" module. For general persistence and transfer of Python objects through RPC calls, see the modules *pickle* and *shelve*. The *marshal* module exists mainly to support reading and writing the "pseudo-compiled" code for Python modules of `.pyc` files. Therefore, the Python maintainers reserve the right to modify the marshal format in backward incompatible ways should the need arise. The format of code objects is not compatible between Python versions, even if the version of the format is the same. De-serializing a code object in the incorrect Python version has undefined behavior. If you're serializing and de-serializing Python objects, use the *pickle* module instead – the performance is comparable, version independence is guaranteed, and pickle supports a substantially wider range of objects than marshal.

Warning

The *marshal* module is not intended to be secure against erroneous or maliciously constructed data. Never unmarshal data received from an untrusted or unauthenticated source.

Not all Python object types are supported; in general, only objects whose value is independent from a particular invocation of Python can be written and read by this module. The following types are supported: booleans, integers, floating-point numbers, complex numbers, strings, bytes, bytearrays, tuples, lists, sets, frozensets, dictionaries, and code objects (if *allow_code* is true), where it should be understood that tuples, lists, sets, frozensets and dictionaries are only supported as long as the values contained therein are themselves supported. The singletons *None*, *Ellipsis* and *StopIteration* can also be marshalled and unmarshalled. For format *version* lower than 3, recursive lists, sets and dictionaries cannot be written (see below).

There are functions that read/write files as well as functions operating on bytes-like objects.

The module defines these functions:

`marshal.`**`dump`**(*value*, *file*, *version=version*, */*, ***, *allow_code=True*)

Write the value on the open file. The value must be a supported type. The file must be a writeable *binary file*.

If the value has (or contains an object that has) an unsupported type, a *ValueError* exception is raised — but garbage data will also be written to the file. The object will not be properly read back by *load()*. Code objects are only supported if *allow_code* is true.

The *version* argument indicates the data format that `dump` should use (see below).

Raises an *auditing event* `marshal.dumps` with arguments `value`, `version`.

Changed in version 3.13: Added the *allow_code* parameter.

`marshal.`**`load`**(*file*, */*, ***, *allow_code=True*)

Read one value from the open file and return it. If no valid value is read (e.g. because the data has a different Python version's incompatible marshal format), raise *EOFError*, *ValueError* or *TypeError*. Code objects are only supported if *allow_code* is true. The file must be a readable *binary file*.

Raises an *auditing event* `marshal.load` with no arguments.

Note

If an object containing an unsupported type was marshalled with *dump()*, *load()* will substitute `None` for the unmarshallable type.

Changed in version 3.10: This call used to raise a `code.__new__` audit event for each code object. Now it raises a single `marshal.load` event for the entire load operation.

Changed in version 3.13: Added the *allow_code* parameter.

marshal.**dumps**(*value*, *version=version*, /, *, *allow_code=True*)

Return the bytes object that would be written to a file by `dump(value, file)`. The value must be a supported type. Raise a *ValueError* exception if value has (or contains an object that has) an unsupported type. Code objects are only supported if *allow_code* is true.

The *version* argument indicates the data format that `dumps` should use (see below).

Raises an *auditing event* `marshal.dumps` with arguments `value`, `version`.

Changed in version 3.13: Added the *allow_code* parameter.

marshal.**loads**(*bytes*, /, *, *allow_code=True*)

Convert the *bytes-like object* to a value. If no valid value is found, raise *EOFError*, *ValueError* or *TypeError*. Code objects are only supported if *allow_code* is true. Extra bytes in the input are ignored.

Raises an *auditing event* `marshal.loads` with argument `bytes`.

Changed in version 3.10: This call used to raise a `code.__new__` audit event for each code object. Now it raises a single `marshal.loads` event for the entire load operation.

Changed in version 3.13: Added the *allow_code* parameter.

In addition, the following constants are defined:

marshal.**version**

Indicates the format that the module uses. Version 0 is the historical format, version 1 shares interned strings and version 2 uses a binary format for floating-point numbers. Version 3 adds support for object instancing and recursion. The current version is 4.

12.5 dbm — Interfaces to Unix "databases"

Source code: Lib/dbm/__init__.py

---

dbm is a generic interface to variants of the DBM database:

- *dbm.sqlite3*
- *dbm.gnu*
- *dbm.ndbm*

If none of these modules are installed, the slow-but-simple implementation in module *dbm.dumb* will be used. There is a third party interface to the Oracle Berkeley DB.

Availability: not WASI, not iOS.

This module does not work or is not available on WebAssembly platforms, or on iOS. See *WebAssembly platforms* for more information on WASM availability; see *iOS* for more information on iOS availability.

exception dbm.**error**

A tuple containing the exceptions that can be raised by each of the supported modules, with a unique exception also named *dbm.error* as the first item — the latter is used when *dbm.error* is raised.

dbm.**whichdb**(*filename*)

This function attempts to guess which of the several simple database modules available — *dbm.sqlite3*, *dbm.gnu*, *dbm.ndbm*, or *dbm.dumb* — should be used to open a given file.

Return one of the following values:

- `None` if the file can't be opened because it's unreadable or doesn't exist
- the empty string (`''`) if the file's format can't be guessed
- a string containing the required module name, such as `'dbm.ndbm'` or `'dbm.gnu'`

Changed in version 3.11: *filename* accepts a *path-like object*.

`dbm.`**`open`**(*file*, *flag='r'*, *mode=0o666*)

Open a database and return the corresponding database object.

Parameters

- **`file`** (*path-like object*) – The database file to open.

  If the database file already exists, the *whichdb()* function is used to determine its type and the appropriate module is used; if it does not exist, the first submodule listed above that can be imported is used.

- **`flag`** (`str`) –
  - `'r'` (default): Open existing database for reading only.
  - `'w'`: Open existing database for reading and writing.
  - `'c'`: Open database for reading and writing, creating it if it doesn't exist.
  - `'n'`: Always create a new, empty database, open for reading and writing.
- **`mode`** (`int`) – The Unix file access mode of the file (default: octal `0o666`), used only when the database has to be created.

Changed in version 3.11: *file* accepts a *path-like object*.

The object returned by *open()* supports the same basic functionality as a *dict*; keys and their corresponding values can be stored, retrieved, and deleted, and the `in` operator and the `keys()` method are available, as well as `get()` and `setdefault()` methods.

Key and values are always stored as *bytes*. This means that when strings are used they are implicitly converted to the default encoding before being stored.

These objects also support being used in a `with` statement, which will automatically close them when done.

Changed in version 3.2: `get()` and `setdefault()` methods are now available for all *dbm* backends.

Changed in version 3.4: Added native support for the context management protocol to the objects returned by *open()*.

Changed in version 3.8: Deleting a key from a read-only database raises a database module specific exception instead of *KeyError*.

The following example records some hostnames and a corresponding title, and then prints out the contents of the database:

```
import dbm

# Open database, creating it if necessary.
with dbm.open('cache', 'c') as db:

    # Record some values
    db[b'hello'] = b'there'
    db['www.python.org'] = 'Python Website'
    db['www.cnn.com'] = 'Cable News Network'

    # Note that the keys are considered bytes now.
    assert db[b'www.python.org'] == b'Python Website'
    # Notice how the value is now in bytes.
```

(continues on next page)

(continued from previous page)

```
    assert db['www.cnn.com'] == b'Cable News Network'

    # Often-used methods of the dict interface work too.
    print(db.get('python.org', b'not present'))

    # Storing a non-string key or value will raise an exception (most
    # likely a TypeError).
    db['www.yahoo.com'] = 4

# db is automatically closed when leaving the with statement.
```

See also

Module *shelve*
Persistence module which stores non-string data.

The individual submodules are described in the following sections.

12.5.1 `dbm.sqlite3` — SQLite backend for dbm

Added in version 3.13.

Source code: Lib/dbm/sqlite3.py

This module uses the standard library *sqlite3* module to provide an SQLite backend for the *dbm* module. The files created by *dbm.sqlite3* can thus be opened by *sqlite3*, or any other SQLite browser, including the SQLite CLI.

`dbm.sqlite3.open`(*filename*, */*, *flag='r'*, *mode=0o666*)

Open an SQLite database. The returned object behaves like a *mapping*, implements a `close()` method, and supports a "closing" context manager via the `with` keyword.

Parameters

- **`filename`** (*path-like object*) – The path to the database to be opened.
- **`flag`** (`str`) –
  - `'r'` (default): Open existing database for reading only.
  - `'w'`: Open existing database for reading and writing.
  - `'c'`: Open database for reading and writing, creating it if it doesn't exist.
  - `'n'`: Always create a new, empty database, open for reading and writing.
- **`mode`** – The Unix file access mode of the file (default: octal `0o666`), used only when the database has to be created.

12.5.2 dbm.gnu — GNU database manager

Source code: Lib/dbm/gnu.py

The *dbm.gnu* module provides an interface to the GDBM (GNU dbm) library, similar to the *dbm.ndbm* module, but with additional functionality like crash tolerance.

> **Note**
>
> The file formats created by *dbm.gnu* and *dbm.ndbm* are incompatible and can not be used interchangeably.

exception dbm.gnu.**error**

Raised on *dbm.gnu*-specific errors, such as I/O errors. *KeyError* is raised for general mapping errors like specifying an incorrect key.

dbm.gnu.**open**(*filename*, *flag='r'*, *mode=0o666*, */*)

Open a GDBM database and return a gdbm object.

Parameters

- **filename** (*path-like object*) – The database file to open.
- **flag** (str) –
  - 'r' (default): Open existing database for reading only.
  - 'w': Open existing database for reading and writing.
  - 'c': Open database for reading and writing, creating it if it doesn't exist.
  - 'n': Always create a new, empty database, open for reading and writing.

  The following additional characters may be appended to control how the database is opened:
  - 'f': Open the database in fast mode. Writes to the database will not be synchronized.
  - 's': Synchronized mode. Changes to the database will be written immediately to the file.
  - 'u': Do not lock database.

  Not all flags are valid for all versions of GDBM. See the *open_flags* member for a list of supported flag characters.
- **mode** (int) – The Unix file access mode of the file (default: octal 0o666), used only when the database has to be created.

Raises

error – If an invalid *flag* argument is passed.

Changed in version 3.11: *filename* accepts a *path-like object*.

dbm.gnu.**open_flags**

A string of characters the *flag* parameter of *open()* supports.

gdbm objects behave similar to *mappings*, but items() and values() methods are not supported. The following methods are also provided:

gdbm.**firstkey**()

It's possible to loop over every key in the database using this method and the *nextkey()* method. The traversal is ordered by GDBM's internal hash values, and won't be sorted by the key values. This method returns the starting key.

gdbm.**nextkey**(*key*)

Returns the key that follows *key* in the traversal. The following code prints every key in the database `db`, without having to create a list in memory that contains them all:

```
k = db.firstkey()
while k is not None:
    print(k)
    k = db.nextkey(k)
```

gdbm.**reorganize**()

If you have carried out a lot of deletions and would like to shrink the space used by the GDBM file, this routine will reorganize the database. `gdbm` objects will not shorten the length of a database file except by using this reorganization; otherwise, deleted file space will be kept and reused as new (key, value) pairs are added.

gdbm.**sync**()

When the database has been opened in fast mode, this method forces any unwritten data to be written to the disk.

gdbm.**close**()

Close the GDBM database.

gdbm.**clear**()

Remove all items from the GDBM database.

Added in version 3.13.

12.5.3 dbm.ndbm — New Database Manager

Source code: Lib/dbm/ndbm.py

The *dbm.ndbm* module provides an interface to the NDBM (New Database Manager) library. This module can be used with the "classic" NDBM interface or the GDBM compatibility interface.

Note

The file formats created by *dbm.gnu* and *dbm.ndbm* are incompatible and can not be used interchangeably.

Warning

The NDBM library shipped as part of macOS has an undocumented limitation on the size of values, which can result in corrupted database files when storing values larger than this limit. Reading such corrupted files can result in a hard crash (segmentation fault).

exception dbm.ndbm.**error**

Raised on *dbm.ndbm*-specific errors, such as I/O errors. *KeyError* is raised for general mapping errors like specifying an incorrect key.

dbm.ndbm.**library**

Name of the NDBM implementation library used.

dbm.ndbm.**open**(*filename*, *flag='r'*, *mode=0o666*, */*)

Open an NDBM database and return an ndbm object.

Parameters

- **filename** (*path-like object*) – The basename of the database file (without the .dir or .pag extensions).
- **flag** (str) –
  - 'r' (default): Open existing database for reading only.
  - 'w': Open existing database for reading and writing.
  - 'c': Open database for reading and writing, creating it if it doesn't exist.
  - 'n': Always create a new, empty database, open for reading and writing.
- **mode** (int) – The Unix file access mode of the file (default: octal 0o666), used only when the database has to be created.

ndbm objects behave similar to *mappings*, but items() and values() methods are not supported. The following methods are also provided:

Changed in version 3.11: Accepts *path-like object* for filename.

ndbm.**close**()

Close the NDBM database.

ndbm.**clear**()

Remove all items from the NDBM database.

Added in version 3.13.

12.5.4 dbm.dumb — Portable DBM implementation

Source code: Lib/dbm/dumb.py

Note

The *dbm.dumb* module is intended as a last resort fallback for the *dbm* module when a more robust module is not available. The *dbm.dumb* module is not written for speed and is not nearly as heavily used as the other database modules.

The *dbm.dumb* module provides a persistent *dict*-like interface which is written entirely in Python. Unlike other *dbm* backends, such as *dbm.gnu*, no external library is required.

The dbm.dumb module defines the following:

exception dbm.dumb.**error**

Raised on *dbm.dumb*-specific errors, such as I/O errors. *KeyError* is raised for general mapping errors like specifying an incorrect key.

dbm.dumb.**open**(*filename*, *flag='c'*, *mode=0o666*)

Open a dbm.dumb database. The returned database object behaves similar to a *mapping*, in addition to providing *sync()* and *close()* methods.

Parameters

- **filename** – The basename of the database file (without extensions). A new database creates the following files:
  - *filename*.dat
  - *filename*.dir
- **flag** (str) –
  - 'r': Open existing database for reading only.
  - 'w': Open existing database for reading and writing.
  - 'c' (default): Open database for reading and writing, creating it if it doesn't exist.
  - 'n': Always create a new, empty database, open for reading and writing.
- **mode** (int) – The Unix file access mode of the file (default: octal 0o666), used only when the database has to be created.

Warning

It is possible to crash the Python interpreter when loading a database with a sufficiently large/complex entry due to stack depth limitations in Python's AST compiler.

Changed in version 3.5: *open()* always creates a new database when *flag* is 'n'.

Changed in version 3.8: A database opened read-only if *flag* is 'r'. A database is not created if it does not exist if *flag* is 'r' or 'w'.

Changed in version 3.11: *filename* accepts a *path-like object*.

In addition to the methods provided by the *collections.abc.MutableMapping* class, the following methods are provided:

dumbdbm.**sync**()

Synchronize the on-disk directory and data files. This method is called by the Shelve.sync() method.

dumbdbm.**close**()

Close the database.

12.6 sqlite3 — DB-API 2.0 interface for SQLite databases

Source code: Lib/sqlite3/ SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

The sqlite3 module was written by Gerhard Häring. It provides an SQL interface compliant with the DB-API 2.0 specification described by **PEP 249**, and requires SQLite 3.15.2 or newer.

This document includes four main sections:

- *Tutorial* teaches how to use the sqlite3 module.
- *Reference* describes the classes and functions this module defines.
- *How-to guides* details how to handle specific tasks.
- *Explanation* provides in-depth background on transaction control.

See also

https://www.sqlite.org
The SQLite web page; the documentation describes the syntax and the available data types for the supported SQL dialect.

https://www.w3schools.com/sql/
Tutorial, reference and examples for learning SQL syntax.

PEP 249 - Database API Specification 2.0
PEP written by Marc-André Lemburg.

12.6.1 Tutorial

In this tutorial, you will create a database of Monty Python movies using basic sqlite3 functionality. It assumes a fundamental understanding of database concepts, including cursors and transactions.

First, we need to create a new database and open a database connection to allow sqlite3 to work with it. Call *sqlite3.connect()* to create a connection to the database tutorial.db in the current working directory, implicitly creating it if it does not exist:

```
import sqlite3
con = sqlite3.connect("tutorial.db")
```

The returned *Connection* object con represents the connection to the on-disk database.

In order to execute SQL statements and fetch results from SQL queries, we will need to use a database cursor. Call *con.cursor()* to create the *Cursor*:

```
cur = con.cursor()
```

Now that we've got a database connection and a cursor, we can create a database table movie with columns for title, release year, and review score. For simplicity, we can just use column names in the table declaration – thanks to the flexible typing feature of SQLite, specifying the data types is optional. Execute the CREATE TABLE statement by calling *cur.execute(...)*:

```
cur.execute("CREATE TABLE movie(title, year, score)")
```

We can verify that the new table has been created by querying the `sqlite_master` table built-in to SQLite, which should now contain an entry for the `movie` table definition (see The Schema Table for details). Execute that query by calling *`cur.execute(...)`*, assign the result to `res`, and call *`res.fetchone()`* to fetch the resulting row:

```
>>> res = cur.execute("SELECT name FROM sqlite_master")
>>> res.fetchone()
('movie',)
```

We can see that the table has been created, as the query returns a *`tuple`* containing the table's name. If we query `sqlite_master` for a non-existent table `spam`, `res.fetchone()` will return `None`:

```
>>> res = cur.execute("SELECT name FROM sqlite_master WHERE name='spam'")
>>> res.fetchone() is None
True
```

Now, add two rows of data supplied as SQL literals by executing an `INSERT` statement, once again by calling *`cur.execute(...)`*:

```
cur.execute("""
    INSERT INTO movie VALUES
        ('Monty Python and the Holy Grail', 1975, 8.2),
        ('And Now for Something Completely Different', 1971, 7.5)
""")
```

The `INSERT` statement implicitly opens a transaction, which needs to be committed before changes are saved in the database (see *Transaction control* for details). Call *`con.commit()`* on the connection object to commit the transaction:

```
con.commit()
```

We can verify that the data was inserted correctly by executing a `SELECT` query. Use the now-familiar *`cur.execute(...)`* to assign the result to `res`, and call *`res.fetchall()`* to return all resulting rows:

```
>>> res = cur.execute("SELECT score FROM movie")
>>> res.fetchall()
[(8.2,), (7.5,)]
```

The result is a *`list`* of two `tuple`s, one per row, each containing that row's `score` value.

Now, insert three more rows by calling *`cur.executemany(...)`*:

```
data = [
    ("Monty Python Live at the Hollywood Bowl", 1982, 7.9),
    ("Monty Python's The Meaning of Life", 1983, 7.5),
    ("Monty Python's Life of Brian", 1979, 8.0),
]
cur.executemany("INSERT INTO movie VALUES(?, ?, ?)", data)
con.commit()  # Remember to commit the transaction after executing INSERT.
```

Notice that `?` placeholders are used to bind `data` to the query. Always use placeholders instead of string formatting to bind Python values to SQL statements, to avoid SQL injection attacks (see *How to use placeholders to bind values in SQL queries* for more details).

We can verify that the new rows were inserted by executing a `SELECT` query, this time iterating over the results of the query:

```
>>> for row in cur.execute("SELECT year, title FROM movie ORDER BY year"):
...     print(row)
(1971, 'And Now for Something Completely Different')
(1975, 'Monty Python and the Holy Grail')
(1979, "Monty Python's Life of Brian")
(1982, 'Monty Python Live at the Hollywood Bowl')
(1983, "Monty Python's The Meaning of Life")
```

Each row is a two-item *tuple* of `(year, title)`, matching the columns selected in the query.

Finally, verify that the database has been written to disk by calling *con.close()* to close the existing connection, opening a new one, creating a new cursor, then querying the database:

```
>>> con.close()
>>> new_con = sqlite3.connect("tutorial.db")
>>> new_cur = new_con.cursor()
>>> res = new_cur.execute("SELECT title, year FROM movie ORDER BY score DESC")
>>> title, year = res.fetchone()
>>> print(f'The highest scoring Monty Python movie is {title!r}, released in {year}')
The highest scoring Monty Python movie is 'Monty Python and the Holy Grail', released␣
↪in 1975
>>> new_con.close()
```

You've now created an SQLite database using the `sqlite3` module, inserted data and retrieved values from it in multiple ways.

See also

- *How-to guides* for further reading:
  - *How to use placeholders to bind values in SQL queries*
  - *How to adapt custom Python types to SQLite values*
  - *How to convert SQLite values to custom Python types*
  - *How to use the connection context manager*
  - *How to create and use row factories*
- *Explanation* for in-depth background on transaction control.

12.6.2 Reference

Module functions

`sqlite3.`**`connect`**(*database*, *timeout=5.0*, *detect_types=0*, *isolation_level='DEFERRED'*, *check_same_thread=True*, *factory=sqlite3.Connection*, *cached_statements=128*, *uri=False*, ***, *autocommit=sqlite3.LEGACY_TRANSACTION_CONTROL*)

Open a connection to an SQLite database.

Parameters

- **`database`** (*path-like object*) – The path to the database file to be opened. You can pass `":memory:"` to create an SQLite database existing only in memory, and open a connection to it.

- **timeout** (float) – How many seconds the connection should wait before raising an *OperationalError* when a table is locked. If another connection opens a transaction to modify a table, that table will be locked until the transaction is committed. Default five seconds.
- **detect_types** (int) – Control whether and how data types not *natively supported by SQLite* are looked up to be converted to Python types, using the converters registered with *register_converter()*. Set it to any combination (using |, bitwise or) of *PARSE_DECLTYPES* and *PARSE_COLNAMES* to enable this. Column names takes precedence over declared types if both flags are set. Types cannot be detected for generated fields (for example max(data)), even when the *detect_types* parameter is set; *str* will be returned instead. By default (0), type detection is disabled.
- **isolation_level** (str | *None*) – Control legacy transaction handling behaviour. See *Connection.isolation_level* and *Transaction control via the isolation_level attribute* for more information. Can be "DEFERRED" (default), "EXCLUSIVE" or "IMMEDIATE"; or None to disable opening transactions implicitly. Has no effect unless *Connection.autocommit* is set to *LEGACY_TRANSACTION_CONTROL* (the default).
- **check_same_thread** (bool) – If True (default), *ProgrammingError* will be raised if the database connection is used by a thread other than the one that created it. If False, the connection may be accessed in multiple threads; write operations may need to be serialized by the user to avoid data corruption. See *threadsafety* for more information.
- **factory** (Connection) – A custom subclass of *Connection* to create the connection with, if not the default *Connection* class.
- **cached_statements** (int) – The number of statements that sqlite3 should internally cache for this connection, to avoid parsing overhead. By default, 128 statements.
- **uri** (bool) – If set to True, *database* is interpreted as a URI (Uniform Resource Identifier) with a file path and an optional query string. The scheme part *must* be "file:", and the path can be relative or absolute. The query string allows passing parameters to SQLite, enabling various *How to work with SQLite URIs*.
- **autocommit** (bool) – Control **PEP 249** transaction handling behaviour. See *Connection.autocommit* and *Transaction control via the autocommit attribute* for more information. *autocommit* currently defaults to *LEGACY_TRANSACTION_CONTROL*. The default will change to False in a future Python release.

Return type
Connection

Raises an *auditing event* sqlite3.connect with argument database.

Raises an *auditing event* sqlite3.connect/handle with argument connection_handle.

Changed in version 3.4: Added the *uri* parameter.

Changed in version 3.7: *database* can now also be a *path-like object*, not only a string.

Changed in version 3.10: Added the sqlite3.connect/handle auditing event.

Changed in version 3.12: Added the *autocommit* parameter.

Changed in version 3.13: Positional use of the parameters *timeout*, *detect_types*, *isolation_level*, *check_same_thread*, *factory*, *cached_statements*, and *uri* is deprecated. They will become keyword-only parameters in Python 3.15.

sqlite3.**complete_statement**(*statement*)

Return True if the string *statement* appears to contain one or more complete SQL statements. No syntactic verification or parsing of any kind is performed, other than checking that there are no unclosed string literals and the statement is terminated by a semicolon.

For example:

```
>>> sqlite3.complete_statement("SELECT foo FROM bar;")
True
>>> sqlite3.complete_statement("SELECT foo")
False
```

This function may be useful during command-line input to determine if the entered text seems to form a complete SQL statement, or if additional input is needed before calling *execute()*.

See `runsource()` in Lib/sqlite3/__main__.py for real-world use.

`sqlite3.`**`enable_callback_tracebacks`**(*flag*, `/`)

Enable or disable callback tracebacks. By default you will not get any tracebacks in user-defined functions, aggregates, converters, authorizer callbacks etc. If you want to debug them, you can call this function with *flag* set to `True`. Afterwards, you will get tracebacks from callbacks on *sys.stderr*. Use `False` to disable the feature again.

> **Note**
>
> Errors in user-defined function callbacks are logged as unraisable exceptions. Use an *unraisable hook handler* for introspection of the failed callback.

`sqlite3.`**`register_adapter`**(*type*, *adapter*, `/`)

Register an *adapter* *callable* to adapt the Python type *type* into an SQLite type. The adapter is called with a Python object of type *type* as its sole argument, and must return a value of a *type that SQLite natively understands*.

`sqlite3.`**`register_converter`**(*typename*, *converter*, `/`)

Register the *converter* *callable* to convert SQLite objects of type *typename* into a Python object of a specific type. The converter is invoked for all SQLite values of type *typename*; it is passed a *bytes* object and should return an object of the desired Python type. Consult the parameter *detect_types* of *connect()* for information regarding how type detection works.

Note: *typename* and the name of the type in your query are matched case-insensitively.

Module constants

`sqlite3.`**`LEGACY_TRANSACTION_CONTROL`**

Set *autocommit* to this constant to select old style (pre-Python 3.12) transaction control behaviour. See *Transaction control via the isolation_level attribute* for more information.

`sqlite3.`**`PARSE_COLNAMES`**

Pass this flag value to the *detect_types* parameter of *connect()* to look up a converter function by using the type name, parsed from the query column name, as the converter dictionary key. The type name must be wrapped in square brackets (`[]`).

```
SELECT p as "p [point]" FROM test;  ! will look up converter "point"
```

This flag may be combined with *PARSE_DECLTYPES* using the `|` (bitwise or) operator.

`sqlite3.`**`PARSE_DECLTYPES`**

Pass this flag value to the *detect_types* parameter of *connect()* to look up a converter function using the declared types for each column. The types are declared when the database table is created. `sqlite3` will look up a converter function using the first word of the declared type as the converter dictionary key. For example:

```
CREATE TABLE test(
   i integer primary key,  ! will look up a converter named "integer"
   p point,                ! will look up a converter named "point"
   n number(10)            ! will look up a converter named "number"
)
```

This flag may be combined with *PARSE_COLNAMES* using the | (bitwise or) operator.

sqlite3.**SQLITE_OK**

sqlite3.**SQLITE_DENY**

sqlite3.**SQLITE_IGNORE**

Flags that should be returned by the *authorizer_callback* *callable* passed to *Connection.set_authorizer()*, to indicate whether:

- Access is allowed (SQLITE_OK),
- The SQL statement should be aborted with an error (SQLITE_DENY)
- The column should be treated as a NULL value (SQLITE_IGNORE)

sqlite3.**apilevel**

String constant stating the supported DB-API level. Required by the DB-API. Hard-coded to "2.0".

sqlite3.**paramstyle**

String constant stating the type of parameter marker formatting expected by the sqlite3 module. Required by the DB-API. Hard-coded to "qmark".

Note

The named DB-API parameter style is also supported.

sqlite3.**sqlite_version**

Version number of the runtime SQLite library as a *string*.

sqlite3.**sqlite_version_info**

Version number of the runtime SQLite library as a *tuple* of *integers*.

sqlite3.**threadsafety**

Integer constant required by the DB-API 2.0, stating the level of thread safety the sqlite3 module supports. This attribute is set based on the default threading mode the underlying SQLite library is compiled with. The SQLite threading modes are:

1. **Single-thread**: In this mode, all mutexes are disabled and SQLite is unsafe to use in more than a single thread at once.
2. **Multi-thread**: In this mode, SQLite can be safely used by multiple threads provided that no single database connection is used simultaneously in two or more threads.
3. **Serialized**: In serialized mode, SQLite can be safely used by multiple threads with no restriction.

The mappings from SQLite threading modes to DB-API 2.0 threadsafety levels are as follows:

SQLite threading mode	thread-safety	SQLITE_THREADSA	DB-API 2.0 meaning
single-thread	0	0	Threads may not share the module
multi-thread	1	2	Threads may share the module, but not connections
serialized	3	1	Threads may share the module, connections and cursors

Changed in version 3.11: Set *threadsafety* dynamically instead of hard-coding it to `1`.

`sqlite3.`**`version`**

Version number of this module as a *string*. This is not the version of the SQLite library.

Deprecated since version 3.12, will be removed in version 3.14: This constant used to reflect the version number of the `pysqlite` package, a third-party library which used to upstream changes to `sqlite3`. Today, it carries no meaning or practical value.

`sqlite3.`**`version_info`**

Version number of this module as a *tuple* of *integers*. This is not the version of the SQLite library.

Deprecated since version 3.12, will be removed in version 3.14: This constant used to reflect the version number of the `pysqlite` package, a third-party library which used to upstream changes to `sqlite3`. Today, it carries no meaning or practical value.

`sqlite3.`**`SQLITE_DBCONFIG_DEFENSIVE`**
`sqlite3.`**`SQLITE_DBCONFIG_DQS_DDL`**
`sqlite3.`**`SQLITE_DBCONFIG_DQS_DML`**
`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_FKEY`**
`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER`**
`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION`**
`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_QPSG`**
`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_TRIGGER`**
`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_VIEW`**
`sqlite3.`**`SQLITE_DBCONFIG_LEGACY_ALTER_TABLE`**
`sqlite3.`**`SQLITE_DBCONFIG_LEGACY_FILE_FORMAT`**
`sqlite3.`**`SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE`**
`sqlite3.`**`SQLITE_DBCONFIG_RESET_DATABASE`**
`sqlite3.`**`SQLITE_DBCONFIG_TRIGGER_EQP`**
`sqlite3.`**`SQLITE_DBCONFIG_TRUSTED_SCHEMA`**
`sqlite3.`**`SQLITE_DBCONFIG_WRITABLE_SCHEMA`**

These constants are used for the *`Connection.setconfig()`* and *`getconfig()`* methods.

The availability of these constants varies depending on the version of SQLite Python was compiled with.

Added in version 3.12.

See also

https://www.sqlite.org/c3ref/c_dbconfig_defensive.html
SQLite docs: Database Connection Configuration Options

Connection objects

class sqlite3.**Connection**

Each open SQLite database is represented by a `Connection` object, which is created using *sqlite3.connect()*. Their main purpose is creating *Cursor* objects, and *Transaction control*.

> **See also**
>
> - *How to use connection shortcut methods*
> - *How to use the connection context manager*

Changed in version 3.13: A *ResourceWarning* is emitted if *close()* is not called before a `Connection` object is deleted.

An SQLite database connection has the following attributes and methods:

cursor(*factory=Cursor*)

Create and return a *Cursor* object. The cursor method accepts a single optional parameter *factory*. If supplied, this must be a *callable* returning an instance of *Cursor* or its subclasses.

blobopen(*table*, *column*, *row*, */*, ***, *readonly=False*, *name='main'*)

Open a *Blob* handle to an existing BLOB (Binary Large OBject).

Parameters

- **table** (str) – The name of the table where the blob is located.
- **column** (str) – The name of the column where the blob is located.
- **row** (str) – The name of the row where the blob is located.
- **readonly** (bool) – Set to `True` if the blob should be opened without write permissions. Defaults to `False`.
- **name** (str) – The name of the database where the blob is located. Defaults to `"main"`.

Raises

OperationalError – When trying to open a blob in a `WITHOUT ROWID` table.

Return type

Blob

> **Note**
>
> The blob size cannot be changed using the *Blob* class. Use the SQL function `zeroblob` to create a blob with a fixed size.

Added in version 3.11.

commit()

Commit any pending transaction to the database. If *autocommit* is `True`, or there is no open transaction, this method does nothing. If `autocommit` is `False`, a new transaction is implicitly opened if a pending transaction was committed by this method.

rollback()

Roll back to the start of any pending transaction. If *autocommit* is `True`, or there is no open transaction, this method does nothing. If `autocommit` is `False`, a new transaction is implicitly opened if a pending transaction was rolled back by this method.

close()

Close the database connection. If *autocommit* is `False`, any pending transaction is implicitly rolled back. If `autocommit` is `True` or *LEGACY_TRANSACTION_CONTROL*, no implicit transaction control is executed. Make sure to *commit()* before closing to avoid losing pending changes.

execute(*sql*, *parameters=()*, */*)

Create a new *Cursor* object and call *execute()* on it with the given *sql* and *parameters*. Return the new cursor object.

executemany(*sql*, *parameters*, */*)

Create a new *Cursor* object and call *executemany()* on it with the given *sql* and *parameters*. Return the new cursor object.

executescript(*sql_script*, */*)

Create a new *Cursor* object and call *executescript()* on it with the given *sql_script*. Return the new cursor object.

create_function(*name*, *narg*, *func*, *, *deterministic=False*)

Create or remove a user-defined SQL function.

Parameters

- **name** (`str`) – The name of the SQL function.
- **narg** (`int`) – The number of arguments the SQL function can accept. If `-1`, it may take any number of arguments.
- **func** (*callback* | None) – A *callable* that is called when the SQL function is invoked. The callable must return *a type natively supported by SQLite*. Set to `None` to remove an existing SQL function.
- **deterministic** (`bool`) – If `True`, the created SQL function is marked as deterministic, which allows SQLite to perform additional optimizations.

Changed in version 3.8: Added the *deterministic* parameter.

Example:

```
>>> import hashlib
>>> def md5sum(t):
...     return hashlib.md5(t).hexdigest()
>>> con = sqlite3.connect(":memory:")
>>> con.create_function("md5", 1, md5sum)
>>> for row in con.execute("SELECT md5(?)", (b"foo",)):
...     print(row)
('acbd18db4cc2f85cedef654fccc4a4d8',)
>>> con.close()
```

Changed in version 3.13: Passing *name*, *narg*, and *func* as keyword arguments is deprecated. These parameters will become positional-only in Python 3.15.

create_aggregate(*name*, *n_arg*, *aggregate_class*)

Create or remove a user-defined SQL aggregate function.

Parameters

- **name** (`str`) – The name of the SQL aggregate function.
- **n_arg** (`int`) – The number of arguments the SQL aggregate function can accept. If `-1`, it may take any number of arguments.
- **aggregate_class** (*class* | None) – A class must implement the following methods:
  - `step()`: Add a row to the aggregate.
  - `finalize()`: Return the final result of the aggregate as *a type natively supported by SQLite*.

  The number of arguments that the `step()` method must accept is controlled by *n_arg*.

  Set to `None` to remove an existing SQL aggregate function.

Example:

```
class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.execute("CREATE TABLE test(i)")
cur.execute("INSERT INTO test(i) VALUES(1)")
cur.execute("INSERT INTO test(i) VALUES(2)")
cur.execute("SELECT mysum(i) FROM test")
print(cur.fetchone()[0])

con.close()
```

Changed in version 3.13: Passing *name*, *n_arg*, and *aggregate_class* as keyword arguments is deprecated. These parameters will become positional-only in Python 3.15.

create_window_function(*name*, *num_params*, *aggregate_class*, */*)

Create or remove a user-defined aggregate window function.

Parameters

- **name** (`str`) – The name of the SQL aggregate window function to create or remove.
- **num_params** (`int`) – The number of arguments the SQL aggregate window function can accept. If `-1`, it may take any number of arguments.
- **aggregate_class** (*class* | None) – A class that must implement the following methods:
  - `step()`: Add a row to the current window.
  - `value()`: Return the current value of the aggregate.
  - `inverse()`: Remove a row from the current window.
  - `finalize()`: Return the final result of the aggregate as *a type natively supported by SQLite*.

  The number of arguments that the `step()` and `value()` methods must accept is controlled by *num_params*.

Set to `None` to remove an existing SQL aggregate window function.

Raises

NotSupportedError – If used with a version of SQLite older than 3.25.0, which does not support aggregate window functions.

Added in version 3.11.

Example:

```
# Example taken from https://www.sqlite.org/windowfunctions.html#udfwinfunc
class WindowSumInt:
    def __init__(self):
        self.count = 0

    def step(self, value):
        """Add a row to the current window."""
        self.count += value

    def value(self):
        """Return the current value of the aggregate."""
        return self.count

    def inverse(self, value):
        """Remove a row from the current window."""
        self.count -= value

    def finalize(self):
        """Return the final value of the aggregate.

        Any clean-up actions should be placed here.
        """
        return self.count


con = sqlite3.connect(":memory:")
cur = con.execute("CREATE TABLE test(x, y)")
values = [
    ("a", 4),
    ("b", 5),
    ("c", 3),
    ("d", 8),
    ("e", 1),
]
cur.executemany("INSERT INTO test VALUES(?, ?)", values)
con.create_window_function("sumint", 1, WindowSumInt)
cur.execute("""
    SELECT x, sumint(y) OVER (
        ORDER BY x ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
    ) AS sum_y
    FROM test ORDER BY x
""")
print(cur.fetchall())
con.close()
```

create_collation(*name*, *callable*, */*)

Create a collation named *name* using the collating function *callable*. *callable* is passed two *string* arguments, and it should return an *integer*:

- `1` if the first is ordered higher than the second

- `-1` if the first is ordered lower than the second
- `0` if they are ordered equal

The following example shows a reverse sorting collation:

```
def collate_reverse(string1, string2):
    if string1 == string2:
        return 0
    elif string1 < string2:
        return 1
    else:
        return -1

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.execute("CREATE TABLE test(x)")
cur.executemany("INSERT INTO test(x) VALUES(?)", [("a",), ("b",)])
cur.execute("SELECT x FROM test ORDER BY x COLLATE reverse")
for row in cur:
    print(row)
con.close()
```

Remove a collation function by setting *callable* to `None`.

Changed in version 3.11: The collation name can contain any Unicode character. Earlier, only ASCII characters were allowed.

interrupt()

Call this method from a different thread to abort any queries that might be executing on the connection. Aborted queries will raise an *OperationalError*.

set_authorizer(*authorizer_callback*)

Register *callable* *authorizer_callback* to be invoked for each attempt to access a column of a table in the database. The callback should return one of *SQLITE_OK*, *SQLITE_DENY*, or *SQLITE_IGNORE* to signal how access to the column should be handled by the underlying SQLite library.

The first argument to the callback signifies what kind of operation is to be authorized. The second and third argument will be arguments or `None` depending on the first argument. The 4th argument is the name of the database ("main", "temp", etc.) if applicable. The 5th argument is the name of the inner-most trigger or view that is responsible for the access attempt or `None` if this access attempt is directly from input SQL code.

Please consult the SQLite documentation about the possible values for the first argument and the meaning of the second and third argument depending on the first one. All necessary constants are available in the `sqlite3` module.

Passing `None` as *authorizer_callback* will disable the authorizer.

Changed in version 3.11: Added support for disabling the authorizer using `None`.

Changed in version 3.13: Passing *authorizer_callback* as a keyword argument is deprecated. The parameter will become positional-only in Python 3.15.

set_progress_handler(*progress_handler*, *n*)

Register *callable* *progress_handler* to be invoked for every *n* instructions of the SQLite virtual machine. This is useful if you want to get called from SQLite during long-running operations, for example to update a GUI.

If you want to clear any previously installed progress handler, call the method with `None` for *progress_handler*.

Returning a non-zero value from the handler function will terminate the currently executing query and cause it to raise a *DatabaseError* exception.

Changed in version 3.13: Passing *progress_handler* as a keyword argument is deprecated. The parameter will become positional-only in Python 3.15.

set_trace_callback(*trace_callback*)

Register *callable* *trace_callback* to be invoked for each SQL statement that is actually executed by the SQLite backend.

The only argument passed to the callback is the statement (as *str*) that is being executed. The return value of the callback is ignored. Note that the backend does not only run statements passed to the *Cursor.execute()* methods. Other sources include the *transaction management* of the sqlite3 module and the execution of triggers defined in the current database.

Passing None as *trace_callback* will disable the trace callback.

> **Note**
>
> Exceptions raised in the trace callback are not propagated. As a development and debugging aid, use *enable_callback_tracebacks()* to enable printing tracebacks from exceptions raised in the trace callback.

Added in version 3.3.

Changed in version 3.13: Passing *trace_callback* as a keyword argument is deprecated. The parameter will become positional-only in Python 3.15.

enable_load_extension(*enabled*, /)

Enable the SQLite engine to load SQLite extensions from shared libraries if *enabled* is True; else, disallow loading SQLite extensions. SQLite extensions can define new functions, aggregates or whole new virtual table implementations. One well-known extension is the fulltext-search extension distributed with SQLite.

> **Note**
>
> The sqlite3 module is not built with loadable extension support by default, because some platforms (notably macOS) have SQLite libraries which are compiled without this feature. To get loadable extension support, you must pass the --enable-loadable-sqlite-extensions option to **configure**.

Raises an *auditing event* sqlite3.enable_load_extension with arguments connection, enabled.

Added in version 3.2.

Changed in version 3.10: Added the sqlite3.enable_load_extension auditing event.

```
con.enable_load_extension(True)

# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")

# alternatively you can load the extension using an API call:
# con.load_extension("./fts3.so")

# disable extension loading again
con.enable_load_extension(False)

# example from SQLite wiki
con.execute("CREATE VIRTUAL TABLE recipe USING fts3(name, ingredients)")
```

(continues on next page)

(continued from previous page)

```
con.executescript("""
    INSERT INTO recipe (name, ingredients) VALUES('broccoli stew', 'broccoli␣
↪peppers cheese tomatoes');
    INSERT INTO recipe (name, ingredients) VALUES('pumpkin stew', 'pumpkin␣
↪onions garlic celery');
    INSERT INTO recipe (name, ingredients) VALUES('broccoli pie', 'broccoli␣
↪cheese onions flour');
    INSERT INTO recipe (name, ingredients) VALUES('pumpkin pie', 'pumpkin␣
↪sugar flour butter');
    """)
for row in con.execute("SELECT rowid, name, ingredients FROM recipe WHERE␣
↪name MATCH 'pie'"):
    print(row)
```

load_extension(*path*, */*, ***, *entrypoint=None*)

Load an SQLite extension from a shared library. Enable extension loading with *enable_load_extension()* before calling this method.

Parameters

- **path** (str) – The path to the SQLite extension.
- **entrypoint** (str | *None*) – Entry point name. If None (the default), SQLite will come up with an entry point name of its own; see the SQLite docs Loading an Extension for details.

Raises an *auditing event* sqlite3.load_extension with arguments connection, path.

Added in version 3.2.

Changed in version 3.10: Added the sqlite3.load_extension auditing event.

Changed in version 3.12: Added the *entrypoint* parameter.

iterdump(***, *filter=None*)

Return an *iterator* to dump the database as SQL source code. Useful when saving an in-memory database for later restoration. Similar to the .dump command in the **sqlite3** shell.

Parameters

filter (str | *None*) – An optional LIKE pattern for database objects to dump, e.g. prefix_%. If None (the default), all database objects will be included.

Example:

```
# Convert file example.db to SQL dump file dump.sql
con = sqlite3.connect('example.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
con.close()
```

See also

How to handle non-UTF-8 text encodings

Changed in version 3.13: Added the *filter* parameter.

backup(*target*, ***, *pages=-1*, *progress=None*, *name='main'*, *sleep=0.250*)

Create a backup of an SQLite database.

Works even if the database is being accessed by other clients or concurrently by the same connection.

Parameters

- **target** (Connection) – The database connection to save the backup to.
- **pages** (int) – The number of pages to copy at a time. If equal to or less than 0, the entire database is copied in a single step. Defaults to -1.
- **progress** (*callback* | None) – If set to a *callable*, it is invoked with three integer arguments for every backup iteration: the *status* of the last iteration, the *remaining* number of pages still to be copied, and the *total* number of pages. Defaults to None.
- **name** (str) – The name of the database to back up. Either "main" (the default) for the main database, "temp" for the temporary database, or the name of a custom database as attached using the ATTACH DATABASE SQL statement.
- **sleep** (float) – The number of seconds to sleep between successive attempts to back up remaining pages.

Example 1, copy an existing database into another:

```
def progress(status, remaining, total):
    print(f'Copied {total-remaining} of {total} pages...')

src = sqlite3.connect('example.db')
dst = sqlite3.connect('backup.db')
with dst:
    src.backup(dst, pages=1, progress=progress)
dst.close()
src.close()
```

Example 2, copy an existing database into a transient copy:

```
src = sqlite3.connect('example.db')
dst = sqlite3.connect(':memory:')
src.backup(dst)
dst.close()
src.close()
```

Added in version 3.7.

See also

How to handle non-UTF-8 text encodings

getlimit(*category*, */*)

Get a connection runtime limit.

Parameters

category (int) – The SQLite limit category to be queried.

Return type

int

Raises

ProgrammingError – If *category* is not recognised by the underlying SQLite library.

Example, query the maximum length of an SQL statement for *Connection* `con` (the default is 1000000000):

```
>>> con.getlimit(sqlite3.SQLITE_LIMIT_SQL_LENGTH)
1000000000
```

Added in version 3.11.

setlimit(*category*, *limit*, /)

Set a connection runtime limit. Attempts to increase a limit above its hard upper bound are silently truncated to the hard upper bound. Regardless of whether or not the limit was changed, the prior value of the limit is returned.

Parameters

- **category** (*int*) – The SQLite limit category to be set.
- **limit** (*int*) – The value of the new limit. If negative, the current limit is unchanged.

Return type
int

Raises
ProgrammingError – If *category* is not recognised by the underlying SQLite library.

Example, limit the number of attached databases to 1 for *Connection* `con` (the default limit is 10):

```
>>> con.setlimit(sqlite3.SQLITE_LIMIT_ATTACHED, 1)
10
>>> con.getlimit(sqlite3.SQLITE_LIMIT_ATTACHED)
1
```

Added in version 3.11.

getconfig(*op*, /)

Query a boolean connection configuration option.

Parameters
op (*int*) – A *SQLITE_DBCONFIG code*.

Return type
bool

Added in version 3.12.

setconfig(*op*, *enable=True*, /)

Set a boolean connection configuration option.

Parameters

- **op** (*int*) – A *SQLITE_DBCONFIG code*.
- **enable** (*bool*) – `True` if the configuration option should be enabled (default); `False` if it should be disabled.

Added in version 3.12.

serialize(*, *name='main'*)

Serialize a database into a *bytes* object. For an ordinary on-disk database file, the serialization is just a copy of the disk file. For an in-memory database or a "temp" database, the serialization is the same sequence of bytes which would be written to disk if that database were backed up to disk.

Parameters
name (str) – The database name to be serialized. Defaults to "main".

Return type
bytes

> **Note**
>
> This method is only available if the underlying SQLite library has the serialize API.

Added in version 3.11.

deserialize(*data*, /, *, *name='main'*)

Deserialize a *serialized* database into a *Connection*. This method causes the database connection to disconnect from database *name*, and reopen *name* as an in-memory database based on the serialization contained in *data*.

Parameters

- **data** (bytes) – A serialized database.
- **name** (str) – The database name to deserialize into. Defaults to "main".

Raises

- ***OperationalError*** – If the database connection is currently involved in a read transaction or a backup operation.
- ***DatabaseError*** – If *data* does not contain a valid SQLite database.
- ***OverflowError*** – If *len(data)* is larger than 2**63 - 1.

> **Note**
>
> This method is only available if the underlying SQLite library has the deserialize API.

Added in version 3.11.

autocommit

This attribute controls **PEP 249**-compliant transaction behaviour. autocommit has three allowed values:

- False: Select **PEP 249**-compliant transaction behaviour, implying that sqlite3 ensures a transaction is always open. Use *commit()* and *rollback()* to close transactions.

  This is the recommended value of autocommit.

- True: Use SQLite's autocommit mode. *commit()* and *rollback()* have no effect in this mode.

- *LEGACY_TRANSACTION_CONTROL*: Pre-Python 3.12 (non-**PEP 249**-compliant) transaction control. See *isolation_level* for more details.

  This is currently the default value of autocommit.

Changing autocommit to False will open a new transaction, and changing it to True will commit any pending transaction.

See *Transaction control via the autocommit attribute* for more details.

> **Note**
>
> The *isolation_level* attribute has no effect unless *autocommit* is *LEGACY_TRANSACTION_CONTROL*.

Added in version 3.12.

in_transaction

This read-only attribute corresponds to the low-level SQLite autocommit mode.

`True` if a transaction is active (there are uncommitted changes), `False` otherwise.

Added in version 3.2.

isolation_level

Controls the *legacy transaction handling mode* of `sqlite3`. If set to `None`, transactions are never implicitly opened. If set to one of `"DEFERRED"`, `"IMMEDIATE"`, or `"EXCLUSIVE"`, corresponding to the underlying SQLite transaction behaviour, *implicit transaction management* is performed.

If not overridden by the *isolation_level* parameter of *connect()*, the default is `""`, which is an alias for `"DEFERRED"`.

> **Note**
>
> Using *autocommit* to control transaction handling is recommended over using `isolation_level`. `isolation_level` has no effect unless *autocommit* is set to *LEGACY_TRANSACTION_CONTROL* (the default).

row_factory

The initial *row_factory* for *Cursor* objects created from this connection. Assigning to this attribute does not affect the `row_factory` of existing cursors belonging to this connection, only new ones. Is `None` by default, meaning each row is returned as a *tuple*.

See *How to create and use row factories* for more details.

text_factory

A *callable* that accepts a *bytes* parameter and returns a text representation of it. The callable is invoked for SQLite values with the `TEXT` data type. By default, this attribute is set to *str*.

See *How to handle non-UTF-8 text encodings* for more details.

total_changes

Return the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.

Cursor objects

A `Cursor` object represents a database cursor which is used to execute SQL statements, and manage the context of a fetch operation. Cursors are created using `Connection.cursor()`, or by using any of the *connection shortcut methods*.

Cursor objects are *iterators*, meaning that if you `execute()` a `SELECT` query, you can simply iterate over the cursor to fetch the resulting rows:

```
for row in cur.execute("SELECT t FROM data"):
    print(row)
```

class sqlite3.**Cursor**

A `Cursor` instance has the following attributes and methods.

execute(*sql*, *parameters=()*, */*)

Execute a single SQL statement, optionally binding Python values using *placeholders*.

Parameters

- **sql** (`str`) – A single SQL statement.
- **parameters** (`dict` | *sequence*) – Python values to bind to placeholders in *sql*. A `dict` if named placeholders are used. A sequence if unnamed placeholders are used. See *How to use placeholders to bind values in SQL queries*.

Raises

ProgrammingError – If *sql* contains more than one SQL statement.

If `autocommit` is `LEGACY_TRANSACTION_CONTROL`, `isolation_level` is not `None`, *sql* is an `INSERT`, `UPDATE`, `DELETE`, or `REPLACE` statement, and there is no open transaction, a transaction is implicitly opened before executing *sql*.

Deprecated since version 3.12, will be removed in version 3.14: `DeprecationWarning` is emitted if *named placeholders* are used and *parameters* is a sequence instead of a `dict`. Starting with Python 3.14, `ProgrammingError` will be raised instead.

Use `executescript()` to execute multiple SQL statements.

executemany(*sql*, *parameters*, */*)

For every item in *parameters*, repeatedly execute the *parameterized* DML (Data Manipulation Language) SQL statement *sql*.

Uses the same implicit transaction handling as `execute()`.

Parameters

- **sql** (`str`) – A single SQL DML statement.
- **parameters** (*iterable*) – An iterable of parameters to bind with the placeholders in *sql*. See *How to use placeholders to bind values in SQL queries*.

Raises

ProgrammingError – If *sql* contains more than one SQL statement, or is not a DML statement.

Example:

```
rows = [
    ("row1",),
    ("row2",),
```

(continues on next page)

(continued from previous page)

```
]
# cur is an sqlite3.Cursor object
cur.executemany("INSERT INTO data VALUES(?)", rows)
```

> **Note**
>
> Any resulting rows are discarded, including DML statements with RETURNING clauses.

Deprecated since version 3.12, will be removed in version 3.14: *DeprecationWarning* is emitted if *named placeholders* are used and the items in *parameters* are sequences instead of *dict*s. Starting with Python 3.14, *ProgrammingError* will be raised instead.

executescript(*sql_script*, */*)

Execute the SQL statements in *sql_script*. If the *autocommit* is *LEGACY_TRANSACTION_CONTROL* and there is a pending transaction, an implicit `COMMIT` statement is executed first. No other implicit transaction control is performed; any transaction control must be added to *sql_script*.

sql_script must be a *string*.

Example:

```
# cur is an sqlite3.Cursor object
cur.executescript("""
    BEGIN;
    CREATE TABLE person(firstname, lastname, age);
    CREATE TABLE book(title, author, published);
    CREATE TABLE publisher(name, address);
    COMMIT;
""")
```

fetchone()

If *row_factory* is `None`, return the next row query result set as a *tuple*. Else, pass it to the row factory and return its result. Return `None` if no more data is available.

fetchmany(*size=cursor.arraysize*)

Return the next set of rows of a query result as a *list*. Return an empty list if no more rows are available.

The number of rows to fetch per call is specified by the *size* parameter. If *size* is not given, *arraysize* determines the number of rows to be fetched. If fewer than *size* rows are available, as many rows as are available are returned.

Note there are performance considerations involved with the *size* parameter. For optimal performance, it is usually best to use the arraysize attribute. If the *size* parameter is used, then it is best for it to retain the same value from one *fetchmany()* call to the next.

fetchall()

Return all (remaining) rows of a query result as a *list*. Return an empty list if no rows are available. Note that the *arraysize* attribute can affect the performance of this operation.

close()

Close the cursor now (rather than whenever `__del__` is called).

The cursor will be unusable from this point forward; a *ProgrammingError* exception will be raised if any operation is attempted with the cursor.

setinputsizes(*sizes*, /)

Required by the DB-API. Does nothing in `sqlite3`.

setoutputsize(*size*, *column=None*, /)

Required by the DB-API. Does nothing in `sqlite3`.

arraysize

Read/write attribute that controls the number of rows returned by *fetchmany()*. The default value is 1 which means a single row would be fetched per call.

connection

Read-only attribute that provides the SQLite database *Connection* belonging to the cursor. A *Cursor* object created by calling *con.cursor()* will have a *connection* attribute that refers to *con*:

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
>>> con.close()
```

description

Read-only attribute that provides the column names of the last query. To remain compatible with the Python DB API, it returns a 7-tuple for each column where the last six items of each tuple are `None`.

It is set for `SELECT` statements without any matching rows as well.

lastrowid

Read-only attribute that provides the row id of the last inserted row. It is only updated after successful `INSERT` or `REPLACE` statements using the *execute()* method. For other statements, after *executemany()* or *executescript()*, or if the insertion failed, the value of `lastrowid` is left unchanged. The initial value of `lastrowid` is `None`.

> **Note**
>
> Inserts into `WITHOUT ROWID` tables are not recorded.

Changed in version 3.6: Added support for the `REPLACE` statement.

rowcount

Read-only attribute that provides the number of modified rows for `INSERT`, `UPDATE`, `DELETE`, and `REPLACE` statements; is `-1` for other statements, including CTE (Common Table Expression) queries. It is only updated by the *execute()* and *executemany()* methods, after the statement has run to completion. This means that any resulting rows must be fetched in order for `rowcount` to be updated.

row_factory

Control how a row fetched from this `Cursor` is represented. If `None`, a row is represented as a *tuple*. Can be set to the included *sqlite3.Row*; or a *callable* that accepts two arguments, a *Cursor* object and the `tuple` of row values, and returns a custom object representing an SQLite row.

Defaults to what *Connection.row_factory* was set to when the `Cursor` was created. Assigning to this attribute does not affect *Connection.row_factory* of the parent connection.

See *How to create and use row factories* for more details.

Row objects

class sqlite3.**Row**

A Row instance serves as a highly optimized *row_factory* for *Connection* objects. It supports iteration, equality testing, *len()*, and *mapping* access by column name and index.

Two Row objects compare equal if they have identical column names and values.

See *How to create and use row factories* for more details.

keys()

Return a *list* of column names as *strings*. Immediately after a query, it is the first member of each tuple in *Cursor.description*.

Changed in version 3.5: Added support of slicing.

Blob objects

class sqlite3.**Blob**

Added in version 3.11.

A *Blob* instance is a *file-like object* that can read and write data in an SQLite BLOB. Call *len(blob)* to get the size (number of bytes) of the blob. Use indices and *slices* for direct access to the blob data.

Use the *Blob* as a *context manager* to ensure that the blob handle is closed after use.

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE test(blob_col blob)")
con.execute("INSERT INTO test(blob_col) VALUES(zeroblob(13))")

# Write to our blob, using two write operations:
with con.blobopen("test", "blob_col", 1) as blob:
    blob.write(b"hello, ")
    blob.write(b"world.")
    # Modify the first and last bytes of our blob
    blob[0] = ord("H")
    blob[-1] = ord("!")

# Read the contents of our blob
with con.blobopen("test", "blob_col", 1) as blob:
    greeting = blob.read()

print(greeting)  # outputs "b'Hello, world!'"
con.close()
```

close()

Close the blob.

The blob will be unusable from this point onward. An *Error* (or subclass) exception will be raised if any further operation is attempted with the blob.

read(*length=-1*, */*)

Read *length* bytes of data from the blob at the current offset position. If the end of the blob is reached, the data up to EOF (End of File) will be returned. When *length* is not specified, or is negative, *read()* will read until the end of the blob.

write(*data*, /)

Write *data* to the blob at the current offset. This function cannot change the blob length. Writing beyond the end of the blob will raise `ValueError`.

tell()

Return the current access position of the blob.

seek(*offset*, *origin=os.SEEK_SET*, /)

Set the current access position of the blob to *offset*. The *origin* argument defaults to `os.SEEK_SET` (absolute blob positioning). Other values for *origin* are `os.SEEK_CUR` (seek relative to the current position) and `os.SEEK_END` (seek relative to the blob's end).

PrepareProtocol objects

class sqlite3.**PrepareProtocol**

The PrepareProtocol type's single purpose is to act as a **PEP 246** style adaption protocol for objects that can *adapt themselves* to *native SQLite types*.

Exceptions

The exception hierarchy is defined by the DB-API 2.0 (**PEP 249**).

exception sqlite3.**Warning**

This exception is not currently raised by the `sqlite3` module, but may be raised by applications using `sqlite3`, for example if a user-defined function truncates data while inserting. `Warning` is a subclass of `Exception`.

exception sqlite3.**Error**

The base class of the other exceptions in this module. Use this to catch all errors with one single `except` statement. `Error` is a subclass of `Exception`.

If the exception originated from within the SQLite library, the following two attributes are added to the exception:

sqlite_errorcode

The numeric error code from the SQLite API

Added in version 3.11.

sqlite_errorname

The symbolic name of the numeric error code from the SQLite API

Added in version 3.11.

exception sqlite3.**InterfaceError**

Exception raised for misuse of the low-level SQLite C API. In other words, if this exception is raised, it probably indicates a bug in the `sqlite3` module. `InterfaceError` is a subclass of `Error`.

exception sqlite3.**DatabaseError**

Exception raised for errors that are related to the database. This serves as the base exception for several types of database errors. It is only raised implicitly through the specialised subclasses. `DatabaseError` is a subclass of `Error`.

exception sqlite3.**DataError**

Exception raised for errors caused by problems with the processed data, like numeric values out of range, and strings which are too long. `DataError` is a subclass of `DatabaseError`.

exception sqlite3.**OperationalError**

Exception raised for errors that are related to the database's operation, and not necessarily under the control of the programmer. For example, the database path is not found, or a transaction could not be processed. `OperationalError` is a subclass of *DatabaseError*.

exception sqlite3.**IntegrityError**

Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails. It is a subclass of *DatabaseError*.

exception sqlite3.**InternalError**

Exception raised when SQLite encounters an internal error. If this is raised, it may indicate that there is a problem with the runtime SQLite library. `InternalError` is a subclass of *DatabaseError*.

exception sqlite3.**ProgrammingError**

Exception raised for `sqlite3` API programming errors, for example supplying the wrong number of bindings to a query, or trying to operate on a closed *Connection*. `ProgrammingError` is a subclass of *DatabaseError*.

exception sqlite3.**NotSupportedError**

Exception raised in case a method or database API is not supported by the underlying SQLite library. For example, setting *deterministic* to `True` in *create_function()*, if the underlying SQLite library does not support deterministic functions. `NotSupportedError` is a subclass of *DatabaseError*.

SQLite and Python types

SQLite natively supports the following types: `NULL`, `INTEGER`, `REAL`, `TEXT`, `BLOB`.

The following Python types can thus be sent to SQLite without any problem:

Python type	SQLite type
`None`	`NULL`
int	`INTEGER`
float	`REAL`
str	`TEXT`
bytes	`BLOB`

This is how SQLite types are converted to Python types by default:

SQLite type	Python type
`NULL`	`None`
`INTEGER`	*int*
`REAL`	*float*
`TEXT`	depends on *text_factory*, *str* by default
`BLOB`	*bytes*

The type system of the `sqlite3` module is extensible in two ways: you can store additional Python types in an SQLite database via *object adapters*, and you can let the `sqlite3` module convert SQLite types to Python types via *converters*.

Default adapters and converters (deprecated)

> **Note**
>
> The default adapters and converters are deprecated as of Python 3.12. Instead, use the *Adapter and converter recipes* and tailor them to your needs.

The deprecated default adapters and converters consist of:

- An adapter for `datetime.date` objects to `strings` in ISO 8601 format.
- An adapter for `datetime.datetime` objects to strings in ISO 8601 format.
- A converter for *declared* "date" types to `datetime.date` objects.
- A converter for declared "timestamp" types to `datetime.datetime` objects. Fractional parts will be truncated to 6 digits (microsecond precision).

> **Note**
>
> The default "timestamp" converter ignores UTC offsets in the database and always returns a naive `datetime.datetime` object. To preserve UTC offsets in timestamps, either leave converters disabled, or register an offset-aware converter with `register_converter()`.

Deprecated since version 3.12.

Command-line interface

The `sqlite3` module can be invoked as a script, using the interpreter's `-m` switch, in order to provide a simple SQLite shell. The argument signature is as follows:

```
python -m sqlite3 [-h] [-v] [filename] [sql]
```

Type `.quit` or CTRL-D to exit the shell.

`-h, --help`
: Print CLI help.

`-v, --version`
: Print underlying SQLite library version.

Added in version 3.12.

12.6.3 How-to guides

How to use placeholders to bind values in SQL queries

SQL operations usually need to use values from Python variables. However, beware of using Python's string operations to assemble queries, as they are vulnerable to SQL injection attacks. For example, an attacker can simply close the single quote and inject `OR TRUE` to select all rows:

```
>>> # Never do this -- insecure!
>>> symbol = input()
' OR TRUE; --
>>> sql = "SELECT * FROM stocks WHERE symbol = '%s'" % symbol
>>> print(sql)
SELECT * FROM stocks WHERE symbol = '' OR TRUE; --'
>>> cur.execute(sql)
```

Instead, use the DB-API's parameter substitution. To insert a variable into a query string, use a placeholder in the string, and substitute the actual values into the query by providing them as a *tuple* of values to the second argument of the cursor's *execute()* method.

An SQL statement may use one of two kinds of placeholders: question marks (qmark style) or named placeholders (named style). For the qmark style, *parameters* must be a *sequence* whose length must match the number of placeholders, or a *ProgrammingError* is raised. For the named style, *parameters* must be an instance of a *dict* (or a subclass), which must contain keys for all named parameters; any extra items are ignored. Here's an example of both styles:

```
con = sqlite3.connect(":memory:")
cur = con.execute("CREATE TABLE lang(name, first_appeared)")

# This is the named style used with executemany():
data = (
    {"name": "C", "year": 1972},
    {"name": "Fortran", "year": 1957},
    {"name": "Python", "year": 1991},
    {"name": "Go", "year": 2009},
)
cur.executemany("INSERT INTO lang VALUES(:name, :year)", data)

# This is the qmark style used in a SELECT query:
params = (1972,)
cur.execute("SELECT * FROM lang WHERE first_appeared = ?", params)
print(cur.fetchall())
con.close()
```

Note

PEP 249 numeric placeholders are *not* supported. If used, they will be interpreted as named placeholders.

How to adapt custom Python types to SQLite values

SQLite supports only a limited set of data types natively. To store custom Python types in SQLite databases, *adapt* them to one of the *Python types SQLite natively understands*.

There are two ways to adapt Python objects to SQLite types: letting your object adapt itself, or using an *adapter callable*. The latter will take precedence above the former. For a library that exports a custom type, it may make sense to enable that type to adapt itself. As an application developer, it may make more sense to take direct control by registering custom adapter functions.

How to write adaptable objects

Suppose we have a `Point` class that represents a pair of coordinates, `x` and `y`, in a Cartesian coordinate system. The coordinate pair will be stored as a text string in the database, using a semicolon to separate the coordinates. This can be implemented by adding a `__conform__(self, protocol)` method which returns the adapted value. The object passed to *protocol* will be of type *PrepareProtocol*.

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return f"{self.x};{self.y}"

con = sqlite3.connect(":memory:")
cur = con.cursor()

cur.execute("SELECT ?", (Point(4.0, -3.2),))
print(cur.fetchone()[0])
con.close()
```

How to register adapter callables

The other possibility is to create a function that converts the Python object to an SQLite-compatible type. This function can then be registered using *register_adapter()*.

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return f"{point.x};{point.y}"

sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()

cur.execute("SELECT ?", (Point(1.0, 2.5),))
print(cur.fetchone()[0])
con.close()
```

How to convert SQLite values to custom Python types

Writing an adapter lets you convert *from* custom Python types *to* SQLite values. To be able to convert *from* SQLite values *to* custom Python types, we use *converters*.

Let's go back to the `Point` class. We stored the x and y coordinates separated via semicolons as strings in SQLite.

First, we'll define a converter function that accepts the string as a parameter and constructs a `Point` object from it.

> **Note**
>
> Converter functions are **always** passed a *bytes* object, no matter the underlying SQLite data type.

```
def convert_point(s):
    x, y = map(float, s.split(b";"))
    return Point(x, y)
```

We now need to tell `sqlite3` when it should convert a given SQLite value. This is done when connecting to a database, using the *detect_types* parameter of *connect()*. There are three options:

- Implicit: set *detect_types* to *PARSE_DECLTYPES*
- Explicit: set *detect_types* to *PARSE_COLNAMES*
- Both: set *detect_types* to `sqlite3.PARSE_DECLTYPES | sqlite3.PARSE_COLNAMES`. Column names take precedence over declared types.

The following example illustrates the implicit and explicit approaches:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return f"Point({self.x}, {self.y})"

def adapt_point(point):
    return f"{point.x};{point.y}"

def convert_point(s):
    x, y = list(map(float, s.split(b";")))
    return Point(x, y)

# Register the adapter and converter
sqlite3.register_adapter(Point, adapt_point)
sqlite3.register_converter("point", convert_point)

# 1) Parse using declared types
p = Point(4.0, -3.2)
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.execute("CREATE TABLE test(p point)")

cur.execute("INSERT INTO test(p) VALUES(?)", (p,))
cur.execute("SELECT p FROM test")
print("with declared types:", cur.fetchone()[0])
cur.close()
con.close()

# 2) Parse using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.execute("CREATE TABLE test(p)")

cur.execute("INSERT INTO test(p) VALUES(?)", (p,))
cur.execute('SELECT p AS "p [point]" FROM test')
print("with column names:", cur.fetchone()[0])
cur.close()
```

(continues on next page)

(continued from previous page)

```
con.close()
```

Adapter and converter recipes

This section shows recipes for common adapters and converters.

```
import datetime
import sqlite3

def adapt_date_iso(val):
    """Adapt datetime.date to ISO 8601 date."""
    return val.isoformat()

def adapt_datetime_iso(val):
    """Adapt datetime.datetime to timezone-naive ISO 8601 date."""
    return val.isoformat()

def adapt_datetime_epoch(val):
    """Adapt datetime.datetime to Unix timestamp."""
    return int(val.timestamp())

sqlite3.register_adapter(datetime.date, adapt_date_iso)
sqlite3.register_adapter(datetime.datetime, adapt_datetime_iso)
sqlite3.register_adapter(datetime.datetime, adapt_datetime_epoch)

def convert_date(val):
    """Convert ISO 8601 date to datetime.date object."""
    return datetime.date.fromisoformat(val.decode())

def convert_datetime(val):
    """Convert ISO 8601 datetime to datetime.datetime object."""
    return datetime.datetime.fromisoformat(val.decode())

def convert_timestamp(val):
    """Convert Unix epoch timestamp to datetime.datetime object."""
    return datetime.datetime.fromtimestamp(int(val))

sqlite3.register_converter("date", convert_date)
sqlite3.register_converter("datetime", convert_datetime)
sqlite3.register_converter("timestamp", convert_timestamp)
```

How to use connection shortcut methods

Using the *execute()*, *executemany()*, and *executescript()* methods of the *Connection* class, your code can be written more concisely because you don't have to create the (often superfluous) *Cursor* objects explicitly. Instead, the *Cursor* objects are created implicitly and these shortcut methods return the cursor objects. This way, you can execute a SELECT statement and iterate over it directly using only a single call on the *Connection* object.

```
# Create and fill the table.
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(name, first_appeared)")
data = [
    ("C++", 1985),
```

(continues on next page)

(continued from previous page)

```
    ("Objective-C", 1984),
]
con.executemany("INSERT INTO lang(name, first_appeared) VALUES(?, ?)", data)

# Print the table contents
for row in con.execute("SELECT name, first_appeared FROM lang"):
    print(row)

print("I just deleted", con.execute("DELETE FROM lang").rowcount, "rows")

# close() is not a shortcut method and it's not called automatically;
# the connection object should be closed manually
con.close()
```

How to use the connection context manager

A *Connection* object can be used as a context manager that automatically commits or rolls back open transactions when leaving the body of the context manager. If the body of the `with` statement finishes without exceptions, the transaction is committed. If this commit fails, or if the body of the `with` statement raises an uncaught exception, the transaction is rolled back. If *autocommit* is `False`, a new transaction is implicitly opened after committing or rolling back.

If there is no open transaction upon leaving the body of the `with` statement, or if *autocommit* is `True`, the context manager does nothing.

Note

The context manager neither implicitly opens a new transaction nor closes the connection. If you need a closing context manager, consider using *contextlib.closing()*.

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(id INTEGER PRIMARY KEY, name VARCHAR UNIQUE)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))

# con.rollback() is called after the with block finishes with an exception,
# the exception is still raised and must be caught
try:
    with con:
        con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))
except sqlite3.IntegrityError:
    print("couldn't add Python twice")

# Connection object used as context manager only commits or rollbacks transactions,
# so the connection object should be closed manually
con.close()
```

How to work with SQLite URIs

Some useful URI tricks include:

- Open a database in read-only mode:

```
>>> con = sqlite3.connect("file:tutorial.db?mode=ro", uri=True)
>>> con.execute("CREATE TABLE readonly(data)")
Traceback (most recent call last):
OperationalError: attempt to write a readonly database
```

- Do not implicitly create a new database file if it does not already exist; will raise *OperationalError* if unable to create a new file:

```
>>> con = sqlite3.connect("file:nosuchdb.db?mode=rw", uri=True)
Traceback (most recent call last):
OperationalError: unable to open database file
```

- Create a shared named in-memory database:

```
db = "file:mem1?mode=memory&cache=shared"
con1 = sqlite3.connect(db, uri=True)
con2 = sqlite3.connect(db, uri=True)
with con1:
    con1.execute("CREATE TABLE shared(data)")
    con1.execute("INSERT INTO shared VALUES(28)")
res = con2.execute("SELECT data FROM shared")
assert res.fetchone() == (28,)

con1.close()
con2.close()
```

More information about this feature, including a list of parameters, can be found in the SQLite URI documentation.

How to create and use row factories

By default, `sqlite3` represents each row as a *tuple*. If a `tuple` does not suit your needs, you can use the *sqlite3.Row* class or a custom *row_factory*.

While `row_factory` exists as an attribute both on the *Cursor* and the *Connection*, it is recommended to set *Connection.row_factory*, so all cursors created from the connection will use the same row factory.

`Row` provides indexed and case-insensitive named access to columns, with minimal memory overhead and performance impact over a `tuple`. To use `Row` as a row factory, assign it to the `row_factory` attribute:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = sqlite3.Row
```

Queries now return `Row` objects:

```
>>> res = con.execute("SELECT 'Earth' AS name, 6378 AS radius")
>>> row = res.fetchone()
>>> row.keys()
['name', 'radius']
>>> row[0]          # Access by index.
'Earth'
>>> row["name"]     # Access by name.
```

(continues on next page)

(continued from previous page)

```
'Earth'
>>> row["RADIUS"]  # Column names are case-insensitive.
6378
>>> con.close()
```

Note

The `FROM` clause can be omitted in the `SELECT` statement, as in the above example. In such cases, SQLite returns a single row with columns defined by expressions, e.g. literals, with the given aliases `expr AS alias`.

You can create a custom *row_factory* that returns each row as a *dict*, with column names mapped to values:

```
def dict_factory(cursor, row):
    fields = [column[0] for column in cursor.description]
    return {key: value for key, value in zip(fields, row)}
```

Using it, queries now return a `dict` instead of a `tuple`:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = dict_factory
>>> for row in con.execute("SELECT 1 AS a, 2 AS b"):
...     print(row)
{'a': 1, 'b': 2}
>>> con.close()
```

The following row factory returns a *named tuple*:

```
from collections import namedtuple

def namedtuple_factory(cursor, row):
    fields = [column[0] for column in cursor.description]
    cls = namedtuple("Row", fields)
    return cls._make(row)
```

`namedtuple_factory()` can be used as follows:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = namedtuple_factory
>>> cur = con.execute("SELECT 1 AS a, 2 AS b")
>>> row = cur.fetchone()
>>> row
Row(a=1, b=2)
>>> row[0]  # Indexed access.
1
>>> row.b   # Attribute access.
2
>>> con.close()
```

With some adjustments, the above recipe can be adapted to use a *dataclass*, or any other custom class, instead of a *namedtuple*.

How to handle non-UTF-8 text encodings

By default, `sqlite3` uses *str* to adapt SQLite values with the `TEXT` data type. This works well for UTF-8 encoded text, but it might fail for other encodings and invalid UTF-8. You can use a custom *text_factory* to handle such cases.

Because of SQLite's flexible typing, it is not uncommon to encounter table columns with the `TEXT` data type containing non-UTF-8 encodings, or even arbitrary data. To demonstrate, let's assume we have a database with ISO-8859-2 (Latin-2) encoded text, for example a table of Czech-English dictionary entries. Assuming we now have a *Connection* instance `con` connected to this database, we can decode the Latin-2 encoded text using this *text_factory*:

```
con.text_factory = lambda data: str(data, encoding="latin2")
```

For invalid UTF-8 or arbitrary data in stored in `TEXT` table columns, you can use the following technique, borrowed from the unicode-howto:

```
con.text_factory = lambda data: str(data, errors="surrogateescape")
```

Note

The `sqlite3` module API does not support strings containing surrogates.

See also

unicode-howto

12.6.4 Explanation

Transaction control

`sqlite3` offers multiple methods of controlling whether, when and how database transactions are opened and closed. *Transaction control via the autocommit attribute* is recommended, while *Transaction control via the isolation_level attribute* retains the pre-Python 3.12 behaviour.

Transaction control via the `autocommit` attribute

The recommended way of controlling transaction behaviour is through the *Connection.autocommit* attribute, which should preferably be set using the *autocommit* parameter of *connect()*.

It is suggested to set *autocommit* to `False`, which implies **PEP 249**-compliant transaction control. This means:

- `sqlite3` ensures that a transaction is always open, so *connect()*, *Connection.commit()*, and *Connection.rollback()* will implicitly open a new transaction (immediately after closing the pending one, for the latter two). `sqlite3` uses `BEGIN DEFERRED` statements when opening transactions.
- Transactions should be committed explicitly using `commit()`.
- Transactions should be rolled back explicitly using `rollback()`.
- An implicit rollback is performed if the database is *close()*-ed with pending changes.

Set *autocommit* to `True` to enable SQLite's autocommit mode. In this mode, *Connection.commit()* and *Connection.rollback()* have no effect. Note that SQLite's autocommit mode is distinct from the **PEP 249**-compliant *Connection.autocommit* attribute; use *Connection.in_transaction* to query the low-level SQLite autocommit mode.

Set *autocommit* to *LEGACY_TRANSACTION_CONTROL* to leave transaction control behaviour to the *Connection.isolation_level* attribute. See *Transaction control via the isolation_level attribute* for more information.

Transaction control via the `isolation_level` attribute

> **Note**
>
> The recommended way of controlling transactions is via the *autocommit* attribute. See *Transaction control via the autocommit attribute*.

If *Connection.autocommit* is set to *LEGACY_TRANSACTION_CONTROL* (the default), transaction behaviour is controlled using the *Connection.isolation_level* attribute. Otherwise, `isolation_level` has no effect.

If the connection attribute *isolation_level* is not `None`, new transactions are implicitly opened before *execute()* and *executemany()* executes `INSERT`, `UPDATE`, `DELETE`, or `REPLACE` statements; for other statements, no implicit transaction handling is performed. Use the *commit()* and *rollback()* methods to respectively commit and roll back pending transactions. You can choose the underlying SQLite transaction behaviour — that is, whether and what type of `BEGIN` statements `sqlite3` implicitly executes – via the *isolation_level* attribute.

If *isolation_level* is set to `None`, no transactions are implicitly opened at all. This leaves the underlying SQLite library in autocommit mode, but also allows the user to perform their own transaction handling using explicit SQL statements. The underlying SQLite library autocommit mode can be queried using the *in_transaction* attribute.

The *executescript()* method implicitly commits any pending transaction before execution of the given SQL script, regardless of the value of *isolation_level*.

Changed in version 3.6: `sqlite3` used to implicitly commit an open transaction before DDL statements. This is no longer the case.

Changed in version 3.12: The recommended way of controlling transactions is now via the *autocommit* attribute.

CHAPTER

THIRTEEN

DATA COMPRESSION AND ARCHIVING

The modules described in this chapter support data compression with the zlib, gzip, bzip2 and lzma algorithms, and the creation of ZIP- and tar-format archives. See also *Archiving operations* provided by the `shutil` module.

13.1 `zlib` — Compression compatible with `gzip`

For applications that require data compression, the functions in this module allow compression and decompression, using the zlib library. The zlib library has its own home page at https://www.zlib.net. There are known incompatibilities between the Python module and versions of the zlib library earlier than 1.1.3; 1.1.3 has a security vulnerability, so we recommend using 1.1.4 or later.

zlib's functions have many options and often need to be used in a particular order. This documentation doesn't attempt to cover all of the permutations; consult the zlib manual at http://www.zlib.net/manual.html for authoritative information.

For reading and writing `.gz` files see the `gzip` module.

The available exception and functions in this module are:

exception zlib.**error**

Exception raised on compression and decompression errors.

zlib.**adler32**(*data*[, *value*])

Computes an Adler-32 checksum of *data*. (An Adler-32 checksum is almost as reliable as a CRC32 but can be computed much more quickly.) The result is an unsigned 32-bit integer. If *value* is present, it is used as the starting value of the checksum; otherwise, a default value of 1 is used. Passing in *value* allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.

Changed in version 3.0: The result is always unsigned.

zlib.**compress**(*data*, */*, *level=-1*, *wbits=MAX_WBITS*)

Compresses the bytes in *data*, returning a bytes object containing compressed data. *level* is an integer from `0` to `9` or `-1` controlling the level of compression; `1` (Z_BEST_SPEED) is fastest and produces the least compression, `9` (Z_BEST_COMPRESSION) is slowest and produces the most. `0` (Z_NO_COMPRESSION) is no compression. The default value is `-1` (Z_DEFAULT_COMPRESSION). Z_DEFAULT_COMPRESSION represents a default compromise between speed and compression (currently equivalent to level 6).

The *wbits* argument controls the size of the history buffer (or the "window size") used when compressing data, and whether a header and trailer is included in the output. It can take several ranges of values, defaulting to `15` (MAX_WBITS):

- +9 to +15: The base-two logarithm of the window size, which therefore ranges between 512 and 32768. Larger values produce better compression at the expense of greater memory usage. The resulting output will include a zlib-specific header and trailer.
- −9 to −15: Uses the absolute value of *wbits* as the window size logarithm, while producing a raw output stream with no header or trailing checksum.
- +25 to +31 = 16 + (9 to 15): Uses the low 4 bits of the value as the window size logarithm, while including a basic **gzip** header and trailing checksum in the output.

Raises the `error` exception if any error occurs.

Changed in version 3.6: *level* can now be used as a keyword parameter.

Changed in version 3.11: The *wbits* parameter is now available to set window bits and compression type.

zlib.**compressobj**(*level=-1*, *method=DEFLATED*, *wbits=MAX_WBITS*, *memLevel=DEF_MEM_LEVEL*, *strategy=Z_DEFAULT_STRATEGY*[, *zdict*])

Returns a compression object, to be used for compressing data streams that won't fit into memory at once.

level is the compression level – an integer from `0` to `9` or `-1`. A value of `1` (Z_BEST_SPEED) is fastest and produces the least compression, while a value of `9` (Z_BEST_COMPRESSION) is slowest and produces the most. `0` (Z_NO_COMPRESSION) is no compression. The default value is `-1` (Z_DEFAULT_COMPRESSION). Z_DEFAULT_COMPRESSION represents a default compromise between speed and compression (currently equivalent to level 6).

method is the compression algorithm. Currently, the only supported value is `DEFLATED`.

The *wbits* parameter controls the size of the history buffer (or the "window size"), and what header and trailer format will be used. It has the same meaning as *described for compress()*.

The *memLevel* argument controls the amount of memory used for the internal compression state. Valid values range from `1` to `9`. Higher values use more memory, but are faster and produce smaller output.

strategy is used to tune the compression algorithm. Possible values are `Z_DEFAULT_STRATEGY`, `Z_FILTERED`, `Z_HUFFMAN_ONLY`, `Z_RLE` (zlib 1.2.0.1) and `Z_FIXED` (zlib 1.2.2.2).

zdict is a predefined compression dictionary. This is a sequence of bytes (such as a `bytes` object) containing subsequences that are expected to occur frequently in the data that is to be compressed. Those subsequences that are expected to be most common should come at the end of the dictionary.

Changed in version 3.3: Added the *zdict* parameter and keyword argument support.

zlib.**crc32**(*data*[, *value*])

Computes a CRC (Cyclic Redundancy Check) checksum of *data*. The result is an unsigned 32-bit integer. If *value* is present, it is used as the starting value of the checksum; otherwise, a default value of 0 is used. Passing in *value* allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.

Changed in version 3.0: The result is always unsigned.

zlib.**decompress**(*data*, */*, *wbits=MAX_WBITS*, *bufsize=DEF_BUF_SIZE*)

Decompresses the bytes in *data*, returning a bytes object containing the uncompressed data. The *wbits* parameter depends on the format of *data*, and is discussed further below. If *bufsize* is given, it is used as the initial size of the output buffer. Raises the `error` exception if any error occurs.

The *wbits* parameter controls the size of the history buffer (or "window size"), and what header and trailer format is expected. It is similar to the parameter for `compressobj()`, but accepts more ranges of values:

- +8 to +15: The base-two logarithm of the window size. The input must include a zlib header and trailer.
- 0: Automatically determine the window size from the zlib header. Only supported since zlib 1.2.3.5.

- −8 to −15: Uses the absolute value of *wbits* as the window size logarithm. The input must be a raw stream with no header or trailer.
- +24 to +31 = 16 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm. The input must include a gzip header and trailer.
- +40 to +47 = 32 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm, and automatically accepts either the zlib or gzip format.

When decompressing a stream, the window size must not be smaller than the size originally used to compress the stream; using a too-small value may result in an `error` exception. The default *wbits* value corresponds to the largest window size and requires a zlib header and trailer to be included.

bufsize is the initial size of the buffer used to hold decompressed data. If more space is required, the buffer size will be increased as needed, so you don't have to get this value exactly right; tuning it will only save a few calls to `malloc()`.

Changed in version 3.6: *wbits* and *bufsize* can be used as keyword arguments.

`zlib.`**`decompressobj`**`(`*wbits=MAX_WBITS*[, *zdict*]`)`

Returns a decompression object, to be used for decompressing data streams that won't fit into memory at once.

The *wbits* parameter controls the size of the history buffer (or the "window size"), and what header and trailer format is expected. It has the same meaning as *described for decompress()*.

The *zdict* parameter specifies a predefined compression dictionary. If provided, this must be the same dictionary as was used by the compressor that produced the data that is to be decompressed.

> **Note**
>
> If *zdict* is a mutable object (such as a `bytearray`), you must not modify its contents between the call to `decompressobj()` and the first call to the decompressor's `decompress()` method.

Changed in version 3.3: Added the *zdict* parameter.

Compression objects support the following methods:

`Compress.`**`compress`**`(`*data*`)`

Compress *data*, returning a bytes object containing compressed data for at least part of the data in *data*. This data should be concatenated to the output produced by any preceding calls to the `compress()` method. Some input may be kept in internal buffers for later processing.

`Compress.`**`flush`**`(`[*mode*]`)`

All pending input is processed, and a bytes object containing the remaining compressed output is returned. *mode* can be selected from the constants `Z_NO_FLUSH`, `Z_PARTIAL_FLUSH`, `Z_SYNC_FLUSH`, `Z_FULL_FLUSH`, `Z_BLOCK` (zlib 1.2.3.4), or `Z_FINISH`, defaulting to `Z_FINISH`. Except `Z_FINISH`, all constants allow compressing further bytestrings of data, while `Z_FINISH` finishes the compressed stream and prevents compressing any more data. After calling `flush()` with *mode* set to `Z_FINISH`, the `compress()` method cannot be called again; the only realistic action is to delete the object.

`Compress.`**`copy`**`()`

Returns a copy of the compression object. This can be used to efficiently compress a set of data that share a common initial prefix.

Changed in version 3.8: Added `copy.copy()` and `copy.deepcopy()` support to compression objects.

Decompression objects support the following methods and attributes:

Decompress.**unused_data**

A bytes object which contains any bytes past the end of the compressed data. That is, this remains `b""` until the last byte that contains compression data is available. If the whole bytestring turned out to contain compressed data, this is `b""`, an empty bytes object.

Decompress.**unconsumed_tail**

A bytes object that contains any data that was not consumed by the last *decompress()* call because it exceeded the limit for the uncompressed data buffer. This data has not yet been seen by the zlib machinery, so you must feed it (possibly with further data concatenated to it) back to a subsequent *decompress()* method call in order to get correct output.

Decompress.**eof**

A boolean indicating whether the end of the compressed data stream has been reached.

This makes it possible to distinguish between a properly formed compressed stream, and an incomplete or truncated one.

Added in version 3.3.

Decompress.**decompress**(*data*, *max_length=0*)

Decompress *data*, returning a bytes object containing the uncompressed data corresponding to at least part of the data in *string*. This data should be concatenated to the output produced by any preceding calls to the *decompress()* method. Some of the input data may be preserved in internal buffers for later processing.

If the optional parameter *max_length* is non-zero then the return value will be no longer than *max_length*. This may mean that not all of the compressed input can be processed; and unconsumed data will be stored in the attribute *unconsumed_tail*. This bytestring must be passed to a subsequent call to *decompress()* if decompression is to continue. If *max_length* is zero then the whole input is decompressed, and *unconsumed_tail* is empty.

Changed in version 3.6: *max_length* can be used as a keyword argument.

Decompress.**flush**([*length*])

All pending input is processed, and a bytes object containing the remaining uncompressed output is returned. After calling *flush()*, the *decompress()* method cannot be called again; the only realistic action is to delete the object.

The optional parameter *length* sets the initial size of the output buffer.

Decompress.**copy**()

Returns a copy of the decompression object. This can be used to save the state of the decompressor midway through the data stream in order to speed up random seeks into the stream at a future point.

Changed in version 3.8: Added *copy.copy()* and *copy.deepcopy()* support to decompression objects.

Information about the version of the zlib library in use is available through the following constants:

zlib.**ZLIB_VERSION**

The version string of the zlib library that was used for building the module. This may be different from the zlib library actually used at runtime, which is available as *ZLIB_RUNTIME_VERSION*.

zlib.**ZLIB_RUNTIME_VERSION**

The version string of the zlib library actually loaded by the interpreter.

Added in version 3.3.

See also

Module *gzip*
Reading and writing **gzip**-format files.

http://www.zlib.net
The zlib library home page.

http://www.zlib.net/manual.html
The zlib manual explains the semantics and usage of the library's many functions.

13.2 `gzip` — Support for `gzip` files

Source code: Lib/gzip.py

This module provides a simple interface to compress and decompress files just like the GNU programs **gzip** and **gunzip** would.

The data compression is provided by the *zlib* module.

The *gzip* module provides the *GzipFile* class, as well as the *open()*, *compress()* and *decompress()* convenience functions. The *GzipFile* class reads and writes **gzip**-format files, automatically compressing or decompressing the data so that it looks like an ordinary *file object*.

Note that additional file formats which can be decompressed by the **gzip** and **gunzip** programs, such as those produced by **compress** and **pack**, are not supported by this module.

The module defines the following items:

`gzip.`**`open`**(*filename*, *mode='rb'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)

Open a gzip-compressed file in binary or text mode, returning a *file object*.

The *filename* argument can be an actual filename (a *str* or *bytes* object), or an existing file object to read from or write to.

The *mode* argument can be any of `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, `'wb'`, `'x'` or `'xb'` for binary mode, or `'rt'`, `'at'`, `'wt'`, or `'xt'` for text mode. The default is `'rb'`.

The *compresslevel* argument is an integer from 0 to 9, as for the *GzipFile* constructor.

For binary mode, this function is equivalent to the *GzipFile* constructor: `GzipFile(filename, mode, compresslevel)`. In this case, the *encoding*, *errors* and *newline* arguments must not be provided.

For text mode, a *GzipFile* object is created, and wrapped in an *io.TextIOWrapper* instance with the specified encoding, error handling behavior, and line ending(s).

Changed in version 3.3: Added support for *filename* being a file object, support for text mode, and the *encoding*, *errors* and *newline* arguments.

Changed in version 3.4: Added support for the `'x'`, `'xb'` and `'xt'` modes.

Changed in version 3.6: Accepts a *path-like object*.

exception `gzip.`**`BadGzipFile`**

An exception raised for invalid gzip files. It inherits from *OSError*. *EOFError* and *zlib.error* can also be raised for invalid gzip files.

Added in version 3.8.

class gzip.**GzipFile**(*filename=None*, *mode=None*, *compresslevel=9*, *fileobj=None*, *mtime=None*)

Constructor for the *GzipFile* class, which simulates most of the methods of a *file object*, with the exception of the *truncate()* method. At least one of *fileobj* and *filename* must be given a non-trivial value.

The new class instance is based on *fileobj*, which can be a regular file, an *io.BytesIO* object, or any other object which simulates a file. It defaults to None, in which case *filename* is opened to provide a file object.

When *fileobj* is not None, the *filename* argument is only used to be included in the **gzip** file header, which may include the original filename of the uncompressed file. It defaults to the filename of *fileobj*, if discernible; otherwise, it defaults to the empty string, and in this case the original filename is not included in the header.

The *mode* argument can be any of 'r', 'rb', 'a', 'ab', 'w', 'wb', 'x', or 'xb', depending on whether the file will be read or written. The default is the mode of *fileobj* if discernible; otherwise, the default is 'rb'. In future Python releases the mode of *fileobj* will not be used. It is better to always specify *mode* for writing.

Note that the file is always opened in binary mode. To open a compressed file in text mode, use *open()* (or wrap your *GzipFile* with an *io.TextIOWrapper*).

The *compresslevel* argument is an integer from 0 to 9 controlling the level of compression; 1 is fastest and produces the least compression, and 9 is slowest and produces the most compression. 0 is no compression. The default is 9.

The optional *mtime* argument is the timestamp requested by gzip. The time is in Unix format, i.e., seconds since 00:00:00 UTC, January 1, 1970. If *mtime* is omitted or None, the current time is used. Use *mtime* = 0 to generate a compressed stream that does not depend on creation time.

See below for the *mtime* attribute that is set when decompressing.

Calling a *GzipFile* object's close() method does not close *fileobj*, since you might wish to append more material after the compressed data. This also allows you to pass an *io.BytesIO* object opened for writing as *fileobj*, and retrieve the resulting memory buffer using the *io.BytesIO* object's *getvalue()* method.

GzipFile supports the *io.BufferedIOBase* interface, including iteration and the with statement. Only the *truncate()* method isn't implemented.

GzipFile also provides the following method and attribute:

peek(*n*)

Read *n* uncompressed bytes without advancing the file position. At most one single read on the compressed stream is done to satisfy the call. The number of bytes returned may be more or less than requested.

> **Note**
>
> While calling *peek()* does not change the file position of the *GzipFile*, it may change the position of the underlying file object (e.g. if the *GzipFile* was constructed with the *fileobj* parameter).

Added in version 3.2.

mode

'rb' for reading and 'wb' for writing.

Changed in version 3.13: In previous versions it was an integer 1 or 2.

mtime

When decompressing, this attribute is set to the last timestamp in the most recently read header. It is an integer, holding the number of seconds since the Unix epoch (00:00:00 UTC, January 1, 1970). The initial value before reading any headers is None.

name

The path to the gzip file on disk, as a *str* or *bytes*. Equivalent to the output of *os.fspath()* on the original input path, with no other normalization, resolution or expansion.

Changed in version 3.1: Support for the `with` statement was added, along with the *mtime* constructor argument and *mtime* attribute.

Changed in version 3.2: Support for zero-padded and unseekable files was added.

Changed in version 3.3: The *io.BufferedIOBase.read1()* method is now implemented.

Changed in version 3.4: Added support for the `'x'` and `'xb'` modes.

Changed in version 3.5: Added support for writing arbitrary *bytes-like objects*. The *read()* method now accepts an argument of `None`.

Changed in version 3.6: Accepts a *path-like object*.

Deprecated since version 3.9: Opening *GzipFile* for writing without specifying the *mode* argument is deprecated.

Changed in version 3.12: Remove the `filename` attribute, use the *name* attribute instead.

`gzip.`**`compress`**(*data*, *compresslevel=9*, ***, *mtime=None*)

Compress the *data*, returning a *bytes* object containing the compressed data. *compresslevel* and *mtime* have the same meaning as in the *GzipFile* constructor above.

Added in version 3.2.

Changed in version 3.8: Added the *mtime* parameter for reproducible output.

Changed in version 3.11: Speed is improved by compressing all data at once instead of in a streamed fashion. Calls with *mtime* set to `0` are delegated to *zlib.compress()* for better speed. In this situation the output may contain a gzip header "OS" byte value other than 255 "unknown" as supplied by the underlying zlib implementation.

Changed in version 3.13: The gzip header OS byte is guaranteed to be set to 255 when this function is used as was the case in 3.10 and earlier.

`gzip.`**`decompress`**(*data*)

Decompress the *data*, returning a *bytes* object containing the uncompressed data. This function is capable of decompressing multi-member gzip data (multiple gzip blocks concatenated together). When the data is certain to contain only one member the *zlib.decompress()* function with *wbits* set to 31 is faster.

Added in version 3.2.

Changed in version 3.11: Speed is improved by decompressing members at once in memory instead of in a streamed fashion.

13.2.1 Examples of usage

Example of how to read a compressed file:

```
import gzip
with gzip.open('/home/joe/file.txt.gz', 'rb') as f:
    file_content = f.read()
```

Example of how to create a compressed GZIP file:

```
import gzip
content = b"Lots of content here"
with gzip.open('/home/joe/file.txt.gz', 'wb') as f:
    f.write(content)
```

Example of how to GZIP compress an existing file:

```
import gzip
import shutil
with open('/home/joe/file.txt', 'rb') as f_in:
    with gzip.open('/home/joe/file.txt.gz', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)
```

Example of how to GZIP compress a binary string:

```
import gzip
s_in = b"Lots of content here"
s_out = gzip.compress(s_in)
```

See also

Module *zlib*
The basic data compression module needed to support the **gzip** file format.

13.2.2 Command Line Interface

The *gzip* module provides a simple command line interface to compress or decompress files.

Once executed the *gzip* module keeps the input file(s).

Changed in version 3.8: Add a new command line interface with a usage. By default, when you will execute the CLI, the default compression level is 6.

Command line options

file
If *file* is not specified, read from *sys.stdin*.

--fast
Indicates the fastest compression method (less compression).

--best
Indicates the slowest compression method (best compression).

-d, --decompress
Decompress the given file.

-h, --help
Show the help message.

13.3 bz2 — Support for bzip2 compression

Source code: Lib/bz2.py

This module provides a comprehensive interface for compressing and decompressing data using the bzip2 compression algorithm.

The *bz2* module contains:

- The *open()* function and *BZ2File* class for reading and writing compressed files.
- The *BZ2Compressor* and *BZ2Decompressor* classes for incremental (de)compression.
- The *compress()* and *decompress()* functions for one-shot (de)compression.

13.3.1 (De)compression of files

bz2.**open**(*filename*, *mode='rb'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)

Open a bzip2-compressed file in binary or text mode, returning a *file object*.

As with the constructor for *BZ2File*, the *filename* argument can be an actual filename (a *str* or *bytes* object), or an existing file object to read from or write to.

The *mode* argument can be any of `'r'`, `'rb'`, `'w'`, `'wb'`, `'x'`, `'xb'`, `'a'` or `'ab'` for binary mode, or `'rt'`, `'wt'`, `'xt'`, or `'at'` for text mode. The default is `'rb'`.

The *compresslevel* argument is an integer from 1 to 9, as for the *BZ2File* constructor.

For binary mode, this function is equivalent to the *BZ2File* constructor: `BZ2File(filename, mode, compresslevel=compresslevel)`. In this case, the *encoding*, *errors* and *newline* arguments must not be provided.

For text mode, a *BZ2File* object is created, and wrapped in an *io.TextIOWrapper* instance with the specified encoding, error handling behavior, and line ending(s).

Added in version 3.3.

Changed in version 3.4: The `'x'` (exclusive creation) mode was added.

Changed in version 3.6: Accepts a *path-like object*.

class bz2.**BZ2File**(*filename*, *mode='r'*, ***, *compresslevel=9*)

Open a bzip2-compressed file in binary mode.

If *filename* is a *str* or *bytes* object, open the named file directly. Otherwise, *filename* should be a *file object*, which will be used to read or write the compressed data.

The *mode* argument can be either `'r'` for reading (default), `'w'` for overwriting, `'x'` for exclusive creation, or `'a'` for appending. These can equivalently be given as `'rb'`, `'wb'`, `'xb'` and `'ab'` respectively.

If *filename* is a file object (rather than an actual file name), a mode of `'w'` does not truncate the file, and is instead equivalent to `'a'`.

If *mode* is `'w'` or `'a'`, *compresslevel* can be an integer between `1` and `9` specifying the level of compression: `1` produces the least compression, and `9` (default) produces the most compression.

If *mode* is `'r'`, the input file may be the concatenation of multiple compressed streams.

BZ2File provides all of the members specified by the *io.BufferedIOBase*, except for *detach()* and *truncate()*. Iteration and the `with` statement are supported.

BZ2File also provides the following methods and attributes:

peek([*n*])

Return buffered data without advancing the file position. At least one byte of data will be returned (unless at EOF). The exact number of bytes returned is unspecified.

> **Note**
>
> While calling *peek()* does not change the file position of the *BZ2File*, it may change the position of the underlying file object (e.g. if the *BZ2File* was constructed by passing a file object for *filename*).

Added in version 3.3.

fileno()

Return the file descriptor for the underlying file.

Added in version 3.3.

readable()

Return whether the file was opened for reading.

Added in version 3.3.

seekable()

Return whether the file supports seeking.

Added in version 3.3.

writable()

Return whether the file was opened for writing.

Added in version 3.3.

read1(*size=-1*)

Read up to *size* uncompressed bytes, while trying to avoid making multiple reads from the underlying stream. Reads up to a buffer's worth of data if size is negative.

Returns `b''` if the file is at EOF.

Added in version 3.3.

readinto(*b*)

Read bytes into *b*.

Returns the number of bytes read (0 for EOF).

Added in version 3.3.

mode

`'rb'` for reading and `'wb'` for writing.

Added in version 3.13.

name

The bzip2 file name. Equivalent to the *name* attribute of the underlying *file object*.

Added in version 3.13.

Changed in version 3.1: Support for the `with` statement was added.

Changed in version 3.3: Support was added for *filename* being a *file object* instead of an actual filename.

The `'a'` (append) mode was added, along with support for reading multi-stream files.

Changed in version 3.4: The `'x'` (exclusive creation) mode was added.

Changed in version 3.5: The *read()* method now accepts an argument of `None`.

Changed in version 3.6: Accepts a *path-like object*.

Changed in version 3.9: The *buffering* parameter has been removed. It was ignored and deprecated since Python 3.0. Pass an open file object to control how the file is opened.

The *compresslevel* parameter became keyword-only.

Changed in version 3.10: This class is thread unsafe in the face of multiple simultaneous readers or writers, just like its equivalent classes in *gzip* and *lzma* have always been.

13.3.2 Incremental (de)compression

class bz2.**BZ2Compressor**(*compresslevel=9*)

Create a new compressor object. This object may be used to compress data incrementally. For one-shot compression, use the *compress()* function instead.

compresslevel, if given, must be an integer between `1` and `9`. The default is `9`.

compress(*data*)

Provide data to the compressor object. Returns a chunk of compressed data if possible, or an empty byte string otherwise.

When you have finished providing data to the compressor, call the *flush()* method to finish the compression process.

flush()

Finish the compression process. Returns the compressed data left in internal buffers.

The compressor object may not be used after this method has been called.

class bz2.**BZ2Decompressor**

Create a new decompressor object. This object may be used to decompress data incrementally. For one-shot compression, use the *decompress()* function instead.

> **Note**
>
> This class does not transparently handle inputs containing multiple compressed streams, unlike *decompress()* and *BZ2File*. If you need to decompress a multi-stream input with *BZ2Decompressor*, you must use a new decompressor for each stream.

decompress(*data*, *max_length=-1*)

Decompress *data* (a *bytes-like object*), returning uncompressed data as bytes. Some of *data* may be buffered internally, for use in later calls to *decompress()*. The returned data should be concatenated with the output of any previous calls to *decompress()*.

If *max_length* is nonnegative, returns at most *max_length* bytes of decompressed data. If this limit is reached and further output can be produced, the *needs_input* attribute will be set to `False`. In this case, the next call to *decompress()* may provide *data* as `b''` to obtain more of the output.

If all of the input data was decompressed and returned (either because this was less than *max_length* bytes, or because *max_length* was negative), the *needs_input* attribute will be set to `True`.

Attempting to decompress data after the end of stream is reached raises an *EOFError*. Any data found after the end of the stream is ignored and saved in the *unused_data* attribute.

Changed in version 3.5: Added the *max_length* parameter.

eof

`True` if the end-of-stream marker has been reached.

Added in version 3.3.

unused_data

Data found after the end of the compressed stream.

If this attribute is accessed before the end of the stream has been reached, its value will be `b''`.

needs_input

`False` if the *decompress()* method can provide more decompressed data before requiring new uncompressed input.

Added in version 3.5.

13.3.3 One-shot (de)compression

bz2.**compress**(*data*, *compresslevel=9*)

Compress *data*, a *bytes-like object*.

compresslevel, if given, must be an integer between `1` and `9`. The default is `9`.

For incremental compression, use a *BZ2Compressor* instead.

bz2.**decompress**(*data*)

Decompress *data*, a *bytes-like object*.

If *data* is the concatenation of multiple compressed streams, decompress all of the streams.

For incremental decompression, use a *BZ2Decompressor* instead.

Changed in version 3.3: Support for multi-stream inputs was added.

13.3.4 Examples of usage

Below are some examples of typical usage of the *bz2* module.

Using *compress()* and *decompress()* to demonstrate round-trip compression:

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
>>> c = bz2.compress(data)
>>> len(data) / len(c)  # Data compression ratio
1.513595166163142
>>> d = bz2.decompress(c)
```

(continues on next page)

(continued from previous page)

```
>>> data == d  # Check equality to original object after round-trip
True
```

Using *BZ2Compressor* for incremental compression:

```
>>> import bz2
>>> def gen_data(chunks=10, chunksize=1000):
...     """Yield incremental blocks of chunksize bytes."""
...     for _ in range(chunks):
...         yield b"z" * chunksize
...
>>> comp = bz2.BZ2Compressor()
>>> out = b""
>>> for chunk in gen_data():
...     # Provide data to the compressor object
...     out = out + comp.compress(chunk)
...
>>> # Finish the compression process.  Call this once you have
>>> # finished providing data to the compressor.
>>> out = out + comp.flush()
```

The example above uses a very "nonrandom" stream of data (a stream of `b"z"` chunks). Random data tends to compress poorly, while ordered, repetitive data usually yields a high compression ratio.

Writing and reading a bzip2-compressed file in binary mode:

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
>>> with bz2.open("myfile.bz2", "wb") as f:
...     # Write compressed data to file
...     unused = f.write(data)
...
>>> with bz2.open("myfile.bz2", "rb") as f:
...     # Decompress data from file
...     content = f.read()
...
>>> content == data  # Check equality to original object after round-trip
True
```

13.4 lzma — Compression using the LZMA algorithm

Added in version 3.3.

Source code: Lib/lzma.py

This module provides classes and convenience functions for compressing and decompressing data using the LZMA compression algorithm. Also included is a file interface supporting the `.xz` and legacy `.lzma` file formats used by the **xz** utility, as well as raw compressed streams.

The interface provided by this module is very similar to that of the *bz2* module. Note that *LZMAFile* and *bz2.BZ2File* are *not* thread-safe, so if you need to use a single *LZMAFile* instance from multiple threads, it is necessary to protect it with a lock.

exception lzma.**LZMAError**

This exception is raised when an error occurs during compression or decompression, or while initializing the compressor/decompressor state.

13.4.1 Reading and writing compressed files

lzma.**open**(*filename*, *mode='rb'*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*, *encoding=None*, *errors=None*, *newline=None*)

Open an LZMA-compressed file in binary or text mode, returning a *file object*.

The *filename* argument can be either an actual file name (given as a *str*, *bytes* or *path-like* object), in which case the named file is opened, or it can be an existing file object to read from or write to.

The *mode* argument can be any of `"r"`, `"rb"`, `"w"`, `"wb"`, `"x"`, `"xb"`, `"a"` or `"ab"` for binary mode, or `"rt"`, `"wt"`, `"xt"`, or `"at"` for text mode. The default is `"rb"`.

When opening a file for reading, the *format* and *filters* arguments have the same meanings as for *LZMADecompressor*. In this case, the *check* and *preset* arguments should not be used.

When opening a file for writing, the *format*, *check*, *preset* and *filters* arguments have the same meanings as for *LZMACompressor*.

For binary mode, this function is equivalent to the *LZMAFile* constructor: `LZMAFile(filename, mode, ...)`. In this case, the *encoding*, *errors* and *newline* arguments must not be provided.

For text mode, a *LZMAFile* object is created, and wrapped in an *io.TextIOWrapper* instance with the specified encoding, error handling behavior, and line ending(s).

Changed in version 3.4: Added support for the `"x"`, `"xb"` and `"xt"` modes.

Changed in version 3.6: Accepts a *path-like object*.

class lzma.**LZMAFile**(*filename=None*, *mode='r'*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*)

Open an LZMA-compressed file in binary mode.

An *LZMAFile* can wrap an already-open *file object*, or operate directly on a named file. The *filename* argument specifies either the file object to wrap, or the name of the file to open (as a *str*, *bytes* or *path-like* object). When wrapping an existing file object, the wrapped file will not be closed when the *LZMAFile* is closed.

The *mode* argument can be either `"r"` for reading (default), `"w"` for overwriting, `"x"` for exclusive creation, or `"a"` for appending. These can equivalently be given as `"rb"`, `"wb"`, `"xb"` and `"ab"` respectively.

If *filename* is a file object (rather than an actual file name), a mode of `"w"` does not truncate the file, and is instead equivalent to `"a"`.

When opening a file for reading, the input file may be the concatenation of multiple separate compressed streams. These are transparently decoded as a single logical stream.

When opening a file for reading, the *format* and *filters* arguments have the same meanings as for `LZMADecompressor`. In this case, the *check* and *preset* arguments should not be used.

When opening a file for writing, the *format*, *check*, *preset* and *filters* arguments have the same meanings as for `LZMACompressor`.

`LZMAFile` supports all the members specified by `io.BufferedIOBase`, except for `detach()` and `truncate()`. Iteration and the `with` statement are supported.

The following method and attributes are also provided:

peek(*size=-1*)

Return buffered data without advancing the file position. At least one byte of data will be returned, unless EOF has been reached. The exact number of bytes returned is unspecified (the *size* argument is ignored).

> **Note**
>
> While calling `peek()` does not change the file position of the `LZMAFile`, it may change the position of the underlying file object (e.g. if the `LZMAFile` was constructed by passing a file object for *filename*).

mode

`'rb'` for reading and `'wb'` for writing.

Added in version 3.13.

name

The lzma file name. Equivalent to the `name` attribute of the underlying *file object*.

Added in version 3.13.

Changed in version 3.4: Added support for the `"x"` and `"xb"` modes.

Changed in version 3.5: The `read()` method now accepts an argument of `None`.

Changed in version 3.6: Accepts a *path-like object*.

13.4.2 Compressing and decompressing data in memory

class lzma.**LZMACompressor**(*format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)

Create a compressor object, which can be used to compress data incrementally.

For a more convenient way of compressing a single chunk of data, see `compress()`.

The *format* argument specifies what container format should be used. Possible values are:

- **FORMAT_XZ: The `.xz` container format.**
  This is the default format.
- **FORMAT_ALONE: The legacy `.lzma` container format.**
  This format is more limited than `.xz` – it does not support integrity checks or multiple filters.
- **FORMAT_RAW: A raw data stream, not using any container format.**
  This format specifier does not support integrity checks, and requires that you always specify a custom filter chain (for both compression and decompression). Additionally, data compressed in this manner cannot be decompressed using `FORMAT_AUTO` (see `LZMADecompressor`).

The *check* argument specifies the type of integrity check to include in the compressed data. This check is used when decompressing, to ensure that the data has not been corrupted. Possible values are:

- `CHECK_NONE`: No integrity check. This is the default (and the only acceptable value) for `FORMAT_ALONE` and `FORMAT_RAW`.
- `CHECK_CRC32`: 32-bit Cyclic Redundancy Check.
- `CHECK_CRC64`: 64-bit Cyclic Redundancy Check. This is the default for `FORMAT_XZ`.
- `CHECK_SHA256`: 256-bit Secure Hash Algorithm.

If the specified check is not supported, an *LZMAError* is raised.

The compression settings can be specified either as a preset compression level (with the *preset* argument), or in detail as a custom filter chain (with the *filters* argument).

The *preset* argument (if provided) should be an integer between `0` and `9` (inclusive), optionally OR-ed with the constant `PRESET_EXTREME`. If neither *preset* nor *filters* are given, the default behavior is to use `PRESET_DEFAULT` (preset level `6`). Higher presets produce smaller output, but make the compression process slower.

> **Note**
>
> In addition to being more CPU-intensive, compression with higher presets also requires much more memory (and produces output that needs more memory to decompress). With preset `9` for example, the overhead for an *LZMACompressor* object can be as high as 800 MiB. For this reason, it is generally best to stick with the default preset.

The *filters* argument (if provided) should be a filter chain specifier. See *Specifying custom filter chains* for details.

compress(*data*)

Compress *data* (a *bytes* object), returning a *bytes* object containing compressed data for at least part of the input. Some of *data* may be buffered internally, for use in later calls to *compress()* and *flush()*. The returned data should be concatenated with the output of any previous calls to *compress()*.

flush()

Finish the compression process, returning a *bytes* object containing any data stored in the compressor's internal buffers.

The compressor cannot be used after this method has been called.

class lzma.**LZMADecompressor**(*format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)

Create a decompressor object, which can be used to decompress data incrementally.

For a more convenient way of decompressing an entire compressed stream at once, see *decompress()*.

The *format* argument specifies the container format that should be used. The default is `FORMAT_AUTO`, which can decompress both `.xz` and `.lzma` files. Other possible values are `FORMAT_XZ`, `FORMAT_ALONE`, and `FORMAT_RAW`.

The *memlimit* argument specifies a limit (in bytes) on the amount of memory that the decompressor can use. When this argument is used, decompression will fail with an *LZMAError* if it is not possible to decompress the input within the given memory limit.

The *filters* argument specifies the filter chain that was used to create the stream being decompressed. This argument is required if *format* is `FORMAT_RAW`, but should not be used for other formats. See *Specifying custom filter chains* for more information about filter chains.

> **Note**
>
> This class does not transparently handle inputs containing multiple compressed streams, unlike *decompress()* and *LZMAFile*. To decompress a multi-stream input with *LZMADecompressor*, you must create a new decompressor for each stream.

decompress(*data*, *max_length=-1*)

Decompress *data* (a *bytes-like object*), returning uncompressed data as bytes. Some of *data* may be buffered internally, for use in later calls to *decompress()*. The returned data should be concatenated with the output of any previous calls to *decompress()*.

If *max_length* is nonnegative, returns at most *max_length* bytes of decompressed data. If this limit is reached and further output can be produced, the *needs_input* attribute will be set to `False`. In this case, the next call to *decompress()* may provide *data* as `b''` to obtain more of the output.

If all of the input data was decompressed and returned (either because this was less than *max_length* bytes, or because *max_length* was negative), the *needs_input* attribute will be set to `True`.

Attempting to decompress data after the end of stream is reached raises an *EOFError*. Any data found after the end of the stream is ignored and saved in the *unused_data* attribute.

Changed in version 3.5: Added the *max_length* parameter.

check

The ID of the integrity check used by the input stream. This may be `CHECK_UNKNOWN` until enough of the input has been decoded to determine what integrity check it uses.

eof

`True` if the end-of-stream marker has been reached.

unused_data

Data found after the end of the compressed stream.

Before the end of the stream is reached, this will be `b""`.

needs_input

`False` if the *decompress()* method can provide more decompressed data before requiring new uncompressed input.

Added in version 3.5.

`lzma.`**compress**(*data*, *format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)

Compress *data* (a *bytes* object), returning the compressed data as a *bytes* object.

See *LZMACompressor* above for a description of the *format*, *check*, *preset* and *filters* arguments.

`lzma.`**decompress**(*data*, *format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)

Decompress *data* (a *bytes* object), returning the uncompressed data as a *bytes* object.

If *data* is the concatenation of multiple distinct compressed streams, decompress all of these streams, and return the concatenation of the results.

See *LZMADecompressor* above for a description of the *format*, *memlimit* and *filters* arguments.

13.4.3 Miscellaneous

lzma.**is_check_supported**(*check*)

Return `True` if the given integrity check is supported on this system.

`CHECK_NONE` and `CHECK_CRC32` are always supported. `CHECK_CRC64` and `CHECK_SHA256` may be unavailable if you are using a version of **liblzma** that was compiled with a limited feature set.

13.4.4 Specifying custom filter chains

A filter chain specifier is a sequence of dictionaries, where each dictionary contains the ID and options for a single filter. Each dictionary must contain the key `"id"`, and may contain additional keys to specify filter-dependent options. Valid filter IDs are as follows:

- Compression filters:
  - `FILTER_LZMA1` (for use with `FORMAT_ALONE`)
  - `FILTER_LZMA2` (for use with `FORMAT_XZ` and `FORMAT_RAW`)
- Delta filter:
  - `FILTER_DELTA`
- Branch-Call-Jump (BCJ) filters:
  - `FILTER_X86`
  - `FILTER_IA64`
  - `FILTER_ARM`
  - `FILTER_ARMTHUMB`
  - `FILTER_POWERPC`
  - `FILTER_SPARC`

A filter chain can consist of up to 4 filters, and cannot be empty. The last filter in the chain must be a compression filter, and any other filters must be delta or BCJ filters.

Compression filters support the following options (specified as additional entries in the dictionary representing the filter):

- `preset`: A compression preset to use as a source of default values for options that are not specified explicitly.
- `dict_size`: Dictionary size in bytes. This should be between 4 KiB and 1.5 GiB (inclusive).
- `lc`: Number of literal context bits.
- `lp`: Number of literal position bits. The sum `lc + lp` must be at most 4.
- `pb`: Number of position bits; must be at most 4.
- `mode`: `MODE_FAST` or `MODE_NORMAL`.
- `nice_len`: What should be considered a "nice length" for a match. This should be 273 or less.
- `mf`: What match finder to use – `MF_HC3`, `MF_HC4`, `MF_BT2`, `MF_BT3`, or `MF_BT4`.
- `depth`: Maximum search depth used by match finder. 0 (default) means to select automatically based on other filter options.

The delta filter stores the differences between bytes, producing more repetitive input for the compressor in certain circumstances. It supports one option, `dist`. This indicates the distance between bytes to be subtracted. The default is 1, i.e. take the differences between adjacent bytes.

The BCJ filters are intended to be applied to machine code. They convert relative branches, calls and jumps in the code to use absolute addressing, with the aim of increasing the redundancy that can be exploited by the compressor. These filters support one option, `start_offset`. This specifies the address that should be mapped to the beginning of the input data. The default is 0.

13.4.5 Examples

Reading in a compressed file:

```
import lzma
with lzma.open("file.xz") as f:
    file_content = f.read()
```

Creating a compressed file:

```
import lzma
data = b"Insert Data Here"
with lzma.open("file.xz", "w") as f:
    f.write(data)
```

Compressing data in memory:

```
import lzma
data_in = b"Insert Data Here"
data_out = lzma.compress(data_in)
```

Incremental compression:

```
import lzma
lzc = lzma.LZMACompressor()
out1 = lzc.compress(b"Some data\n")
out2 = lzc.compress(b"Another piece of data\n")
out3 = lzc.compress(b"Even more data\n")
out4 = lzc.flush()
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])
```

Writing compressed data to an already-open file:

```
import lzma
with open("file.xz", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with lzma.open(f, "w") as lzf:
        lzf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")
```

Creating a compressed file using a custom filter chain:

```
import lzma
my_filters = [
    {"id": lzma.FILTER_DELTA, "dist": 5},
    {"id": lzma.FILTER_LZMA2, "preset": 7 | lzma.PRESET_EXTREME},
```

(continues on next page)

(continued from previous page)

```
]
with lzma.open("file.xz", "w", filters=my_filters) as f:
    f.write(b"blah blah blah")
```

13.5 zipfile — Work with ZIP archives

Source code: Lib/zipfile/

The ZIP file format is a common archive and compression standard. This module provides tools to create, read, write, append, and list a ZIP file. Any advanced use of this module will require an understanding of the format, as defined in PKZIP Application Note.

This module does not currently handle multi-disk ZIP files. It can handle ZIP files that use the ZIP64 extensions (that is ZIP files that are more than 4 GiB in size). It supports decryption of encrypted files in ZIP archives, but it currently cannot create an encrypted file. Decryption is extremely slow as it is implemented in native Python rather than C.

The module defines the following items:

exception zipfile.**BadZipFile**

The error raised for bad ZIP files.

Added in version 3.2.

exception zipfile.**BadZipfile**

Alias of *BadZipFile*, for compatibility with older Python versions.

Deprecated since version 3.2.

exception zipfile.**LargeZipFile**

The error raised when a ZIP file would require ZIP64 functionality but that has not been enabled.

class zipfile.**ZipFile**

The class for reading and writing ZIP files. See section *ZipFile Objects* for constructor details.

class zipfile.**Path**

Class that implements a subset of the interface provided by *pathlib.Path*, including the full *importlib.resources.abc.Traversable* interface.

Added in version 3.8.

class zipfile.**PyZipFile**

Class for creating ZIP archives containing Python libraries.

class zipfile.**ZipInfo**(*filename='NoName'*, *date_time=(1980, 1, 1, 0, 0, 0)*)

Class used to represent information about a member of an archive. Instances of this class are returned by the *getinfo()* and *infolist()* methods of *ZipFile* objects. Most users of the *zipfile* module will not need to create these, but only use those created by this module. *filename* should be the full name of the archive member, and *date_time* should be a tuple containing six fields which describe the time of the last modification to the file; the fields are described in section *ZipInfo Objects*.

Changed in version 3.13: A public `compress_level` attribute has been added to expose the formerly protected `_compresslevel`. The older protected name continues to work as a property for backwards compatibility.

zipfile.**is_zipfile**(*filename*)

Returns `True` if *filename* is a valid ZIP file based on its magic number, otherwise returns `False`. *filename* may be a file or file-like object too.

Changed in version 3.1: Support for file and file-like objects.

zipfile.**ZIP_STORED**

The numeric constant for an uncompressed archive member.

zipfile.**ZIP_DEFLATED**

The numeric constant for the usual ZIP compression method. This requires the *zlib* module.

zipfile.**ZIP_BZIP2**

The numeric constant for the BZIP2 compression method. This requires the *bz2* module.

Added in version 3.3.

zipfile.**ZIP_LZMA**

The numeric constant for the LZMA compression method. This requires the *lzma* module.

Added in version 3.3.

Note

The ZIP file format specification has included support for bzip2 compression since 2001, and for LZMA compression since 2006. However, some tools (including older Python releases) do not support these compression methods, and may either refuse to process the ZIP file altogether, or fail to extract individual files.

See also

PKZIP Application Note
Documentation on the ZIP file format by Phil Katz, the creator of the format and algorithms used.

Info-ZIP Home Page
Information about the Info-ZIP project's ZIP archive programs and development libraries.

13.5.1 ZipFile Objects

class zipfile.**ZipFile**(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *compresslevel=None*, ***, *strict_timestamps=True*, *metadata_encoding=None*)

Open a ZIP file, where *file* can be a path to a file (a string), a file-like object or a *path-like object*.

The *mode* parameter should be `'r'` to read an existing file, `'w'` to truncate and write a new file, `'a'` to append to an existing file, or `'x'` to exclusively create and write a new file. If *mode* is `'x'` and *file* refers to an existing file, a *FileExistsError* will be raised. If *mode* is `'a'` and *file* refers to an existing ZIP file, then additional files are added to it. If *file* does not refer to a ZIP file, then a new ZIP archive is appended to the file. This is meant for adding a ZIP archive to another file (such as `python.exe`). If *mode* is `'a'` and the file does not exist at all, it is created. If *mode* is `'r'` or `'a'`, the file should be seekable.

compression is the ZIP compression method to use when writing the archive, and should be *ZIP_STORED*, *ZIP_DEFLATED*, *ZIP_BZIP2* or *ZIP_LZMA*; unrecognized values will cause *NotImplementedError* to be raised. If *ZIP_DEFLATED*, *ZIP_BZIP2* or *ZIP_LZMA* is specified but the corresponding module (*zlib*, *bz2* or *lzma*) is not available, *RuntimeError* is raised. The default is *ZIP_STORED*.

If *allowZip64* is `True` (the default) zipfile will create ZIP files that use the ZIP64 extensions when the zipfile is larger than 4 GiB. If it is `false` *zipfile* will raise an exception when the ZIP file would require ZIP64 extensions.

The *compresslevel* parameter controls the compression level to use when writing files to the archive. When using *ZIP_STORED* or *ZIP_LZMA* it has no effect. When using *ZIP_DEFLATED* integers `0` through `9` are accepted (see *zlib* for more information). When using *ZIP_BZIP2* integers `1` through `9` are accepted (see *bz2* for more information).

The *strict_timestamps* argument, when set to `False`, allows to zip files older than 1980-01-01 at the cost of setting the timestamp to 1980-01-01. Similar behavior occurs with files newer than 2107-12-31, the timestamp is also set to the limit.

When mode is `'r'`, *metadata_encoding* may be set to the name of a codec, which will be used to decode metadata such as the names of members and ZIP comments.

If the file is created with mode `'w'`, `'x'` or `'a'` and then *closed* without adding any files to the archive, the appropriate ZIP structures for an empty archive will be written to the file.

ZipFile is also a context manager and therefore supports the `with` statement. In the example, *myzip* is closed after the `with` statement's suite is finished—even if an exception occurs:

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

Note

metadata_encoding is an instance-wide setting for the ZipFile. It is not currently possible to set this on a per-member basis.

This attribute is a workaround for legacy implementations which produce archives with names in the current locale encoding or code page (mostly on Windows). According to the .ZIP standard, the encoding of metadata may be specified to be either IBM code page (default) or UTF-8 by a flag in the archive header. That flag takes precedence over *metadata_encoding*, which is a Python-specific extension.

Changed in version 3.2: Added the ability to use *ZipFile* as a context manager.

Changed in version 3.3: Added support for *bzip2* and *lzma* compression.

Changed in version 3.4: ZIP64 extensions are enabled by default.

Changed in version 3.5: Added support for writing to unseekable streams. Added support for the `'x'` mode.

Changed in version 3.6: Previously, a plain *RuntimeError* was raised for unrecognized compression values.

Changed in version 3.6.2: The *file* parameter accepts a *path-like object*.

Changed in version 3.7: Add the *compresslevel* parameter.

Changed in version 3.8: The *strict_timestamps* keyword-only parameter.

Changed in version 3.11: Added support for specifying member name encoding for reading metadata in the zipfile's directory and file headers.

`ZipFile.`**`close`**`()`

Close the archive file. You must call *close()* before exiting your program or essential records will not be written.

`ZipFile.`**`getinfo`**`(`*name*`)`

Return a *ZipInfo* object with information about the archive member *name*. Calling *getinfo()* for a name not currently contained in the archive will raise a *KeyError*.

ZipFile.**infolist**()

Return a list containing a *ZipInfo* object for each member of the archive. The objects are in the same order as their entries in the actual ZIP file on disk if an existing archive was opened.

ZipFile.**namelist**()

Return a list of archive members by name.

ZipFile.**open**(*name*, *mode='r'*, *pwd=None*, ***, *force_zip64=False*)

Access a member of the archive as a binary file-like object. *name* can be either the name of a file within the archive or a *ZipInfo* object. The *mode* parameter, if included, must be `'r'` (the default) or `'w'`. *pwd* is the password used to decrypt encrypted ZIP files as a *bytes* object.

open() is also a context manager and therefore supports the `with` statement:

```
with ZipFile('spam.zip') as myzip:
    with myzip.open('eggs.txt') as myfile:
        print(myfile.read())
```

With *mode* `'r'` the file-like object (`ZipExtFile`) is read-only and provides the following methods: *read()*, *readline()*, *readlines()*, *seek()*, *tell()*, *__iter__()*, *__next__()*. These objects can operate independently of the ZipFile.

With `mode='w'`, a writable file handle is returned, which supports the *write()* method. While a writable file handle is open, attempting to read or write other files in the ZIP file will raise a *ValueError*.

In both cases the file-like object has also attributes `name`, which is equivalent to the name of a file within the archive, and `mode`, which is `'rb'` or `'wb'` depending on the input mode.

When writing a file, if the file size is not known in advance but may exceed 2 GiB, pass `force_zip64=True` to ensure that the header format is capable of supporting large files. If the file size is known in advance, construct a *ZipInfo* object with *file_size* set, and use that as the *name* parameter.

> **Note**
>
> The *open()*, *read()* and *extract()* methods can take a filename or a *ZipInfo* object. You will appreciate this when trying to read a ZIP file that contains members with duplicate names.

Changed in version 3.6: Removed support of `mode='U'`. Use *io.TextIOWrapper* for reading compressed text files in *universal newlines* mode.

Changed in version 3.6: *ZipFile.open()* can now be used to write files into the archive with the `mode='w'` option.

Changed in version 3.6: Calling *open()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

Changed in version 3.13: Added attributes `name` and `mode` for the writeable file-like object. The value of the `mode` attribute for the readable file-like object was changed from `'r'` to `'rb'`.

ZipFile.**extract**(*member*, *path=None*, *pwd=None*)

Extract a member from the archive to the current working directory; *member* must be its full name or a *ZipInfo* object. Its file information is extracted as accurately as possible. *path* specifies a different directory to extract to. *member* can be a filename or a *ZipInfo* object. *pwd* is the password used for encrypted files as a *bytes* object.

Returns the normalized path created (a directory or new file).

> **Note**
>
> If a member filename is an absolute path, a drive/UNC sharepoint and leading (back)slashes will be stripped, e.g.: `///foo/bar` becomes `foo/bar` on Unix, and `C:\foo\bar` becomes `foo\bar` on Windows. And all `"..."` components in a member filename will be removed, e.g.: `../../foo../../ba..r` becomes `foo../ba..r`. On Windows illegal characters (`:`, `<`, `>`, `|`, `"`, `?`, and `*`) replaced by underscore (`_`).

Changed in version 3.6: Calling *extract()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

Changed in version 3.6.2: The *path* parameter accepts a *path-like object*.

ZipFile.**extractall**(*path=None*, *members=None*, *pwd=None*)

Extract all members from the archive to the current working directory. *path* specifies a different directory to extract to. *members* is optional and must be a subset of the list returned by *namelist()*. *pwd* is the password used for encrypted files as a *bytes* object.

> **Warning**
>
> Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of *path*, e.g. members that have absolute filenames starting with `"/"` or filenames with two dots `".."`. This module attempts to prevent that. See *extract()* note.

Changed in version 3.6: Calling *extractall()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

Changed in version 3.6.2: The *path* parameter accepts a *path-like object*.

ZipFile.**printdir**()

Print a table of contents for the archive to `sys.stdout`.

ZipFile.**setpassword**(*pwd*)

Set *pwd* (a *bytes* object) as default password to extract encrypted files.

ZipFile.**read**(*name*, *pwd=None*)

Return the bytes of the file *name* in the archive. *name* is the name of the file in the archive, or a *ZipInfo* object. The archive must be open for read or append. *pwd* is the password used for encrypted files as a *bytes* object and, if specified, overrides the default password set with *setpassword()*. Calling *read()* on a ZipFile that uses a compression method other than *ZIP_STORED*, *ZIP_DEFLATED*, *ZIP_BZIP2* or *ZIP_LZMA* will raise a *NotImplementedError*. An error will also be raised if the corresponding compression module is not available.

Changed in version 3.6: Calling *read()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

ZipFile.**testzip**()

Read all the files in the archive and check their CRC's and file headers. Return the name of the first bad file, or else return `None`.

Changed in version 3.6: Calling *testzip()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

ZipFile.**write**(*filename*, *arcname=None*, *compress_type=None*, *compresslevel=None*)

Write the file named *filename* to the archive, giving it the archive name *arcname* (by default, this will be the same as *filename*, but without a drive letter and with leading path separators removed). If given, *compress_type* overrides

the value given for the *compression* parameter to the constructor for the new entry. Similarly, *compresslevel* will override the constructor if given. The archive must be open with mode `'w'`, `'x'` or `'a'`.

> **Note**
>
> The ZIP file standard historically did not specify a metadata encoding, but strongly recommended CP437 (the original IBM PC encoding) for interoperability. Recent versions allow use of UTF-8 (only). In this module, UTF-8 will automatically be used to write the member names if they contain any non-ASCII characters. It is not possible to write member names in any encoding other than ASCII or UTF-8.

> **Note**
>
> Archive names should be relative to the archive root, that is, they should not start with a path separator.

> **Note**
>
> If `arcname` (or `filename`, if `arcname` is not given) contains a null byte, the name of the file in the archive will be truncated at the null byte.

> **Note**
>
> A leading slash in the filename may lead to the archive being impossible to open in some zip programs on Windows systems.

Changed in version 3.6: Calling *write()* on a ZipFile created with mode `'r'` or a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

ZipFile.**writestr**(*zinfo_or_arcname*, *data*, *compress_type=None*, *compresslevel=None*)

Write a file into the archive. The contents is *data*, which may be either a *str* or a *bytes* instance; if it is a *str*, it is encoded as UTF-8 first. *zinfo_or_arcname* is either the file name it will be given in the archive, or a *ZipInfo* instance. If it's an instance, at least the filename, date, and time must be given. If it's a name, the date and time is set to the current date and time. The archive must be opened with mode `'w'`, `'x'` or `'a'`.

If given, *compress_type* overrides the value given for the *compression* parameter to the constructor for the new entry, or in the *zinfo_or_arcname* (if that is a *ZipInfo* instance). Similarly, *compresslevel* will override the constructor if given.

> **Note**
>
> When passing a *ZipInfo* instance as the *zinfo_or_arcname* parameter, the compression method used will be that specified in the *compress_type* member of the given *ZipInfo* instance. By default, the *ZipInfo* constructor sets this member to *ZIP_STORED*.

Changed in version 3.2: The *compress_type* argument.

Changed in version 3.6: Calling *writestr()* on a ZipFile created with mode `'r'` or a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

ZipFile.**mkdir**(*zinfo_or_directory*, *mode=511*)

Create a directory inside the archive. If *zinfo_or_directory* is a string, a directory is created inside the archive with the mode that is specified in the *mode* argument. If, however, *zinfo_or_directory* is a *ZipInfo* instance then the *mode* argument is ignored.

The archive must be opened with mode `'w'`, `'x'` or `'a'`.

Added in version 3.11.

The following data attributes are also available:

ZipFile.**filename**

Name of the ZIP file.

ZipFile.**debug**

The level of debug output to use. This may be set from `0` (the default, no output) to `3` (the most output). Debugging information is written to `sys.stdout`.

ZipFile.**comment**

The comment associated with the ZIP file as a *bytes* object. If assigning a comment to a *ZipFile* instance created with mode `'w'`, `'x'` or `'a'`, it should be no longer than 65535 bytes. Comments longer than this will be truncated.

13.5.2 Path Objects

class zipfile.**Path**(*root*, *at=''*)

Construct a Path object from a `root` zipfile (which may be a *ZipFile* instance or `file` suitable for passing to the *ZipFile* constructor).

`at` specifies the location of this Path within the zipfile, e.g. 'dir/file.txt', 'dir/', or ''. Defaults to the empty string, indicating the root.

Path objects expose the following features of *pathlib.Path* objects:

Path objects are traversable using the `/` operator or `joinpath`.

Path.**name**

The final path component.

Path.**open**(*mode='r'*, ***, *pwd*, ****)

Invoke *ZipFile.open()* on the current path. Allows opening for read or write, text or binary through supported modes: 'r', 'w', 'rb', 'wb'. Positional and keyword arguments are passed through to *io.TextIOWrapper* when opened as text and ignored otherwise. `pwd` is the `pwd` parameter to *ZipFile.open()*.

Changed in version 3.9: Added support for text and binary modes for open. Default mode is now text.

Changed in version 3.11.2: The `encoding` parameter can be supplied as a positional argument without causing a *TypeError*. As it could in 3.9. Code needing to be compatible with unpatched 3.10 and 3.11 versions must pass all *io.TextIOWrapper* arguments, `encoding` included, as keywords.

Path.**iterdir**()

Enumerate the children of the current directory.

Path.**is_dir**()

Return `True` if the current context references a directory.

Path.**is_file**()

Return `True` if the current context references a file.

Path.**is_symlink**()

Return `True` if the current context references a symbolic link.

Added in version 3.12.

Changed in version 3.13: Previously, `is_symlink` would unconditionally return `False`.

Path.**exists**()

Return `True` if the current context references a file or directory in the zip file.

Path.**suffix**

The last dot-separated portion of the final component, if any. This is commonly called the file extension.

Added in version 3.11: Added *Path.suffix* property.

Path.**stem**

The final path component, without its suffix.

Added in version 3.11: Added *Path.stem* property.

Path.**suffixes**

A list of the path's suffixes, commonly called file extensions.

Added in version 3.11: Added *Path.suffixes* property.

Path.**read_text**(*, **)

Read the current file as unicode text. Positional and keyword arguments are passed through to *io.TextIOWrapper* (except `buffer`, which is implied by the context).

Changed in version 3.11.2: The `encoding` parameter can be supplied as a positional argument without causing a *TypeError*. As it could in 3.9. Code needing to be compatible with unpatched 3.10 and 3.11 versions must pass all *io.TextIOWrapper* arguments, `encoding` included, as keywords.

Path.**read_bytes**()

Read the current file as bytes.

Path.**joinpath**(**other*)

Return a new Path object with each of the *other* arguments joined. The following are equivalent:

```
>>> Path(...).joinpath('child').joinpath('grandchild')
>>> Path(...).joinpath('child', 'grandchild')
>>> Path(...) / 'child' / 'grandchild'
```

Changed in version 3.10: Prior to 3.10, `joinpath` was undocumented and accepted exactly one parameter.

The zipp project provides backports of the latest path object functionality to older Pythons. Use `zipp.Path` in place of `zipfile.Path` for early access to changes.

13.5.3 PyZipFile Objects

The *PyZipFile* constructor takes the same parameters as the *ZipFile* constructor, and one additional parameter, *optimize*.

class zipfile.**PyZipFile**(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *optimize=-1*)

Changed in version 3.2: Added the *optimize* parameter.

Changed in version 3.4: ZIP64 extensions are enabled by default.

Instances have one method in addition to those of *ZipFile* objects:

writepy(*pathname*, *basename=''*, *filterfunc=None*)

Search for files `*.py` and add the corresponding file to the archive.

If the *optimize* parameter to *PyZipFile* was not given or `-1`, the corresponding file is a `*.pyc` file, compiling if necessary.

If the *optimize* parameter to *PyZipFile* was `0`, `1` or `2`, only files with that optimization level (see *compile()*) are added to the archive, compiling if necessary.

If *pathname* is a file, the filename must end with `.py`, and just the (corresponding `*.pyc`) file is added at the top level (no path information). If *pathname* is a file that does not end with `.py`, a *RuntimeError* will be raised. If it is a directory, and the directory is not a package directory, then all the files `*.pyc` are added at the top level. If the directory is a package directory, then all `*.pyc` are added under the package name as a file path, and if any subdirectories are package directories, all of these are added recursively in sorted order.

basename is intended for internal use only.

filterfunc, if given, must be a function taking a single string argument. It will be passed each path (including each individual full file path) before it is added to the archive. If *filterfunc* returns a false value, the path will not be added, and if it is a directory its contents will be ignored. For example, if our test files are all either in `test` directories or start with the string `test_`, we can use a *filterfunc* to exclude them:

```
>>> zf = PyZipFile('myprog.zip')
>>> def notests(s):
...     fn = os.path.basename(s)
...     return (not (fn == 'test' or fn.startswith('test_')))
...
>>> zf.writepy('myprog', filterfunc=notests)
```

The *writepy()* method makes archives with file names like this:

```
string.pyc                    # Top level name
test/__init__.pyc             # Package directory
test/testall.pyc              # Module test.testall
test/bogus/__init__.pyc       # Subpackage directory
test/bogus/myfile.pyc         # Submodule test.bogus.myfile
```

Changed in version 3.4: Added the *filterfunc* parameter.

Changed in version 3.6.2: The *pathname* parameter accepts a *path-like object*.

Changed in version 3.7: Recursion sorts directory entries.

13.5.4 ZipInfo Objects

Instances of the *ZipInfo* class are returned by the *getinfo()* and *infolist()* methods of *ZipFile* objects. Each object stores information about a single member of the ZIP archive.

There is one classmethod to make a *ZipInfo* instance for a filesystem file:

classmethod ZipInfo.**from_file**(*filename*, *arcname=None*, *, *strict_timestamps=True*)

Construct a *ZipInfo* instance for a file on the filesystem, in preparation for adding it to a zip file.

filename should be the path to a file or directory on the filesystem.

If *arcname* is specified, it is used as the name within the archive. If *arcname* is not specified, the name will be the same as *filename*, but with any drive letter and leading path separators removed.

The *strict_timestamps* argument, when set to `False`, allows to zip files older than 1980-01-01 at the cost of setting the timestamp to 1980-01-01. Similar behavior occurs with files newer than 2107-12-31, the timestamp is also set to the limit.

Added in version 3.6.

Changed in version 3.6.2: The *filename* parameter accepts a *path-like object*.

Changed in version 3.8: Added the *strict_timestamps* keyword-only parameter.

Instances have the following methods and attributes:

`ZipInfo.`**`is_dir`**`()`

Return `True` if this archive member is a directory.

This uses the entry's name: directories should always end with `/`.

Added in version 3.6.

`ZipInfo.`**`filename`**

Name of the file in the archive.

`ZipInfo.`**`date_time`**

The time and date of the last modification to the archive member. This is a tuple of six values:

Index	Value
0	Year (>= 1980)
1	Month (one-based)
2	Day of month (one-based)
3	Hours (zero-based)
4	Minutes (zero-based)
5	Seconds (zero-based)

Note

The ZIP file format does not support timestamps before 1980.

`ZipInfo.`**`compress_type`**

Type of compression for the archive member.

`ZipInfo.`**`comment`**

Comment for the individual archive member as a `bytes` object.

`ZipInfo.`**`extra`**

Expansion field data. The PKZIP Application Note contains some comments on the internal structure of the data contained in this `bytes` object.

`ZipInfo.`**`create_system`**

System which created ZIP archive.

`ZipInfo.`**`create_version`**

PKZIP version which created ZIP archive.

`ZipInfo.`**`extract_version`**

PKZIP version needed to extract archive.

ZipInfo.**reserved**
: Must be zero.

ZipInfo.**flag_bits**
: ZIP flag bits.

ZipInfo.**volume**
: Volume number of file header.

ZipInfo.**internal_attr**
: Internal attributes.

ZipInfo.**external_attr**
: External file attributes.

ZipInfo.**header_offset**
: Byte offset to the file header.

ZipInfo.**CRC**
: CRC-32 of the uncompressed file.

ZipInfo.**compress_size**
: Size of the compressed data.

ZipInfo.**file_size**
: Size of the uncompressed file.

13.5.5 Command-Line Interface

The *zipfile* module provides a simple command-line interface to interact with ZIP archives.

If you want to create a new ZIP archive, specify its name after the *-c* option and then list the filename(s) that should be included:

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

Passing a directory is also acceptable:

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

If you want to extract a ZIP archive into the specified directory, use the *-e* option:

```
$ python -m zipfile -e monty.zip target-dir/
```

For a list of the files in a ZIP archive, use the *-l* option:

```
$ python -m zipfile -l monty.zip
```

Command-line options

-l <zipfile>
--list <zipfile>
List files in a zipfile.

-c <zipfile> <source1> ... <sourceN>
--create <zipfile> <source1> ... <sourceN>
Create zipfile from source files.

-e <zipfile> <output_dir>
--extract <zipfile> <output_dir>
Extract zipfile into target directory.

-t <zipfile>
--test <zipfile>
Test whether the zipfile is valid or not.

--metadata-encoding <encoding>
Specify encoding of member names for *-l*, *-e* and *-t*.

Added in version 3.11.

13.5.6 Decompression pitfalls

The extraction in zipfile module might fail due to some pitfalls listed below.

From file itself

Decompression may fail due to incorrect password / CRC checksum / ZIP format or unsupported compression method / decryption.

File System limitations

Exceeding limitations on different file systems can cause decompression failed. Such as allowable characters in the directory entries, length of the file name, length of the pathname, size of a single file, and number of files, etc.

Resources limitations

The lack of memory or disk volume would lead to decompression failed. For example, decompression bombs (aka ZIP bomb) apply to zipfile library that can cause disk volume exhaustion.

Interruption

Interruption during the decompression, such as pressing control-C or killing the decompression process may result in incomplete decompression of the archive.

Default behaviors of extraction

Not knowing the default extraction behaviors can cause unexpected decompression results. For example, when extracting the same archive twice, it overwrites files without asking.

13.6 `tarfile` — Read and write tar archive files

Source code: Lib/tarfile.py

The `tarfile` module makes it possible to read and write tar archives, including those using gzip, bz2 and lzma compression. Use the `zipfile` module to read or write `.zip` files, or the higher-level functions in *shutil*.

Some facts and figures:

- reads and writes `gzip`, `bz2` and `lzma` compressed archives if the respective modules are available.
- read/write support for the POSIX.1-1988 (ustar) format.
- read/write support for the GNU tar format including *longname* and *longlink* extensions, read-only support for all variants of the *sparse* extension including restoration of sparse files.
- read/write support for the POSIX.1-2001 (pax) format.
- handles directories, regular files, hardlinks, symbolic links, fifos, character devices and block devices and is able to acquire and restore file information like timestamp, access permissions and owner.

Changed in version 3.3: Added support for `lzma` compression.

Changed in version 3.12: Archives are extracted using a *filter*, which makes it possible to either limit surprising/dangerous features, or to acknowledge that they are expected and the archive is fully trusted. By default, archives are fully trusted, but this default is deprecated and slated to change in Python 3.14.

`tarfile.`**`open`**(*name=None*, *mode='r'*, *fileobj=None*, *bufsize=10240*, ***kwargs*)

Return a `TarFile` object for the pathname *name*. For detailed information on `TarFile` objects and the keyword arguments that are allowed, see *TarFile Objects*.

mode has to be a string of the form `'filemode[:compression]'`, it defaults to `'r'`. Here is a full list of mode combinations:

mode	action
`'r'` or `'r:*'`	Open for reading with transparent compression (recommended).
`'r:'`	Open for reading exclusively without compression.
`'r:gz'`	Open for reading with gzip compression.
`'r:bz2'`	Open for reading with bzip2 compression.
`'r:xz'`	Open for reading with lzma compression.
`'x'` or `'x:'`	Create a tarfile exclusively without compression. Raise a *FileExistsError* exception if it already exists.
`'x:gz'`	Create a tarfile with gzip compression. Raise a *FileExistsError* exception if it already exists.
`'x:bz2'`	Create a tarfile with bzip2 compression. Raise a *FileExistsError* exception if it already exists.
`'x:xz'`	Create a tarfile with lzma compression. Raise a *FileExistsError* exception if it already exists.
`'a'` or `'a:'`	Open for appending with no compression. The file is created if it does not exist.
`'w'` or `'w:'`	Open for uncompressed writing.
`'w:gz'`	Open for gzip compressed writing.
`'w:bz2'`	Open for bzip2 compressed writing.
`'w:xz'`	Open for lzma compressed writing.

Note that `'a:gz'`, `'a:bz2'` or `'a:xz'` is not possible. If *mode* is not suitable to open a certain (compressed) file for reading, *ReadError* is raised. Use *mode* `'r'` to avoid this. If a compression method is not supported, *CompressionError* is raised.

If *fileobj* is specified, it is used as an alternative to a *file object* opened in binary mode for *name*. It is supposed to be at position 0.

For modes `'w:gz'`, `'x:gz'`, `'w|gz'`, `'w:bz2'`, `'x:bz2'`, `'w|bz2'`, *tarfile.open()* accepts the keyword argument *compresslevel* (default `9`) to specify the compression level of the file.

For modes `'w:xz'` and `'x:xz'`, *tarfile.open()* accepts the keyword argument *preset* to specify the compression level of the file.

For special purposes, there is a second format for *mode*: `'filemode|[compression]'`. *tarfile.open()* will return a *TarFile* object that processes its data as a stream of blocks. No random seeking will be done on the file. If given, *fileobj* may be any object that has a *read()* or *write()* method (depending on the *mode*) that works with bytes. *bufsize* specifies the blocksize and defaults to `20 * 512` bytes. Use this variant in combination with e.g. `sys.stdin.buffer`, a socket *file object* or a tape device. However, such a *TarFile* object is limited in that it does not allow random access, see *Examples*. The currently possible modes:

Mode	Action
`'r|*'`	Open a *stream* of tar blocks for reading with transparent compression.
`'r|'`	Open a *stream* of uncompressed tar blocks for reading.
`'r|gz'`	Open a gzip compressed *stream* for reading.
`'r|bz2'`	Open a bzip2 compressed *stream* for reading.
`'r|xz'`	Open an lzma compressed *stream* for reading.
`'w|'`	Open an uncompressed *stream* for writing.
`'w|gz'`	Open a gzip compressed *stream* for writing.
`'w|bz2'`	Open a bzip2 compressed *stream* for writing.
`'w|xz'`	Open an lzma compressed *stream* for writing.

Changed in version 3.5: The `'x'` (exclusive creation) mode was added.

Changed in version 3.6: The *name* parameter accepts a *path-like object*.

Changed in version 3.12: The *compresslevel* keyword argument also works for streams.

class tarfile.**TarFile**

Class for reading and writing tar archives. Do not use this class directly: use `tarfile.open()` instead. See *TarFile Objects*.

tarfile.**is_tarfile**(*name*)

Return `True` if *name* is a tar archive file, that the `tarfile` module can read. *name* may be a `str`, file, or file-like object.

Changed in version 3.9: Support for file and file-like objects.

The `tarfile` module defines the following exceptions:

exception tarfile.**TarError**

Base class for all `tarfile` exceptions.

exception tarfile.**ReadError**

Is raised when a tar archive is opened, that either cannot be handled by the `tarfile` module or is somehow invalid.

exception tarfile.**CompressionError**

Is raised when a compression method is not supported or when the data cannot be decoded properly.

exception tarfile.**StreamError**

Is raised for the limitations that are typical for stream-like `TarFile` objects.

exception tarfile.**ExtractError**

Is raised for *non-fatal* errors when using `TarFile.extract()`, but only if `TarFile.errorlevel`== 2.

exception tarfile.**HeaderError**

Is raised by `TarInfo.frombuf()` if the buffer it gets is invalid.

exception tarfile.**FilterError**

Base class for members *refused* by filters.

tarinfo

Information about the member that the filter refused to extract, as *TarInfo*.

exception tarfile.**AbsolutePathError**

Raised to refuse extracting a member with an absolute path.

exception tarfile.**OutsideDestinationError**

Raised to refuse extracting a member outside the destination directory.

exception tarfile.**SpecialFileError**

Raised to refuse extracting a special file (e.g. a device or pipe).

exception tarfile.**AbsoluteLinkError**

Raised to refuse extracting a symbolic link with an absolute path.

exception tarfile.**LinkOutsideDestinationError**

Raised to refuse extracting a symbolic link pointing outside the destination directory.

The following constants are available at the module level:

tarfile.**ENCODING**

The default character encoding: `'utf-8'` on Windows, the value returned by *sys.getfilesystemencoding()* otherwise.

tarfile.**REGTYPE**

tarfile.**AREGTYPE**

A regular file *type*.

tarfile.**LNKTYPE**

A link (inside tarfile) *type*.

tarfile.**SYMTYPE**

A symbolic link *type*.

tarfile.**CHRTYPE**

A character special device *type*.

tarfile.**BLKTYPE**

A block special device *type*.

tarfile.**DIRTYPE**

A directory *type*.

tarfile.**FIFOTYPE**

A FIFO special device *type*.

tarfile.**CONTTYPE**

A contiguous file *type*.

tarfile.**GNUTYPE_LONGNAME**

A GNU tar longname *type*.

tarfile.**GNUTYPE_LONGLINK**

A GNU tar longlink *type*.

tarfile.**GNUTYPE_SPARSE**

A GNU tar sparse file *type*.

Each of the following constants defines a tar archive format that the *tarfile* module is able to create. See section *Supported tar formats* for details.

tarfile.**USTAR_FORMAT**

POSIX.1-1988 (ustar) format.

tarfile.**GNU_FORMAT**

GNU tar format.

tarfile.**PAX_FORMAT**

POSIX.1-2001 (pax) format.

tarfile.**DEFAULT_FORMAT**

The default format for creating archives. This is currently *PAX_FORMAT*.

Changed in version 3.8: The default format for new archives was changed to *PAX_FORMAT* from *GNU_FORMAT*.

See also

Module *zipfile*
Documentation of the *zipfile* standard module.

Archiving operations
Documentation of the higher-level archiving facilities provided by the standard *shutil* module.

GNU tar manual, Basic Tar Format
Documentation for tar archive files, including GNU tar extensions.

13.6.1 TarFile Objects

The *TarFile* object provides an interface to a tar archive. A tar archive is a sequence of blocks. An archive member (a stored file) is made up of a header block followed by data blocks. It is possible to store a file in a tar archive several times. Each archive member is represented by a *TarInfo* object, see *TarInfo Objects* for details.

A *TarFile* object can be used as a context manager in a `with` statement. It will automatically be closed when the block is completed. Please note that in the event of an exception an archive opened for writing will not be finalized; only the internally used file object will be closed. See the *Examples* section for a use case.

Added in version 3.2: Added support for the context management protocol.

class tarfile.**TarFile**(*name=None*, *mode='r'*, *fileobj=None*, *format=DEFAULT_FORMAT*, *tarinfo=TarInfo*, *dereference=False*, *ignore_zeros=False*, *encoding=ENCODING*, *errors='surrogateescape'*, *pax_headers=None*, *debug=0*, *errorlevel=1*, *stream=False*)

All following arguments are optional and can be accessed as instance attributes as well.

name is the pathname of the archive. *name* may be a *path-like object*. It can be omitted if *fileobj* is given. In this case, the file object's `name` attribute is used if it exists.

mode is either `'r'` to read from an existing archive, `'a'` to append data to an existing file, `'w'` to create a new file overwriting an existing one, or `'x'` to create a new file only if it does not already exist.

If *fileobj* is given, it is used for reading or writing data. If it can be determined, *mode* is overridden by *fileobj*'s mode. *fileobj* will be used from position 0.

> **Note**
>
> *fileobj* is not closed, when *TarFile* is closed.

format controls the archive format for writing. It must be one of the constants *USTAR_FORMAT*, *GNU_FORMAT* or *PAX_FORMAT* that are defined at module level. When reading, format will be automatically detected, even if different formats are present in a single archive.

The *tarinfo* argument can be used to replace the default *TarInfo* class with a different one.

If *dereference* is *False*, add symbolic and hard links to the archive. If it is *True*, add the content of the target files to the archive. This has no effect on systems that do not support symbolic links.

If *ignore_zeros* is *False*, treat an empty block as the end of the archive. If it is *True*, skip empty (and invalid) blocks and try to get as many members as possible. This is only useful for reading concatenated or damaged archives.

debug can be set from `0` (no debug messages) up to `3` (all debug messages). The messages are written to `sys.stderr`.

errorlevel controls how extraction errors are handled, see *the corresponding attribute*.

The *encoding* and *errors* arguments define the character encoding to be used for reading or writing the archive and how conversion errors are going to be handled. The default settings will work for most users. See section *Unicode issues* for in-depth information.

The *pax_headers* argument is an optional dictionary of strings which will be added as a pax global header if *format* is `PAX_FORMAT`.

If *stream* is set to `True` then while reading the archive info about files in the archive are not cached, saving memory.

Changed in version 3.2: Use `'surrogateescape'` as the default for the *errors* argument.

Changed in version 3.5: The `'x'` (exclusive creation) mode was added.

Changed in version 3.6: The *name* parameter accepts a *path-like object*.

Changed in version 3.13: Add the *stream* parameter.

classmethod TarFile.**open**(...)

Alternative constructor. The `tarfile.open()` function is actually a shortcut to this classmethod.

TarFile.**getmember**(*name*)

Return a `TarInfo` object for member *name*. If *name* can not be found in the archive, `KeyError` is raised.

> **Note**
>
> If a member occurs more than once in the archive, its last occurrence is assumed to be the most up-to-date version.

TarFile.**getmembers**()

Return the members of the archive as a list of `TarInfo` objects. The list has the same order as the members in the archive.

TarFile.**getnames**()

Return the members as a list of their names. It has the same order as the list returned by `getmembers()`.

TarFile.**list**(*verbose=True*, ***, *members=None*)

Print a table of contents to `sys.stdout`. If *verbose* is `False`, only the names of the members are printed. If it is `True`, output similar to that of **`ls -l`** is produced. If optional *members* is given, it must be a subset of the list returned by `getmembers()`.

Changed in version 3.5: Added the *members* parameter.

TarFile.**next**()

Return the next member of the archive as a `TarInfo` object, when `TarFile` is opened for reading. Return `None` if there is no more available.

TarFile.**extractall**(*path='.'*, *members=None*, ***, *numeric_owner=False*, *filter=None*)

Extract all members from the archive to the current working directory or directory *path*. If optional *members* is given, it must be a subset of the list returned by `getmembers()`. Directory information like owner, modification time and permissions are set after all members have been extracted. This is done to work around two problems: A directory's modification time is reset each time a file is created in it. And, if a directory's permissions do not allow writing, extracting files to it will fail.

If *numeric_owner* is `True`, the uid and gid numbers from the tarfile are used to set the owner/group for the extracted files. Otherwise, the named values from the tarfile are used.

The *filter* argument specifies how `members` are modified or rejected before extraction. See *Extraction filters* for details. It is recommended to set this explicitly depending on which *tar* features you need to support.

> **Warning**
>
> Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of *path*, e.g. members that have absolute filenames starting with `"/"` or filenames with two dots `".."`.
>
> Set `filter='data'` to prevent the most dangerous security issues, and read the *Extraction filters* section for details.

Changed in version 3.5: Added the *numeric_owner* parameter.

Changed in version 3.6: The *path* parameter accepts a *path-like object*.

Changed in version 3.12: Added the *filter* parameter.

`TarFile.`**`extract`**(*member*, *path=''*, *set_attrs=True*, ***, *numeric_owner=False*, *filter=None*)

Extract a member from the archive to the current working directory, using its full name. Its file information is extracted as accurately as possible. *member* may be a filename or a `TarInfo` object. You can specify a different directory using *path*. *path* may be a *path-like object*. File attributes (owner, mtime, mode) are set unless *set_attrs* is false.

The *numeric_owner* and *filter* arguments are the same as for `extractall()`.

> **Note**
>
> The `extract()` method does not take care of several extraction issues. In most cases you should consider using the `extractall()` method.

> **Warning**
>
> See the warning for `extractall()`.
>
> Set `filter='data'` to prevent the most dangerous security issues, and read the *Extraction filters* section for details.

Changed in version 3.2: Added the *set_attrs* parameter.

Changed in version 3.5: Added the *numeric_owner* parameter.

Changed in version 3.6: The *path* parameter accepts a *path-like object*.

Changed in version 3.12: Added the *filter* parameter.

`TarFile.`**`extractfile`**(*member*)

Extract a member from the archive as a file object. *member* may be a filename or a `TarInfo` object. If *member* is a regular file or a link, an `io.BufferedReader` object is returned. For all other existing members, `None` is returned. If *member* does not appear in the archive, `KeyError` is raised.

Changed in version 3.3: Return an `io.BufferedReader` object.

Changed in version 3.13: The returned `io.BufferedReader` object has the `mode` attribute which is always equal to `'rb'`.

`TarFile.`**`errorlevel`**`: int`

If *errorlevel* is `0`, errors are ignored when using `TarFile.extract()` and `TarFile.extractall()`. Nevertheless, they appear as error messages in the debug output when *debug* is greater than 0. If `1` (the default),

all *fatal* errors are raised as `OSError` or `FilterError` exceptions. If `2`, all *non-fatal* errors are raised as `TarError` exceptions as well.

Some exceptions, e.g. ones caused by wrong argument types or data corruption, are always raised.

Custom *extraction filters* should raise `FilterError` for *fatal* errors and `ExtractError` for *non-fatal* ones.

Note that when an exception is raised, the archive may be partially extracted. It is the user's responsibility to clean up.

`TarFile.`**`extraction_filter`**

Added in version 3.12.

The *extraction filter* used as a default for the *filter* argument of `extract()` and `extractall()`.

The attribute may be `None` or a callable. String names are not allowed for this attribute, unlike the *filter* argument to `extract()`.

If `extraction_filter` is `None` (the default), calling an extraction method without a *filter* argument will raise a `DeprecationWarning`, and fall back to the `fully_trusted` filter, whose dangerous behavior matches previous versions of Python.

In Python 3.14+, leaving `extraction_filter=None` will cause extraction methods to use the `data` filter by default.

The attribute may be set on instances or overridden in subclasses. It also is possible to set it on the `TarFile` class itself to set a global default, although, since it affects all uses of *tarfile*, it is best practice to only do so in top-level applications or `site configuration`. To set a global default this way, a filter function needs to be wrapped in `staticmethod()` to prevent injection of a `self` argument.

`TarFile.`**`add`**(*name*, *arcname=None*, *recursive=True*, ***, *filter=None*)

Add the file *name* to the archive. *name* may be any type of file (directory, fifo, symbolic link, etc.). If given, *arcname* specifies an alternative name for the file in the archive. Directories are added recursively by default. This can be avoided by setting *recursive* to `False`. Recursion adds entries in sorted order. If *filter* is given, it should be a function that takes a `TarInfo` object argument and returns the changed `TarInfo` object. If it instead returns `None` the `TarInfo` object will be excluded from the archive. See *Examples* for an example.

Changed in version 3.2: Added the *filter* parameter.

Changed in version 3.7: Recursion adds entries in sorted order.

`TarFile.`**`addfile`**(*tarinfo*, *fileobj=None*)

Add the `TarInfo` object *tarinfo* to the archive. If *tarinfo* represents a non zero-size regular file, the *fileobj* argument should be a *binary file*, and `tarinfo.size` bytes are read from it and added to the archive. You can create `TarInfo` objects directly, or by using `gettarinfo()`.

Changed in version 3.13: *fileobj* must be given for non-zero-sized regular files.

`TarFile.`**`gettarinfo`**(*name=None*, *arcname=None*, *fileobj=None*)

Create a `TarInfo` object from the result of `os.stat()` or equivalent on an existing file. The file is either named by *name*, or specified as a *file object* *fileobj* with a file descriptor. *name* may be a *path-like object*. If given, *arcname* specifies an alternative name for the file in the archive, otherwise, the name is taken from *fileobj*'s `name` attribute, or the *name* argument. The name should be a text string.

You can modify some of the `TarInfo`'s attributes before you add it using `addfile()`. If the file object is not an ordinary file object positioned at the beginning of the file, attributes such as `size` may need modifying. This is the case for objects such as `GzipFile`. The `name` may also be modified, in which case *arcname* could be a dummy string.

Changed in version 3.6: The *name* parameter accepts a *path-like object*.

TarFile.**close**()

Close the *TarFile*. In write mode, two finishing zero blocks are appended to the archive.

TarFile.**pax_headers**: *dict*

A dictionary containing key-value pairs of pax global headers.

13.6.2 TarInfo Objects

A *TarInfo* object represents one member in a *TarFile*. Aside from storing all required attributes of a file (like file type, size, time, permissions, owner etc.), it provides some useful methods to determine its type. It does *not* contain the file's data itself.

TarInfo objects are returned by *TarFile*'s methods *getmember()*, *getmembers()* and *gettarinfo()*.

Modifying the objects returned by *getmember()* or *getmembers()* will affect all subsequent operations on the archive. For cases where this is unwanted, you can use *copy.copy()* or call the *replace()* method to create a modified copy in one step.

Several attributes can be set to None to indicate that a piece of metadata is unused or unknown. Different *TarInfo* methods handle None differently:

- The *extract()* or *extractall()* methods will ignore the corresponding metadata, leaving it set to a default.
- *addfile()* will fail.
- *list()* will print a placeholder string.

class tarfile.**TarInfo**(*name=''*)

Create a *TarInfo* object.

classmethod TarInfo.**frombuf**(*buf*, *encoding*, *errors*)

Create and return a *TarInfo* object from string buffer *buf*.

Raises *HeaderError* if the buffer is invalid.

classmethod TarInfo.**fromtarfile**(*tarfile*)

Read the next member from the *TarFile* object *tarfile* and return it as a *TarInfo* object.

TarInfo.**tobuf**(*format=DEFAULT_FORMAT*, *encoding=ENCODING*, *errors='surrogateescape'*)

Create a string buffer from a *TarInfo* object. For information on the arguments see the constructor of the *TarFile* class.

Changed in version 3.2: Use 'surrogateescape' as the default for the *errors* argument.

A TarInfo object has the following public data attributes:

TarInfo.**name**: *str*

Name of the archive member.

TarInfo.**size**: *int*

Size in bytes.

TarInfo.**mtime**: *int* | *float*

Time of last modification in seconds since the *epoch*, as in *os.stat_result.st_mtime*.

Changed in version 3.12: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**mode**: *int*

Permission bits, as for *os.chmod()*.

Changed in version 3.12: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**type**

File type. *type* is usually one of these constants: *REGTYPE*, *AREGTYPE*, *LNKTYPE*, *SYMTYPE*, *DIRTYPE*, *FIFOTYPE*, *CONTTYPE*, *CHRTYPE*, *BLKTYPE*, *GNUTYPE_SPARSE*. To determine the type of a *TarInfo* object more conveniently, use the is*() methods below.

TarInfo.**linkname**: *str*

Name of the target file name, which is only present in *TarInfo* objects of type *LNKTYPE* and *SYMTYPE*.

For symbolic links (SYMTYPE), the *linkname* is relative to the directory that contains the link. For hard links (LNKTYPE), the *linkname* is relative to the root of the archive.

TarInfo.**uid**: *int*

User ID of the user who originally stored this member.

Changed in version 3.12: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**gid**: *int*

Group ID of the user who originally stored this member.

Changed in version 3.12: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**uname**: *str*

User name.

Changed in version 3.12: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**gname**: *str*

Group name.

Changed in version 3.12: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**chksum**: *int*

Header checksum.

TarInfo.**devmajor**: *int*

Device major number.

TarInfo.**devminor**: *int*

Device minor number.

TarInfo.**offset**: *int*

The tar header starts here.

TarInfo.**offset_data**: *int*

The file's data starts here.

TarInfo.**sparse**

Sparse member information.

TarInfo.**pax_headers**: *dict*

A dictionary containing key-value pairs of an associated pax extended header.

TarInfo.**replace**(*name=...*, *mtime=...*, *mode=...*, *linkname=...*, *uid=...*, *gid=...*, *uname=...*, *gname=...*, *deep=True*)

Added in version 3.12.

Return a *new* copy of the `TarInfo` object with the given attributes changed. For example, to return a `TarInfo` with the group name set to `'staff'`, use:

```
new_tarinfo = old_tarinfo.replace(gname='staff')
```

By default, a deep copy is made. If *deep* is false, the copy is shallow, i.e. `pax_headers` and any custom attributes are shared with the original `TarInfo` object.

A *TarInfo* object also provides some convenient query methods:

TarInfo.**isfile**()

Return *True* if the *TarInfo* object is a regular file.

TarInfo.**isreg**()

Same as *isfile()*.

TarInfo.**isdir**()

Return *True* if it is a directory.

TarInfo.**issym**()

Return *True* if it is a symbolic link.

TarInfo.**islnk**()

Return *True* if it is a hard link.

TarInfo.**ischr**()

Return *True* if it is a character device.

TarInfo.**isblk**()

Return *True* if it is a block device.

TarInfo.**isfifo**()

Return *True* if it is a FIFO.

TarInfo.**isdev**()

Return *True* if it is one of character device, block device or FIFO.

13.6.3 Extraction filters

Added in version 3.12.

The *tar* format is designed to capture all details of a UNIX-like filesystem, which makes it very powerful. Unfortunately, the features make it easy to create tar files that have unintended – and possibly malicious – effects when extracted. For example, extracting a tar file can overwrite arbitrary files in various ways (e.g. by using absolute paths, `..` path components, or symlinks that affect later members).

In most cases, the full functionality is not needed. Therefore, *tarfile* supports extraction filters: a mechanism to limit functionality, and thus mitigate some of the security issues.

See also

PEP 706
Contains further motivation and rationale behind the design.

The *filter* argument to *TarFile.extract()* or *extractall()* can be:

- the string `'fully_trusted'`: Honor all metadata as specified in the archive. Should be used if the user trusts the archive completely, or implements their own complex verification.
- the string `'tar'`: Honor most *tar*-specific features (i.e. features of UNIX-like filesystems), but block features that are very likely to be surprising or malicious. See *tar_filter()* for details.
- the string `'data'`: Ignore or block most features specific to UNIX-like filesystems. Intended for extracting cross-platform data archives. See *data_filter()* for details.
- `None` (default): Use *TarFile.extraction_filter*.

  If that is also `None` (the default), raise a `DeprecationWarning`, and fall back to the `'fully_trusted'` filter, whose dangerous behavior matches previous versions of Python.

  In Python 3.14, the `'data'` filter will become the default instead. It's possible to switch earlier; see *TarFile.extraction_filter*.
- A callable which will be called for each extracted member with a *TarInfo* describing the member and the destination path to where the archive is extracted (i.e. the same path is used for all members):

  ```
  filter(member: TarInfo, path: str, /) -> TarInfo | None
  ```

  The callable is called just before each member is extracted, so it can take the current state of the disk into account. It can:

  – return a *TarInfo* object which will be used instead of the metadata in the archive, or

  – return `None`, in which case the member will be skipped, or

  – raise an exception to abort the operation or skip the member, depending on *errorlevel*. Note that when extraction is aborted, *extractall()* may leave the archive partially extracted. It does not attempt to clean up.

Default named filters

The pre-defined, named filters are available as functions, so they can be reused in custom filters:

`tarfile.`**`fully_trusted_filter`**(*member*, *path*)

Return *member* unchanged.

This implements the `'fully_trusted'` filter.

`tarfile.`**`tar_filter`**(*member*, *path*)

Implements the `'tar'` filter.

- Strip leading slashes (`/` and *os.sep*) from filenames.
- *Refuse* to extract files with absolute paths (in case the name is absolute even after stripping slashes, e.g. `C:/foo` on Windows). This raises *AbsolutePathError*.
- *Refuse* to extract files whose absolute path (after following symlinks) would end up outside the destination. This raises *OutsideDestinationError*.

- Clear high mode bits (setuid, setgid, sticky) and group/other write bits (*S_IWGRP* | *S_IWOTH*).

Return the modified `TarInfo` member.

`tarfile.`**`data_filter`**(*member*, *path*)

Implements the `'data'` filter. In addition to what `tar_filter` does:

- *Refuse* to extract links (hard or soft) that link to absolute paths, or ones that link outside the destination.

  This raises *AbsoluteLinkError* or *LinkOutsideDestinationError*.

  Note that such files are refused even on platforms that do not support symbolic links.

- *Refuse* to extract device files (including pipes). This raises *SpecialFileError*.

- For regular files, including hard links:

  - Set the owner read and write permissions (*S_IRUSR* | *S_IWUSR*).
  - Remove the group & other executable permission (*S_IXGRP* | *S_IXOTH*) if the owner doesn't have it (*S_IXUSR*).

- For other files (directories), set `mode` to `None`, so that extraction methods skip applying permission bits.

- Set user and group info (`uid`, `gid`, `uname`, `gname`) to `None`, so that extraction methods skip setting it.

Return the modified `TarInfo` member.

Filter errors

When a filter refuses to extract a file, it will raise an appropriate exception, a subclass of *FilterError*. This will abort the extraction if *TarFile.errorlevel* is 1 or more. With `errorlevel=0` the error will be logged and the member will be skipped, but extraction will continue.

Hints for further verification

Even with `filter='data'`, *tarfile* is not suited for extracting untrusted files without prior inspection. Among other issues, the pre-defined filters do not prevent denial-of-service attacks. Users should do additional checks.

Here is an incomplete list of things to consider:

- Extract to a *new temporary directory* to prevent e.g. exploiting pre-existing links, and to make it easier to clean up after a failed extraction.
- When working with untrusted data, use external (e.g. OS-level) limits on disk, memory and CPU usage.
- Check filenames against an allow-list of characters (to filter out control characters, confusables, foreign path separators, etc.).
- Check that filenames have expected extensions (discouraging files that execute when you "click on them", or extension-less files like Windows special device names).
- Limit the number of extracted files, total size of extracted data, filename length (including symlink length), and size of individual files.
- Check for files that would be shadowed on case-insensitive filesystems.

Also note that:

- Tar files may contain multiple versions of the same file. Later ones are expected to overwrite any earlier ones. This feature is crucial to allow updating tape archives, but can be abused maliciously.

- *tarfile* does not protect against issues with "live" data, e.g. an attacker tinkering with the destination (or source) directory while extraction (or archiving) is in progress.

Supporting older Python versions

Extraction filters were added to Python 3.12, but may be backported to older versions as security updates. To check whether the feature is available, use e.g. `hasattr(tarfile, 'data_filter')` rather than checking the Python version.

The following examples show how to support Python versions with and without the feature. Note that setting `extraction_filter` will affect any subsequent operations.

- Fully trusted archive:

```
my_tarfile.extraction_filter = (lambda member, path: member)
my_tarfile.extractall()
```

- Use the `'data'` filter if available, but revert to Python 3.11 behavior (`'fully_trusted'`) if this feature is not available:

```
my_tarfile.extraction_filter = getattr(tarfile, 'data_filter',
                                       (lambda member, path: member))
my_tarfile.extractall()
```

- Use the `'data'` filter; *fail* if it is not available:

```
my_tarfile.extractall(filter=tarfile.data_filter)
```

  or:

```
my_tarfile.extraction_filter = tarfile.data_filter
my_tarfile.extractall()
```

- Use the `'data'` filter; *warn* if it is not available:

```
if hasattr(tarfile, 'data_filter'):
    my_tarfile.extractall(filter='data')
else:
    # remove this when no longer needed
    warn_the_user('Extracting may be unsafe; consider updating Python')
    my_tarfile.extractall()
```

Stateful extraction filter example

While *tarfile*'s extraction methods take a simple *filter* callable, custom filters may be more complex objects with an internal state. It may be useful to write these as context managers, to be used like this:

```
with StatefulFilter() as filter_func:
    tar.extractall(path, filter=filter_func)
```

Such a filter can be written as, for example:

```
class StatefulFilter:
    def __init__(self):
        self.file_count = 0
```

(continues on next page)

(continued from previous page)

```
    def __enter__(self):
        return self

    def __call__(self, member, path):
        self.file_count += 1
        return member

    def __exit__(self, *exc_info):
        print(f'{self.file_count} files extracted')
```

13.6.4 Command-Line Interface

Added in version 3.4.

The *tarfile* module provides a simple command-line interface to interact with tar archives.

If you want to create a new tar archive, specify its name after the *-c* option and then list the filename(s) that should be included:

```
$ python -m tarfile -c monty.tar  spam.txt eggs.txt
```

Passing a directory is also acceptable:

```
$ python -m tarfile -c monty.tar life-of-brian_1979/
```

If you want to extract a tar archive into the current directory, use the *-e* option:

```
$ python -m tarfile -e monty.tar
```

You can also extract a tar archive into a different directory by passing the directory's name:

```
$ python -m tarfile -e monty.tar  other-dir/
```

For a list of the files in a tar archive, use the *-l* option:

```
$ python -m tarfile -l monty.tar
```

Command-line options

-l <tarfile>
--list <tarfile>
: List files in a tarfile.

-c <tarfile> <source1> ... <sourceN>
--create <tarfile> <source1> ... <sourceN>
: Create tarfile from source files.

-e <tarfile> [<output_dir>]
--extract <tarfile> [<output_dir>]
: Extract tarfile into the current directory if *output_dir* is not specified.

-t <tarfile>

--test <tarfile>

Test whether the tarfile is valid or not.

-v, --verbose

Verbose output.

--filter <filtername>

Specifies the *filter* for `--extract`. See *Extraction filters* for details. Only string names are accepted (that is, `fully_trusted`, `tar`, and `data`).

13.6.5 Examples

How to extract an entire tar archive to the current working directory:

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall(filter='data')
tar.close()
```

How to extract a subset of a tar archive with *`TarFile.extractall()`* using a generator function instead of a list:

```
import os
import tarfile

def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo

tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

How to create an uncompressed tar archive from a list of filenames:

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

The same example using the `with` statement:

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

How to read a gzip compressed tar archive and display some member information:

```
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print(tarinfo.name, "is", tarinfo.size, "bytes in size and is ", end="")
    if tarinfo.isreg():
        print("a regular file.")
```

(continues on next page)

(continued from previous page)

```
    elif tarinfo.isdir():
        print("a directory.")
    else:
        print("something else.")
tar.close()
```

How to create an archive and reset the user information using the *filter* parameter in `TarFile.add()`:

```
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```

13.6.6 Supported tar formats

There are three tar formats that can be created with the `tarfile` module:

- The POSIX.1-1988 ustar format (`USTAR_FORMAT`). It supports filenames up to a length of at best 256 characters and linknames up to 100 characters. The maximum file size is 8 GiB. This is an old and limited but widely supported format.
- The GNU tar format (`GNU_FORMAT`). It supports long filenames and linknames, files bigger than 8 GiB and sparse files. It is the de facto standard on GNU/Linux systems. `tarfile` fully supports the GNU tar extensions for long names, sparse file support is read-only.
- The POSIX.1-2001 pax format (`PAX_FORMAT`). It is the most flexible format with virtually no limits. It supports long filenames and linknames, large files and stores pathnames in a portable way. Modern tar implementations, including GNU tar, bsdtar/libarchive and star, fully support extended *pax* features; some old or unmaintained libraries may not, but should treat *pax* archives as if they were in the universally supported *ustar* format. It is the current default format for new archives.

  It extends the existing *ustar* format with extra headers for information that cannot be stored otherwise. There are two flavours of pax headers: Extended headers only affect the subsequent file header, global headers are valid for the complete archive and affect all following files. All the data in a pax header is encoded in *UTF-8* for portability reasons.

There are some more variants of the tar format which can be read, but not created:

- The ancient V7 format. This is the first tar format from Unix Seventh Edition, storing only regular files and directories. Names must not be longer than 100 characters, there is no user/group name information. Some archives have miscalculated header checksums in case of fields with non-ASCII characters.
- The SunOS tar extended format. This format is a variant of the POSIX.1-2001 pax format, but is not compatible.

13.6.7 Unicode issues

The tar format was originally conceived to make backups on tape drives with the main focus on preserving file system information. Nowadays tar archives are commonly used for file distribution and exchanging archives over networks. One problem of the original format (which is the basis of all other formats) is that there is no concept of supporting different character encodings. For example, an ordinary tar archive created on a *UTF-8* system cannot be read correctly on a *Latin-1* system if it contains non-*ASCII* characters. Textual metadata (like filenames, linknames, user/group names) will appear damaged. Unfortunately, there is no way to autodetect the encoding of an archive. The pax format was designed to solve this problem. It stores non-ASCII metadata using the universal character encoding *UTF-8*.

The details of character conversion in `tarfile` are controlled by the *encoding* and *errors* keyword arguments of the `TarFile` class.

encoding defines the character encoding to use for the metadata in the archive. The default value is `sys.getfilesystemencoding()` or `'ascii'` as a fallback. Depending on whether the archive is read or written, the metadata must be either decoded or encoded. If *encoding* is not set appropriately, this conversion may fail.

The *errors* argument defines how characters are treated that cannot be converted. Possible values are listed in section *Error Handlers*. The default scheme is `'surrogateescape'` which Python also uses for its file system calls, see *File Names, Command Line Arguments, and Environment Variables*.

For `PAX_FORMAT` archives (the default), *encoding* is generally not needed because all the metadata is stored using *UTF-8*. *encoding* is only used in the rare cases when binary pax headers are decoded or when strings with surrogate characters are stored.

CHAPTER

FOURTEEN

FILE FORMATS

The modules described in this chapter parse various miscellaneous file formats that aren't markup languages and are not related to e-mail.

14.1 csv — CSV File Reading and Writing

Source code: Lib/csv.py

The so-called CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases. CSV format was used for many years prior to attempts to describe the format in a standardized way in **RFC 4180**. The lack of a well-defined standard means that subtle differences often exist in the data produced and consumed by different applications. These differences can make it annoying to process CSV files from multiple sources. Still, while the delimiters and quoting characters vary, the overall format is similar enough that it is possible to write a single module which can efficiently manipulate such data, hiding the details of reading and writing the data from the programmer.

The *csv* module implements classes to read and write tabular data in CSV format. It allows programmers to say, "write this data in the format preferred by Excel," or "read data from this file which was generated by Excel," without knowing the precise details of the CSV format used by Excel. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats.

The *csv* module's *reader* and *writer* objects read and write sequences. Programmers can also read and write data in dictionary form using the *DictReader* and *DictWriter* classes.

> **See also**
>
> **PEP 305 - CSV File API**
> The Python Enhancement Proposal which proposed this addition to Python.

14.1.1 Module Contents

The *csv* module defines the following functions:

csv.**reader**(*csvfile*, *dialect='excel'*, ***fmtparams*)

Return a *reader object* that will process lines from the given *csvfile*. A csvfile must be an iterable of strings, each in the reader's defined csv format. A csvfile is most commonly a file-like object or list. If *csvfile* is a file object, it should be opened with `newline=''`.[1] An optional *dialect* parameter can be given which is used to define a set of

[1] If `newline=''` is not specified, newlines embedded inside quoted fields will not be interpreted correctly, and on platforms that use `\r\n` linendings on write an extra `\r` will be added. It should always be safe to specify `newline=''`, since the csv module does its own (*universal*) newline handling.

parameters specific to a particular CSV dialect. It may be an instance of a subclass of the `Dialect` class or one of the strings returned by the `list_dialects()` function. The other optional *fmtparams* keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about the dialect and formatting parameters, see section *Dialects and Formatting Parameters*.

Each row read from the csv file is returned as a list of strings. No automatic data type conversion is performed unless the `QUOTE_NONNUMERIC` format option is specified (in which case unquoted fields are transformed into floats).

A short usage example:

```
>>> import csv
>>> with open('eggs.csv', newline='') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print(', '.join(row))
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

`csv.`**`writer`**(*csvfile*, *dialect='excel'*, ***fmtparams*)

Return a writer object responsible for converting the user's data into delimited strings on the given file-like object. *csvfile* can be any object with a `write()` method. If *csvfile* is a file object, it should be opened with `newline=''` [Page 637, 1]. An optional *dialect* parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the `Dialect` class or one of the strings returned by the `list_dialects()` function. The other optional *fmtparams* keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about dialects and formatting parameters, see the *Dialects and Formatting Parameters* section. To make it as easy as possible to interface with modules which implement the DB API, the value `None` is written as the empty string. While this isn't a reversible transformation, it makes it easier to dump SQL NULL data values to CSV files without preprocessing the data returned from a `cursor.fetch*` call. All other non-string data are stringified with `str()` before being written.

A short usage example:

```
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                            quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

`csv.`**`register_dialect`**(*name*[, *dialect*[, ***fmtparams*]])

Associate *dialect* with *name*. *name* must be a string. The dialect can be specified either by passing a sub-class of `Dialect`, or by *fmtparams* keyword arguments, or both, with keyword arguments overriding parameters of the dialect. For full details about dialects and formatting parameters, see section *Dialects and Formatting Parameters*.

`csv.`**`unregister_dialect`**(*name*)

Delete the dialect associated with *name* from the dialect registry. An `Error` is raised if *name* is not a registered dialect name.

`csv.`**`get_dialect`**(*name*)

Return the dialect associated with *name*. An `Error` is raised if *name* is not a registered dialect name. This function returns an immutable `Dialect`.

`csv.`**`list_dialects`**()

Return the names of all registered dialects.

`csv.`**`field_size_limit`**([*new_limit*])

Returns the current maximum field size allowed by the parser. If *new_limit* is given, this becomes the new limit.

The *csv* module defines the following classes:

class csv.**DictReader**(*f*, *fieldnames=None*, *restkey=None*, *restval=None*, *dialect='excel'*, **args*, ***kwds*)

Create an object that operates like a regular reader but maps the information in each row to a *dict* whose keys are given by the optional *fieldnames* parameter.

The *fieldnames* parameter is a *sequence*. If *fieldnames* is omitted, the values in the first row of file *f* will be used as the fieldnames and will be omitted from the results. If *fieldnames* is provided, they will be used and the first row will be included in the results. Regardless of how the fieldnames are determined, the dictionary preserves their original ordering.

If a row has more fields than fieldnames, the remaining data is put in a list and stored with the fieldname specified by *restkey* (which defaults to `None`). If a non-blank row has fewer fields than fieldnames, the missing values are filled-in with the value of *restval* (which defaults to `None`).

All other optional or keyword arguments are passed to the underlying *reader* instance.

If the argument passed to *fieldnames* is an iterator, it will be coerced to a *list*.

Changed in version 3.6: Returned rows are now of type `OrderedDict`.

Changed in version 3.8: Returned rows are now of type *dict*.

A short usage example:

```
>>> import csv
>>> with open('names.csv', newline='') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Eric Idle
John Cleese

>>> print(row)
{'first_name': 'John', 'last_name': 'Cleese'}
```

class csv.**DictWriter**(*f*, *fieldnames*, *restval=''*, *extrasaction='raise'*, *dialect='excel'*, **args*, ***kwds*)

Create an object which operates like a regular writer but maps dictionaries onto output rows. The *fieldnames* parameter is a *sequence* of keys that identify the order in which values in the dictionary passed to the *writerow()* method are written to file *f*. The optional *restval* parameter specifies the value to be written if the dictionary is missing a key in *fieldnames*. If the dictionary passed to the *writerow()* method contains a key not found in *fieldnames*, the optional *extrasaction* parameter indicates what action to take. If it is set to `'raise'`, the default value, a *ValueError* is raised. If it is set to `'ignore'`, extra values in the dictionary are ignored. Any other optional or keyword arguments are passed to the underlying *writer* instance.

Note that unlike the *DictReader* class, the *fieldnames* parameter of the *DictWriter* class is not optional.

If the argument passed to *fieldnames* is an iterator, it will be coerced to a *list*.

A short usage example:

```
import csv

with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
```

(continues on next page)

(continued from previous page)

```
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

class csv.**Dialect**

The *Dialect* class is a container class whose attributes contain information for how to handle doublequotes, whitespace, delimiters, etc. Due to the lack of a strict CSV specification, different applications produce subtly different CSV data. *Dialect* instances define how *reader* and *writer* instances behave.

All available *Dialect* names are returned by *list_dialects()*, and they can be registered with specific *reader* and *writer* classes through their initializer (__init__) functions like this:

```
import csv

with open('students.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile, dialect='unix')
```

class csv.**excel**

The *excel* class defines the usual properties of an Excel-generated CSV file. It is registered with the dialect name `'excel'`.

class csv.**excel_tab**

The *excel_tab* class defines the usual properties of an Excel-generated TAB-delimited file. It is registered with the dialect name `'excel-tab'`.

class csv.**unix_dialect**

The *unix_dialect* class defines the usual properties of a CSV file generated on UNIX systems, i.e. using `'\n'` as line terminator and quoting all fields. It is registered with the dialect name `'unix'`.

Added in version 3.2.

class csv.**Sniffer**

The *Sniffer* class is used to deduce the format of a CSV file.

The *Sniffer* class provides two methods:

sniff(*sample*, *delimiters=None*)

Analyze the given *sample* and return a *Dialect* subclass reflecting the parameters found. If the optional *delimiters* parameter is given, it is interpreted as a string containing possible valid delimiter characters.

has_header(*sample*)

Analyze the sample text (presumed to be in CSV format) and return *True* if the first row appears to be a series of column headers. Inspecting each column, one of two key criteria will be considered to estimate if the sample contains a header:

- the second through n-th rows contain numeric values
- the second through n-th rows contain strings where at least one value's length differs from that of the putative header of that column.

Twenty rows after the first row are sampled; if more than half of columns + rows meet the criteria, *True* is returned.

> **Note**
>
> This method is a rough heuristic and may produce both false positives and negatives.

An example for *Sniffer* use:

```
with open('example.csv', newline='') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... process CSV file contents here ...
```

The *csv* module defines the following constants:

csv.**QUOTE_ALL**

Instructs *writer* objects to quote all fields.

csv.**QUOTE_MINIMAL**

Instructs *writer* objects to only quote those fields which contain special characters such as *delimiter*, *quotechar* or any of the characters in *lineterminator*.

csv.**QUOTE_NONNUMERIC**

Instructs *writer* objects to quote all non-numeric fields.

Instructs *reader* objects to convert all non-quoted fields to type *float*.

csv.**QUOTE_NONE**

Instructs *writer* objects to never quote fields. When the current *delimiter* occurs in output data it is preceded by the current *escapechar* character. If *escapechar* is not set, the writer will raise *Error* if any characters that require escaping are encountered.

Instructs *reader* objects to perform no special processing of quote characters.

csv.**QUOTE_NOTNULL**

Instructs *writer* objects to quote all fields which are not `None`. This is similar to *QUOTE_ALL*, except that if a field value is `None` an empty (unquoted) string is written.

Instructs *reader* objects to interpret an empty (unquoted) field as `None` and to otherwise behave as *QUOTE_ALL*.

Added in version 3.12.

csv.**QUOTE_STRINGS**

Instructs *writer* objects to always place quotes around fields which are strings. This is similar to *QUOTE_NONNUMERIC*, except that if a field value is `None` an empty (unquoted) string is written.

Instructs *reader* objects to interpret an empty (unquoted) string as `None` and to otherwise behave as *QUOTE_NONNUMERIC*.

Added in version 3.12.

The *csv* module defines the following exception:

exception csv.**Error**

Raised by any of the functions when an error is detected.

14.1.2 Dialects and Formatting Parameters

To make it easier to specify the format of input and output records, specific formatting parameters are grouped together into dialects. A dialect is a subclass of the *Dialect* class containing various attributes describing the format of the CSV file. When creating *reader* or *writer* objects, the programmer can specify a string or a subclass of the *Dialect* class as the dialect parameter. In addition to, or instead of, the *dialect* parameter, the programmer can also specify individual formatting parameters, which have the same names as the attributes defined below for the *Dialect* class.

Dialects support the following attributes:

`Dialect.`**`delimiter`**

A one-character string used to separate fields. It defaults to `','`.

`Dialect.`**`doublequote`**

Controls how instances of *quotechar* appearing inside a field should themselves be quoted. When *True*, the character is doubled. When *False*, the *escapechar* is used as a prefix to the *quotechar*. It defaults to *True*.

On output, if *doublequote* is *False* and no *escapechar* is set, *Error* is raised if a *quotechar* is found in a field.

`Dialect.`**`escapechar`**

A one-character string used by the writer to escape the *delimiter* if *quoting* is set to *QUOTE_NONE* and the *quotechar* if *doublequote* is *False*. On reading, the *escapechar* removes any special meaning from the following character. It defaults to *None*, which disables escaping.

Changed in version 3.11: An empty *escapechar* is not allowed.

`Dialect.`**`lineterminator`**

The string used to terminate lines produced by the *writer*. It defaults to `'\r\n'`.

> **Note**
>
> The *reader* is hard-coded to recognise either `'\r'` or `'\n'` as end-of-line, and ignores *lineterminator*. This behavior may change in the future.

`Dialect.`**`quotechar`**

A one-character string used to quote fields containing special characters, such as the *delimiter* or *quotechar*, or which contain new-line characters. It defaults to `'"'`.

Changed in version 3.11: An empty *quotechar* is not allowed.

`Dialect.`**`quoting`**

Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the *QUOTE_* constants* and defaults to *QUOTE_MINIMAL*.

`Dialect.`**`skipinitialspace`**

When *True*, spaces immediately following the *delimiter* are ignored. The default is *False*.

`Dialect.`**`strict`**

When `True`, raise exception *Error* on bad CSV input. The default is `False`.

14.1.3 Reader Objects

Reader objects (*DictReader* instances and objects returned by the *reader()* function) have the following public methods:

csvreader.**__next__**()

Return the next row of the reader's iterable object as a list (if the object was returned from *reader()*) or a dict (if it is a *DictReader* instance), parsed according to the current *Dialect*. Usually you should call this as `next(reader)`.

Reader objects have the following public attributes:

csvreader.**dialect**

A read-only description of the dialect in use by the parser.

csvreader.**line_num**

The number of lines read from the source iterator. This is not the same as the number of records returned, as records can span multiple lines.

DictReader objects have the following public attribute:

DictReader.**fieldnames**

If not passed as a parameter when creating the object, this attribute is initialized upon first access or when the first record is read from the file.

14.1.4 Writer Objects

writer objects (*DictWriter* instances and objects returned by the *writer()* function) have the following public methods. A *row* must be an iterable of strings or numbers for *writer* objects and a dictionary mapping fieldnames to strings or numbers (by passing them through *str()* first) for *DictWriter* objects. Note that complex numbers are written out surrounded by parens. This may cause some problems for other programs which read CSV files (assuming they support complex numbers at all).

csvwriter.**writerow**(*row*)

Write the *row* parameter to the writer's file object, formatted according to the current *Dialect*. Return the return value of the call to the *write* method of the underlying file object.

Changed in version 3.5: Added support of arbitrary iterables.

csvwriter.**writerows**(*rows*)

Write all elements in *rows* (an iterable of *row* objects as described above) to the writer's file object, formatted according to the current dialect.

Writer objects have the following public attribute:

csvwriter.**dialect**

A read-only description of the dialect in use by the writer.

DictWriter objects have the following public method:

DictWriter.**writeheader**()

Write a row with the field names (as specified in the constructor) to the writer's file object, formatted according to the current dialect. Return the return value of the *csvwriter.writerow()* call used internally.

Added in version 3.2.

Changed in version 3.8: *writeheader()* now also returns the value returned by the *csvwriter.writerow()* method it uses internally.

14.1.5 Examples

The simplest example of reading a CSV file:

```
import csv
with open('some.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Reading a file with an alternate format:

```
import csv
with open('passwd', newline='') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_NONE)
    for row in reader:
        print(row)
```

The corresponding simplest possible writing example is:

```
import csv
with open('some.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

Since *open()* is used to open a CSV file for reading, the file will by default be decoded into unicode using the system default encoding (see *locale.getencoding()*). To decode a file using a different encoding, use the `encoding` argument of open:

```
import csv
with open('some.csv', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

The same applies to writing in something other than the system default encoding: specify the encoding argument when opening the output file.

Registering a new dialect:

```
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', newline='') as f:
    reader = csv.reader(f, 'unixpwd')
```

A slightly more advanced use of the reader — catching and reporting errors:

```
import csv, sys
filename = 'some.csv'
with open(filename, newline='') as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            print(row)
    except csv.Error as e:
        sys.exit('file {}, line {}: {}'.format(filename, reader.line_num, e))
```

And while the module doesn't directly support parsing strings, it can easily be done:

```
import csv
for row in csv.reader(['one,two,three']):
    print(row)
```

14.2 `configparser` — Configuration file parser

Source code: Lib/configparser.py

This module provides the *ConfigParser* class which implements a basic configuration language which provides a structure similar to what's found in Microsoft Windows INI files. You can use this to write Python programs which can be customized by end users easily.

Note

This library does *not* interpret or write the value-type prefixes used in the Windows Registry extended version of INI syntax.

See also

Module *tomllib*
TOML is a well-specified format for application configuration files. It is specifically designed to be an improved version of INI.

Module *shlex*
Support for creating Unix shell-like mini-languages which can also be used for application configuration files.

Module *json*
The `json` module implements a subset of JavaScript syntax which is sometimes used for configuration, but does not support comments.

14.2.1 Quick Start

Let's take a very basic configuration file that looks like this:

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes

[forge.example]
User = hg

[topsecret.server.example]
Port = 50022
ForwardX11 = no
```

The structure of INI files is described *in the following section*. Essentially, the file consists of sections, each of which contains keys with values. `configparser` classes can read and write such files. Let's start by creating the above configuration file programmatically.

```
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config['DEFAULT'] = {'ServerAliveInterval': '45',
...                      'Compression': 'yes',
...                      'CompressionLevel': '9'}
>>> config['forge.example'] = {}
>>> config['forge.example']['User'] = 'hg'
>>> config['topsecret.server.example'] = {}
>>> topsecret = config['topsecret.server.example']
>>> topsecret['Port'] = '50022'     # mutates the parser
>>> topsecret['ForwardX11'] = 'no'  # same here
>>> config['DEFAULT']['ForwardX11'] = 'yes'
>>> with open('example.ini', 'w') as configfile:
...   config.write(configfile)
...
```

As you can see, we can treat a config parser much like a dictionary. There are differences, *outlined later*, but the behavior is very close to what you would expect from a dictionary.

Now that we have created and saved a configuration file, let's read it back and explore the data it holds.

```
>>> config = configparser.ConfigParser()
>>> config.sections()
[]
>>> config.read('example.ini')
['example.ini']
>>> config.sections()
['forge.example', 'topsecret.server.example']
>>> 'forge.example' in config
True
>>> 'python.org' in config
False
>>> config['forge.example']['User']
'hg'
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.example']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['forge.example']:
...     print(key)
user
compressionlevel
serveraliveinterval
compression
forwardx11
>>> config['forge.example']['ForwardX11']
'yes'
```

As we can see above, the API is pretty straightforward. The only bit of magic involves the `DEFAULT` section which provides default values for all other sections[1]. Note also that keys in sections are case-insensitive and stored in lowercase[1].

[1] Config parsers allow for heavy customization. If you are interested in changing the behaviour outlined by the footnote reference, consult the

It is possible to read several configurations into a single *ConfigParser*, where the most recently added configuration has the highest priority. Any conflicting keys are taken from the more recent configuration while the previously existing keys are retained. The example below reads in an `override.ini` file, which will override any conflicting keys from the `example.ini` file.

```
[DEFAULT]
ServerAliveInterval = -1
```

```
>>> config_override = configparser.ConfigParser()
>>> config_override['DEFAULT'] = {'ServerAliveInterval': '-1'}
>>> with open('override.ini', 'w') as configfile:
...     config_override.write(configfile)
...
>>> config_override = configparser.ConfigParser()
>>> config_override.read(['example.ini', 'override.ini'])
['example.ini', 'override.ini']
>>> print(config_override.get('DEFAULT', 'ServerAliveInterval'))
-1
```

This behaviour is equivalent to a *ConfigParser.read()* call with several files passed to the *filenames* parameter.

14.2.2 Supported Datatypes

Config parsers do not guess datatypes of values in configuration files, always storing them internally as strings. This means that if you need other datatypes, you should convert on your own:

```
>>> int(topsecret['Port'])
50022
>>> float(topsecret['CompressionLevel'])
9.0
```

Since this task is so common, config parsers provide a range of handy getter methods to handle integers, floats and booleans. The last one is the most interesting because simply passing the value to `bool()` would do no good since `bool('False')` is still `True`. This is why config parsers also provide *getboolean()*. This method is case-insensitive and recognizes Boolean values from `'yes'`/`'no'`, `'on'`/`'off'`, `'true'`/`'false'` and `'1'`/`'0'`[Page 646, 1]. For example:

```
>>> topsecret.getboolean('ForwardX11')
False
>>> config['forge.example'].getboolean('ForwardX11')
True
>>> config.getboolean('forge.example', 'Compression')
True
```

Apart from *getboolean()*, config parsers also provide equivalent *getint()* and *getfloat()* methods. You can register your own converters and customize the provided ones.[Page 646, 1]

Customizing Parser Behaviour section.

14.2.3 Fallback Values

As with a dictionary, you can use a section's *get()* method to provide fallback values:

```
>>> topsecret.get('Port')
'50022'
>>> topsecret.get('CompressionLevel')
'9'
>>> topsecret.get('Cipher')
>>> topsecret.get('Cipher', '3des-cbc')
'3des-cbc'
```

Please note that default values have precedence over fallback values. For instance, in our example the `'CompressionLevel'` key was specified only in the `'DEFAULT'` section. If we try to get it from the section `'topsecret.server.example'`, we will always get the default, even if we specify a fallback:

```
>>> topsecret.get('CompressionLevel', '3')
'9'
```

One more thing to be aware of is that the parser-level *get()* method provides a custom, more complex interface, maintained for backwards compatibility. When using this method, a fallback value can be provided via the `fallback` keyword-only argument:

```
>>> config.get('forge.example', 'monster',
...            fallback='No such things as monsters')
'No such things as monsters'
```

The same `fallback` argument can be used with the *getint()*, *getfloat()* and *getboolean()* methods, for example:

```
>>> 'BatchMode' in topsecret
False
>>> topsecret.getboolean('BatchMode', fallback=True)
True
>>> config['DEFAULT']['BatchMode'] = 'no'
>>> topsecret.getboolean('BatchMode', fallback=True)
False
```

14.2.4 Supported INI File Structure

A configuration file consists of sections, each led by a `[section]` header, followed by key/value entries separated by a specific string (= or `:` by default[Page 646, 1]). By default, section names are case sensitive but keys are not[Page 646, 1]. Leading and trailing whitespace is removed from keys and values. Values can be omitted if the parser is configured to allow it[Page 646, 1], in which case the key/value delimiter may also be left out. Values can also span multiple lines, as long as they are indented deeper than the first line of the value. Depending on the parser's mode, blank lines may be treated as parts of multiline values or ignored.

By default, a valid section name can be any string that does not contain '\n'. To change this, see *ConfigParser.SECTCRE*.

The first section name may be omitted if the parser is configured to allow an unnamed top level section with `allow_unnamed_section=True`. In this case, the keys/values may be retrieved by *UNNAMED_SECTION* as in `config[UNNAMED_SECTION]`.

Configuration files may include comments, prefixed by specific characters (`#` and `;` by default[Page 646, 1]). Comments may appear on their own on an otherwise empty line, possibly indented.[Page 646, 1]

For example:

```
[Simple Values]
key=value
spaces in keys=allowed
spaces in values=allowed as well
spaces around the delimiter = obviously
you can also use : to delimit keys from values

[All Values Are Strings]
values like this: 1000000
or this: 3.14159265359
are they treated as numbers? : no
integers, floats and booleans are held as: strings
can use the API to get converted values directly: true

[Multiline Values]
chorus: I'm a lumberjack, and I'm okay
    I sleep all night and I work all day

[No Values]
key_without_value
empty string value here =

[You can use comments]
# like this
; or this

# By default only in an empty line.
# Inline comments can be harmful because they prevent users
# from using the delimiting characters as parts of values.
# That being said, this can be customized.

    [Sections Can Be Indented]
        can_values_be_as_well = True
        does_that_mean_anything_special = False
        purpose = formatting for readability
        multiline_values = are
            handled just fine as
            long as they are indented
            deeper than the first line
            of a value
        # Did I mention we can indent comments, too?
```

14.2.5 Unnamed Sections

The name of the first section (or unique) may be omitted and values retrieved by the *UNNAMED_SECTION* attribute.

```
>>> config = """
... option = value
...
... [  Section 2  ]
... another = val
... """
>>> unnamed = configparser.ConfigParser(allow_unnamed_section=True)
>>> unnamed.read_string(config)
```

(continues on next page)

(continued from previous page)

```
>>> unnamed.get(configparser.UNNAMED_SECTION, 'option')
'value'
```

14.2.6 Interpolation of values

On top of the core functionality, *ConfigParser* supports interpolation. This means values can be preprocessed before returning them from `get()` calls.

class configparser.**BasicInterpolation**

The default implementation used by *ConfigParser*. It enables values to contain format strings which refer to other values in the same section, or values in the special default section[Page 646, 1]. Additional default values can be provided on initialization.

For example:

```
[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures

[Escape]
# use a %% to escape the % sign (% is the only character that needs to be␣
↪escaped):
gain: 80%%
```

In the example above, *ConfigParser* with *interpolation* set to `BasicInterpolation()` would resolve `%(home_dir)s` to the value of `home_dir` (`/Users` in this case). `%(my_dir)s` in effect would resolve to `/Users/lumberjack`. All interpolations are done on demand so keys used in the chain of references do not have to be specified in any specific order in the configuration file.

With `interpolation` set to `None`, the parser would simply return `%(my_dir)s/Pictures` as the value of `my_pictures` and `%(home_dir)s/lumberjack` as the value of `my_dir`.

class configparser.**ExtendedInterpolation**

An alternative handler for interpolation which implements a more advanced syntax, used for instance in `zc.buildout`. Extended interpolation is using `${section:option}` to denote a value from a foreign section. Interpolation can span multiple levels. For convenience, if the `section:` part is omitted, interpolation defaults to the current section (and possibly the default values from the special section).

For example, the configuration specified above with basic interpolation, would look like this with extended interpolation:

```
[Paths]
home_dir: /Users
my_dir: ${home_dir}/lumberjack
my_pictures: ${my_dir}/Pictures

[Escape]
# use a $$ to escape the $ sign ($ is the only character that needs to be␣
↪escaped):
cost: $$80
```

Values from other sections can be fetched as well:

```
[Common]
home_dir: /Users
library_dir: /Library
system_dir: /System
macports_dir: /opt/local

[Frameworks]
Python: 3.2
path: ${Common:system_dir}/Library/Frameworks/

[Arthur]
nickname: Two Sheds
last_name: Jackson
my_dir: ${Common:home_dir}/twosheds
my_pictures: ${my_dir}/Pictures
python_dir: ${Frameworks:path}/Python/Versions/${Frameworks:Python}
```

14.2.7 Mapping Protocol Access

Added in version 3.2.

Mapping protocol access is a generic name for functionality that enables using custom objects as if they were dictionaries. In case of *configparser*, the mapping interface implementation is using the `parser['section']['option']` notation.

`parser['section']` in particular returns a proxy for the section's data in the parser. This means that the values are not copied but they are taken from the original parser on demand. What's even more important is that when values are changed on a section proxy, they are actually mutated in the original parser.

configparser objects behave as close to actual dictionaries as possible. The mapping interface is complete and adheres to the *MutableMapping* ABC. However, there are a few differences that should be taken into account:

- By default, all keys in sections are accessible in a case-insensitive manner[Page 646, 1]. E.g. `for option in parser["section"]` yields only `optionxform`'ed option key names. This means lowercased keys by default. At the same time, for a section that holds the key `'a'`, both expressions return `True`:

```
"a" in parser["section"]
"A" in parser["section"]
```

- All sections include `DEFAULTSECT` values as well which means that `.clear()` on a section may not leave the section visibly empty. This is because default values cannot be deleted from the section (because technically they are not there). If they are overridden in the section, deleting causes the default value to be visible again. Trying to delete a default value causes a *KeyError*.
- `DEFAULTSECT` cannot be removed from the parser:
  - trying to delete it raises *ValueError*,
  - `parser.clear()` leaves it intact,
  - `parser.popitem()` never returns it.
- `parser.get(section, option, **kwargs)` - the second argument is **not** a fallback value. Note however that the section-level `get()` methods are compatible both with the mapping protocol and the classic configparser API.
- `parser.items()` is compatible with the mapping protocol (returns a list of *section_name*, *section_proxy* pairs including the DEFAULTSECT). However, this method can also be invoked with arguments: `parser.`

`items(section, raw, vars)`. The latter call returns a list of *option*, *value* pairs for a specified `section`, with all interpolations expanded (unless `raw=True` is provided).

The mapping protocol is implemented on top of the existing legacy API so that subclasses overriding the original interface still should have mappings working as expected.

14.2.8 Customizing Parser Behaviour

There are nearly as many INI format variants as there are applications using it. *configparser* goes a long way to provide support for the largest sensible set of INI styles available. The default functionality is mainly dictated by historical background and it's very likely that you will want to customize some of the features.

The most common way to change the way a specific config parser works is to use the `__init__()` options:

- *defaults*, default value: `None`

  This option accepts a dictionary of key-value pairs which will be initially put in the `DEFAULT` section. This makes for an elegant way to support concise configuration files that don't specify values which are the same as the documented default.

  Hint: if you want to specify default values for a specific section, use *read_dict()* before you read the actual file.

- *dict_type*, default value: *dict*

  This option has a major impact on how the mapping protocol will behave and how the written configuration files look. With the standard dictionary, every section is stored in the order they were added to the parser. Same goes for options within sections.

  An alternative dictionary type can be used for example to sort sections and options on write-back.

  Please note: there are ways to add a set of key-value pairs in a single operation. When you use a regular dictionary in those operations, the order of the keys will be ordered. For example:

```
>>> parser = configparser.ConfigParser()
>>> parser.read_dict({'section1': {'key1': 'value1',
...                                'key2': 'value2',
...                                'key3': 'value3'},
...                   'section2': {'keyA': 'valueA',
...                                'keyB': 'valueB',
...                                'keyC': 'valueC'},
...                   'section3': {'foo': 'x',
...                                'bar': 'y',
...                                'baz': 'z'}
... })
>>> parser.sections()
['section1', 'section2', 'section3']
>>> [option for option in parser['section3']]
['foo', 'bar', 'baz']
```

- *allow_no_value*, default value: `False`

  Some configuration files are known to include settings without values, but which otherwise conform to the syntax supported by *configparser*. The *allow_no_value* parameter to the constructor can be used to indicate that such values should be accepted:

```
>>> import configparser

>>> sample_config = """
```

(continues on next page)

(continued from previous page)

```
... [mysqld]
...   user = mysql
...   pid-file = /var/run/mysqld/mysqld.pid
...   skip-external-locking
...   old_passwords = 1
...   skip-bdb
...   # we don't need ACID today
...   skip-innodb
... """
>>> config = configparser.ConfigParser(allow_no_value=True)
>>> config.read_string(sample_config)

>>> # Settings with values are treated as before:
>>> config["mysqld"]["user"]
'mysql'

>>> # Settings without values provide None:
>>> config["mysqld"]["skip-bdb"]

>>> # Settings which aren't specified still raise an error:
>>> config["mysqld"]["does-not-exist"]
Traceback (most recent call last):
  ...
KeyError: 'does-not-exist'
```

- *delimiters*, default value: `('=', ':')`

  Delimiters are substrings that delimit keys from values within a section. The first occurrence of a delimiting substring on a line is considered a delimiter. This means values (but not keys) can contain the delimiters.

  See also the *space_around_delimiters* argument to *ConfigParser.write()*.

- *comment_prefixes*, default value: `('#', ';')`

- *inline_comment_prefixes*, default value: `None`

  Comment prefixes are strings that indicate the start of a valid comment within a config file. *comment_prefixes* are used only on otherwise empty lines (optionally indented) whereas *inline_comment_prefixes* can be used after every valid value (e.g. section names, options and empty lines as well). By default inline comments are disabled and `'#'` and `';'` are used as prefixes for whole line comments.

  Changed in version 3.2: In previous versions of *configparser* behaviour matched `comment_prefixes=('#',';')` and `inline_comment_prefixes=(';',)`.

  Please note that config parsers don't support escaping of comment prefixes so using *inline_comment_prefixes* may prevent users from specifying option values with characters used as comment prefixes. When in doubt, avoid setting *inline_comment_prefixes*. In any circumstances, the only way of storing comment prefix characters at the beginning of a line in multiline values is to interpolate the prefix, for example:

```
>>> from configparser import ConfigParser, ExtendedInterpolation
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> # the default BasicInterpolation could be used as well
>>> parser.read_string("""
... [DEFAULT]
... hash = #
...
... [hashes]
... shebang =
```

(continues on next page)

(continued from previous page)

```
...   ${hash}!/usr/bin/env python
...   ${hash} -*- coding: utf-8 -*-
...
... extensions =
...   enabled_extension
...   another_extension
...   #disabled_by_comment
...   yet_another_extension
...
... interpolation not necessary = if # is not at line start
... even in multiline values = line #1
...   line #2
...   line #3
... """)
>>> print(parser['hashes']['shebang'])

#!/usr/bin/env python
# -*- coding: utf-8 -*-
>>> print(parser['hashes']['extensions'])

enabled_extension
another_extension
yet_another_extension
>>> print(parser['hashes']['interpolation not necessary'])
if # is not at line start
>>> print(parser['hashes']['even in multiline values'])
line #1
line #2
line #3
```

- *strict*, default value: `True`

  When set to `True`, the parser will not allow for any section or option duplicates while reading from a single source (using *read_file()*, *read_string()* or *read_dict()*). It is recommended to use strict parsers in new applications.

  Changed in version 3.2: In previous versions of *configparser* behaviour matched `strict=False`.

- *empty_lines_in_values*, default value: `True`

  In config parsers, values can span multiple lines as long as they are indented more than the key that holds them. By default parsers also let empty lines to be parts of values. At the same time, keys can be arbitrarily indented themselves to improve readability. In consequence, when configuration files get big and complex, it is easy for the user to lose track of the file structure. Take for instance:

```
[Section]
key = multiline
  value with a gotcha

 this = is still a part of the multiline value of 'key'
```

  This can be especially problematic for the user to see if she's using a proportional font to edit the file. That is why when your application does not need values with empty lines, you should consider disallowing them. This will make empty lines split keys every time. In the example above, it would produce two keys, `key` and `this`.

- *default_section*, default value: `configparser.DEFAULTSECT` (that is: `"DEFAULT"`)

  The convention of allowing a special section of default values for other sections or interpolation purposes is a powerful concept of this library, letting users create complex declarative configurations. This section is normally

called "DEFAULT" but this can be customized to point to any other valid section name. Some typical values include: "general" or "common". The name provided is used for recognizing default sections when reading from any source and is used when writing configuration back to a file. Its current value can be retrieved using the parser_instance.default_section attribute and may be modified at runtime (i.e. to convert files from one format to another).

- *interpolation*, default value: configparser.BasicInterpolation

  Interpolation behaviour may be customized by providing a custom handler through the *interpolation* argument. None can be used to turn off interpolation completely, ExtendedInterpolation() provides a more advanced variant inspired by zc.buildout. More on the subject in the *dedicated documentation section*. *RawConfigParser* has a default value of None.

- *converters*, default value: not set

  Config parsers provide option value getters that perform type conversion. By default *getint()*, *getfloat()*, and *getboolean()* are implemented. Should other getters be desirable, users may define them in a subclass or pass a dictionary where each key is a name of the converter and each value is a callable implementing said conversion. For instance, passing {'decimal': decimal.Decimal} would add getdecimal() on both the parser object and all section proxies. In other words, it will be possible to write both parser_instance.getdecimal('section', 'key', fallback=0) and parser_instance['section'].getdecimal('key', 0).

  If the converter needs to access the state of the parser, it can be implemented as a method on a config parser subclass. If the name of this method starts with get, it will be available on all section proxies, in the dict-compatible form (see the getdecimal() example above).

More advanced customization may be achieved by overriding default values of these parser attributes. The defaults are defined on the classes, so they may be overridden by subclasses or by attribute assignment.

ConfigParser.**BOOLEAN_STATES**

By default when using *getboolean()*, config parsers consider the following values True: '1', 'yes', 'true', 'on' and the following values False: '0', 'no', 'false', 'off'. You can override this by specifying a custom dictionary of strings and their Boolean outcomes. For example:

```
>>> custom = configparser.ConfigParser()
>>> custom['section1'] = {'funky': 'nope'}
>>> custom['section1'].getboolean('funky')
Traceback (most recent call last):
...
ValueError: Not a boolean: nope
>>> custom.BOOLEAN_STATES = {'sure': True, 'nope': False}
>>> custom['section1'].getboolean('funky')
False
```

Other typical Boolean pairs include accept/reject or enabled/disabled.

ConfigParser.**optionxform**(*option*)

This method transforms option names on every read, get, or set operation. The default converts the name to lowercase. This also means that when a configuration file gets written, all keys will be lowercase. Override this method if that's unsuitable. For example:

```
>>> config = """
... [Section1]
... Key = Value
...
... [Section2]
... AnotherKey = Value
```

(continues on next page)

(continued from previous page)

```
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> list(typical['Section1'].keys())
['key']
>>> list(typical['Section2'].keys())
['anotherkey']
>>> custom = configparser.RawConfigParser()
>>> custom.optionxform = lambda option: option
>>> custom.read_string(config)
>>> list(custom['Section1'].keys())
['Key']
>>> list(custom['Section2'].keys())
['AnotherKey']
```

Note

The optionxform function transforms option names to a canonical form. This should be an idempotent function: if the name is already in canonical form, it should be returned unchanged.

`ConfigParser.SECTCRE`

A compiled regular expression used to parse section headers. The default matches `[section]` to the name `"section"`. Whitespace is considered part of the section name, thus `[ larch ]` will be read as a section of name `" larch "`. Override this attribute if that's unsuitable. For example:

```
>>> import re
>>> config = """
... [Section 1]
... option = value
...
... [  Section 2  ]
... another = val
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> typical.sections()
['Section 1', '  Section 2  ']
>>> custom = configparser.ConfigParser()
>>> custom.SECTCRE = re.compile(r"\[ *(?P<header>[^]]+?) *\]")
>>> custom.read_string(config)
>>> custom.sections()
['Section 1', 'Section 2']
```

Note

While ConfigParser objects also use an `OPTCRE` attribute for recognizing option lines, it's not recommended to override it because that would interfere with constructor options *allow_no_value* and *delimiters*.

14.2.9 Legacy API Examples

Mainly because of backwards compatibility concerns, *configparser* provides also a legacy API with explicit `get`/`set` methods. While there are valid use cases for the methods outlined below, mapping protocol access is preferred for new projects. The legacy API is at times more advanced, low-level and downright counterintuitive.

An example of writing to a configuration file:

```
import configparser

config = configparser.RawConfigParser()

# Please note that using RawConfigParser's set functions, you can assign
# non-string values to keys internally, but will receive an error when
# attempting to write to a file or when you get it in non-raw mode. Setting
# values using the mapping protocol or ConfigParser's set() does not allow
# such assignments to take place.
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'w') as configfile:
    config.write(configfile)
```

An example of reading the configuration file again:

```
import configparser

config = configparser.RawConfigParser()
config.read('example.cfg')

# getfloat() raises an exception if the value is not a float
# getint() and getboolean() also do this for their respective types
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print(a_float + an_int)

# Notice that the next output does not interpolate '%(bar)s' or '%(baz)s'.
# This is because we are using a RawConfigParser().
if config.getboolean('Section1', 'a_bool'):
    print(config.get('Section1', 'foo'))
```

To get interpolation, use *ConfigParser*:

```
import configparser

cfg = configparser.ConfigParser()
cfg.read('example.cfg')

# Set the optional *raw* argument of get() to True if you wish to disable
# interpolation in a single get operation.
print(cfg.get('Section1', 'foo', raw=False))  # -> "Python is fun!"
print(cfg.get('Section1', 'foo', raw=True))   # -> "%(bar)s is %(baz)s!"
```

(continues on next page)

(continued from previous page)

```
# The optional *vars* argument is a dict with members that will take
# precedence in interpolation.
print(cfg.get('Section1', 'foo', vars={'bar': 'Documentation',
                                       'baz': 'evil'}))

# The optional *fallback* argument can be used to provide a fallback value
print(cfg.get('Section1', 'foo'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'foo', fallback='Monty is not.'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'monster', fallback='No such things as monsters.'))
      # -> "No such things as monsters."

# A bare print(cfg.get('Section1', 'monster')) would raise NoOptionError
# but we can also use:

print(cfg.get('Section1', 'monster', fallback=None))
      # -> None
```

Default values are available in both types of ConfigParsers. They are used in interpolation if an option used is not defined elsewhere.

```
import configparser

# New instance with 'bar' and 'baz' defaulting to 'Life' and 'hard' each
config = configparser.ConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

print(config.get('Section1', 'foo'))      # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print(config.get('Section1', 'foo'))      # -> "Life is hard!"
```

14.2.10 ConfigParser Objects

class configparser.**ConfigParser**(*defaults=None*, *dict_type=dict*, *allow_no_value=False*, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*, *interpolation=BasicInterpolation()*, *converters={}*)

The main configuration parser. When *defaults* is given, it is initialized into the dictionary of intrinsic defaults. When *dict_type* is given, it will be used to create the dictionary objects for the list of sections, for the options within a section, and for the default values.

When *delimiters* is given, it is used as the set of substrings that divide keys from values. When *comment_prefixes* is given, it will be used as the set of substrings that prefix comments in otherwise empty lines. Comments can be indented. When *inline_comment_prefixes* is given, it will be used as the set of substrings that prefix comments in non-empty lines.

When *strict* is `True` (the default), the parser won't allow for any section or option duplicates while reading from a single source (file, string or dictionary), raising *DuplicateSectionError* or *DuplicateOptionError*. When *empty_lines_in_values* is `False` (default: `True`), each empty line marks the end of an option. Otherwise,

internal empty lines of a multiline option are kept as part of the value. When *allow_no_value* is `True` (default: `False`), options without values are accepted; the value held for these is `None` and they are serialized without the trailing delimiter.

When *default_section* is given, it specifies the name for the special section holding default values for other sections and interpolation purposes (normally named `"DEFAULT"`). This value can be retrieved and changed at runtime using the `default_section` instance attribute. This won't re-evaluate an already parsed config file, but will be used when writing parsed settings to a new config file.

Interpolation behaviour may be customized by providing a custom handler through the *interpolation* argument. `None` can be used to turn off interpolation completely, `ExtendedInterpolation()` provides a more advanced variant inspired by `zc.buildout`. More on the subject in the *dedicated documentation section*.

All option names used in interpolation will be passed through the *optionxform()* method just like any other option name reference. For example, using the default implementation of *optionxform()* (which converts option names to lower case), the values `foo %(bar)s` and `foo %(BAR)s` are equivalent.

When *converters* is given, it should be a dictionary where each key represents the name of a type converter and each value is a callable implementing the conversion from string to the desired datatype. Every converter gets its own corresponding `get*()` method on the parser object and section proxies.

It is possible to read several configurations into a single *ConfigParser*, where the most recently added configuration has the highest priority. Any conflicting keys are taken from the more recent configuration while the previously existing keys are retained. The example below reads in an `override.ini` file, which will override any conflicting keys from the `example.ini` file.

```
[DEFAULT]
ServerAliveInterval = -1
```

```
>>> config_override = configparser.ConfigParser()
>>> config_override['DEFAULT'] = {'ServerAliveInterval': '-1'}
>>> with open('override.ini', 'w') as configfile:
...     config_override.write(configfile)
...
>>> config_override = configparser.ConfigParser()
>>> config_override.read(['example.ini', 'override.ini'])
['example.ini', 'override.ini']
>>> print(config_override.get('DEFAULT', 'ServerAliveInterval'))
-1
```

Changed in version 3.1: The default *dict_type* is *collections.OrderedDict*.

Changed in version 3.2: *allow_no_value*, *delimiters*, *comment_prefixes*, *strict*, *empty_lines_in_values*, *default_section* and *interpolation* were added.

Changed in version 3.5: The *converters* argument was added.

Changed in version 3.7: The *defaults* argument is read with *read_dict()*, providing consistent behavior across the parser: non-string keys and values are implicitly converted to strings.

Changed in version 3.8: The default *dict_type* is *dict*, since it now preserves insertion order.

Changed in version 3.13: Raise a *MultilineContinuationError* when *allow_no_value* is `True`, and a key without a value is continued with an indented line.

defaults()

Return a dictionary containing the instance-wide defaults.

sections()

Return a list of the sections available; the *default section* is not included in the list.

add_section(*section*)

Add a section named *section* to the instance. If a section by the given name already exists, *DuplicateSectionError* is raised. If the *default section* name is passed, *ValueError* is raised. The name of the section must be a string; if not, *TypeError* is raised.

Changed in version 3.2: Non-string section names raise *TypeError*.

has_section(*section*)

Indicates whether the named *section* is present in the configuration. The *default section* is not acknowledged.

options(*section*)

Return a list of options available in the specified *section*.

has_option(*section*, *option*)

If the given *section* exists, and contains the given *option*, return *True*; otherwise return *False*. If the specified *section* is *None* or an empty string, DEFAULT is assumed.

read(*filenames*, *encoding=None*)

Attempt to read and parse an iterable of filenames, returning a list of filenames which were successfully parsed.

If *filenames* is a string, a *bytes* object or a *path-like object*, it is treated as a single filename. If a file named in *filenames* cannot be opened, that file will be ignored. This is designed so that you can specify an iterable of potential configuration file locations (for example, the current directory, the user's home directory, and some system-wide directory), and all existing configuration files in the iterable will be read.

If none of the named files exist, the *ConfigParser* instance will contain an empty dataset. An application which requires initial values to be loaded from a file should load the required file or files using *read_file()* before calling *read()* for any optional files:

```
import configparser, os

config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')],
            encoding='cp1250')
```

Changed in version 3.2: Added the *encoding* parameter. Previously, all files were read using the default encoding for *open()*.

Changed in version 3.6.1: The *filenames* parameter accepts a *path-like object*.

Changed in version 3.7: The *filenames* parameter accepts a *bytes* object.

read_file(*f*, *source=None*)

Read and parse configuration data from *f* which must be an iterable yielding Unicode strings (for example files opened in text mode).

Optional argument *source* specifies the name of the file being read. If not given and *f* has a `name` attribute, that is used for *source*; the default is `'<???>'`.

Added in version 3.2: Replaces `readfp()`.

read_string(*string*, *source='<string>'*)

Parse configuration data from a string.

Optional argument *source* specifies a context-specific name of the string passed. If not given, `'<string>'` is used. This should commonly be a filesystem path or a URL.

Added in version 3.2.

read_dict(*dictionary*, *source='<dict>'*)

Load configuration from any object that provides a dict-like `items()` method. Keys are section names, values are dictionaries with keys and values that should be present in the section. If the used dictionary type preserves order, sections and their keys will be added in order. Values are automatically converted to strings.

Optional argument *source* specifies a context-specific name of the dictionary passed. If not given, `<dict>` is used.

This method can be used to copy state between parsers.

Added in version 3.2.

get(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

Get an *option* value for the named *section*. If *vars* is provided, it must be a dictionary. The *option* is looked up in *vars* (if provided), *section*, and in *DEFAULTSECT* in that order. If the key is not found and *fallback* is provided, it is used as a fallback value. `None` can be provided as a *fallback* value.

All the `'%'` interpolations are expanded in the return values, unless the *raw* argument is true. Values for interpolation keys are looked up in the same manner as the option.

Changed in version 3.2: Arguments *raw*, *vars* and *fallback* are keyword only to protect users from trying to use the third argument as the *fallback* fallback (especially when using the mapping protocol).

getint(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

A convenience method which coerces the *option* in the specified *section* to an integer. See *get()* for explanation of *raw*, *vars* and *fallback*.

getfloat(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

A convenience method which coerces the *option* in the specified *section* to a floating-point number. See *get()* for explanation of *raw*, *vars* and *fallback*.

getboolean(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

A convenience method which coerces the *option* in the specified *section* to a Boolean value. Note that the accepted values for the option are `'1'`, `'yes'`, `'true'`, and `'on'`, which cause this method to return `True`, and `'0'`, `'no'`, `'false'`, and `'off'`, which cause it to return `False`. These string values are checked in a case-insensitive manner. Any other value will cause it to raise *ValueError*. See *get()* for explanation of *raw*, *vars* and *fallback*.

items(*raw=False*, *vars=None*)

items(*section*, *raw=False*, *vars=None*)

When *section* is not given, return a list of *section_name*, *section_proxy* pairs, including DEFAULTSECT.

Otherwise, return a list of *name*, *value* pairs for the options in the given *section*. Optional arguments have the same meaning as for the *get()* method.

Changed in version 3.8: Items present in *vars* no longer appear in the result. The previous behaviour mixed actual parser options with variables provided for interpolation.

set(*section*, *option*, *value*)

If the given section exists, set the given option to the specified value; otherwise raise *NoSectionError*. *option* and *value* must be strings; if not, *TypeError* is raised.

write(*fileobject*, *space_around_delimiters=True*)

Write a representation of the configuration to the specified *file object*, which must be opened in text mode (accepting strings). This representation can be parsed by a future *read()* call. If *space_around_delimiters* is true, delimiters between keys and values are surrounded by spaces.

> **Note**
>
> Comments in the original configuration file are not preserved when writing the configuration back. What is considered a comment, depends on the given values for *comment_prefix* and *inline_comment_prefix*.

remove_option(*section*, *option*)

Remove the specified *option* from the specified *section*. If the section does not exist, raise *NoSectionError*. If the option existed to be removed, return *True*; otherwise return *False*.

remove_section(*section*)

Remove the specified *section* from the configuration. If the section in fact existed, return `True`. Otherwise return `False`.

optionxform(*option*)

Transforms the option name *option* as found in an input file or as passed in by client code to the form that should be used in the internal structures. The default implementation returns a lower-case version of *option*; subclasses may override this or client code can set an attribute of this name on instances to affect this behavior.

You don't need to subclass the parser to use this method, you can also set it on an instance, to a function that takes a string argument and returns a string. Setting it to `str`, for example, would make option names case sensitive:

```
cfgparser = ConfigParser()
cfgparser.optionxform = str
```

Note that when reading configuration files, whitespace around the option names is stripped before *optionxform()* is called.

configparser.**UNNAMED_SECTION**

A special object representing a section name used to reference the unnamed section (see *Unnamed Sections*).

configparser.**MAX_INTERPOLATION_DEPTH**

The maximum depth for recursive interpolation for *get()* when the *raw* parameter is false. This is relevant only when the default *interpolation* is used.

14.2.11 RawConfigParser Objects

class configparser.**RawConfigParser**(*defaults=None*, *dict_type=dict*, *allow_no_value=False*, ***, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*[, *interpolation*])

Legacy variant of the *ConfigParser*. It has interpolation disabled by default and allows for non-string section names, option names, and values via its unsafe `add_section` and `set` methods, as well as the legacy `defaults=` keyword argument handling.

Changed in version 3.8: The default *dict_type* is *dict*, since it now preserves insertion order.

> **Note**
>
> Consider using *ConfigParser* instead which checks types of the values to be stored internally. If you don't want interpolation, you can use `ConfigParser(interpolation=None)`.

add_section(*section*)

Add a section named *section* to the instance. If a section by the given name already exists, *DuplicateSectionError* is raised. If the *default section* name is passed, *ValueError* is raised.

Type of *section* is not checked which lets users create non-string named sections. This behaviour is unsupported and may cause internal errors.

set(*section*, *option*, *value*)

If the given section exists, set the given option to the specified value; otherwise raise *NoSectionError*. While it is possible to use *RawConfigParser* (or *ConfigParser* with *raw* parameters set to true) for *internal* storage of non-string values, full functionality (including interpolation and output to files) can only be achieved using string values.

This method lets users assign non-string values to keys internally. This behaviour is unsupported and will cause errors when attempting to write to a file or get it in non-raw mode. **Use the mapping protocol API** which does not allow such assignments to take place.

14.2.12 Exceptions

exception configparser.**Error**

Base class for all other *configparser* exceptions.

exception configparser.**NoSectionError**

Exception raised when a specified section is not found.

exception configparser.**DuplicateSectionError**

Exception raised if *add_section()* is called with the name of a section that is already present or in strict parsers when a section if found more than once in a single input file, string or dictionary.

Changed in version 3.2: Added the optional *source* and *lineno* attributes and parameters to `__init__()`.

exception configparser.**DuplicateOptionError**

Exception raised by strict parsers if a single option appears twice during reading from a single file, string or dictionary. This catches misspellings and case sensitivity-related errors, e.g. a dictionary may have two keys representing the same case-insensitive configuration key.

exception configparser.**NoOptionError**

Exception raised when a specified option is not found in the specified section.

exception configparser.**InterpolationError**

Base class for exceptions raised when problems occur performing string interpolation.

exception configparser.**InterpolationDepthError**

Exception raised when string interpolation cannot be completed because the number of iterations exceeds *MAX_INTERPOLATION_DEPTH*. Subclass of *InterpolationError*.

exception configparser.**InterpolationMissingOptionError**

Exception raised when an option referenced from a value does not exist. Subclass of *InterpolationError*.

exception configparser.**InterpolationSyntaxError**

Exception raised when the source text into which substitutions are made does not conform to the required syntax. Subclass of *InterpolationError*.

exception configparser.**MissingSectionHeaderError**

Exception raised when attempting to parse a file which has no section headers.

exception configparser.**ParsingError**

Exception raised when errors occur attempting to parse a file.

Changed in version 3.12: The `filename` attribute and `__init__()` constructor argument were removed. They have been available using the name `source` since 3.2.

exception configparser.**MultilineContinuationError**

Exception raised when a key without a corresponding value is continued with an indented line.

Added in version 3.13.

14.3 tomllib — Parse TOML files

Added in version 3.11.

Source code: Lib/tomllib

This module provides an interface for parsing TOML (Tom's Obvious Minimal Language, https://toml.io). This module does not support writing TOML.

See also

The Tomli-W package is a TOML writer that can be used in conjunction with this module, providing a write API familiar to users of the standard library *marshal* and *pickle* modules.

See also

The TOML Kit package is a style-preserving TOML library with both read and write capability. It is a recommended replacement for this module for editing already existing TOML files.

This module defines the following functions:

tomllib.**load**(*fp*, */*, ***, *parse_float=float*)

Read a TOML file. The first argument should be a readable and binary file object. Return a *dict*. Convert TOML types to Python using this *conversion table*.

parse_float will be called with the string of every TOML float to be decoded. By default, this is equivalent to `float(num_str)`. This can be used to use another datatype or parser for TOML floats (e.g. *decimal.Decimal*). The callable must not return a *dict* or a *list*, else a *ValueError* is raised.

A *TOMLDecodeError* will be raised on an invalid TOML document.

tomllib.**loads**(*s*, */*, ***, *parse_float=float*)

Load TOML from a *str* object. Return a *dict*. Convert TOML types to Python using this *conversion table*. The *parse_float* argument has the same meaning as in *load()*.

A *TOMLDecodeError* will be raised on an invalid TOML document.

The following exceptions are available:

exception tomllib.**TOMLDecodeError**

Subclass of *ValueError*.

14.3.1 Examples

Parsing a TOML file:

```
import tomllib

with open("pyproject.toml", "rb") as f:
    data = tomllib.load(f)
```

Parsing a TOML string:

```
import tomllib

toml_str = """
python-version = "3.11.0"
python-implementation = "CPython"
"""

data = tomllib.loads(toml_str)
```

14.3.2 Conversion Table

TOML	Python
TOML document	dict
string	str
integer	int
float	float (configurable with *parse_float*)
boolean	bool
offset date-time	datetime.datetime (`tzinfo` attribute set to an instance of `datetime.timezone`)
local date-time	datetime.datetime (`tzinfo` attribute set to `None`)
local date	datetime.date
local time	datetime.time
array	list
table	dict
inline table	dict
array of tables	list of dicts

14.4 `netrc` — netrc file processing

Source code: Lib/netrc.py

The *netrc* class parses and encapsulates the netrc file format used by the Unix **ftp** program and other FTP clients.

class netrc.**netrc**([*file*])

A *netrc* instance or subclass instance encapsulates data from a netrc file. The initialization argument, if present, specifies the file to parse. If no argument is given, the file `.netrc` in the user's home directory – as determined by *os.path.expanduser()* – will be read. Otherwise, a *FileNotFoundError* exception will be raised. Parse errors will raise *NetrcParseError* with diagnostic information including the file name, line number, and terminating token. If no argument is specified on a POSIX system, the presence of passwords in the `.netrc` file

will raise a *NetrcParseError* if the file ownership or permissions are insecure (owned by a user other than the user running the process, or accessible for read or write by any other user). This implements security behavior equivalent to that of ftp and other programs that use `.netrc`.

Changed in version 3.4: Added the POSIX permission check.

Changed in version 3.7: *os.path.expanduser()* is used to find the location of the `.netrc` file when *file* is not passed as argument.

Changed in version 3.10: *netrc* try UTF-8 encoding before using locale specific encoding. The entry in the netrc file no longer needs to contain all tokens. The missing tokens' value default to an empty string. All the tokens and their values now can contain arbitrary characters, like whitespace and non-ASCII characters. If the login name is anonymous, it won't trigger the security check.

exception netrc.**NetrcParseError**

Exception raised by the *netrc* class when syntactical errors are encountered in source text. Instances of this exception provide three interesting attributes:

msg

Textual explanation of the error.

filename

The name of the source file.

lineno

The line number on which the error was found.

14.4.1 netrc Objects

A *netrc* instance has the following methods:

netrc.**authenticators**(*host*)

Return a 3-tuple `(login, account, password)` of authenticators for *host*. If the netrc file did not contain an entry for the given host, return the tuple associated with the 'default' entry. If neither matching host nor default entry is available, return `None`.

netrc.**__repr__**()

Dump the class data as a string in the format of a netrc file. (This discards comments and may reorder the entries.)

Instances of *netrc* have public instance variables:

netrc.**hosts**

Dictionary mapping host names to `(login, account, password)` tuples. The 'default' entry, if any, is represented as a pseudo-host by that name.

netrc.**macros**

Dictionary mapping macro names to string lists.

14.5 plistlib — Generate and parse Apple .plist files

Source code: Lib/plistlib.py

This module provides an interface for reading and writing the "property list" files used by Apple, primarily on macOS and iOS. This module supports both binary and XML plist files.

The property list (`.plist`) file format is a simple serialization supporting basic object types, like dictionaries, lists, numbers and strings. Usually the top level object is a dictionary.

To write out and to parse a plist file, use the *dump()* and *load()* functions.

To work with plist data in bytes or string objects, use *dumps()* and *loads()*.

Values can be strings, integers, floats, booleans, tuples, lists, dictionaries (but only with string keys), *bytes*, *bytearray* or *datetime.datetime* objects.

Changed in version 3.4: New API, old API deprecated. Support for binary format plists added.

Changed in version 3.8: Support added for reading and writing *UID* tokens in binary plists as used by NSKeyedArchiver and NSKeyedUnarchiver.

Changed in version 3.9: Old API removed.

See also

PList manual page
Apple's documentation of the file format.

This module defines the following functions:

plistlib.**load**(*fp*, *, *fmt=None*, *dict_type=dict*, *aware_datetime=False*)

Read a plist file. *fp* should be a readable and binary file object. Return the unpacked root object (which usually is a dictionary).

The *fmt* is the format of the file and the following values are valid:

- *None*: Autodetect the file format
- *FMT_XML*: XML file format
- *FMT_BINARY*: Binary plist format

The *dict_type* is the type used for dictionaries that are read from the plist file.

When *aware_datetime* is true, fields with type `datetime.datetime` will be created as *aware object*, with `tzinfo` as *datetime.UTC*.

XML data for the *FMT_XML* format is parsed using the Expat parser from *xml.parsers.expat* – see its documentation for possible exceptions on ill-formed XML. Unknown elements will simply be ignored by the plist parser.

The parser for the binary format raises `InvalidFileException` when the file cannot be parsed.

Added in version 3.4.

Changed in version 3.13: The keyword-only parameter *aware_datetime* has been added.

plistlib.**loads**(*data*, *, *fmt=None*, *dict_type=dict*, *aware_datetime=False*)

Load a plist from a bytes or string object. See *load()* for an explanation of the keyword arguments.

Added in version 3.4.

Changed in version 3.13: *data* can be a string when *fmt* equals *FMT_XML*.

plistlib.**dump**(*value*, *fp*, *, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*, *aware_datetime=False*)

Write *value* to a plist file. *Fp* should be a writable, binary file object.

The *fmt* argument specifies the format of the plist file and can be one of the following values:

- *FMT_XML*: XML formatted plist file
- *FMT_BINARY*: Binary formatted plist file

When *sort_keys* is true (the default) the keys for dictionaries will be written to the plist in sorted order, otherwise they will be written in the iteration order of the dictionary.

When *skipkeys* is false (the default) the function raises *TypeError* when a key of a dictionary is not a string, otherwise such keys are skipped.

When *aware_datetime* is true and any field with type `datetime.datetime` is set as an *aware object*, it will convert to UTC timezone before writing it.

A *TypeError* will be raised if the object is of an unsupported type or a container that contains objects of unsupported types.

An *OverflowError* will be raised for integer values that cannot be represented in (binary) plist files.

Added in version 3.4.

Changed in version 3.13: The keyword-only parameter *aware_datetime* has been added.

plistlib.**dumps**(*value*, *, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*, *aware_datetime=False*)

Return *value* as a plist-formatted bytes object. See the documentation for *dump()* for an explanation of the keyword arguments of this function.

Added in version 3.4.

The following classes are available:

class plistlib.**UID**(*data*)

Wraps an *int*. This is used when reading or writing NSKeyedArchiver encoded data, which contains UID (see PList manual).

It has one attribute, `data`, which can be used to retrieve the int value of the UID. `data` must be in the range `0 <= data < 2**64`.

Added in version 3.8.

The following constants are available:

plistlib.**FMT_XML**

The XML format for plist files.

Added in version 3.4.

plistlib.**FMT_BINARY**

The binary format for plist files

Added in version 3.4.

14.5.1 Examples

Generating a plist:

```
import datetime
import plistlib

pl = dict(
    aString = "Doodah",
    aList = ["A", "B", 12, 32.1, [1, 2, 3]],
    aFloat = 0.1,
    anInt = 728,
    aDict = dict(
        anotherString = "<hello & hi there!>",
        aThirdString = "M\xe4ssig, Ma\xdf",
        aTrueValue = True,
        aFalseValue = False,
    ),
    someData = b"<binary gunk>",
    someMoreData = b"<lots of binary gunk>" * 10,
    aDate = datetime.datetime.now()
)
print(plistlib.dumps(pl).decode())
```

Parsing a plist:

```
import plistlib

plist = b"""<plist version="1.0">
<dict>
    <key>foo</key>
    <string>bar</string>
</dict>
</plist>"""
pl = plistlib.loads(plist)
print(pl["foo"])
```

CHAPTER

FIFTEEN

CRYPTOGRAPHIC SERVICES

The modules described in this chapter implement various algorithms of a cryptographic nature. They are available at the discretion of the installation. Here's an overview:

15.1 `hashlib` — Secure hashes and message digests

Source code: Lib/hashlib.py

This module implements a common interface to many different secure hash and message digest algorithms. Included are the FIPS secure hash algorithms SHA1, SHA224, SHA256, SHA384, SHA512, (defined in the FIPS 180-4 standard), the SHA-3 series (defined in the FIPS 202 standard) as well as RSA's MD5 algorithm (defined in internet **RFC 1321**). The terms "secure hash" and "message digest" are interchangeable. Older algorithms were called message digests. The modern term is secure hash.

> **Note**
>
> If you want the adler32 or crc32 hash functions, they are available in the `zlib` module.

15.1.1 Hash algorithms

There is one constructor method named for each type of *hash*. All return a hash object with the same simple interface. For example: use `sha256()` to create a SHA-256 hash object. You can now feed this object with *bytes-like objects* (normally `bytes`) using the `update` method. At any point you can ask it for the *digest* of the concatenation of the data fed to it so far using the `digest()` or `hexdigest()` methods.

To allow multithreading, the Python *GIL* is released while computing a hash supplied more than 2047 bytes of data at once in its constructor or `.update` method.

Constructors for hash algorithms that are always present in this module are `sha1()`, `sha224()`, `sha256()`, `sha384()`, `sha512()`, `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()`, `shake_256()`, `blake2b()`, and `blake2s()`. `md5()` is normally available as well, though it may be missing or blocked if you are using a rare "FIPS compliant" build of Python. These correspond to `algorithms_guaranteed`.

Additional algorithms may also be available if your Python distribution's `hashlib` was linked against a build of OpenSSL that provides others. Others *are not guaranteed available* on all installations and will only be accessible by name via `new()`. See `algorithms_available`.

Warning

Some algorithms have known hash collision weaknesses (including MD5 and SHA1). Refer to Attacks on cryptographic hash algorithms and the *hashlib-seealso* section at the end of this document.

Added in version 3.6: SHA3 (Keccak) and SHAKE constructors *sha3_224()*, *sha3_256()*, *sha3_384()*, *sha3_512()*, *shake_128()*, *shake_256()* were added. *blake2b()* and *blake2s()* were added. Changed in version 3.9: All hashlib constructors take a keyword-only argument *usedforsecurity* with default value `True`. A false value allows the use of insecure and blocked hashing algorithms in restricted environments. `False` indicates that the hashing algorithm is not used in a security context, e.g. as a non-cryptographic one-way compression function.

Changed in version 3.9: Hashlib now uses SHA3 and SHAKE from OpenSSL if it provides it.

Changed in version 3.12: For any of the MD5, SHA1, SHA2, or SHA3 algorithms that the linked OpenSSL does not provide we fall back to a verified implementation from the HACL* project.

15.1.2 Usage

To obtain the digest of the byte string `b"Nobody inspects the spammish repetition"`:

```
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\x03\x1e\xdd}Ae\x15\x93\xc5\xfe\\\x00o\xa5u+7\xfd\xdf\xf7\xbcN\x84:\xa6\xaf\x0c\x95\
↪x0fK\x94\x06'
>>> m.hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406'
```

More condensed:

```
>>> hashlib.sha256(b"Nobody inspects the spammish repetition").hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406'
```

15.1.3 Constructors

`hashlib.`**`new`**(*name*, [*data*,]*, *usedforsecurity=True*)

Is a generic constructor that takes the string *name* of the desired algorithm as its first parameter. It also exists to allow access to the above listed hashes as well as any other algorithms that your OpenSSL library may offer.

Using *new()* with an algorithm name:

```
>>> h = hashlib.new('sha256')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406'
```

`hashlib.`**`md5`**([*data*,]*, *usedforsecurity=True*)

`hashlib.`**`sha1`**([*data*,]*, *usedforsecurity=True*)

`hashlib.`**`sha224`**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha256**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha384**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha512**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_224**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_256**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_384**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_512**([*data*,]*, *usedforsecurity=True*)

Named constructors such as these are faster than passing an algorithm name to *new()*.

15.1.4 Attributes

Hashlib provides the following constant module attributes:

hashlib.**algorithms_guaranteed**

A set containing the names of the hash algorithms guaranteed to be supported by this module on all platforms. Note that 'md5' is in this list despite some upstream vendors offering an odd "FIPS compliant" Python build that excludes it.

Added in version 3.2.

hashlib.**algorithms_available**

A set containing the names of the hash algorithms that are available in the running Python interpreter. These names will be recognized when passed to *new()*. *algorithms_guaranteed* will always be a subset. The same algorithm may appear multiple times in this set under different names (thanks to OpenSSL).

Added in version 3.2.

15.1.5 Hash Objects

The following values are provided as constant attributes of the hash objects returned by the constructors:

hash.**digest_size**

The size of the resulting hash in bytes.

hash.**block_size**

The internal block size of the hash algorithm in bytes.

A hash object has the following attributes:

hash.**name**

The canonical name of this hash, always lowercase and always suitable as a parameter to *new()* to create another hash of this type.

Changed in version 3.4: The name attribute has been present in CPython since its inception, but until Python 3.4 was not formally specified, so may not exist on some platforms.

A hash object has the following methods:

hash.**update**(*data*)

Update the hash object with the *bytes-like object*. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a+b)`.

hash.**digest**()

Return the digest of the data passed to the *update()* method so far. This is a bytes object of size *digest_size* which may contain bytes in the whole range from 0 to 255.

hash.**hexdigest**()

Like *digest()* except the digest is returned as a string object of double length, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

hash.**copy**()

Return a copy ("clone") of the hash object. This can be used to efficiently compute the digests of data sharing a common initial substring.

15.1.6 SHAKE variable length digests

hashlib.**shake_128**([*data*,]*, *usedforsecurity=True*)

hashlib.**shake_256**([*data*,]*, *usedforsecurity=True*)

The *shake_128()* and *shake_256()* algorithms provide variable length digests with length_in_bits//2 up to 128 or 256 bits of security. As such, their digest methods require a length. Maximum length is not limited by the SHAKE algorithm.

shake.**digest**(*length*)

Return the digest of the data passed to the *update()* method so far. This is a bytes object of size *length* which may contain bytes in the whole range from 0 to 255.

shake.**hexdigest**(*length*)

Like *digest()* except the digest is returned as a string object of double length, containing only hexadecimal digits. This may be used to exchange the value in email or other non-binary environments.

Example use:

```
>>> h = hashlib.shake_256(b'Nobody inspects the spammish repetition')
>>> h.hexdigest(20)
'44709d6fcb83d92a76dcb0b668c98e1b1d3dafe7'
```

15.1.7 File hashing

The hashlib module provides a helper function for efficient hashing of a file or file-like object.

hashlib.**file_digest**(*fileobj*, *digest*, */*)

Return a digest object that has been updated with contents of file object.

fileobj must be a file-like object opened for reading in binary mode. It accepts file objects from builtin *open()*, *BytesIO* instances, SocketIO objects from *socket.socket.makefile()*, and similar. The function may bypass Python's I/O and use the file descriptor from *fileno()* directly. *fileobj* must be assumed to be in an unknown state after this function returns or raises. It is up to the caller to close *fileobj*.

digest must either be a hash algorithm name as a *str*, a hash constructor, or a callable that returns a hash object.

Example:

```
>>> import io, hashlib, hmac
>>> with open(hashlib.__file__, "rb") as f:
...     digest = hashlib.file_digest(f, "sha256")
...
>>> digest.hexdigest()
'...'
```

```
>>> buf = io.BytesIO(b"somedata")
>>> mac1 = hmac.HMAC(b"key", digestmod=hashlib.sha512)
>>> digest = hashlib.file_digest(buf, lambda: mac1)
```

```
>>> digest is mac1
True
>>> mac2 = hmac.HMAC(b"key", b"somedata", digestmod=hashlib.sha512)
>>> mac1.digest() == mac2.digest()
True
```

Added in version 3.11.

15.1.8 Key derivation

Key derivation and key stretching algorithms are designed for secure password hashing. Naive algorithms such as `sha1(password)` are not resistant against brute-force attacks. A good password hashing function must be tunable, slow, and include a salt.

`hashlib.`**`pbkdf2_hmac`**(*hash_name*, *password*, *salt*, *iterations*, *dklen=None*)

The function provides PKCS#5 password-based key derivation function 2. It uses HMAC as pseudorandom function.

The string *hash_name* is the desired name of the hash digest algorithm for HMAC, e.g. 'sha1' or 'sha256'. *password* and *salt* are interpreted as buffers of bytes. Applications and libraries should limit *password* to a sensible length (e.g. 1024). *salt* should be about 16 or more bytes from a proper source, e.g. `os.urandom()`.

The number of *iterations* should be chosen based on the hash algorithm and computing power. As of 2022, hundreds of thousands of iterations of SHA-256 are suggested. For rationale as to why and how to choose what is best for your application, read *Appendix A.2.2* of NIST-SP-800-132. The answers on the stackexchange pbkdf2 iterations question explain in detail.

dklen is the length of the derived key in bytes. If *dklen* is `None` then the digest size of the hash algorithm *hash_name* is used, e.g. 64 for SHA-512.

```
>>> from hashlib import pbkdf2_hmac
>>> our_app_iters = 500_000  # Application specific, read above.
>>> dk = pbkdf2_hmac('sha256', b'password', b'bad salt' * 2, our_app_iters)
>>> dk.hex()
'15530bba69924174860db778f2c6f8104d3aaf9d26241840c8c4a641c8d000a9'
```

Function only available when Python is compiled with OpenSSL.

Added in version 3.4.

Changed in version 3.12: Function now only available when Python is built with OpenSSL. The slow pure Python implementation has been removed.

`hashlib.`**`scrypt`**(*password*, ***, *salt*, *n*, *r*, *p*, *maxmem=0*, *dklen=64*)

The function provides scrypt password-based key derivation function as defined in **RFC 7914**.

password and *salt* must be *bytes-like objects*. Applications and libraries should limit *password* to a sensible length (e.g. 1024). *salt* should be about 16 or more bytes from a proper source, e.g. `os.urandom()`.

n is the CPU/Memory cost factor, *r* the block size, *p* parallelization factor and *maxmem* limits memory (OpenSSL 1.1.0 defaults to 32 MiB). *dklen* is the length of the derived key in bytes.

Added in version 3.6.

15.1.9 BLAKE2

BLAKE2 is a cryptographic hash function defined in **RFC 7693** that comes in two flavors:

- **BLAKE2b**, optimized for 64-bit platforms and produces digests of any size between 1 and 64 bytes,
- **BLAKE2s**, optimized for 8- to 32-bit platforms and produces digests of any size between 1 and 32 bytes.

BLAKE2 supports **keyed mode** (a faster and simpler replacement for HMAC), **salted hashing**, **personalization**, and **tree hashing**.

Hash objects from this module follow the API of standard library's `hashlib` objects.

Creating hash objects

New hash objects are created by calling constructor functions:

`hashlib.`**`blake2b`**(*data=b''*, ***, *digest_size=64*, *key=b''*, *salt=b''*, *person=b''*, *fanout=1*, *depth=1*, *leaf_size=0*, *node_offset=0*, *node_depth=0*, *inner_size=0*, *last_node=False*, *usedforsecurity=True*)

`hashlib.`**`blake2s`**(*data=b''*, ***, *digest_size=32*, *key=b''*, *salt=b''*, *person=b''*, *fanout=1*, *depth=1*, *leaf_size=0*, *node_offset=0*, *node_depth=0*, *inner_size=0*, *last_node=False*, *usedforsecurity=True*)

These functions return the corresponding hash objects for calculating BLAKE2b or BLAKE2s. They optionally take these general parameters:

- *data*: initial chunk of data to hash, which must be *bytes-like object*. It can be passed only as positional argument.
- *digest_size*: size of output digest in bytes.
- *key*: key for keyed hashing (up to 64 bytes for BLAKE2b, up to 32 bytes for BLAKE2s).
- *salt*: salt for randomized hashing (up to 16 bytes for BLAKE2b, up to 8 bytes for BLAKE2s).
- *person*: personalization string (up to 16 bytes for BLAKE2b, up to 8 bytes for BLAKE2s).

The following table shows limits for general parameters (in bytes):

Hash	digest_size	len(key)	len(salt)	len(person)
BLAKE2b	64	64	16	16
BLAKE2s	32	32	8	8

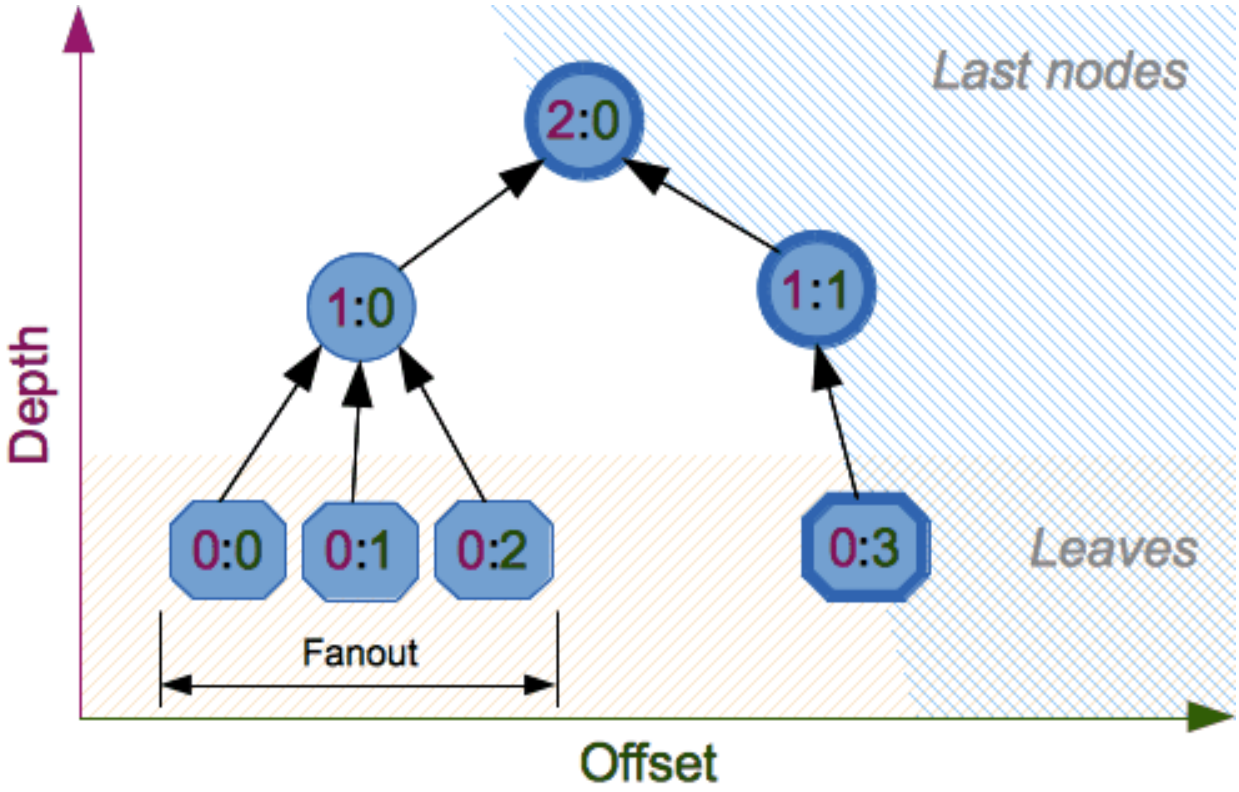
Note

BLAKE2 specification defines constant lengths for salt and personalization parameters, however, for convenience, this implementation accepts byte strings of any size up to the specified length. If the length of the parameter is less than specified, it is padded with zeros, thus, for example, `b'salt'` and `b'salt\x00'` is the same value. (This is not the case for *key*.)

These sizes are available as module *constants* described below.

Constructor functions also accept the following tree hashing parameters:

- *fanout*: fanout (0 to 255, 0 if unlimited, 1 in sequential mode).
- *depth*: maximal depth of tree (1 to 255, 255 if unlimited, 1 in sequential mode).
- *leaf_size*: maximal byte length of leaf (0 to `2**32-1`, 0 if unlimited or in sequential mode).
- *node_offset*: node offset (0 to `2**64-1` for BLAKE2b, 0 to `2**48-1` for BLAKE2s, 0 for the first, leftmost, leaf, or in sequential mode).
- *node_depth*: node depth (0 to 255, 0 for leaves, or in sequential mode).
- *inner_size*: inner digest size (0 to 64 for BLAKE2b, 0 to 32 for BLAKE2s, 0 in sequential mode).
- *last_node*: boolean indicating whether the processed node is the last one (`False` for sequential mode).



See section 2.10 in BLAKE2 specification for comprehensive review of tree hashing.

Constants

`blake2b.`**`SALT_SIZE`**

`blake2s.`**`SALT_SIZE`**

Salt length (maximum length accepted by constructors).

`blake2b.`**`PERSON_SIZE`**

blake2s.**PERSON_SIZE**

Personalization string length (maximum length accepted by constructors).

blake2b.**MAX_KEY_SIZE**

blake2s.**MAX_KEY_SIZE**

Maximum key size.

blake2b.**MAX_DIGEST_SIZE**

blake2s.**MAX_DIGEST_SIZE**

Maximum digest size that the hash function can output.

Examples

Simple hashing

To calculate hash of some data, you should first construct a hash object by calling the appropriate constructor function (*blake2b()* or *blake2s()*), then update it with the data by calling *update()* on the object, and, finally, get the digest out of the object by calling *digest()* (or *hexdigest()* for hex-encoded string).

```
>>> from hashlib import blake2b
>>> h = blake2b()
>>> h.update(b'Hello world')
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc
↪'
```

As a shortcut, you can pass the first chunk of data to update directly to the constructor as the positional argument:

```
>>> from hashlib import blake2b
>>> blake2b(b'Hello world').hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc
↪'
```

You can call *hash.update()* as many times as you need to iteratively update the hash:

```
>>> from hashlib import blake2b
>>> items = [b'Hello', b' ', b'world']
>>> h = blake2b()
>>> for item in items:
...     h.update(item)
...
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc
↪'
```

Using different digest sizes

BLAKE2 has configurable size of digests up to 64 bytes for BLAKE2b and up to 32 bytes for BLAKE2s. For example, to replace SHA-1 with BLAKE2b without changing the size of output, we can tell BLAKE2b to produce 20-byte digests:

```
>>> from hashlib import blake2b
>>> h = blake2b(digest_size=20)
>>> h.update(b'Replacing SHA1 with the more secure function')
>>> h.hexdigest()
'd24f26cf8de66472d58d4e1b1774b4c9158b1f4c'
>>> h.digest_size
20
>>> len(h.digest())
20
```

Hash objects with different digest sizes have completely different outputs (shorter hashes are *not* prefixes of longer hashes); BLAKE2b and BLAKE2s produce different outputs even if the output length is the same:

```
>>> from hashlib import blake2b, blake2s
>>> blake2b(digest_size=10).hexdigest()
'6fa1d8fcfd719046d762'
>>> blake2b(digest_size=11).hexdigest()
'eb6ec15daf9546254f0809'
>>> blake2s(digest_size=10).hexdigest()
'1bf21a98c78a1c376ae9'
>>> blake2s(digest_size=11).hexdigest()
'567004bf96e4a25773ebf4'
```

Keyed hashing

Keyed hashing can be used for authentication as a faster and simpler replacement for Hash-based message authentication code (HMAC). BLAKE2 can be securely used in prefix-MAC mode thanks to the indifferentiability property inherited from BLAKE.

This example shows how to get a (hex-encoded) 128-bit authentication code for message `b'message data'` with key `b'pseudorandom key'`:

```
>>> from hashlib import blake2b
>>> h = blake2b(key=b'pseudorandom key', digest_size=16)
>>> h.update(b'message data')
>>> h.hexdigest()
'3d363ff7401e02026f4a4687d4863ced'
```

As a practical example, a web application can symmetrically sign cookies sent to users and later verify them to make sure they weren't tampered with:

```
>>> from hashlib import blake2b
>>> from hmac import compare_digest
>>>
>>> SECRET_KEY = b'pseudorandomly generated server secret key'
>>> AUTH_SIZE = 16
>>>
>>> def sign(cookie):
...     h = blake2b(digest_size=AUTH_SIZE, key=SECRET_KEY)
...     h.update(cookie)
```

(continues on next page)

(continued from previous page)

```
...     return h.hexdigest().encode('utf-8')
>>>
>>> def verify(cookie, sig):
...     good_sig = sign(cookie)
...     return compare_digest(good_sig, sig)
>>>
>>> cookie = b'user-alice'
>>> sig = sign(cookie)
>>> print("{0},{1}".format(cookie.decode('utf-8'), sig))
user-alice,b'43b3c982cf697e0c5ab22172d1ca7421'
>>> verify(cookie, sig)
True
>>> verify(b'user-bob', sig)
False
>>> verify(cookie, b'0102030405060708090a0b0c0d0e0f00')
False
```

Even though there's a native keyed hashing mode, BLAKE2 can, of course, be used in HMAC construction with *hmac* module:

```
>>> import hmac, hashlib
>>> m = hmac.new(b'secret key', digestmod=hashlib.blake2s)
>>> m.update(b'message')
>>> m.hexdigest()
'e3c8102868d28b5ff85fc35dda07329970d1a01e273c37481326fe0c861c8142'
```

Randomized hashing

By setting *salt* parameter users can introduce randomization to the hash function. Randomized hashing is useful for protecting against collision attacks on the hash function used in digital signatures.

> Randomized hashing is designed for situations where one party, the message preparer, generates all or part of a message to be signed by a second party, the message signer. If the message preparer is able to find cryptographic hash function collisions (i.e., two messages producing the same hash value), then they might prepare meaningful versions of the message that would produce the same hash value and digital signature, but with different results (e.g., transferring $1,000,000 to an account, rather than $10). Cryptographic hash functions have been designed with collision resistance as a major goal, but the current concentration on attacking cryptographic hash functions may result in a given cryptographic hash function providing less collision resistance than expected. Randomized hashing offers the signer additional protection by reducing the likelihood that a preparer can generate two or more messages that ultimately yield the same hash value during the digital signature generation process — even if it is practical to find collisions for the hash function. However, the use of randomized hashing may reduce the amount of security provided by a digital signature when all portions of the message are prepared by the signer.
>
> (NIST SP-800-106 "Randomized Hashing for Digital Signatures")

In BLAKE2 the salt is processed as a one-time input to the hash function during initialization, rather than as an input to each compression function.

Warning

Salted hashing (or just hashing) with BLAKE2 or any other general-purpose cryptographic hash function, such as SHA-256, is not suitable for hashing passwords. See BLAKE2 FAQ for more information.

```
>>> import os
>>> from hashlib import blake2b
>>> msg = b'some message'
>>> # Calculate the first hash with a random salt.
>>> salt1 = os.urandom(blake2b.SALT_SIZE)
>>> h1 = blake2b(salt=salt1)
>>> h1.update(msg)
>>> # Calculate the second hash with a different random salt.
>>> salt2 = os.urandom(blake2b.SALT_SIZE)
>>> h2 = blake2b(salt=salt2)
>>> h2.update(msg)
>>> # The digests are different.
>>> h1.digest() != h2.digest()
True
```

Personalization

Sometimes it is useful to force hash function to produce different digests for the same input for different purposes. Quoting the authors of the Skein hash function:

> We recommend that all application designers seriously consider doing this; we have seen many protocols where a hash that is computed in one part of the protocol can be used in an entirely different part because two hash computations were done on similar or related data, and the attacker can force the application to make the hash inputs the same. Personalizing each hash function used in the protocol summarily stops this type of attack.
>
> (The Skein Hash Function Family, p. 21)

BLAKE2 can be personalized by passing bytes to the *person* argument:

```
>>> from hashlib import blake2b
>>> FILES_HASH_PERSON = b'MyApp Files Hash'
>>> BLOCK_HASH_PERSON = b'MyApp Block Hash'
>>> h = blake2b(digest_size=32, person=FILES_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'20d9cd024d4fb086aae819a1432dd2466de12947831b75c5a30cf2676095d3b4'
>>> h = blake2b(digest_size=32, person=BLOCK_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'cf68fb5761b9c44e7878bfb2c4c9aea52264a80b75005e65619778de59f383a3'
```

Personalization together with the keyed mode can also be used to derive different keys from a single one.

```
>>> from hashlib import blake2s
>>> from base64 import b64decode, b64encode
>>> orig_key = b64decode(b'Rm5EPJai72qcK3RGBpW3vPNfZy5OZothY+kHY6h21KM=')
>>> enc_key = blake2s(key=orig_key, person=b'kEncrypt').digest()
>>> mac_key = blake2s(key=orig_key, person=b'kMAC').digest()
>>> print(b64encode(enc_key).decode('utf-8'))
rbPb15S/Z9t+agffno5wuhB77VbRi6F9Iv2qIxU7WHw=
>>> print(b64encode(mac_key).decode('utf-8'))
G9GtHFE1YluXY1zWPlYk1e/nWfu0WSEb0KRcjhDeP/o=
```

Tree mode

Here's an example of hashing a minimal tree with two leaf nodes:

```
  10
 /  \
00  01
```

This example uses 64-byte internal digests, and returns the 32-byte final digest:

```
>>> from hashlib import blake2b
>>>
>>> FANOUT = 2
>>> DEPTH = 2
>>> LEAF_SIZE = 4096
>>> INNER_SIZE = 64
>>>
>>> buf = bytearray(6000)
>>>
>>> # Left leaf
... h00 = blake2b(buf[0:LEAF_SIZE], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=0, last_node=False)
>>> # Right leaf
... h01 = blake2b(buf[LEAF_SIZE:], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=1, node_depth=0, last_node=True)
>>> # Root node
... h10 = blake2b(digest_size=32, fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=1, last_node=True)
>>> h10.update(h00.digest())
>>> h10.update(h01.digest())
>>> h10.hexdigest()
'3ad2a9b37c6070e374c7a8c508fe20ca86b6ed54e286e93a0318e95e881db5aa'
```

Credits

BLAKE2 was designed by *Jean-Philippe Aumasson*, *Samuel Neves*, *Zooko Wilcox-O'Hearn*, and *Christian Winnerlein* based on SHA-3 finalist BLAKE created by *Jean-Philippe Aumasson*, *Luca Henzen*, *Willi Meier*, and *Raphael C.-W. Phan*.

It uses core algorithm from ChaCha cipher designed by *Daniel J. Bernstein*.

The stdlib implementation is based on pyblake2 module. It was written by *Dmitry Chestnykh* based on C implementation written by *Samuel Neves*. The documentation was copied from pyblake2 and written by *Dmitry Chestnykh*.

The C code was partly rewritten for Python by *Christian Heimes*.

The following public domain dedication applies for both C hash function implementation, extension code, and this documentation:

> To the extent possible under law, the author(s) have dedicated all copyright and related and neighboring rights to this software to the public domain worldwide. This software is distributed without any warranty.
>
> You should have received a copy of the CC0 Public Domain Dedication along with this software. If not, see https://creativecommons.org/publicdomain/zero/1.0/.

The following people have helped with development or contributed their changes to the project and the public domain according to the Creative Commons Public Domain Dedication 1.0 Universal:

- *Alexandr Sokolovskiy*

See also

Module ***hmac***
A module to generate message authentication codes using hashes.

Module ***base64***
Another way to encode binary hashes for non-binary environments.

https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.180-4.pdf
The FIPS 180-4 publication on Secure Hash Algorithms.

https://csrc.nist.gov/publications/detail/fips/202/final
The FIPS 202 publication on the SHA-3 Standard.

https://www.blake2.net/
Official BLAKE2 website.

https://en.wikipedia.org/wiki/Cryptographic_hash_function
Wikipedia article with information on which algorithms have known issues and what that means regarding their use.

https://www.ietf.org/rfc/rfc8018.txt
PKCS #5: Password-Based Cryptography Specification Version 2.1

https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf
NIST Recommendation for Password-Based Key Derivation.

15.2 `hmac` — Keyed-Hashing for Message Authentication

Source code: Lib/hmac.py

This module implements the HMAC algorithm as described by **RFC 2104**.

`hmac.new`(*key*, *msg=None*, *digestmod*)

Return a new hmac object. *key* is a bytes or bytearray object giving the secret key. If *msg* is present, the method call `update(msg)` is made. *digestmod* is the digest name, digest constructor or module for the HMAC object to use. It may be any name suitable to *hashlib.new()*. Despite its argument position, it is required.

Changed in version 3.4: Parameter *key* can be a bytes or bytearray object. Parameter *msg* can be of any type supported by *hashlib*. Parameter *digestmod* can be the name of a hash algorithm.

Changed in version 3.8: The *digestmod* argument is now required. Pass it as a keyword argument to avoid awkwardness when you do not have an initial *msg*.

`hmac.digest`(*key*, *msg*, *digest*)

Return digest of *msg* for given secret *key* and *digest*. The function is equivalent to `HMAC(key, msg, digest).digest()`, but uses an optimized C or inline implementation, which is faster for messages that fit into memory. The parameters *key*, *msg*, and *digest* have the same meaning as in *new()*.

CPython implementation detail, the optimized C implementation is only used when *digest* is a string and name of a digest algorithm, which is supported by OpenSSL.

Added in version 3.7.

An HMAC object has the following methods:

HMAC.**update**(*msg*)

Update the hmac object with *msg*. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a + b)`.

Changed in version 3.4: Parameter *msg* can be of any type supported by *hashlib*.

HMAC.**digest**()

Return the digest of the bytes passed to the *update()* method so far. This bytes object will be the same length as the *digest_size* of the digest given to the constructor. It may contain non-ASCII bytes, including NUL bytes.

> **Warning**
>
> When comparing the output of *digest()* to an externally supplied digest during a verification routine, it is recommended to use the *compare_digest()* function instead of the == operator to reduce the vulnerability to timing attacks.

HMAC.**hexdigest**()

Like *digest()* except the digest is returned as a string twice the length containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

> **Warning**
>
> When comparing the output of *hexdigest()* to an externally supplied digest during a verification routine, it is recommended to use the *compare_digest()* function instead of the == operator to reduce the vulnerability to timing attacks.

HMAC.**copy**()

Return a copy ("clone") of the hmac object. This can be used to efficiently compute the digests of strings that share a common initial substring.

A hash object has the following attributes:

HMAC.**digest_size**

The size of the resulting HMAC digest in bytes.

HMAC.**block_size**

The internal block size of the hash algorithm in bytes.

Added in version 3.4.

HMAC.**name**

The canonical name of this HMAC, always lowercase, e.g. `hmac-md5`.

Added in version 3.4.

Changed in version 3.10: Removed the undocumented attributes `HMAC.digest_cons`, `HMAC.inner`, and `HMAC.outer`.

This module also provides the following helper function:

hmac.**compare_digest**(*a*, *b*)

Return `a == b`. This function uses an approach designed to prevent timing analysis by avoiding content-based short circuiting behaviour, making it appropriate for cryptography. *a* and *b* must both be of the same type: either *str* (ASCII only, as e.g. returned by *HMAC.hexdigest()*), or a *bytes-like object*.

Note

If *a* and *b* are of different lengths, or if an error occurs, a timing attack could theoretically reveal information about the types and lengths of *a* and *b*—but not their values.

Added in version 3.3.

Changed in version 3.10: The function uses OpenSSL's `CRYPTO_memcmp()` internally when available.

See also

Module *hashlib*
The Python module providing secure hash functions.

15.3 `secrets` — Generate secure random numbers for managing secrets

Added in version 3.6.

Source code: Lib/secrets.py

The *secrets* module is used for generating cryptographically strong random numbers suitable for managing data such as passwords, account authentication, security tokens, and related secrets.

In particular, *secrets* should be used in preference to the default pseudo-random number generator in the *random* module, which is designed for modelling and simulation, not security or cryptography.

See also

PEP 506

15.3.1 Random numbers

The *secrets* module provides access to the most secure source of randomness that your operating system provides.

class secrets.**SystemRandom**

A class for generating random numbers using the highest-quality sources provided by the operating system. See *random.SystemRandom* for additional details.

secrets.**choice**(*seq*)

Return a randomly chosen element from a non-empty sequence.

secrets.**randbelow**(*exclusive_upper_bound*)

Return a random int in the range [0, *exclusive_upper_bound*).

secrets.**randbits**(*k*)

Return an int with *k* random bits.

15.3.2 Generating tokens

The *secrets* module provides functions for generating secure tokens, suitable for applications such as password resets, hard-to-guess URLs, and similar.

secrets.**token_bytes**([*nbytes=None*])

Return a random byte string containing *nbytes* number of bytes. If *nbytes* is `None` or not supplied, a reasonable default is used.

```
>>> token_bytes(16)
b'\xebr\x17D*t\xae\xd4\xe3S\xb6\xe2\xebP1\x8b'
```

secrets.**token_hex**([*nbytes=None*])

Return a random text string, in hexadecimal. The string has *nbytes* random bytes, each byte converted to two hex digits. If *nbytes* is `None` or not supplied, a reasonable default is used.

```
>>> token_hex(16)
'f9bf78b9a18ce6d46a0cd2b0b86df9da'
```

secrets.**token_urlsafe**([*nbytes=None*])

Return a random URL-safe text string, containing *nbytes* random bytes. The text is Base64 encoded, so on average each byte results in approximately 1.3 characters. If *nbytes* is `None` or not supplied, a reasonable default is used.

```
>>> token_urlsafe(16)
'Drmhze6EPcv0fN_81Bj-nA'
```

How many bytes should tokens use?

To be secure against brute-force attacks, tokens need to have sufficient randomness. Unfortunately, what is considered sufficient will necessarily increase as computers get more powerful and able to make more guesses in a shorter period. As of 2015, it is believed that 32 bytes (256 bits) of randomness is sufficient for the typical use-case expected for the *secrets* module.

For those who want to manage their own token length, you can explicitly specify how much randomness is used for tokens by giving an *int* argument to the various `token_*` functions. That argument is taken as the number of bytes of randomness to use.

Otherwise, if no argument is provided, or if the argument is `None`, the `token_*` functions will use a reasonable default instead.

> **Note**
>
> That default is subject to change at any time, including during maintenance releases.

15.3.3 Other functions

secrets.**compare_digest**(*a*, *b*)

Return `True` if strings or *bytes-like objects* *a* and *b* are equal, otherwise `False`, using a "constant-time compare" to reduce the risk of timing attacks. See *`hmac.compare_digest()`* for additional details.

15.3.4 Recipes and best practices

This section shows recipes and best practices for using *`secrets`* to manage a basic level of security.

Generate an eight-character alphanumeric password:

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
password = ''.join(secrets.choice(alphabet) for i in range(8))
```

Note

Applications should not store passwords in a recoverable format, whether plain text or encrypted. They should be salted and hashed using a cryptographically strong one-way (irreversible) hash function.

Generate a ten-character alphanumeric password with at least one lowercase character, at least one uppercase character, and at least three digits:

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(secrets.choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
            and any(c.isupper() for c in password)
            and sum(c.isdigit() for c in password) >= 3):
        break
```

Generate an XKCD-style passphrase:

```
import secrets
# On standard Linux systems, use a convenient dictionary file.
# Other platforms may need to provide their own word-list.
with open('/usr/share/dict/words') as f:
    words = [word.strip() for word in f]
    password = ' '.join(secrets.choice(words) for i in range(4))
```

Generate a hard-to-guess temporary URL containing a security token suitable for password recovery applications:

```
import secrets
url = 'https://example.com/reset=' + secrets.token_urlsafe()
```

CHAPTER

SIXTEEN

GENERIC OPERATING SYSTEM SERVICES

The modules described in this chapter provide interfaces to operating system features that are available on (almost) all operating systems, such as files and a clock. The interfaces are generally modeled after the Unix or C interfaces, but they are available on most other systems as well. Here's an overview:

16.1 os — Miscellaneous operating system interfaces

Source code: Lib/os.py

This module provides a portable way of using operating system dependent functionality. If you just want to read or write a file see *open()*, if you want to manipulate paths, see the *os.path* module, and if you want to read all the lines in all the files on the command line see the *fileinput* module. For creating temporary files and directories see the *tempfile* module, and for high-level file and directory handling see the *shutil* module.

Notes on the availability of these functions:

- The design of all built-in operating system dependent modules of Python is such that as long as the same functionality is available, it uses the same interface; for example, the function `os.stat(path)` returns stat information about *path* in the same format (which happens to have originated with the POSIX interface).
- Extensions peculiar to a particular operating system are also available through the *os* module, but using them is of course a threat to portability.
- All functions accepting path or file names accept both bytes and string objects, and result in an object of the same type, if a path or file name is returned.
- On VxWorks, os.popen, os.fork, os.execv and os.spawn*p* are not supported.
- On WebAssembly platforms, and on iOS, large parts of the *os* module are not available or behave differently. API related to processes (e.g. *fork()*, *execve()*) and resources (e.g. *nice()*) are not available. Others like *getuid()* and *getpid()* are emulated or stubs. WebAssembly platforms also lack support for signals (e.g. *kill()*, *wait()*).

> **Note**
>
> All functions in this module raise *OSError* (or subclasses thereof) in the case of invalid or inaccessible file names and paths, or other arguments that have the correct type, but are not accepted by the operating system.

exception `os.error`

An alias for the built-in *OSError* exception.

os.**name**

The name of the operating system dependent module imported. The following names have currently been registered: 'posix', 'nt', 'java'.

> **See also**
>
> *sys.platform* has a finer granularity. *os.uname()* gives system-dependent version information.
>
> The *platform* module provides detailed checks for the system's identity.

16.1.1 File Names, Command Line Arguments, and Environment Variables

In Python, file names, command line arguments, and environment variables are represented using the string type. On some systems, decoding these strings to and from bytes is necessary before passing them to the operating system. Python uses the *filesystem encoding and error handler* to perform this conversion (see *sys.getfilesystemencoding()*).

The *filesystem encoding and error handler* are configured at Python startup by the PyConfig_Read() function: see filesystem_encoding and filesystem_errors members of PyConfig.

Changed in version 3.1: On some systems, conversion using the file system encoding may fail. In this case, Python uses the *surrogateescape encoding error handler*, which means that undecodable bytes are replaced by a Unicode character U+DC*xx* on decoding, and these are again translated to the original byte on encoding.

The *file system encoding* must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise *UnicodeError*.

See also the *locale encoding*.

16.1.2 Python UTF-8 Mode

Added in version 3.7: See **PEP 540** for more details.

The Python UTF-8 Mode ignores the *locale encoding* and forces the usage of the UTF-8 encoding:

- Use UTF-8 as the *filesystem encoding*.
- *sys.getfilesystemencoding()* returns 'utf-8'.
- *locale.getpreferredencoding()* returns 'utf-8' (the *do_setlocale* argument has no effect).
- *sys.stdin*, *sys.stdout*, and *sys.stderr* all use UTF-8 as their text encoding, with the surrogateescape *error handler* being enabled for *sys.stdin* and *sys.stdout* (*sys.stderr* continues to use backslashreplace as it does in the default locale-aware mode)
- On Unix, *os.device_encoding()* returns 'utf-8' rather than the device encoding.

Note that the standard stream settings in UTF-8 mode can be overridden by PYTHONIOENCODING (just as they can be in the default locale-aware mode).

As a consequence of the changes in those lower level APIs, other higher level APIs also exhibit different default behaviours:

- Command line arguments, environment variables and filenames are decoded to text using the UTF-8 encoding.
- *os.fsdecode()* and *os.fsencode()* use the UTF-8 encoding.
- *open()*, *io.open()*, and *codecs.open()* use the UTF-8 encoding by default. However, they still use the strict error handler by default so that attempting to open a binary file in text mode is likely to raise an exception rather than producing nonsense data.

The *Python UTF-8 Mode* is enabled if the LC_CTYPE locale is `C` or `POSIX` at Python startup (see the `PyConfig_Read()` function).

It can be enabled or disabled using the `-X utf8` command line option and the `PYTHONUTF8` environment variable.

If the `PYTHONUTF8` environment variable is not set at all, then the interpreter defaults to using the current locale settings, *unless* the current locale is identified as a legacy ASCII-based locale (as described for `PYTHONCOERCECLOCALE`), and locale coercion is either disabled or fails. In such legacy locales, the interpreter will default to enabling UTF-8 mode unless explicitly instructed not to do so.

The Python UTF-8 Mode can only be enabled at the Python startup. Its value can be read from *sys.flags.utf8_mode*.

See also the UTF-8 mode on Windows and the *filesystem encoding and error handler*.

See also

PEP 686
Python 3.15 will make *Python UTF-8 Mode* default.

16.1.3 Process Parameters

These functions and data items provide information and operate on the current process and user.

`os.ctermid()`

Return the filename corresponding to the controlling terminal of the process.

Availability: Unix, not WASI.

`os.environ`

A *mapping* object where keys and values are strings that represent the process environment. For example, `environ['HOME']` is the pathname of your home directory (on some platforms), and is equivalent to `getenv("HOME")` in C.

This mapping is captured the first time the *os* module is imported, typically during Python startup as part of processing `site.py`. Changes to the environment made after this time are not reflected in *os.environ*, except for changes made by modifying *os.environ* directly.

This mapping may be used to modify the environment as well as query the environment. *putenv()* will be called automatically when the mapping is modified.

On Unix, keys and values use *sys.getfilesystemencoding()* and `'surrogateescape'` error handler. Use *environb* if you would like to use a different encoding.

On Windows, the keys are converted to uppercase. This also applies when getting, setting, or deleting an item. For example, `environ['monty'] = 'python'` maps the key `'MONTY'` to the value `'python'`.

Note

Calling *putenv()* directly does not change *os.environ*, so it's better to modify *os.environ*.

Note

On some platforms, including FreeBSD and macOS, setting `environ` may cause memory leaks. Refer to the system documentation for `putenv()`.

You can delete items in this mapping to unset environment variables. *unsetenv()* will be called automatically when an item is deleted from *os.environ*, and when one of the `pop()` or `clear()` methods is called.

Changed in version 3.9: Updated to support **PEP 584**'s merge (`|`) and update (`|=`) operators.

os.environb

Bytes version of *environ*: a *mapping* object where both keys and values are *bytes* objects representing the process environment. *environ* and *environb* are synchronized (modifying *environb* updates *environ*, and vice versa).

environb is only available if *supports_bytes_environ* is `True`.

Added in version 3.2.

Changed in version 3.9: Updated to support **PEP 584**'s merge (`|`) and update (`|=`) operators.

os.chdir(*path*)

os.fchdir(*fd*)

os.getcwd()

These functions are described in *Files and Directories*.

os.fsencode(*filename*)

Encode *path-like filename* to the *filesystem encoding and error handler*; return *bytes* unchanged.

fsdecode() is the reverse function.

Added in version 3.2.

Changed in version 3.6: Support added to accept objects implementing the *os.PathLike* interface.

os.fsdecode(*filename*)

Decode the *path-like filename* from the *filesystem encoding and error handler*; return *str* unchanged.

fsencode() is the reverse function.

Added in version 3.2.

Changed in version 3.6: Support added to accept objects implementing the *os.PathLike* interface.

os.fspath(*path*)

Return the file system representation of the path.

If *str* or *bytes* is passed in, it is returned unchanged. Otherwise *__fspath__()* is called and its value is returned as long as it is a *str* or *bytes* object. In all other cases, *TypeError* is raised.

Added in version 3.6.

class os.PathLike

An *abstract base class* for objects representing a file system path, e.g. *pathlib.PurePath*.

Added in version 3.6.

abstractmethod __fspath__()

Return the file system path representation of the object.

The method should only return a *str* or *bytes* object, with the preference being for *str*.

os.**getenv**(*key*, *default=None*)

Return the value of the environment variable *key* as a string if it exists, or *default* if it doesn't. *key* is a string. Note that since *getenv()* uses *os.environ*, the mapping of *getenv()* is similarly also captured on import, and the function may not reflect future environment changes.

On Unix, keys and values are decoded with *sys.getfilesystemencoding()* and 'surrogateescape' error handler. Use *os.getenvb()* if you would like to use a different encoding.

Availability: Unix, Windows.

os.**getenvb**(*key*, *default=None*)

Return the value of the environment variable *key* as bytes if it exists, or *default* if it doesn't. *key* must be bytes. Note that since *getenvb()* uses *os.environb*, the mapping of *getenvb()* is similarly also captured on import, and the function may not reflect future environment changes.

getenvb() is only available if *supports_bytes_environ* is True.

Availability: Unix.

Added in version 3.2.

os.**get_exec_path**(*env=None*)

Returns the list of directories that will be searched for a named executable, similar to a shell, when launching a process. *env*, when specified, should be an environment variable dictionary to lookup the PATH in. By default, when *env* is None, *environ* is used.

Added in version 3.2.

os.**getegid**()

Return the effective group id of the current process. This corresponds to the "set id" bit on the file being executed in the current process.

Availability: Unix, not WASI.

os.**geteuid**()

Return the current process's effective user id.

Availability: Unix, not WASI.

os.**getgid**()

Return the real group id of the current process.

Availability: Unix.

The function is a stub on WASI, see *WebAssembly platforms* for more information.

os.**getgrouplist**(*user*, *group*, */*)

Return list of group ids that *user* belongs to. If *group* is not in the list, it is included; typically, *group* is specified as the group ID field from the password record for *user*, because that group ID will otherwise be potentially omitted.

Availability: Unix, not WASI.

Added in version 3.3.

os.**getgroups**()

Return list of supplemental group ids associated with the current process.

Availability: Unix, not WASI.

> **Note**
>
> On macOS, *getgroups()* behavior differs somewhat from other Unix platforms. If the Python interpreter was built with a deployment target of `10.5` or earlier, *getgroups()* returns the list of effective group ids associated with the current user process; this list is limited to a system-defined number of entries, typically 16, and may be modified by calls to *setgroups()* if suitably privileged. If built with a deployment target greater than `10.5`, *getgroups()* returns the current group access list for the user associated with the effective user id of the process; the group access list may change over the lifetime of the process, it is not affected by calls to *setgroups()*, and its length is not limited to 16. The deployment target value, `MACOSX_DEPLOYMENT_TARGET`, can be obtained with *sysconfig.get_config_var()*.

os.**getlogin**()

Return the name of the user logged in on the controlling terminal of the process. For most purposes, it is more useful to use *getpass.getuser()* since the latter checks the environment variables `LOGNAME` or `USERNAME` to find out who the user is, and falls back to `pwd.getpwuid(os.getuid())[0]` to get the login name of the current real user id.

Availability: Unix, Windows, not WASI.

os.**getpgid**(*pid*)

Return the process group id of the process with process id *pid*. If *pid* is 0, the process group id of the current process is returned.

Availability: Unix, not WASI.

os.**getpgrp**()

Return the id of the current process group.

Availability: Unix, not WASI.

os.**getpid**()

Return the current process id.

The function is a stub on WASI, see *WebAssembly platforms* for more information.

os.**getppid**()

Return the parent's process id. When the parent process has exited, on Unix the id returned is the one of the init process (1), on Windows it is still the same id, which may be already reused by another process.

Availability: Unix, Windows, not WASI.

Changed in version 3.2: Added support for Windows.

os.**getpriority**(*which*, *who*)

Get program scheduling priority. The value *which* is one of *PRIO_PROCESS*, *PRIO_PGRP*, or *PRIO_USER*, and *who* is interpreted relative to *which* (a process identifier for *PRIO_PROCESS*, process group identifier for *PRIO_PGRP*, and a user ID for *PRIO_USER*). A zero value for *who* denotes (respectively) the calling process, the process group of the calling process, or the real user ID of the calling process.

Availability: Unix, not WASI.

Added in version 3.3.

os.**PRIO_PROCESS**

os.**PRIO_PGRP**

os.**PRIO_USER**

Parameters for the *getpriority()* and *setpriority()* functions.

Availability: Unix, not WASI.

Added in version 3.3.

os.**PRIO_DARWIN_THREAD**

os.**PRIO_DARWIN_PROCESS**

os.**PRIO_DARWIN_BG**

os.**PRIO_DARWIN_NONUI**

Parameters for the *getpriority()* and *setpriority()* functions.

Availability: macOS

Added in version 3.12.

os.**getresuid**()

Return a tuple (ruid, euid, suid) denoting the current process's real, effective, and saved user ids.

Availability: Unix, not WASI.

Added in version 3.2.

os.**getresgid**()

Return a tuple (rgid, egid, sgid) denoting the current process's real, effective, and saved group ids.

Availability: Unix, not WASI.

Added in version 3.2.

os.**getuid**()

Return the current process's real user id.

Availability: Unix.

The function is a stub on WASI, see *WebAssembly platforms* for more information.

os.**initgroups**(*username*, *gid*, */*)

Call the system initgroups() to initialize the group access list with all of the groups of which the specified username is a member, plus the specified group id.

Availability: Unix, not WASI.

Added in version 3.2.

os.**putenv**(*key*, *value*, */*)

Set the environment variable named *key* to the string *value*. Such changes to the environment affect subprocesses started with *os.system()*, *popen()* or *fork()* and *execv()*.

Assignments to items in *os.environ* are automatically translated into corresponding calls to *putenv()*; however, calls to *putenv()* don't update *os.environ*, so it is actually preferable to assign to items of *os.environ*. This also applies to *getenv()* and *getenvb()*, which respectively use *os.environ* and *os.environb* in their implementations.

Note

On some platforms, including FreeBSD and macOS, setting `environ` may cause memory leaks. Refer to the system documentation for `putenv()`.

Raises an *auditing event* `os.putenv` with arguments `key`, `value`.

Changed in version 3.9: The function is now always available.

`os.`**`setegid`**(*egid*, `/`)

Set the current process's effective group id.

Availability: Unix, not WASI.

`os.`**`seteuid`**(*euid*, `/`)

Set the current process's effective user id.

Availability: Unix, not WASI.

`os.`**`setgid`**(*gid*, `/`)

Set the current process' group id.

Availability: Unix, not WASI.

`os.`**`setgroups`**(*groups*, `/`)

Set the list of supplemental group ids associated with the current process to *groups*. *groups* must be a sequence, and each element must be an integer identifying a group. This operation is typically available only to the superuser.

Availability: Unix, not WASI.

> **Note**
>
> On macOS, the length of *groups* may not exceed the system-defined maximum number of effective group ids, typically 16. See the documentation for *`getgroups()`* for cases where it may not return the same group list set by calling setgroups().

`os.`**`setns`**(*fd*, *nstype=0*)

Reassociate the current thread with a Linux namespace. See the *`setns(2)`* and *`namespaces(7)`* man pages for more details.

If *fd* refers to a `/proc/`*`pid`*`/ns/` link, `setns()` reassociates the calling thread with the namespace associated with that link, and *nstype* may be set to one of the *CLONE_NEW* constants* to impose constraints on the operation (`0` means no constraints).

Since Linux 5.8, *fd* may refer to a PID file descriptor obtained from *`pidfd_open()`*. In this case, `setns()` reassociates the calling thread into one or more of the same namespaces as the thread referred to by *fd*. This is subject to any constraints imposed by *nstype*, which is a bit mask combining one or more of the *CLONE_NEW* constants*, e.g. `setns(fd, os.CLONE_NEWUTS | os.CLONE_NEWPID)`. The caller's memberships in unspecified namespaces are left unchanged.

fd can be any object with a *`fileno()`* method, or a raw file descriptor.

This example reassociates the thread with the `init` process's network namespace:

```
fd = os.open("/proc/1/ns/net", os.O_RDONLY)
os.setns(fd, os.CLONE_NEWNET)
os.close(fd)
```

Availability: Linux >= 3.0 with glibc >= 2.14.

Added in version 3.12.

> **See also**
>
> The *unshare()* function.

os.**setpgrp**()

Call the system call `setpgrp()` or `setpgrp(0, 0)` depending on which version is implemented (if any). See the Unix manual for the semantics.

Availability: Unix, not WASI.

os.**setpgid**(*pid*, *pgrp*, */*)

Call the system call `setpgid()` to set the process group id of the process with id *pid* to the process group with id *pgrp*. See the Unix manual for the semantics.

Availability: Unix, not WASI.

os.**setpriority**(*which*, *who*, *priority*)

Set program scheduling priority. The value *which* is one of *PRIO_PROCESS*, *PRIO_PGRP*, or *PRIO_USER*, and *who* is interpreted relative to *which* (a process identifier for *PRIO_PROCESS*, process group identifier for *PRIO_PGRP*, and a user ID for *PRIO_USER*). A zero value for *who* denotes (respectively) the calling process, the process group of the calling process, or the real user ID of the calling process. *priority* is a value in the range -20 to 19. The default priority is 0; lower priorities cause more favorable scheduling.

Availability: Unix, not WASI.

Added in version 3.3.

os.**setregid**(*rgid*, *egid*, */*)

Set the current process's real and effective group ids.

Availability: Unix, not WASI.

os.**setresgid**(*rgid*, *egid*, *sgid*, */*)

Set the current process's real, effective, and saved group ids.

Availability: Unix, not WASI.

Added in version 3.2.

os.**setresuid**(*ruid*, *euid*, *suid*, */*)

Set the current process's real, effective, and saved user ids.

Availability: Unix, not WASI.

Added in version 3.2.

os.**setreuid**(*ruid*, *euid*, */*)

Set the current process's real and effective user ids.

Availability: Unix, not WASI.

os.**getsid**(*pid*, */*)

Call the system call `getsid()`. See the Unix manual for the semantics.

Availability: Unix, not WASI.

os.**setsid**()

Call the system call `setsid()`. See the Unix manual for the semantics.

Availability: Unix, not WASI.

os.**setuid**(*uid*, /)

Set the current process's user id.

Availability: Unix, not WASI.

os.**strerror**(*code*, /)

Return the error message corresponding to the error code in *code*. On platforms where `strerror()` returns `NULL` when given an unknown error number, *`ValueError`* is raised.

os.**supports_bytes_environ**

`True` if the native OS type of the environment is bytes (eg. `False` on Windows).

Added in version 3.2.

os.**umask**(*mask*, /)

Set the current numeric umask and return the previous umask.

The function is a stub on WASI, see *WebAssembly platforms* for more information.

os.**uname**()

Returns information identifying the current operating system. The return value is an object with five attributes:

- `sysname` - operating system name
- `nodename` - name of machine on network (implementation-defined)
- `release` - operating system release
- `version` - operating system version
- `machine` - hardware identifier

For backwards compatibility, this object is also iterable, behaving like a five-tuple containing `sysname`, `nodename`, `release`, `version`, and `machine` in that order.

Some systems truncate `nodename` to 8 characters or to the leading component; a better way to get the hostname is *`socket.gethostname()`* or even `socket.gethostbyaddr(socket.gethostname())`.

On macOS, iOS and Android, this returns the *kernel* name and version (i.e., `'Darwin'` on macOS and iOS; `'Linux'` on Android). *`platform.uname()`* can be used to get the user-facing operating system name and version on iOS and Android.

Availability: Unix.

Changed in version 3.3: Return type changed from a tuple to a tuple-like object with named attributes.

os.**unsetenv**(*key*, /)

Unset (delete) the environment variable named *key*. Such changes to the environment affect subprocesses started with *`os.system()`*, *`popen()`* or *`fork()`* and *`execv()`*.

Deletion of items in *`os.environ`* is automatically translated into a corresponding call to *`unsetenv()`*; however, calls to *`unsetenv()`* don't update *`os.environ`*, so it is actually preferable to delete items of *`os.environ`*.

Raises an *auditing event* `os.unsetenv` with argument `key`.

Changed in version 3.9: The function is now always available and is also available on Windows.

os.**unshare**(*flags*)

Disassociate parts of the process execution context, and move them into a newly created namespace. See the *`unshare(2)`* man page for more details. The *flags* argument is a bit mask, combining zero or more of the *CLONE_* constants*, that specifies which parts of the execution context should be unshared from their existing

associations and moved to a new namespace. If the *flags* argument is `0`, no changes are made to the calling process's execution context.

Availability: Linux >= 2.6.16.

Added in version 3.12.

See also

The `setns()` function.

Flags to the `unshare()` function, if the implementation supports them. See `unshare(2)` in the Linux manual for their exact effect and availability.

os.CLONE_FILES

os.CLONE_FS

os.CLONE_NEWCGROUP

os.CLONE_NEWIPC

os.CLONE_NEWNET

os.CLONE_NEWNS

os.CLONE_NEWPID

os.CLONE_NEWTIME

os.CLONE_NEWUSER

os.CLONE_NEWUTS

os.CLONE_SIGHAND

os.CLONE_SYSVSEM

os.CLONE_THREAD

os.CLONE_VM

16.1.4 File Object Creation

These functions create new *file objects*. (See also `open()` for opening file descriptors.)

os.fdopen(*fd*, **args*, ***kwargs*)

Return an open file object connected to the file descriptor *fd*. This is an alias of the `open()` built-in function and accepts the same arguments. The only difference is that the first argument of `fdopen()` must always be an integer.

16.1.5 File Descriptor Operations

These functions operate on I/O streams referenced using file descriptors.

File descriptors are small integers corresponding to a file that has been opened by the current process. For example, standard input is usually file descriptor 0, standard output is 1, and standard error is 2. Further files opened by a process will then be assigned 3, 4, 5, and so forth. The name "file descriptor" is slightly deceptive; on Unix platforms, sockets and pipes are also referenced by file descriptors.

The `fileno()` method can be used to obtain the file descriptor associated with a *file object* when required. Note that using the file descriptor directly will bypass the file object methods, ignoring aspects such as internal buffering of data.

os.**close**(*fd*)

Close file descriptor *fd*.

> **Note**
>
> This function is intended for low-level I/O and must be applied to a file descriptor as returned by *os.open()* or *pipe()*. To close a "file object" returned by the built-in function *open()* or by *popen()* or *fdopen()*, use its *close()* method.

os.**closerange**(*fd_low*, *fd_high*, */*)

Close all file descriptors from *fd_low* (inclusive) to *fd_high* (exclusive), ignoring errors. Equivalent to (but much faster than):

```
for fd in range(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

os.**copy_file_range**(*src*, *dst*, *count*, *offset_src=None*, *offset_dst=None*)

Copy *count* bytes from file descriptor *src*, starting from offset *offset_src*, to file descriptor *dst*, starting from offset *offset_dst*. If *offset_src* is `None`, then *src* is read from the current position; respectively for *offset_dst*.

In Linux kernel older than 5.3, the files pointed to by *src* and *dst* must reside in the same filesystem, otherwise an *OSError* is raised with *errno* set to *errno.EXDEV*.

This copy is done without the additional cost of transferring data from the kernel to user space and then back into the kernel. Additionally, some filesystems could implement extra optimizations, such as the use of reflinks (i.e., two or more inodes that share pointers to the same copy-on-write disk blocks; supported file systems include btrfs and XFS) and server-side copy (in the case of NFS).

The function copies bytes between two file descriptors. Text options, like the encoding and the line ending, are ignored.

The return value is the amount of bytes copied. This could be less than the amount requested.

> **Note**
>
> On Linux, *os.copy_file_range()* should not be used for copying a range of a pseudo file from a special filesystem like procfs and sysfs. It will always copy no bytes and return 0 as if the file was empty because of a known Linux kernel issue.

Availability: Linux >= 4.5 with glibc >= 2.27.

Added in version 3.8.

os.**device_encoding**(*fd*)

Return a string describing the encoding of the device associated with *fd* if it is connected to a terminal; else return *None*.

On Unix, if the *Python UTF-8 Mode* is enabled, return `'UTF-8'` rather than the device encoding.

Changed in version 3.10: On Unix, the function now implements the Python UTF-8 Mode.

os.**dup**(*fd*, /)

Return a duplicate of file descriptor *fd*. The new file descriptor is *non-inheritable*.

On Windows, when duplicating a standard stream (0: stdin, 1: stdout, 2: stderr), the new file descriptor is *inheritable*.

Availability: not WASI.

Changed in version 3.4: The new file descriptor is now non-inheritable.

os.**dup2**(*fd*, *fd2*, *inheritable=True*)

Duplicate file descriptor *fd* to *fd2*, closing the latter first if necessary. Return *fd2*. The new file descriptor is *inheritable* by default or non-inheritable if *inheritable* is `False`.

Availability: not WASI.

Changed in version 3.4: Add the optional *inheritable* parameter.

Changed in version 3.7: Return *fd2* on success. Previously, `None` was always returned.

os.**fchmod**(*fd*, *mode*)

Change the mode of the file given by *fd* to the numeric *mode*. See the docs for *chmod()* for possible values of *mode*. As of Python 3.3, this is equivalent to `os.chmod(fd, mode)`.

Raises an *auditing event* `os.chmod` with arguments `path`, `mode`, `dir_fd`.

Availability: Unix, Windows.

The function is limited on WASI, see *WebAssembly platforms* for more information.

Changed in version 3.13: Added support on Windows.

os.**fchown**(*fd*, *uid*, *gid*)

Change the owner and group id of the file given by *fd* to the numeric *uid* and *gid*. To leave one of the ids unchanged, set it to -1. See *chown()*. As of Python 3.3, this is equivalent to `os.chown(fd, uid, gid)`.

Raises an *auditing event* `os.chown` with arguments `path`, `uid`, `gid`, `dir_fd`.

Availability: Unix.

The function is limited on WASI, see *WebAssembly platforms* for more information.

os.**fdatasync**(*fd*)

Force write of file with filedescriptor *fd* to disk. Does not force update of metadata.

Availability: Unix.

> **Note**
>
> This function is not available on MacOS.

os.**fpathconf**(*fd*, *name*, /)

Return system configuration information relevant to an open file. *name* specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the `pathconf_names` dictionary. For configuration variables not included in that mapping, passing an integer for *name* is also accepted.

If *name* is a string and is not known, *ValueError* is raised. If a specific value for *name* is not supported by the host system, even if it is included in `pathconf_names`, an *OSError* is raised with *errno.EINVAL* for the error number.

As of Python 3.3, this is equivalent to `os.pathconf(fd, name)`.

Availability: Unix.

`os.`**`fstat`**`(`*fd*`)`

Get the status of the file descriptor *fd*. Return a *stat_result* object.

As of Python 3.3, this is equivalent to `os.stat(fd)`.

> **See also**
>
> The *stat()* function.

`os.`**`fstatvfs`**`(`*fd*`, /)`

Return information about the filesystem containing the file associated with file descriptor *fd*, like *statvfs()*. As of Python 3.3, this is equivalent to `os.statvfs(fd)`.

Availability: Unix.

`os.`**`fsync`**`(`*fd*`)`

Force write of file with filedescriptor *fd* to disk. On Unix, this calls the native `fsync()` function; on Windows, the MS `_commit()` function.

If you're starting with a buffered Python *file object* *f*, first do `f.flush()`, and then do `os.fsync(f.fileno())`, to ensure that all internal buffers associated with *f* are written to disk.

Availability: Unix, Windows.

`os.`**`ftruncate`**`(`*fd*`, `*length*`, /)`

Truncate the file corresponding to file descriptor *fd*, so that it is at most *length* bytes in size. As of Python 3.3, this is equivalent to `os.truncate(fd, length)`.

Raises an *auditing event* `os.truncate` with arguments `fd`, `length`.

Availability: Unix, Windows.

Changed in version 3.5: Added support for Windows

`os.`**`get_blocking`**`(`*fd*`, /)`

Get the blocking mode of the file descriptor: `False` if the *O_NONBLOCK* flag is set, `True` if the flag is cleared.

See also *set_blocking()* and *socket.socket.setblocking()*.

Availability: Unix, Windows.

The function is limited on WASI, see *WebAssembly platforms* for more information.

On Windows, this function is limited to pipes.

Added in version 3.5.

Changed in version 3.12: Added support for pipes on Windows.

`os.`**`grantpt`**`(`*fd*`, /)`

Grant access to the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the C standard library function `grantpt()`.

Availability: Unix, not WASI.

Added in version 3.13.

os.**isatty**(*fd*, /)

Return `True` if the file descriptor *fd* is open and connected to a tty(-like) device, else `False`.

os.**lockf**(*fd*, *cmd*, *len*, /)

Apply, test or remove a POSIX lock on an open file descriptor. *fd* is an open file descriptor. *cmd* specifies the command to use - one of *F_LOCK*, *F_TLOCK*, *F_ULOCK* or *F_TEST*. *len* specifies the section of the file to lock.

Raises an *auditing event* `os.lockf` with arguments `fd`, `cmd`, `len`.

Availability: Unix.

Added in version 3.3.

os.**F_LOCK**

os.**F_TLOCK**

os.**F_ULOCK**

os.**F_TEST**

Flags that specify what action *lockf()* will take.

Availability: Unix.

Added in version 3.3.

os.**login_tty**(*fd*, /)

Prepare the tty of which fd is a file descriptor for a new login session. Make the calling process a session leader; make the tty the controlling tty, the stdin, the stdout, and the stderr of the calling process; close fd.

Availability: Unix, not WASI.

Added in version 3.11.

os.**lseek**(*fd*, *pos*, *whence*, /)

Set the current position of file descriptor *fd* to position *pos*, modified by *whence*, and return the new position in bytes relative to the start of the file. Valid values for *whence* are:

- *SEEK_SET* or `0` – set *pos* relative to the beginning of the file
- *SEEK_CUR* or `1` – set *pos* relative to the current file position
- *SEEK_END* or `2` – set *pos* relative to the end of the file
- *SEEK_HOLE* – set *pos* to the next data location, relative to *pos*
- *SEEK_DATA* – set *pos* to the next data hole, relative to *pos*

Changed in version 3.3: Add support for `SEEK_HOLE` and `SEEK_DATA`.

os.**SEEK_SET**

os.**SEEK_CUR**

os.**SEEK_END**

Parameters to the *lseek()* function and the *seek()* method on *file-like objects*, for whence to adjust the file position indicator.

SEEK_SET
Adjust the file position relative to the beginning of the file.

SEEK_CUR
Adjust the file position relative to the current file position.

SEEK_END
Adjust the file position relative to the end of the file.

Their values are 0, 1, and 2, respectively.

os.**SEEK_HOLE**

os.**SEEK_DATA**

Parameters to the *lseek()* function and the *seek()* method on *file-like objects*, for seeking file data and holes on sparsely allocated files.

SEEK_DATA
Adjust the file offset to the next location containing data, relative to the seek position.

SEEK_HOLE
Adjust the file offset to the next location containing a hole, relative to the seek position. A hole is defined as a sequence of zeros.

> **Note**
>
> These operations only make sense for filesystems that support them.

Availability: Linux >= 3.1, macOS, Unix

Added in version 3.3.

os.**open**(*path*, *flags*, *mode=0o777*, ***, *dir_fd=None*)

Open the file *path* and set various flags according to *flags* and possibly its mode according to *mode*. When computing *mode*, the current umask value is first masked out. Return the file descriptor for the newly opened file. The new file descriptor is *non-inheritable*.

For a description of the flag and mode values, see the C run-time documentation; flag constants (like *O_RDONLY* and *O_WRONLY*) are defined in the *os* module. In particular, on Windows adding *O_BINARY* is needed to open files in binary mode.

This function can support *paths relative to directory descriptors* with the *dir_fd* parameter.

Raises an *auditing event* `open` with arguments `path`, `mode`, `flags`.

Changed in version 3.4: The new file descriptor is now non-inheritable.

> **Note**
>
> This function is intended for low-level I/O. For normal usage, use the built-in function *open()*, which returns a *file object* with `read()` and `write()` methods (and many more). To wrap a file descriptor in a file object, use *fdopen()*.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

Changed in version 3.6: Accepts a *path-like object*.

The following constants are options for the *flags* parameter to the *open()* function. They can be combined using the bitwise OR operator `|`. Some of them are not available on all platforms. For descriptions of their availability and use, consult the *open(2)* manual page on Unix or the MSDN on Windows.

os.**O_RDONLY**

os.**O_WRONLY**

os.**O_RDWR**

os.**O_APPEND**

os.**O_CREAT**

os.**O_EXCL**

os.**O_TRUNC**

The above constants are available on Unix and Windows.

os.**O_DSYNC**

os.**O_RSYNC**

os.**O_SYNC**

os.**O_NDELAY**

os.**O_NONBLOCK**

os.**O_NOCTTY**

os.**O_CLOEXEC**

The above constants are only available on Unix.

Changed in version 3.3: Add *O_CLOEXEC* constant.

os.**O_BINARY**

os.**O_NOINHERIT**

os.**O_SHORT_LIVED**

os.**O_TEMPORARY**

os.**O_RANDOM**

os.**O_SEQUENTIAL**

os.**O_TEXT**

The above constants are only available on Windows.

os.**O_EVTONLY**

os.**O_FSYNC**

os.**O_SYMLINK**

os.**O_NOFOLLOW_ANY**

The above constants are only available on macOS.

Changed in version 3.10: Add *O_EVTONLY*, *O_FSYNC*, *O_SYMLINK* and *O_NOFOLLOW_ANY* constants.

os.**O_ASYNC**

os.**O_DIRECT**

os.**O_DIRECTORY**

os.**O_NOFOLLOW**

os.**O_NOATIME**

os.**O_PATH**

os.**O_TMPFILE**

os.**O_SHLOCK**

os.**O_EXLOCK**

The above constants are extensions and not present if they are not defined by the C library.

Changed in version 3.4: Add *O_PATH* on systems that support it. Add *O_TMPFILE*, only available on Linux Kernel 3.11 or newer.

os.**openpty**()

Open a new pseudo-terminal pair. Return a pair of file descriptors `(master, slave)` for the pty and the tty, respectively. The new file descriptors are *non-inheritable*. For a (slightly) more portable approach, use the *pty* module.

Availability: Unix, not WASI.

Changed in version 3.4: The new file descriptors are now non-inheritable.

os.**pipe**()

Create a pipe. Return a pair of file descriptors `(r, w)` usable for reading and writing, respectively. The new file descriptor is *non-inheritable*.

Availability: Unix, Windows.

Changed in version 3.4: The new file descriptors are now non-inheritable.

os.**pipe2**(*flags*, */*)

Create a pipe with *flags* set atomically. *flags* can be constructed by ORing together one or more of these values: *O_NONBLOCK*, *O_CLOEXEC*. Return a pair of file descriptors `(r, w)` usable for reading and writing, respectively.

Availability: Unix, not WASI.

Added in version 3.3.

os.**posix_fallocate**(*fd*, *offset*, *len*, */*)

Ensures that enough disk space is allocated for the file specified by *fd* starting from *offset* and continuing for *len* bytes.

Availability: Unix.

Added in version 3.3.

os.**posix_fadvise**(*fd*, *offset*, *len*, *advice*, */*)

Announces an intention to access data in a specific pattern thus allowing the kernel to make optimizations. The advice applies to the region of the file specified by *fd* starting at *offset* and continuing for *len* bytes. *advice* is one of *POSIX_FADV_NORMAL*, *POSIX_FADV_SEQUENTIAL*, *POSIX_FADV_RANDOM*, *POSIX_FADV_NOREUSE*, *POSIX_FADV_WILLNEED* or *POSIX_FADV_DONTNEED*.

Availability: Unix.

Added in version 3.3.

os.**POSIX_FADV_NORMAL**

os.**POSIX_FADV_SEQUENTIAL**

os.**POSIX_FADV_RANDOM**

os.**POSIX_FADV_NOREUSE**

os.**POSIX_FADV_WILLNEED**

os.**POSIX_FADV_DONTNEED**

Flags that can be used in *advice* in *posix_fadvise()* that specify the access pattern that is likely to be used.

Availability: Unix.

Added in version 3.3.

os.**pread**(*fd*, *n*, *offset*, */*)

Read at most *n* bytes from file descriptor *fd* at a position of *offset*, leaving the file offset unchanged.

Return a bytestring containing the bytes read. If the end of the file referred to by *fd* has been reached, an empty bytes object is returned.

Availability: Unix.

Added in version 3.3.

os.**posix_openpt**(*oflag*, /)

Open and return a file descriptor for a master pseudo-terminal device.

Calls the C standard library function `posix_openpt()`. The *oflag* argument is used to set file status flags and file access modes as specified in the manual page of `posix_openpt()` of your system.

The returned file descriptor is *non-inheritable*. If the value *O_CLOEXEC* is available on the system, it is added to *oflag*.

Availability: Unix, not WASI.

Added in version 3.13.

os.**preadv**(*fd*, *buffers*, *offset*, *flags=0*, /)

Read from a file descriptor *fd* at a position of *offset* into mutable *bytes-like objects* *buffers*, leaving the file offset unchanged. Transfer data into each buffer until it is full and then move on to the next buffer in the sequence to hold the rest of the data.

The flags argument contains a bitwise OR of zero or more of the following flags:

- *RWF_HIPRI*
- *RWF_NOWAIT*

Return the total number of bytes actually read which can be less than the total capacity of all the objects.

The operating system may set a limit (*sysconf()* value `'SC_IOV_MAX'`) on the number of buffers that can be used.

Combine the functionality of *os.readv()* and *os.pread()*.

Availability: Linux >= 2.6.30, FreeBSD >= 6.0, OpenBSD >= 2.7, AIX >= 7.1.

Using flags requires Linux >= 4.6.

Added in version 3.7.

os.**RWF_NOWAIT**

Do not wait for data which is not immediately available. If this flag is specified, the system call will return instantly if it would have to read data from the backing storage or wait for a lock.

If some data was successfully read, it will return the number of bytes read. If no bytes were read, it will return `-1` and set errno to *errno.EAGAIN*.

Availability: Linux >= 4.14.

Added in version 3.7.

os.**RWF_HIPRI**

High priority read/write. Allows block-based filesystems to use polling of the device, which provides lower latency, but may use additional resources.

Currently, on Linux, this feature is usable only on a file descriptor opened using the *O_DIRECT* flag.

Availability: Linux >= 4.6.

Added in version 3.7.

os.**ptsname**(*fd*, */*)

Return the name of the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the reentrant C standard library function `ptsname_r()` if it is available; otherwise, the C standard library function `ptsname()`, which is not guaranteed to be thread-safe, is called.

Availability: Unix, not WASI.

Added in version 3.13.

os.**pwrite**(*fd*, *str*, *offset*, */*)

Write the bytestring in *str* to file descriptor *fd* at position of *offset*, leaving the file offset unchanged.

Return the number of bytes actually written.

Availability: Unix.

Added in version 3.3.

os.**pwritev**(*fd*, *buffers*, *offset*, *flags=0*, */*)

Write the *buffers* contents to file descriptor *fd* at an offset *offset*, leaving the file offset unchanged. *buffers* must be a sequence of *bytes-like objects*. Buffers are processed in array order. Entire contents of the first buffer is written before proceeding to the second, and so on.

The flags argument contains a bitwise OR of zero or more of the following flags:

- *RWF_DSYNC*
- *RWF_SYNC*
- *RWF_APPEND*

Return the total number of bytes actually written.

The operating system may set a limit (*sysconf()* value `'SC_IOV_MAX'`) on the number of buffers that can be used.

Combine the functionality of *os.writev()* and *os.pwrite()*.

Availability: Linux >= 2.6.30, FreeBSD >= 6.0, OpenBSD >= 2.7, AIX >= 7.1.

Using flags requires Linux >= 4.6.

Added in version 3.7.

os.**RWF_DSYNC**

Provide a per-write equivalent of the *O_DSYNC* *os.open()* flag. This flag effect applies only to the data range written by the system call.

Availability: Linux >= 4.7.

Added in version 3.7.

os.**RWF_SYNC**

Provide a per-write equivalent of the *O_SYNC* *os.open()* flag. This flag effect applies only to the data range written by the system call.

Availability: Linux >= 4.7.

Added in version 3.7.

os.**RWF_APPEND**

Provide a per-write equivalent of the *O_APPEND* `os.open()` flag. This flag is meaningful only for `os.pwritev()`, and its effect applies only to the data range written by the system call. The *offset* argument does not affect the write operation; the data is always appended to the end of the file. However, if the *offset* argument is `-1`, the current file *offset* is updated.

Availability: Linux >= 4.16.

Added in version 3.10.

os.**read**(*fd*, *n*, */*)

Read at most *n* bytes from file descriptor *fd*.

Return a bytestring containing the bytes read. If the end of the file referred to by *fd* has been reached, an empty bytes object is returned.

> **Note**
>
> This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `pipe()`. To read a "file object" returned by the built-in function `open()` or by `popen()` or `fdopen()`, or `sys.stdin`, use its `read()` or `readline()` methods.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** for the rationale).

os.**sendfile**(*out_fd*, *in_fd*, *offset*, *count*)

os.**sendfile**(*out_fd*, *in_fd*, *offset*, *count*, *headers=()*, *trailers=()*, *flags=0*)

Copy *count* bytes from file descriptor *in_fd* to file descriptor *out_fd* starting at *offset*. Return the number of bytes sent. When EOF is reached return `0`.

The first function notation is supported by all platforms that define `sendfile()`.

On Linux, if *offset* is given as `None`, the bytes are read from the current position of *in_fd* and the position of *in_fd* is updated.

The second case may be used on macOS and FreeBSD where *headers* and *trailers* are arbitrary sequences of buffers that are written before and after the data from *in_fd* is written. It returns the same as the first case.

On macOS and FreeBSD, a value of `0` for *count* specifies to send until the end of *in_fd* is reached.

All platforms support sockets as *out_fd* file descriptor, and some platforms allow other types (e.g. regular file, pipe) as well.

Cross-platform applications should not use *headers*, *trailers* and *flags* arguments.

Availability: Unix, not WASI.

> **Note**
>
> For a higher-level wrapper of `sendfile()`, see `socket.socket.sendfile()`.

Added in version 3.3.

Changed in version 3.9: Parameters *out* and *in* was renamed to *out_fd* and *in_fd*.

os.**SF_NODISKIO**

os.**SF_MNOWAIT**

os.**SF_SYNC**

Parameters to the *sendfile()* function, if the implementation supports them.

Availability: Unix, not WASI.

Added in version 3.3.

os.**SF_NOCACHE**

Parameter to the *sendfile()* function, if the implementation supports it. The data won't be cached in the virtual memory and will be freed afterwards.

Availability: Unix, not WASI.

Added in version 3.11.

os.**set_blocking**(*fd*, *blocking*, */*)

Set the blocking mode of the specified file descriptor. Set the *O_NONBLOCK* flag if blocking is `False`, clear the flag otherwise.

See also *get_blocking()* and *socket.socket.setblocking()*.

Availability: Unix, Windows.

The function is limited on WASI, see *WebAssembly platforms* for more information.

On Windows, this function is limited to pipes.

Added in version 3.5.

Changed in version 3.12: Added support for pipes on Windows.

os.**splice**(*src*, *dst*, *count*, *offset_src=None*, *offset_dst=None*)

Transfer *count* bytes from file descriptor *src*, starting from offset *offset_src*, to file descriptor *dst*, starting from offset *offset_dst*. At least one of the file descriptors must refer to a pipe. If *offset_src* is `None`, then *src* is read from the current position; respectively for *offset_dst*. The offset associated to the file descriptor that refers to a pipe must be `None`. The files pointed to by *src* and *dst* must reside in the same filesystem, otherwise an *OSError* is raised with *errno* set to *errno.EXDEV*.

This copy is done without the additional cost of transferring data from the kernel to user space and then back into the kernel. Additionally, some filesystems could implement extra optimizations. The copy is done as if both files are opened as binary.

Upon successful completion, returns the number of bytes spliced to or from the pipe. A return value of 0 means end of input. If *src* refers to a pipe, then this means that there was no data to transfer, and it would not make sense to block because there are no writers connected to the write end of the pipe.

Availability: Linux >= 2.6.17 with glibc >= 2.5

Added in version 3.10.

os.**SPLICE_F_MOVE**

os.**SPLICE_F_NONBLOCK**

os.**SPLICE_F_MORE**

Added in version 3.10.

os.**readv**(*fd*, *buffers*, */*)

Read from a file descriptor *fd* into a number of mutable *bytes-like objects* *buffers*. Transfer data into each buffer until it is full and then move on to the next buffer in the sequence to hold the rest of the data.

Return the total number of bytes actually read which can be less than the total capacity of all the objects.

The operating system may set a limit (*sysconf()* value `'SC_IOV_MAX'`) on the number of buffers that can be used.

Availability: Unix.

Added in version 3.3.

os.**tcgetpgrp**(*fd*, /)

Return the process group associated with the terminal given by *fd* (an open file descriptor as returned by `os.open()`).

Availability: Unix, not WASI.

os.**tcsetpgrp**(*fd*, *pg*, /)

Set the process group associated with the terminal given by *fd* (an open file descriptor as returned by `os.open()`) to *pg*.

Availability: Unix, not WASI.

os.**ttyname**(*fd*, /)

Return a string which specifies the terminal device associated with file descriptor *fd*. If *fd* is not associated with a terminal device, an exception is raised.

Availability: Unix.

os.**unlockpt**(*fd*, /)

Unlock the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the C standard library function `unlockpt()`.

Availability: Unix, not WASI.

Added in version 3.13.

os.**write**(*fd*, *str*, /)

Write the bytestring in *str* to file descriptor *fd*.

Return the number of bytes actually written.

> **Note**
>
> This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `pipe()`. To write a "file object" returned by the built-in function `open()` or by `popen()` or `fdopen()`, or `sys.stdout` or `sys.stderr`, use its `write()` method.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** for the rationale).

os.**writev**(*fd*, *buffers*, /)

Write the contents of *buffers* to file descriptor *fd*. *buffers* must be a sequence of *bytes-like objects*. Buffers are processed in array order. Entire contents of the first buffer is written before proceeding to the second, and so on.

Returns the total number of bytes actually written.

The operating system may set a limit (`sysconf()` value `'SC_IOV_MAX'`) on the number of buffers that can be used.

Availability: Unix.

Added in version 3.3.

Querying the size of a terminal

Added in version 3.3.

os.**get_terminal_size**(*fd=STDOUT_FILENO*, /)

Return the size of the terminal window as `(columns, lines)`, tuple of type *terminal_size*.

The optional argument `fd` (default `STDOUT_FILENO`, or standard output) specifies which file descriptor should be queried.

If the file descriptor is not connected to a terminal, an *OSError* is raised.

shutil.get_terminal_size() is the high-level function which should normally be used, `os.get_terminal_size` is the low-level implementation.

Availability: Unix, Windows.

class os.**terminal_size**

A subclass of tuple, holding `(columns, lines)` of the terminal window size.

columns

Width of the terminal window in characters.

lines

Height of the terminal window in characters.

Inheritance of File Descriptors

Added in version 3.4.

A file descriptor has an "inheritable" flag which indicates if the file descriptor can be inherited by child processes. Since Python 3.4, file descriptors created by Python are non-inheritable by default.

On UNIX, non-inheritable file descriptors are closed in child processes at the execution of a new program, other file descriptors are inherited.

On Windows, non-inheritable handles and file descriptors are closed in child processes, except for standard streams (file descriptors 0, 1 and 2: stdin, stdout and stderr), which are always inherited. Using *spawn** functions, all inheritable handles and all inheritable file descriptors are inherited. Using the *subprocess* module, all file descriptors except standard streams are closed, and inheritable handles are only inherited if the *close_fds* parameter is `False`.

On WebAssembly platforms, the file descriptor cannot be modified.

os.**get_inheritable**(*fd*, /)

Get the "inheritable" flag of the specified file descriptor (a boolean).

os.**set_inheritable**(*fd*, *inheritable*, /)

Set the "inheritable" flag of the specified file descriptor.

os.**get_handle_inheritable**(*handle*, /)

Get the "inheritable" flag of the specified handle (a boolean).

Availability: Windows.

os.**set_handle_inheritable**(*handle*, *inheritable*, /)

Set the "inheritable" flag of the specified handle.

Availability: Windows.

16.1.6 Files and Directories

On some Unix platforms, many of these functions support one or more of these features:

- **specifying a file descriptor:** Normally the *path* argument provided to functions in the `os` module must be a string specifying a file path. However, some functions now alternatively accept an open file descriptor for their *path* argument. The function will then operate on the file referred to by the descriptor. (For POSIX systems, Python will call the variant of the function prefixed with `f` (e.g. call `fchdir` instead of `chdir`).)

  You can check whether or not *path* can be specified as a file descriptor for a particular function on your platform using `os.supports_fd`. If this functionality is unavailable, using it will raise a `NotImplementedError`.

  If the function also supports *dir_fd* or *follow_symlinks* arguments, it's an error to specify one of those when supplying *path* as a file descriptor.

- **paths relative to directory descriptors:** If *dir_fd* is not `None`, it should be a file descriptor referring to a directory, and the path to operate on should be relative; path will then be relative to that directory. If the path is absolute, *dir_fd* is ignored. (For POSIX systems, Python will call the variant of the function with an `at` suffix and possibly prefixed with `f` (e.g. call `faccessat` instead of `access`).

  You can check whether or not *dir_fd* is supported for a particular function on your platform using `os.supports_dir_fd`. If it's unavailable, using it will raise a `NotImplementedError`.

- **not following symlinks:** If *follow_symlinks* is `False`, and the last element of the path to operate on is a symbolic link, the function will operate on the symbolic link itself rather than the file pointed to by the link. (For POSIX systems, Python will call the `l...` variant of the function.)

  You can check whether or not *follow_symlinks* is supported for a particular function on your platform using `os.supports_follow_symlinks`. If it's unavailable, using it will raise a `NotImplementedError`.

`os.`**`access`**(*path*, *mode*, *, *dir_fd=None*, *effective_ids=False*, *follow_symlinks=True*)

Use the real uid/gid to test for access to *path*. Note that most operations will use the effective uid/gid, therefore this routine can be used in a suid/sgid environment to test if the invoking user has the specified access to *path*. *mode* should be `F_OK` to test the existence of *path*, or it can be the inclusive OR of one or more of `R_OK`, `W_OK`, and `X_OK` to test permissions. Return `True` if access is allowed, `False` if not. See the Unix man page `access(2)` for more information.

This function can support specifying *paths relative to directory descriptors* and *not following symlinks*.

If *effective_ids* is `True`, `access()` will perform its access checks using the effective uid/gid instead of the real uid/gid. *effective_ids* may not be supported on your platform; you can check whether or not it is available using `os.supports_effective_ids`. If it is unavailable, using it will raise a `NotImplementedError`.

> **Note**
>
> Using `access()` to check if a user is authorized to e.g. open a file before actually doing so using `open()` creates a security hole, because the user might exploit the short time interval between checking and opening the file to manipulate it. It's preferable to use *EAFP* techniques. For example:
>
> ```
> if os.access("myfile", os.R_OK):
>     with open("myfile") as fp:
>         return fp.read()
> return "some default data"
> ```
>
> is better written as:
>
> ```
> try:
>     fp = open("myfile")
> except PermissionError:
>     return "some default data"
> else:
>     with fp:
>         return fp.read()
> ```

```
```

Note

I/O operations may fail even when `access()` indicates that they would succeed, particularly for operations on network filesystems which may have permissions semantics beyond the usual POSIX permission-bit model.

Changed in version 3.3: Added the *dir_fd*, *effective_ids*, and *follow_symlinks* parameters.

Changed in version 3.6: Accepts a *path-like object*.

`os.F_OK`

`os.R_OK`

`os.W_OK`

`os.X_OK`

Values to pass as the *mode* parameter of `access()` to test the existence, readability, writability and executability of *path*, respectively.

`os.chdir`(*path*)

Change the current working directory to *path*.

This function can support *specifying a file descriptor*. The descriptor must refer to an opened directory, not an open file.

This function can raise `OSError` and subclasses such as `FileNotFoundError`, `PermissionError`, and `NotADirectoryError`.

Raises an *auditing event* `os.chdir` with argument `path`.

Changed in version 3.3: Added support for specifying *path* as a file descriptor on some platforms.

Changed in version 3.6: Accepts a *path-like object*.

`os.chflags`(*path*, *flags*, *, *follow_symlinks=True*)

Set the flags of *path* to the numeric *flags*. *flags* may take a combination (bitwise OR) of the following values (as defined in the `stat` module):

- `stat.UF_NODUMP`
- `stat.UF_IMMUTABLE`
- `stat.UF_APPEND`
- `stat.UF_OPAQUE`
- `stat.UF_NOUNLINK`
- `stat.UF_COMPRESSED`
- `stat.UF_HIDDEN`
- `stat.SF_ARCHIVED`
- `stat.SF_IMMUTABLE`
- `stat.SF_APPEND`
- `stat.SF_NOUNLINK`
- `stat.SF_SNAPSHOT`

This function can support *not following symlinks*.

Raises an *auditing event* `os.chflags` with arguments `path`, `flags`.

Availability: Unix, not WASI.

Changed in version 3.3: Added the *follow_symlinks* parameter.

Changed in version 3.6: Accepts a *path-like object*.

`os.`**`chmod`**(*path*, *mode*, *, *dir_fd=None*, *follow_symlinks=True*)

Change the mode of *path* to the numeric *mode*. *mode* may take one of the following values (as defined in the *stat* module) or bitwise ORed combinations of them:

- *stat.S_ISUID*
- *stat.S_ISGID*
- *stat.S_ENFMT*
- *stat.S_ISVTX*
- *stat.S_IREAD*
- *stat.S_IWRITE*
- *stat.S_IEXEC*
- *stat.S_IRWXU*
- *stat.S_IRUSR*
- *stat.S_IWUSR*
- *stat.S_IXUSR*
- *stat.S_IRWXG*
- *stat.S_IRGRP*
- *stat.S_IWGRP*
- *stat.S_IXGRP*
- *stat.S_IRWXO*
- *stat.S_IROTH*
- *stat.S_IWOTH*
- *stat.S_IXOTH*

This function can support *specifying a file descriptor*, *paths relative to directory descriptors* and *not following symlinks*.

Note

Although Windows supports *chmod()*, you can only set the file's read-only flag with it (via the `stat.S_IWRITE` and `stat.S_IREAD` constants or a corresponding integer value). All other bits are ignored. The default value of *follow_symlinks* is `False` on Windows.

The function is limited on WASI, see *WebAssembly platforms* for more information.

Raises an *auditing event* `os.chmod` with arguments `path`, `mode`, `dir_fd`.

Changed in version 3.3: Added support for specifying *path* as an open file descriptor, and the *dir_fd* and *follow_symlinks* arguments.

Changed in version 3.6: Accepts a *path-like object*.

Changed in version 3.13: Added support for a file descriptor and the *follow_symlinks* argument on Windows.

`os.chown`(*path*, *uid*, *gid*, ***, *dir_fd=None*, *follow_symlinks=True*)

Change the owner and group id of *path* to the numeric *uid* and *gid*. To leave one of the ids unchanged, set it to -1.

This function can support *specifying a file descriptor*, *paths relative to directory descriptors* and *not following symlinks*.

See `shutil.chown()` for a higher-level function that accepts names in addition to numeric ids.

Raises an *auditing event* `os.chown` with arguments `path`, `uid`, `gid`, `dir_fd`.

Availability: Unix.

The function is limited on WASI, see *WebAssembly platforms* for more information.

Changed in version 3.3: Added support for specifying *path* as an open file descriptor, and the *dir_fd* and *follow_symlinks* arguments.

Changed in version 3.6: Supports a *path-like object*.

`os.chroot`(*path*)

Change the root directory of the current process to *path*.

Availability: Unix, not WASI.

Changed in version 3.6: Accepts a *path-like object*.

`os.fchdir`(*fd*)

Change the current working directory to the directory represented by the file descriptor *fd*. The descriptor must refer to an opened directory, not an open file. As of Python 3.3, this is equivalent to `os.chdir(fd)`.

Raises an *auditing event* `os.chdir` with argument `path`.

Availability: Unix.

`os.getcwd`()

Return a string representing the current working directory.

`os.getcwdb`()

Return a bytestring representing the current working directory.

Changed in version 3.8: The function now uses the UTF-8 encoding on Windows, rather than the ANSI code page: see **PEP 529** for the rationale. The function is no longer deprecated on Windows.

`os.lchflags`(*path*, *flags*)

Set the flags of *path* to the numeric *flags*, like `chflags()`, but do not follow symbolic links. As of Python 3.3, this is equivalent to `os.chflags(path, flags, follow_symlinks=False)`.

Raises an *auditing event* `os.chflags` with arguments `path`, `flags`.

Availability: Unix, not WASI.

Changed in version 3.6: Accepts a *path-like object*.

`os.lchmod`(*path*, *mode*)

Change the mode of *path* to the numeric *mode*. If path is a symlink, this affects the symlink rather than the target. See the docs for `chmod()` for possible values of *mode*. As of Python 3.3, this is equivalent to `os.chmod(path, mode, follow_symlinks=False)`.

`lchmod()` is not part of POSIX, but Unix implementations may have it if changing the mode of symbolic links is supported.

Raises an *auditing event* `os.chmod` with arguments `path`, `mode`, `dir_fd`.

Availability: Unix, Windows, not Linux, FreeBSD >= 1.3, NetBSD >= 1.3, not OpenBSD

Changed in version 3.6: Accepts a *path-like object*.

Changed in version 3.13: Added support on Windows.

os.lchown(*path*, *uid*, *gid*)

Change the owner and group id of *path* to the numeric *uid* and *gid*. This function will not follow symbolic links. As of Python 3.3, this is equivalent to `os.chown(path, uid, gid, follow_symlinks=False)`.

Raises an *auditing event* `os.chown` with arguments `path`, `uid`, `gid`, `dir_fd`.

Availability: Unix.

Changed in version 3.6: Accepts a *path-like object*.

os.link(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*, *follow_symlinks=True*)

Create a hard link pointing to *src* named *dst*.

This function can support specifying *src_dir_fd* and/or *dst_dir_fd* to supply *paths relative to directory descriptors*, and *not following symlinks*.

Raises an *auditing event* `os.link` with arguments `src`, `dst`, `src_dir_fd`, `dst_dir_fd`.

Availability: Unix, Windows.

Changed in version 3.2: Added Windows support.

Changed in version 3.3: Added the *src_dir_fd*, *dst_dir_fd*, and *follow_symlinks* parameters.

Changed in version 3.6: Accepts a *path-like object* for *src* and *dst*.

os.listdir(*path='.'*)

Return a list containing the names of the entries in the directory given by *path*. The list is in arbitrary order, and does not include the special entries `'.'` and `'..'` even if they are present in the directory. If a file is removed from or added to the directory during the call of this function, whether a name for that file be included is unspecified.

path may be a *path-like object*. If *path* is of type `bytes` (directly or indirectly through the *PathLike* interface), the filenames returned will also be of type `bytes`; in all other circumstances, they will be of type `str`.

This function can also support *specifying a file descriptor*; the file descriptor must refer to a directory.

Raises an *auditing event* `os.listdir` with argument `path`.

Note

To encode `str` filenames to `bytes`, use *fsencode()*.

See also

The *scandir()* function returns directory entries along with file attribute information, giving better performance for many common use cases.

Changed in version 3.2: The *path* parameter became optional.

Changed in version 3.3: Added support for specifying *path* as an open file descriptor.

Changed in version 3.6: Accepts a *path-like object*.

os.**listdrives**()

Return a list containing the names of drives on a Windows system.

A drive name typically looks like `'C:\\'`. Not every drive name will be associated with a volume, and some may be inaccessible for a variety of reasons, including permissions, network connectivity or missing media. This function does not test for access.

May raise *OSError* if an error occurs collecting the drive names.

Raises an *auditing event* `os.listdrives` with no arguments.

Availability: Windows

Added in version 3.12.

os.**listmounts**(*volume*)

Return a list containing the mount points for a volume on a Windows system.

volume must be represented as a GUID path, like those returned by *os.listvolumes()*. Volumes may be mounted in multiple locations or not at all. In the latter case, the list will be empty. Mount points that are not associated with a volume will not be returned by this function.

The mount points return by this function will be absolute paths, and may be longer than the drive name.

Raises *OSError* if the volume is not recognized or if an error occurs collecting the paths.

Raises an *auditing event* `os.listmounts` with argument `volume`.

Availability: Windows

Added in version 3.12.

os.**listvolumes**()

Return a list containing the volumes in the system.

Volumes are typically represented as a GUID path that looks like `\\?\Volume{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}\`. Files can usually be accessed through a GUID path, permissions allowing. However, users are generally not familiar with them, and so the recommended use of this function is to retrieve mount points using *os.listmounts()*.

May raise *OSError* if an error occurs collecting the volumes.

Raises an *auditing event* `os.listvolumes` with no arguments.

Availability: Windows

Added in version 3.12.

os.**lstat**(*path*, *, *dir_fd=None*)

Perform the equivalent of an `lstat()` system call on the given path. Similar to *stat()*, but does not follow symbolic links. Return a *stat_result* object.

On platforms that do not support symbolic links, this is an alias for *stat()*.

As of Python 3.3, this is equivalent to `os.stat(path, dir_fd=dir_fd, follow_symlinks=False)`.

This function can also support *paths relative to directory descriptors*.

See also

The *stat()* function.

Changed in version 3.2: Added support for Windows 6.0 (Vista) symbolic links.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.6: Accepts a *path-like object*.

Changed in version 3.8: On Windows, now opens reparse points that represent another path (name surrogates), including symbolic links and directory junctions. Other kinds of reparse points are resolved by the operating system as for `stat()`.

`os.`**`mkdir`**(*path*, *mode=0o777*, ***, *dir_fd=None*)

Create a directory named *path* with numeric mode *mode*.

If the directory already exists, `FileExistsError` is raised. If a parent directory in the path does not exist, `FileNotFoundError` is raised.

On some systems, *mode* is ignored. Where it is used, the current umask value is first masked out. If bits other than the last 9 (i.e. the last 3 digits of the octal representation of the *mode*) are set, their meaning is platform-dependent. On some platforms, they are ignored and you should call `chmod()` explicitly to set them.

On Windows, a *mode* of `0o700` is specifically handled to apply access control to the new directory such that only the current user and administrators have access. Other values of *mode* are ignored.

This function can also support *paths relative to directory descriptors*.

It is also possible to create temporary directories; see the `tempfile` module's `tempfile.mkdtemp()` function.

Raises an *auditing event* `os.mkdir` with arguments `path`, `mode`, `dir_fd`.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.6: Accepts a *path-like object*.

Changed in version 3.13: Windows now handles a *mode* of `0o700`.

`os.`**`makedirs`**(*name*, *mode=0o777*, *exist_ok=False*)

Recursive directory creation function. Like `mkdir()`, but makes all intermediate-level directories needed to contain the leaf directory.

The *mode* parameter is passed to `mkdir()` for creating the leaf directory; see *the mkdir() description* for how it is interpreted. To set the file permission bits of any newly created parent directories you can set the umask before invoking `makedirs()`. The file permission bits of existing parent directories are not changed.

If *exist_ok* is `False` (the default), a `FileExistsError` is raised if the target directory already exists.

> **Note**
>
> `makedirs()` will become confused if the path elements to create include `pardir` (eg. ".." on UNIX systems).

This function handles UNC paths correctly.

Raises an *auditing event* `os.mkdir` with arguments `path`, `mode`, `dir_fd`.

Changed in version 3.2: Added the *exist_ok* parameter.

Changed in version 3.4.1: Before Python 3.4.1, if *exist_ok* was `True` and the directory existed, `makedirs()` would still raise an error if *mode* did not match the mode of the existing directory. Since this behavior was impossible to implement safely, it was removed in Python 3.4.1. See bpo-21082.

Changed in version 3.6: Accepts a *path-like object*.

Changed in version 3.7: The *mode* argument no longer affects the file permission bits of newly created intermediate-level directories.

os.**mkfifo**(*path*, *mode=0o666*, *, *dir_fd=None*)

Create a FIFO (a named pipe) named *path* with numeric mode *mode*. The current umask value is first masked out from the mode.

This function can also support *paths relative to directory descriptors*.

FIFOs are pipes that can be accessed like regular files. FIFOs exist until they are deleted (for example with `os.unlink()`). Generally, FIFOs are used as rendezvous between "client" and "server" type processes: the server opens the FIFO for reading, and the client opens it for writing. Note that `mkfifo()` doesn't open the FIFO — it just creates the rendezvous point.

Availability: Unix, not WASI.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.6: Accepts a *path-like object*.

os.**mknod**(*path*, *mode=0o600*, *device=0*, *, *dir_fd=None*)

Create a filesystem node (file, device special file or named pipe) named *path*. *mode* specifies both the permissions to use and the type of node to be created, being combined (bitwise OR) with one of `stat.S_IFREG`, `stat.S_IFCHR`, `stat.S_IFBLK`, and `stat.S_IFIFO` (those constants are available in `stat`). For `stat.S_IFCHR` and `stat.S_IFBLK`, *device* defines the newly created device special file (probably using `os.makedev()`), otherwise it is ignored.

This function can also support *paths relative to directory descriptors*.

Availability: Unix, not WASI.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.6: Accepts a *path-like object*.

os.**major**(*device*, /)

Extract the device major number from a raw device number (usually the `st_dev` or `st_rdev` field from `stat`).

os.**minor**(*device*, /)

Extract the device minor number from a raw device number (usually the `st_dev` or `st_rdev` field from `stat`).

os.**makedev**(*major*, *minor*, /)

Compose a raw device number from the major and minor device numbers.

os.**pathconf**(*path*, *name*)

Return system configuration information relevant to a named file. *name* specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the `pathconf_names` dictionary. For configuration variables not included in that mapping, passing an integer for *name* is also accepted.

If *name* is a string and is not known, `ValueError` is raised. If a specific value for *name* is not supported by the host system, even if it is included in `pathconf_names`, an `OSError` is raised with `errno.EINVAL` for the error number.

This function can support *specifying a file descriptor*.

Availability: Unix.

Changed in version 3.6: Accepts a *path-like object*.

os.**pathconf_names**

Dictionary mapping names accepted by *pathconf()* and *fpathconf()* to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

Availability: Unix.

os.**readlink**(*path*, *, *dir_fd=None*)

Return a string representing the path to which the symbolic link points. The result may be either an absolute or relative pathname; if it is relative, it may be converted to an absolute pathname using `os.path.join(os.path.dirname(path), result)`.

If the *path* is a string object (directly or indirectly through a *PathLike* interface), the result will also be a string object, and the call may raise a UnicodeDecodeError. If the *path* is a bytes object (direct or indirectly), the result will be a bytes object.

This function can also support *paths relative to directory descriptors*.

When trying to resolve a path that may contain links, use *realpath()* to properly handle recursion and platform differences.

Availability: Unix, Windows.

Changed in version 3.2: Added support for Windows 6.0 (Vista) symbolic links.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.6: Accepts a *path-like object* on Unix.

Changed in version 3.8: Accepts a *path-like object* and a bytes object on Windows.

Added support for directory junctions, and changed to return the substitution path (which typically includes `\\?\` prefix) rather than the optional "print name" field that was previously returned.

os.**remove**(*path*, *, *dir_fd=None*)

Remove (delete) the file *path*. If *path* is a directory, an *OSError* is raised. Use *rmdir()* to remove directories. If the file does not exist, a *FileNotFoundError* is raised.

This function can support *paths relative to directory descriptors*.

On Windows, attempting to remove a file that is in use causes an exception to be raised; on Unix, the directory entry is removed but the storage allocated to the file is not made available until the original file is no longer in use.

This function is semantically identical to *unlink()*.

Raises an *auditing event* `os.remove` with arguments `path`, `dir_fd`.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.6: Accepts a *path-like object*.

os.**removedirs**(*name*)

Remove directories recursively. Works like *rmdir()* except that, if the leaf directory is successfully removed, *removedirs()* tries to successively remove every parent directory mentioned in *path* until an error is raised (which is ignored, because it generally means that a parent directory is not empty). For example, `os.removedirs('foo/bar/baz')` will first remove the directory `'foo/bar/baz'`, and then remove `'foo/bar'` and `'foo'` if they are empty. Raises *OSError* if the leaf directory could not be successfully removed.

Raises an *auditing event* `os.remove` with arguments `path`, `dir_fd`.

Changed in version 3.6: Accepts a *path-like object*.

os.**rename**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)

Rename the file or directory *src* to *dst*. If *dst* exists, the operation will fail with an `OSError` subclass in a number of cases:

On Windows, if *dst* exists a `FileExistsError` is always raised. The operation may fail if *src* and *dst* are on different filesystems. Use `shutil.move()` to support moves to a different filesystem.

On Unix, if *src* is a file and *dst* is a directory or vice-versa, an `IsADirectoryError` or a `NotADirectoryError` will be raised respectively. If both are directories and *dst* is empty, *dst* will be silently replaced. If *dst* is a non-empty directory, an `OSError` is raised. If both are files, *dst* will be replaced silently if the user has permission. The operation may fail on some Unix flavors if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

This function can support specifying *src_dir_fd* and/or *dst_dir_fd* to supply *paths relative to directory descriptors*.

If you want cross-platform overwriting of the destination, use `replace()`.

Raises an *auditing event* `os.rename` with arguments `src`, `dst`, `src_dir_fd`, `dst_dir_fd`.

Changed in version 3.3: Added the *src_dir_fd* and *dst_dir_fd* parameters.

Changed in version 3.6: Accepts a *path-like object* for *src* and *dst*.

os.**renames**(*old*, *new*)

Recursive directory or file renaming function. Works like `rename()`, except creation of any intermediate directories needed to make the new pathname good is attempted first. After the rename, directories corresponding to rightmost path segments of the old name will be pruned away using `removedirs()`.

> **Note**
>
> This function can fail with the new directory structure made if you lack permissions needed to remove the leaf directory or file.

Raises an *auditing event* `os.rename` with arguments `src`, `dst`, `src_dir_fd`, `dst_dir_fd`.

Changed in version 3.6: Accepts a *path-like object* for *old* and *new*.

os.**replace**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)

Rename the file or directory *src* to *dst*. If *dst* is a non-empty directory, `OSError` will be raised. If *dst* exists and is a file, it will be replaced silently if the user has permission. The operation may fail if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

This function can support specifying *src_dir_fd* and/or *dst_dir_fd* to supply *paths relative to directory descriptors*.

Raises an *auditing event* `os.rename` with arguments `src`, `dst`, `src_dir_fd`, `dst_dir_fd`.

Added in version 3.3.

Changed in version 3.6: Accepts a *path-like object* for *src* and *dst*.

os.**rmdir**(*path*, *, *dir_fd=None*)

Remove (delete) the directory *path*. If the directory does not exist or is not empty, a `FileNotFoundError` or an `OSError` is raised respectively. In order to remove whole directory trees, `shutil.rmtree()` can be used.

This function can support *paths relative to directory descriptors*.

Raises an *auditing event* `os.rmdir` with arguments `path`, `dir_fd`.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.6: Accepts a *path-like object*.

os.**scandir**(*path='.'*)

Return an iterator of *os.DirEntry* objects corresponding to the entries in the directory given by *path*. The entries are yielded in arbitrary order, and the special entries `'.'` and `'..'` are not included. If a file is removed from or added to the directory after creating the iterator, whether an entry for that file be included is unspecified.

Using *scandir()* instead of *listdir()* can significantly increase the performance of code that also needs file type or file attribute information, because *os.DirEntry* objects expose this information if the operating system provides it when scanning a directory. All *os.DirEntry* methods may perform a system call, but *is_dir()* and *is_file()* usually only require a system call for symbolic links; *os.DirEntry.stat()* always requires a system call on Unix but only requires one for symbolic links on Windows.

path may be a *path-like object*. If *path* is of type `bytes` (directly or indirectly through the *PathLike* interface), the type of the *name* and *path* attributes of each *os.DirEntry* will be `bytes`; in all other circumstances, they will be of type `str`.

This function can also support *specifying a file descriptor*; the file descriptor must refer to a directory.

Raises an *auditing event* `os.scandir` with argument `path`.

The *scandir()* iterator supports the *context manager* protocol and has the following method:

scandir.**close**()

Close the iterator and free acquired resources.

This is called automatically when the iterator is exhausted or garbage collected, or when an error happens during iterating. However it is advisable to call it explicitly or use the `with` statement.

Added in version 3.6.

The following example shows a simple use of *scandir()* to display all the files (excluding directories) in the given *path* that don't start with `'.'`. The `entry.is_file()` call will generally not make an additional system call:

```
with os.scandir(path) as it:
    for entry in it:
        if not entry.name.startswith('.') and entry.is_file():
            print(entry.name)
```

Note

On Unix-based systems, *scandir()* uses the system's opendir() and readdir() functions. On Windows, it uses the Win32 FindFirstFileW and FindNextFileW functions.

Added in version 3.5.

Changed in version 3.6: Added support for the *context manager* protocol and the *close()* method. If a *scandir()* iterator is neither exhausted nor explicitly closed a *ResourceWarning* will be emitted in its destructor.

The function accepts a *path-like object*.

Changed in version 3.7: Added support for *file descriptors* on Unix.

class os.**DirEntry**

Object yielded by *scandir()* to expose the file path and other file attributes of a directory entry.

scandir() will provide as much of this information as possible without making additional system calls. When a `stat()` or `lstat()` system call is made, the `os.DirEntry` object will cache the result.

`os.DirEntry` instances are not intended to be stored in long-lived data structures; if you know the file metadata has changed or if a long time has elapsed since calling *scandir()*, call `os.stat(entry.path)` to fetch up-to-date information.

Because the `os.DirEntry` methods can make operating system calls, they may also raise *OSError*. If you need very fine-grained control over errors, you can catch *OSError* when calling one of the `os.DirEntry` methods and handle as appropriate.

To be directly usable as a *path-like object*, `os.DirEntry` implements the *PathLike* interface.

Attributes and methods on a `os.DirEntry` instance are as follows:

name

The entry's base filename, relative to the *scandir()* *path* argument.

The *name* attribute will be `bytes` if the *scandir()* *path* argument is of type `bytes` and `str` otherwise. Use *fsdecode()* to decode byte filenames.

path

The entry's full path name: equivalent to `os.path.join(scandir_path, entry.name)` where *scandir_path* is the *scandir()* *path* argument. The path is only absolute if the *scandir()* *path* argument was absolute. If the *scandir()* *path* argument was a *file descriptor*, the *path* attribute is the same as the *name* attribute.

The *path* attribute will be `bytes` if the *scandir()* *path* argument is of type `bytes` and `str` otherwise. Use *fsdecode()* to decode byte filenames.

inode()

Return the inode number of the entry.

The result is cached on the `os.DirEntry` object. Use `os.stat(entry.path, follow_symlinks=False).st_ino` to fetch up-to-date information.

On the first, uncached call, a system call is required on Windows but not on Unix.

is_dir(*, *follow_symlinks=True*)

Return `True` if this entry is a directory or a symbolic link pointing to a directory; return `False` if the entry is or points to any other kind of file, or if it doesn't exist anymore.

If *follow_symlinks* is `False`, return `True` only if this entry is a directory (without following symlinks); return `False` if the entry is any other kind of file or if it doesn't exist anymore.

The result is cached on the `os.DirEntry` object, with a separate cache for *follow_symlinks* `True` and `False`. Call *os.stat()* along with *stat.S_ISDIR()* to fetch up-to-date information.

On the first, uncached call, no system call is required in most cases. Specifically, for non-symlinks, neither Windows or Unix require a system call, except on certain Unix file systems, such as network file systems, that return `dirent.d_type == DT_UNKNOWN`. If the entry is a symlink, a system call will be required to follow the symlink unless *follow_symlinks* is `False`.

This method can raise *OSError*, such as *PermissionError*, but *FileNotFoundError* is caught and not raised.

is_file(*, *follow_symlinks=True*)

Return `True` if this entry is a file or a symbolic link pointing to a file; return `False` if the entry is or points to a directory or other non-file entry, or if it doesn't exist anymore.

If *follow_symlinks* is `False`, return `True` only if this entry is a file (without following symlinks); return `False` if the entry is a directory or other non-file entry, or if it doesn't exist anymore.

The result is cached on the `os.DirEntry` object. Caching, system calls made, and exceptions raised are as per *is_dir()*.

is_symlink()

Return `True` if this entry is a symbolic link (even if broken); return `False` if the entry points to a directory or any kind of file, or if it doesn't exist anymore.

The result is cached on the `os.DirEntry` object. Call *os.path.islink()* to fetch up-to-date information.

On the first, uncached call, no system call is required in most cases. Specifically, neither Windows or Unix require a system call, except on certain Unix file systems, such as network file systems, that return `dirent.d_type == DT_UNKNOWN`.

This method can raise *OSError*, such as *PermissionError*, but *FileNotFoundError* is caught and not raised.

is_junction()

Return `True` if this entry is a junction (even if broken); return `False` if the entry points to a regular directory, any kind of file, a symlink, or if it doesn't exist anymore.

The result is cached on the `os.DirEntry` object. Call *os.path.isjunction()* to fetch up-to-date information.

Added in version 3.12.

stat(*, *follow_symlinks=True*)

Return a *stat_result* object for this entry. This method follows symbolic links by default; to stat a symbolic link add the `follow_symlinks=False` argument.

On Unix, this method always requires a system call. On Windows, it only requires a system call if *follow_symlinks* is `True` and the entry is a reparse point (for example, a symbolic link or directory junction).

On Windows, the `st_ino`, `st_dev` and `st_nlink` attributes of the *stat_result* are always set to zero. Call *os.stat()* to get these attributes.

The result is cached on the `os.DirEntry` object, with a separate cache for *follow_symlinks* `True` and `False`. Call *os.stat()* to fetch up-to-date information.

Note that there is a nice correspondence between several attributes and methods of `os.DirEntry` and of *pathlib.Path*. In particular, the `name` attribute has the same meaning, as do the `is_dir()`, `is_file()`, `is_symlink()`, `is_junction()`, and `stat()` methods.

Added in version 3.5.

Changed in version 3.6: Added support for the *PathLike* interface. Added support for *bytes* paths on Windows.

Changed in version 3.12: The `st_ctime` attribute of a stat result is deprecated on Windows. The file creation time is properly available as `st_birthtime`, and in the future `st_ctime` may be changed to return zero or the metadata change time, if available.

`os.`**stat**(*path*, *, *dir_fd=None*, *follow_symlinks=True*)

Get the status of a file or a file descriptor. Perform the equivalent of a `stat()` system call on the given path. *path* may be specified as either a string or bytes – directly or indirectly through the *PathLike* interface – or as an open file descriptor. Return a *stat_result* object.

This function normally follows symlinks; to stat a symlink add the argument `follow_symlinks=False`, or use *lstat()*.

This function can support *specifying a file descriptor* and *not following symlinks*.

On Windows, passing `follow_symlinks=False` will disable following all name-surrogate reparse points, which includes symlinks and directory junctions. Other types of reparse points that do not resemble links or that the operating system is unable to follow will be opened directly. When following a chain of multiple links, this

may result in the original link being returned instead of the non-link that prevented full traversal. To obtain stat results for the final path in this case, use the *os.path.realpath()* function to resolve the path name as far as possible and call *lstat()* on the result. This does not apply to dangling symlinks or junction points, which will raise the usual exceptions.

Example:

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
os.stat_result(st_mode=33188, st_ino=7876932, st_dev=234881026,
st_nlink=1, st_uid=501, st_gid=501, st_size=264, st_atime=1297230295,
st_mtime=1297230027, st_ctime=1297230027)
>>> statinfo.st_size
264
```

See also

fstat() and *lstat()* functions.

Changed in version 3.3: Added the *dir_fd* and *follow_symlinks* parameters, specifying a file descriptor instead of a path.

Changed in version 3.6: Accepts a *path-like object*.

Changed in version 3.8: On Windows, all reparse points that can be resolved by the operating system are now followed, and passing `follow_symlinks=False` disables following all name surrogate reparse points. If the operating system reaches a reparse point that it is not able to follow, *stat* now returns the information for the original path as if `follow_symlinks=False` had been specified instead of raising an error.

class os.**stat_result**

Object whose attributes correspond roughly to the members of the `stat` structure. It is used for the result of *os.stat()*, *os.fstat()* and *os.lstat()*.

Attributes:

st_mode

File mode: file type and file mode bits (permissions).

st_ino

Platform dependent, but if non-zero, uniquely identifies the file for a given value of `st_dev`. Typically:

- the inode number on Unix,
- the file index on Windows

st_dev

Identifier of the device on which this file resides.

st_nlink

Number of hard links.

st_uid

User identifier of the file owner.

st_gid

Group identifier of the file owner.

st_size

Size of the file in bytes, if it is a regular file or a symbolic link. The size of a symbolic link is the length of the pathname it contains, without a terminating null byte.

Timestamps:

st_atime

Time of most recent access expressed in seconds.

st_mtime

Time of most recent content modification expressed in seconds.

st_ctime

Time of most recent metadata change expressed in seconds.

Changed in version 3.12: `st_ctime` is deprecated on Windows. Use `st_birthtime` for the file creation time. In the future, `st_ctime` will contain the time of the most recent metadata change, as for other platforms.

st_atime_ns

Time of most recent access expressed in nanoseconds as an integer.

Added in version 3.3.

st_mtime_ns

Time of most recent content modification expressed in nanoseconds as an integer.

Added in version 3.3.

st_ctime_ns

Time of most recent metadata change expressed in nanoseconds as an integer.

Added in version 3.3.

Changed in version 3.12: `st_ctime_ns` is deprecated on Windows. Use `st_birthtime_ns` for the file creation time. In the future, `st_ctime` will contain the time of the most recent metadata change, as for other platforms.

st_birthtime

Time of file creation expressed in seconds. This attribute is not always available, and may raise *AttributeError*.

Changed in version 3.12: `st_birthtime` is now available on Windows.

st_birthtime_ns

Time of file creation expressed in nanoseconds as an integer. This attribute is not always available, and may raise *AttributeError*.

Added in version 3.12.

Note

The exact meaning and resolution of the *st_atime*, *st_mtime*, *st_ctime* and *st_birthtime* attributes depend on the operating system and the file system. For example, on Windows systems using the FAT32 file systems, *st_mtime* has 2-second resolution, and *st_atime* has only 1-day resolution. See your operating system documentation for details.

Similarly, although *st_atime_ns*, *st_mtime_ns*, *st_ctime_ns* and *st_birthtime_ns* are always expressed in nanoseconds, many systems do not provide nanosecond precision. On systems that do provide nanosecond precision, the floating-point object used to store *st_atime*, *st_mtime*, *st_ctime* and

st_birthtime cannot preserve all of it, and as such will be slightly inexact. If you need the exact timestamps you should always use *st_atime_ns*, *st_mtime_ns*, *st_ctime_ns* and *st_birthtime_ns*.

On some Unix systems (such as Linux), the following attributes may also be available:

st_blocks

Number of 512-byte blocks allocated for file. This may be smaller than *st_size*/512 when the file has holes.

st_blksize

"Preferred" blocksize for efficient file system I/O. Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.

st_rdev

Type of device if an inode device.

st_flags

User defined flags for file.

On other Unix systems (such as FreeBSD), the following attributes may be available (but may be only filled out if root tries to use them):

st_gen

File generation number.

On Solaris and derivatives, the following attributes may also be available:

st_fstype

String that uniquely identifies the type of the filesystem that contains the file.

On macOS systems, the following attributes may also be available:

st_rsize

Real size of the file.

st_creator

Creator of the file.

st_type

File type.

On Windows systems, the following attributes are also available:

st_file_attributes

Windows file attributes: `dwFileAttributes` member of the `BY_HANDLE_FILE_INFORMATION` structure returned by `GetFileInformationByHandle()`. See the `FILE_ATTRIBUTE_* <stat.FILE_ATTRIBUTE_ARCHIVE>` constants in the *stat* module.

Added in version 3.5.

st_reparse_tag

When *st_file_attributes* has the *FILE_ATTRIBUTE_REPARSE_POINT* set, this field contains the tag identifying the type of reparse point. See the *IO_REPARSE_TAG_** constants in the *stat* module.

The standard module *stat* defines functions and constants that are useful for extracting information from a `stat` structure. (On Windows, some items are filled with dummy values.)

For backward compatibility, a *stat_result* instance is also accessible as a tuple of at least 10 integers giving the most important (and portable) members of the `stat` structure, in the order *st_mode*, *st_ino*, *st_dev*,

st_nlink, *st_uid*, *st_gid*, *st_size*, *st_atime*, *st_mtime*, *st_ctime*. More items may be added at the end by some implementations. For compatibility with older Python versions, accessing *stat_result* as a tuple always returns integers.

Changed in version 3.5: Windows now returns the file index as *st_ino* when available.

Changed in version 3.7: Added the *st_fstype* member to Solaris/derivatives.

Changed in version 3.8: Added the *st_reparse_tag* member on Windows.

Changed in version 3.8: On Windows, the *st_mode* member now identifies special files as `S_IFCHR`, `S_IFIFO` or `S_IFBLK` as appropriate.

Changed in version 3.12: On Windows, *st_ctime* is deprecated. Eventually, it will contain the last metadata change time, for consistency with other platforms, but for now still contains creation time. Use *st_birthtime* for the creation time.

On Windows, *st_ino* may now be up to 128 bits, depending on the file system. Previously it would not be above 64 bits, and larger file identifiers would be arbitrarily packed.

On Windows, *st_rdev* no longer returns a value. Previously it would contain the same as *st_dev*, which was incorrect.

Added the *st_birthtime* member on Windows.

`os.`**`statvfs`**(*path*)

Perform a `statvfs()` system call on the given path. The return value is an object whose attributes describe the filesystem on the given path, and correspond to the members of the `statvfs` structure, namely: `f_bsize`, `f_frsize`, `f_blocks`, `f_bfree`, `f_bavail`, `f_files`, `f_ffree`, `f_favail`, `f_flag`, `f_namemax`, `f_fsid`.

Two module-level constants are defined for the `f_flag` attribute's bit-flags: if `ST_RDONLY` is set, the filesystem is mounted read-only, and if `ST_NOSUID` is set, the semantics of setuid/setgid bits are disabled or not supported.

Additional module-level constants are defined for GNU/glibc based systems. These are `ST_NODEV` (disallow access to device special files), `ST_NOEXEC` (disallow program execution), `ST_SYNCHRONOUS` (writes are synced at once), `ST_MANDLOCK` (allow mandatory locks on an FS), `ST_WRITE` (write on file/directory/symlink), `ST_APPEND` (append-only file), `ST_IMMUTABLE` (immutable file), `ST_NOATIME` (do not update access times), `ST_NODIRATIME` (do not update directory access times), `ST_RELATIME` (update atime relative to mtime/ctime).

This function can support *specifying a file descriptor*.

Availability: Unix.

Changed in version 3.2: The `ST_RDONLY` and `ST_NOSUID` constants were added.

Changed in version 3.3: Added support for specifying *path* as an open file descriptor.

Changed in version 3.4: The `ST_NODEV`, `ST_NOEXEC`, `ST_SYNCHRONOUS`, `ST_MANDLOCK`, `ST_WRITE`, `ST_APPEND`, `ST_IMMUTABLE`, `ST_NOATIME`, `ST_NODIRATIME`, and `ST_RELATIME` constants were added.

Changed in version 3.6: Accepts a *path-like object*.

Changed in version 3.7: Added the `f_fsid` attribute.

`os.`**`supports_dir_fd`**

A *set* object indicating which functions in the *os* module accept an open file descriptor for their *dir_fd* parameter. Different platforms provide different features, and the underlying functionality Python uses to implement the *dir_fd* parameter is not available on all platforms Python supports. For consistency's sake, functions that may support *dir_fd* always allow specifying the parameter, but will throw an exception if the functionality is used when it's not locally available. (Specifying `None` for *dir_fd* is always supported on all platforms.)

To check whether a particular function accepts an open file descriptor for its *dir_fd* parameter, use the `in` operator on `supports_dir_fd`. As an example, this expression evaluates to `True` if *os.stat()* accepts open file descriptors for *dir_fd* on the local platform:

```
os.stat in os.supports_dir_fd
```

Currently *dir_fd* parameters only work on Unix platforms; none of them work on Windows.

Added in version 3.3.

os.**supports_effective_ids**

A *set* object indicating whether *os.access()* permits specifying `True` for its *effective_ids* parameter on the local platform. (Specifying `False` for *effective_ids* is always supported on all platforms.) If the local platform supports it, the collection will contain *os.access()*; otherwise it will be empty.

This expression evaluates to `True` if *os.access()* supports `effective_ids=True` on the local platform:

```
os.access in os.supports_effective_ids
```

Currently *effective_ids* is only supported on Unix platforms; it does not work on Windows.

Added in version 3.3.

os.**supports_fd**

A *set* object indicating which functions in the *os* module permit specifying their *path* parameter as an open file descriptor on the local platform. Different platforms provide different features, and the underlying functionality Python uses to accept open file descriptors as *path* arguments is not available on all platforms Python supports.

To determine whether a particular function permits specifying an open file descriptor for its *path* parameter, use the `in` operator on `supports_fd`. As an example, this expression evaluates to `True` if *os.chdir()* accepts open file descriptors for *path* on your local platform:

```
os.chdir in os.supports_fd
```

Added in version 3.3.

os.**supports_follow_symlinks**

A *set* object indicating which functions in the *os* module accept `False` for their *follow_symlinks* parameter on the local platform. Different platforms provide different features, and the underlying functionality Python uses to implement *follow_symlinks* is not available on all platforms Python supports. For consistency's sake, functions that may support *follow_symlinks* always allow specifying the parameter, but will throw an exception if the functionality is used when it's not locally available. (Specifying `True` for *follow_symlinks* is always supported on all platforms.)

To check whether a particular function accepts `False` for its *follow_symlinks* parameter, use the `in` operator on `supports_follow_symlinks`. As an example, this expression evaluates to `True` if you may specify `follow_symlinks=False` when calling *os.stat()* on the local platform:

```
os.stat in os.supports_follow_symlinks
```

Added in version 3.3.

os.**symlink**(*src*, *dst*, *target_is_directory=False*, ***, *dir_fd=None*)

Create a symbolic link pointing to *src* named *dst*.

On Windows, a symlink represents either a file or a directory, and does not morph to the target dynamically. If the target is present, the type of the symlink will be created to match. Otherwise, the symlink will be created as a directory if *target_is_directory* is `True` or a file symlink (the default) otherwise. On non-Windows platforms, *target_is_directory* is ignored.

This function can support *paths relative to directory descriptors*.

> **Note**
>
> On newer versions of Windows 10, unprivileged accounts can create symlinks if Developer Mode is enabled. When Developer Mode is not available/enabled, the *SeCreateSymbolicLinkPrivilege* privilege is required, or the process must be run as an administrator.
>
> `OSError` is raised when the function is called by an unprivileged user.

Raises an *auditing event* `os.symlink` with arguments `src`, `dst`, `dir_fd`.

Availability: Unix, Windows.

The function is limited on WASI, see *WebAssembly platforms* for more information.

Changed in version 3.2: Added support for Windows 6.0 (Vista) symbolic links.

Changed in version 3.3: Added the *dir_fd* parameter, and now allow *target_is_directory* on non-Windows platforms.

Changed in version 3.6: Accepts a *path-like object* for *src* and *dst*.

Changed in version 3.8: Added support for unelevated symlinks on Windows with Developer Mode.

`os.`**`sync`**`()`

Force write of everything to disk.

Availability: Unix.

Added in version 3.3.

`os.`**`truncate`**`(`*path*, *length*`)`

Truncate the file corresponding to *path*, so that it is at most *length* bytes in size.

This function can support *specifying a file descriptor*.

Raises an *auditing event* `os.truncate` with arguments `path`, `length`.

Availability: Unix, Windows.

Added in version 3.3.

Changed in version 3.5: Added support for Windows

Changed in version 3.6: Accepts a *path-like object*.

`os.`**`unlink`**`(`*path*, *, *dir_fd=None*`)`

Remove (delete) the file *path*. This function is semantically identical to `remove()`; the `unlink` name is its traditional Unix name. Please see the documentation for `remove()` for further information.

Raises an *auditing event* `os.remove` with arguments `path`, `dir_fd`.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.6: Accepts a *path-like object*.

`os.`**`utime`**`(`*path*, *times=None*, *, [*ns*,]*dir_fd=None*, *follow_symlinks=True*`)`

Set the access and modified times of the file specified by *path*.

`utime()` takes two optional parameters, *times* and *ns*. These specify the times set on *path* and are used as follows:

- If *ns* is specified, it must be a 2-tuple of the form `(atime_ns, mtime_ns)` where each member is an int expressing nanoseconds.
- If *times* is not `None`, it must be a 2-tuple of the form `(atime, mtime)` where each member is an int or float expressing seconds.

- If *times* is `None` and *ns* is unspecified, this is equivalent to specifying `ns=(atime_ns, mtime_ns)` where both times are the current time.

It is an error to specify tuples for both *times* and *ns*.

Note that the exact times you set here may not be returned by a subsequent `stat()` call, depending on the resolution with which your operating system records access and modification times; see `stat()`. The best way to preserve exact times is to use the *st_atime_ns* and *st_mtime_ns* fields from the `os.stat()` result object with the *ns* parameter to `utime()`.

This function can support *specifying a file descriptor*, *paths relative to directory descriptors* and *not following symlinks*.

Raises an *auditing event* `os.utime` with arguments `path`, `times`, `ns`, `dir_fd`.

Changed in version 3.3: Added support for specifying *path* as an open file descriptor, and the *dir_fd*, *follow_symlinks*, and *ns* parameters.

Changed in version 3.6: Accepts a *path-like object*.

`os.`**`walk`**(*top*, *topdown=True*, *onerror=None*, *followlinks=False*)

Generate the file names in a directory tree by walking the tree either top-down or bottom-up. For each directory in the tree rooted at directory *top* (including *top* itself), it yields a 3-tuple `(dirpath, dirnames, filenames)`.

dirpath is a string, the path to the directory. *dirnames* is a list of the names of the subdirectories in *dirpath* (including symlinks to directories, and excluding `'.'` and `'..'`). *filenames* is a list of the names of the non-directory files in *dirpath*. Note that the names in the lists contain no path components. To get a full path (which begins with *top*) to a file or directory in *dirpath*, do `os.path.join(dirpath, name)`. Whether or not the lists are sorted depends on the file system. If a file is removed from or added to the *dirpath* directory during generating the lists, whether a name for that file be included is unspecified.

If optional argument *topdown* is `True` or not specified, the triple for a directory is generated before the triples for any of its subdirectories (directories are generated top-down). If *topdown* is `False`, the triple for a directory is generated after the triples for all of its subdirectories (directories are generated bottom-up). No matter the value of *topdown*, the list of subdirectories is retrieved before the tuples for the directory and its subdirectories are generated.

When *topdown* is `True`, the caller can modify the *dirnames* list in-place (perhaps using `del` or slice assignment), and `walk()` will only recurse into the subdirectories whose names remain in *dirnames*; this can be used to prune the search, impose a specific order of visiting, or even to inform `walk()` about directories the caller creates or renames before it resumes `walk()` again. Modifying *dirnames* when *topdown* is `False` has no effect on the behavior of the walk, because in bottom-up mode the directories in *dirnames* are generated before *dirpath* itself is generated.

By default, errors from the `scandir()` call are ignored. If optional argument *onerror* is specified, it should be a function; it will be called with one argument, an `OSError` instance. It can report the error to continue with the walk, or raise the exception to abort the walk. Note that the filename is available as the `filename` attribute of the exception object.

By default, `walk()` will not walk down into symbolic links that resolve to directories. Set *followlinks* to `True` to visit directories pointed to by symlinks, on systems that support them.

> **Note**
>
> Be aware that setting *followlinks* to `True` can lead to infinite recursion if a link points to a parent directory of itself. `walk()` does not keep track of the directories it visited already.

> **Note**
>
> If you pass a relative pathname, don't change the current working directory between resumptions of *walk()*. *walk()* never changes the current directory, and assumes that its caller doesn't either.

This example displays the number of bytes taken by non-directory files in each directory under the starting directory, except that it doesn't look under any CVS subdirectory:

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name)) for name in files), end=" ")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

In the next example (simple implementation of *shutil.rmtree()*), walking the tree bottom-up is essential, *rmdir()* doesn't allow deleting a directory before the directory is empty:

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```

Raises an *auditing event* `os.walk` with arguments `top`, `topdown`, `onerror`, `followlinks`.

Changed in version 3.5: This function now calls *os.scandir()* instead of *os.listdir()*, making it faster by reducing the number of calls to *os.stat()*.

Changed in version 3.6: Accepts a *path-like object*.

`os.`**`fwalk`**(*top='.'*, *topdown=True*, *onerror=None*, ***, *follow_symlinks=False*, *dir_fd=None*)

This behaves exactly like *walk()*, except that it yields a 4-tuple `(dirpath, dirnames, filenames, dirfd)`, and it supports `dir_fd`.

dirpath, *dirnames* and *filenames* are identical to *walk()* output, and *dirfd* is a file descriptor referring to the directory *dirpath*.

This function always supports *paths relative to directory descriptors* and *not following symlinks*. Note however that, unlike other functions, the *fwalk()* default value for *follow_symlinks* is `False`.

> **Note**
>
> Since *fwalk()* yields file descriptors, those are only valid until the next iteration step, so you should duplicate them (e.g. with *dup()*) if you want to keep them longer.

This example displays the number of bytes taken by non-directory files in each directory under the starting directory, except that it doesn't look under any CVS subdirectory:

```
import os
for root, dirs, files, rootfd in os.fwalk('python/Lib/email'):
    print(root, "consumes", end="")
    print(sum([os.stat(name, dir_fd=rootfd).st_size for name in files]),
          end="")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

In the next example, walking the tree bottom-up is essential: *rmdir()* doesn't allow deleting a directory before the directory is empty:

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files, rootfd in os.fwalk(top, topdown=False):
    for name in files:
        os.unlink(name, dir_fd=rootfd)
    for name in dirs:
        os.rmdir(name, dir_fd=rootfd)
```

Raises an *auditing event* `os.fwalk` with arguments `top`, `topdown`, `onerror`, `follow_symlinks`, `dir_fd`.

Availability: Unix.

Added in version 3.3.

Changed in version 3.6: Accepts a *path-like object*.

Changed in version 3.7: Added support for *bytes* paths.

`os.`**`memfd_create`**(*name*[, *flags=os.MFD_CLOEXEC*])

Create an anonymous file and return a file descriptor that refers to it. *flags* must be one of the `os.MFD_*` constants available on the system (or a bitwise ORed combination of them). By default, the new file descriptor is *non-inheritable*.

The name supplied in *name* is used as a filename and will be displayed as the target of the corresponding symbolic link in the directory `/proc/self/fd/`. The displayed name is always prefixed with `memfd:` and serves only for debugging purposes. Names do not affect the behavior of the file descriptor, and as such multiple files can have the same name without any side effects.

Availability: Linux >= 3.17 with glibc >= 2.27.

Added in version 3.8.

`os.`**`MFD_CLOEXEC`**

`os.`**`MFD_ALLOW_SEALING`**

`os.`**`MFD_HUGETLB`**

`os.`**`MFD_HUGE_SHIFT`**

`os.`**`MFD_HUGE_MASK`**

`os.`**`MFD_HUGE_64KB`**

`os.`**`MFD_HUGE_512KB`**

`os.`**`MFD_HUGE_1MB`**

`os.`**`MFD_HUGE_2MB`**

os.**MFD_HUGE_8MB**

os.**MFD_HUGE_16MB**

os.**MFD_HUGE_32MB**

os.**MFD_HUGE_256MB**

os.**MFD_HUGE_512MB**

os.**MFD_HUGE_1GB**

os.**MFD_HUGE_2GB**

os.**MFD_HUGE_16GB**

These flags can be passed to *memfd_create()*.

Availability: Linux >= 3.17 with glibc >= 2.27

The MFD_HUGE* flags are only available since Linux 4.14.

Added in version 3.8.

os.**eventfd**(*initval*[, *flags=os.EFD_CLOEXEC*])

Create and return an event file descriptor. The file descriptors supports raw *read()* and *write()* with a buffer size of 8, *select()*, *poll()* and similar. See man page *eventfd(2)* for more information. By default, the new file descriptor is *non-inheritable*.

initval is the initial value of the event counter. The initial value must be a 32 bit unsigned integer. Please note that the initial value is limited to a 32 bit unsigned int although the event counter is an unsigned 64 bit integer with a maximum value of 2^{64}-2.

flags can be constructed from *EFD_CLOEXEC*, *EFD_NONBLOCK*, and *EFD_SEMAPHORE*.

If *EFD_SEMAPHORE* is specified and the event counter is non-zero, *eventfd_read()* returns 1 and decrements the counter by one.

If *EFD_SEMAPHORE* is not specified and the event counter is non-zero, *eventfd_read()* returns the current event counter value and resets the counter to zero.

If the event counter is zero and *EFD_NONBLOCK* is not specified, *eventfd_read()* blocks.

eventfd_write() increments the event counter. Write blocks if the write operation would increment the counter to a value larger than 2^{64}-2.

Example:

```
import os

# semaphore with start value '1'
fd = os.eventfd(1, os.EFD_SEMAPHORE | os.EFC_CLOEXEC)
try:
    # acquire semaphore
    v = os.eventfd_read(fd)
    try:
        do_work()
    finally:
        # release semaphore
        os.eventfd_write(fd, v)
finally:
    os.close(fd)
```

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.10.

os.**eventfd_read**(*fd*)

Read value from an *eventfd()* file descriptor and return a 64 bit unsigned int. The function does not verify that *fd* is an *eventfd()*.

Availability: Linux >= 2.6.27

Added in version 3.10.

os.**eventfd_write**(*fd*, *value*)

Add value to an *eventfd()* file descriptor. *value* must be a 64 bit unsigned int. The function does not verify that *fd* is an *eventfd()*.

Availability: Linux >= 2.6.27

Added in version 3.10.

os.**EFD_CLOEXEC**

Set close-on-exec flag for new *eventfd()* file descriptor.

Availability: Linux >= 2.6.27

Added in version 3.10.

os.**EFD_NONBLOCK**

Set *O_NONBLOCK* status flag for new *eventfd()* file descriptor.

Availability: Linux >= 2.6.27

Added in version 3.10.

os.**EFD_SEMAPHORE**

Provide semaphore-like semantics for reads from an *eventfd()* file descriptor. On read the internal counter is decremented by one.

Availability: Linux >= 2.6.30

Added in version 3.10.

Timer File Descriptors

Added in version 3.13.

These functions provide support for Linux's *timer file descriptor* API. Naturally, they are all only available on Linux.

os.**timerfd_create**(*clockid*, */*, *, *flags=0*)

Create and return a timer file descriptor (*timerfd*).

The file descriptor returned by *timerfd_create()* supports:

- *read()*
- *select()*
- *poll()*

The file descriptor's *read()* method can be called with a buffer size of 8. If the timer has already expired one or more times, *read()* returns the number of expirations with the host's endianness, which may be converted to an *int* by `int.from_bytes(x, byteorder=sys.byteorder)`.

select() and *poll()* can be used to wait until timer expires and the file descriptor is readable.

clockid must be a valid *clock ID*, as defined in the *time* module:

- *time.CLOCK_REALTIME*

- *time.CLOCK_MONOTONIC*
- *time.CLOCK_BOOTTIME* (Since Linux 3.15 for timerfd_create)

If *clockid* is *time.CLOCK_REALTIME*, a settable system-wide real-time clock is used. If system clock is changed, timer setting need to be updated. To cancel timer when system clock is changed, see *TFD_TIMER_CANCEL_ON_SET*.

If *clockid* is *time.CLOCK_MONOTONIC*, a non-settable monotonically increasing clock is used. Even if the system clock is changed, the timer setting will not be affected.

If *clockid* is *time.CLOCK_BOOTTIME*, same as *time.CLOCK_MONOTONIC* except it includes any time that the system is suspended.

The file descriptor's behaviour can be modified by specifying a *flags* value. Any of the following variables may used, combined using bitwise OR (the `|` operator):

- *TFD_NONBLOCK*
- *TFD_CLOEXEC*

If *TFD_NONBLOCK* is not set as a flag, *read()* blocks until the timer expires. If it is set as a flag, *read()* doesn't block, but If there hasn't been an expiration since the last call to read, *read()* raises *OSError* with `errno` is set to *errno.EAGAIN*.

TFD_CLOEXEC is always set by Python automatically.

The file descriptor must be closed with *os.close()* when it is no longer needed, or else the file descriptor will be leaked.

See also

The *timerfd_create(2)* man page.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

`os.`**`timerfd_settime`**(*fd*, /, *, *flags=flags*, *initial=0.0*, *interval=0.0*)

Alter a timer file descriptor's internal timer. This function operates the same interval timer as *timerfd_settime_ns()*.

fd must be a valid timer file descriptor.

The timer's behaviour can be modified by specifying a *flags* value. Any of the following variables may used, combined using bitwise OR (the `|` operator):

- *TFD_TIMER_ABSTIME*
- *TFD_TIMER_CANCEL_ON_SET*

The timer is disabled by setting *initial* to zero (`0`). If *initial* is equal to or greater than zero, the timer is enabled. If *initial* is less than zero, it raises an *OSError* exception with `errno` set to *errno.EINVAL*

By default the timer will fire when *initial* seconds have elapsed. (If *initial* is zero, timer will fire immediately.)

However, if the *TFD_TIMER_ABSTIME* flag is set, the timer will fire when the timer's clock (set by *clockid* in *timerfd_create()*) reaches *initial* seconds.

The timer's interval is set by the *interval* *float*. If *interval* is zero, the timer only fires once, on the initial expiration. If *interval* is greater than zero, the timer fires every time *interval* seconds have elapsed since the previous expiration. If *interval* is less than zero, it raises *OSError* with `errno` set to *errno.EINVAL*

If the *TFD_TIMER_CANCEL_ON_SET* flag is set along with *TFD_TIMER_ABSTIME* and the clock for this timer is *time.CLOCK_REALTIME*, the timer is marked as cancelable if the real-time clock is changed discontinuously. Reading the descriptor is aborted with the error ECANCELED.

Linux manages system clock as UTC. A daylight-savings time transition is done by changing time offset only and doesn't cause discontinuous system clock change.

Discontinuous system clock change will be caused by the following events:

- `settimeofday`
- `clock_settime`
- set the system date and time by `date` command

Return a two-item tuple of (`next_expiration`, `interval`) from the previous timer state, before this function executed.

See also

timerfd_create(2), *timerfd_settime(2)*, *settimeofday(2)*, *clock_settime(2)*, and *date(1)*.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

`os.`**`timerfd_settime_ns`**(*fd*, */*, ***, *flags=0*, *initial=0*, *interval=0*)

Similar to *timerfd_settime()*, but use time as nanoseconds. This function operates the same interval timer as *timerfd_settime()*.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

`os.`**`timerfd_gettime`**(*fd*, */*)

Return a two-item tuple of floats (`next_expiration`, `interval`).

`next_expiration` denotes the relative time until next the timer next fires, regardless of if the *TFD_TIMER_ABSTIME* flag is set.

`interval` denotes the timer's interval. If zero, the timer will only fire once, after `next_expiration` seconds have elapsed.

See also

timerfd_gettime(2)

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

`os.`**`timerfd_gettime_ns`**(*fd*, */*)

Similar to *timerfd_gettime()*, but return time as nanoseconds.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**TFD_NONBLOCK**

A flag for the *timerfd_create()* function, which sets the *O_NONBLOCK* status flag for the new timer file descriptor. If *TFD_NONBLOCK* is not set as a flag, *read()* blocks.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**TFD_CLOEXEC**

A flag for the *timerfd_create()* function, If *TFD_CLOEXEC* is set as a flag, set close-on-exec flag for new file descriptor.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**TFD_TIMER_ABSTIME**

A flag for the *timerfd_settime()* and *timerfd_settime_ns()* functions. If this flag is set, *initial* is interpreted as an absolute value on the timer's clock (in UTC seconds or nanoseconds since the Unix Epoch).

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**TFD_TIMER_CANCEL_ON_SET**

A flag for the *timerfd_settime()* and *timerfd_settime_ns()* functions along with *TFD_TIMER_ABSTIME*. The timer is cancelled when the time of the underlying clock changes discontinuously.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

Linux extended attributes

Added in version 3.3.

These functions are all available on Linux only.

os.**getxattr**(*path*, *attribute*, ***, *follow_symlinks=True*)

Return the value of the extended filesystem attribute *attribute* for *path*. *attribute* can be bytes or str (directly or indirectly through the *PathLike* interface). If it is str, it is encoded with the filesystem encoding.

This function can support *specifying a file descriptor* and *not following symlinks*.

Raises an *auditing event* `os.getxattr` with arguments `path`, `attribute`.

Changed in version 3.6: Accepts a *path-like object* for *path* and *attribute*.

os.**listxattr**(*path=None*, ***, *follow_symlinks=True*)

Return a list of the extended filesystem attributes on *path*. The attributes in the list are represented as strings decoded with the filesystem encoding. If *path* is `None`, *listxattr()* will examine the current directory.

This function can support *specifying a file descriptor* and *not following symlinks*.

Raises an *auditing event* `os.listxattr` with argument `path`.

Changed in version 3.6: Accepts a *path-like object*.

os.**removexattr**(*path*, *attribute*, *, *follow_symlinks=True*)

Removes the extended filesystem attribute *attribute* from *path*. *attribute* should be bytes or str (directly or indirectly through the *PathLike* interface). If it is a string, it is encoded with the *filesystem encoding and error handler*.

This function can support *specifying a file descriptor* and *not following symlinks*.

Raises an *auditing event* `os.removexattr` with arguments `path`, `attribute`.

Changed in version 3.6: Accepts a *path-like object* for *path* and *attribute*.

os.**setxattr**(*path*, *attribute*, *value*, *flags=0*, *, *follow_symlinks=True*)

Set the extended filesystem attribute *attribute* on *path* to *value*. *attribute* must be a bytes or str with no embedded NULs (directly or indirectly through the *PathLike* interface). If it is a str, it is encoded with the *filesystem encoding and error handler*. *flags* may be *XATTR_REPLACE* or *XATTR_CREATE*. If *XATTR_REPLACE* is given and the attribute does not exist, `ENODATA` will be raised. If *XATTR_CREATE* is given and the attribute already exists, the attribute will not be created and `EEXISTS` will be raised.

This function can support *specifying a file descriptor* and *not following symlinks*.

> **Note**
>
> A bug in Linux kernel versions less than 2.6.39 caused the flags argument to be ignored on some filesystems.

Raises an *auditing event* `os.setxattr` with arguments `path`, `attribute`, `value`, `flags`.

Changed in version 3.6: Accepts a *path-like object* for *path* and *attribute*.

os.**XATTR_SIZE_MAX**

The maximum size the value of an extended attribute can be. Currently, this is 64 KiB on Linux.

os.**XATTR_CREATE**

This is a possible value for the flags argument in *setxattr()*. It indicates the operation must create an attribute.

os.**XATTR_REPLACE**

This is a possible value for the flags argument in *setxattr()*. It indicates the operation must replace an existing attribute.

16.1.7 Process Management

These functions may be used to create and manage processes.

The various *exec** functions take a list of arguments for the new program loaded into the process. In each case, the first of these arguments is passed to the new program as its own name rather than as an argument a user may have typed on a command line. For the C programmer, this is the `argv[0]` passed to a program's `main()`. For example, `os.execv('/bin/echo', ['foo', 'bar'])` will only print `bar` on standard output; `foo` will seem to be ignored.

os.**abort**()

Generate a `SIGABRT` signal to the current process. On Unix, the default behavior is to produce a core dump; on Windows, the process immediately returns an exit code of `3`. Be aware that calling this function will not call the Python signal handler registered for `SIGABRT` with *signal.signal()*.

os.**add_dll_directory**(*path*)

Add a path to the DLL search path.

This search path is used when resolving dependencies for imported extension modules (the module itself is resolved through *sys.path*), and also by *ctypes*.

Remove the directory by calling **close()** on the returned object or using it in a `with` statement.

See the Microsoft documentation for more information about how DLLs are loaded.

Raises an *auditing event* `os.add_dll_directory` with argument `path`.

Availability: Windows.

Added in version 3.8: Previous versions of CPython would resolve DLLs using the default behavior for the current process. This led to inconsistencies, such as only sometimes searching `PATH` or the current working directory, and OS functions such as `AddDllDirectory` having no effect.

In 3.8, the two primary ways DLLs are loaded now explicitly override the process-wide behavior to ensure consistency. See the porting notes for information on updating libraries.

`os.`**`execl`**(*path*, *arg0*, *arg1*, ...)

`os.`**`execle`**(*path*, *arg0*, *arg1*, ..., *env*)

`os.`**`execlp`**(*file*, *arg0*, *arg1*, ...)

`os.`**`execlpe`**(*file*, *arg0*, *arg1*, ..., *env*)

`os.`**`execv`**(*path*, *args*)

`os.`**`execve`**(*path*, *args*, *env*)

`os.`**`execvp`**(*file*, *args*)

`os.`**`execvpe`**(*file*, *args*, *env*)

These functions all execute a new program, replacing the current process; they do not return. On Unix, the new executable is loaded into the current process, and will have the same process id as the caller. Errors will be reported as *`OSError`* exceptions.

The current process is replaced immediately. Open file objects and descriptors are not flushed, so if there may be data buffered on these open files, you should flush them using `sys.stdout.flush()` or *`os.fsync()`* before calling an *`exec*`* function.

The "l" and "v" variants of the *`exec*`* functions differ in how command-line arguments are passed. The "l" variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the `execl*()` functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the *args* parameter. In either case, the arguments to the child process should start with the name of the command being run, but this is not enforced.

The variants which include a "p" near the end (*`execlp()`*, *`execlpe()`*, *`execvp()`*, and *`execvpe()`*) will use the `PATH` environment variable to locate the program *file*. When the environment is being replaced (using one of the *`exec*e`* variants, discussed in the next paragraph), the new environment is used as the source of the `PATH` variable. The other variants, *`execl()`*, *`execle()`*, *`execv()`*, and *`execve()`*, will not use the `PATH` variable to locate the executable; *path* must contain an appropriate absolute or relative path. Relative paths must include at least one slash, even on Windows, as plain names will not be resolved.

For *`execle()`*, *`execlpe()`*, *`execve()`*, and *`execvpe()`* (note that these all end in "e"), the *env* parameter must be a mapping which is used to define the environment variables for the new process (these are used instead of the current process' environment); the functions *`execl()`*, *`execlp()`*, *`execv()`*, and *`execvp()`* all cause the new process to inherit the environment of the current process.

For *`execve()`* on some platforms, *path* may also be specified as an open file descriptor. This functionality may not be supported on your platform; you can check whether or not it is available using *`os.supports_fd`*. If it is unavailable, using it will raise a *`NotImplementedError`*.

Raises an *auditing event* `os.exec` with arguments `path`, `args`, `env`.

Availability: Unix, Windows, not WASI, not iOS.

Changed in version 3.3: Added support for specifying *path* as an open file descriptor for *`execve()`*.

Changed in version 3.6: Accepts a *path-like object*.

os.**_exit**(*n*)

Exit the process with status *n*, without calling cleanup handlers, flushing stdio buffers, etc.

> **Note**
>
> The standard way to exit is `sys.exit(n)`. `_exit()` should normally only be used in the child process after a `fork()`.

The following exit codes are defined and can be used with `_exit()`, although they are not required. These are typically used for system programs written in Python, such as a mail server's external command delivery program.

> **Note**
>
> Some of these may not be available on all Unix platforms, since there is some variation. These constants are defined where they are defined by the underlying platform.

os.**EX_OK**

Exit code that means no error occurred. May be taken from the defined value of `EXIT_SUCCESS` on some platforms. Generally has a value of zero.

Availability: Unix, Windows.

os.**EX_USAGE**

Exit code that means the command was used incorrectly, such as when the wrong number of arguments are given.

Availability: Unix, not WASI.

os.**EX_DATAERR**

Exit code that means the input data was incorrect.

Availability: Unix, not WASI.

os.**EX_NOINPUT**

Exit code that means an input file did not exist or was not readable.

Availability: Unix, not WASI.

os.**EX_NOUSER**

Exit code that means a specified user did not exist.

Availability: Unix, not WASI.

os.**EX_NOHOST**

Exit code that means a specified host did not exist.

Availability: Unix, not WASI.

os.**EX_UNAVAILABLE**

Exit code that means that a required service is unavailable.

Availability: Unix, not WASI.

os.**EX_SOFTWARE**

Exit code that means an internal software error was detected.

Availability: Unix, not WASI.

os.**EX_OSERR**

Exit code that means an operating system error was detected, such as the inability to fork or create a pipe.

Availability: Unix, not WASI.

os.**EX_OSFILE**

Exit code that means some system file did not exist, could not be opened, or had some other kind of error.

Availability: Unix, not WASI.

os.**EX_CANTCREAT**

Exit code that means a user specified output file could not be created.

Availability: Unix, not WASI.

os.**EX_IOERR**

Exit code that means that an error occurred while doing I/O on some file.

Availability: Unix, not WASI.

os.**EX_TEMPFAIL**

Exit code that means a temporary failure occurred. This indicates something that may not really be an error, such as a network connection that couldn't be made during a retryable operation.

Availability: Unix, not WASI.

os.**EX_PROTOCOL**

Exit code that means that a protocol exchange was illegal, invalid, or not understood.

Availability: Unix, not WASI.

os.**EX_NOPERM**

Exit code that means that there were insufficient permissions to perform the operation (but not intended for file system problems).

Availability: Unix, not WASI.

os.**EX_CONFIG**

Exit code that means that some kind of configuration error occurred.

Availability: Unix, not WASI.

os.**EX_NOTFOUND**

Exit code that means something like "an entry was not found".

Availability: Unix, not WASI.

os.**fork**()

Fork a child process. Return `0` in the child and the child's process id in the parent. If an error occurs *OSError* is raised.

Note that some platforms including FreeBSD <= 6.3 and Cygwin have known issues when using `fork()` from a thread.

Raises an *auditing event* `os.fork` with no arguments.

> **Warning**
>
> If you use TLS sockets in an application calling `fork()`, see the warning in the *ssl* documentation.

> **Warning**
>
> On macOS the use of this function is unsafe when mixed with using higher-level system APIs, and that includes using *urllib.request*.

Changed in version 3.8: Calling `fork()` in a subinterpreter is no longer supported (*RuntimeError* is raised).

Changed in version 3.12: If Python is able to detect that your process has multiple threads, *os.fork()* now raises a *DeprecationWarning*.

We chose to surface this as a warning, when detectable, to better inform developers of a design problem that the POSIX platform specifically notes as not supported. Even in code that *appears* to work, it has never been safe to mix threading with *os.fork()* on POSIX platforms. The CPython runtime itself has always made API calls that are not safe for use in the child process when threads existed in the parent (such as `malloc` and `free`).

Users of macOS or users of libc or malloc implementations other than those typically found in glibc to date are among those already more likely to experience deadlocks running such code.

See this discussion on fork being incompatible with threads for technical details of why we're surfacing this long-standing platform compatibility problem to developers.

Availability: POSIX, not WASI, not iOS.

`os.`**`forkpty`**`()`

Fork a child process, using a new pseudo-terminal as the child's controlling terminal. Return a pair of `(pid, fd)`, where *pid* is `0` in the child, the new child's process id in the parent, and *fd* is the file descriptor of the master end of the pseudo-terminal. For a more portable approach, use the *pty* module. If an error occurs *OSError* is raised.

Raises an *auditing event* `os.forkpty` with no arguments.

> **Warning**
>
> On macOS the use of this function is unsafe when mixed with using higher-level system APIs, and that includes using *urllib.request*.

Changed in version 3.8: Calling `forkpty()` in a subinterpreter is no longer supported (*RuntimeError* is raised).

Changed in version 3.12: If Python is able to detect that your process has multiple threads, this now raises a *DeprecationWarning*. See the longer explanation on *os.fork()*.

Availability: Unix, not WASI, not iOS.

`os.`**`kill`**`(`*pid*, *sig*, `/)`

Send signal *sig* to the process *pid*. Constants for the specific signals available on the host platform are defined in the *signal* module.

Windows: The *signal.CTRL_C_EVENT* and *signal.CTRL_BREAK_EVENT* signals are special signals which can only be sent to console processes which share a common console window, e.g., some subprocesses. Any other value for *sig* will cause the process to be unconditionally killed by the TerminateProcess API, and the exit code will be set to *sig*. The Windows version of *kill()* additionally takes process handles to be killed.

See also *signal.pthread_kill()*.

Raises an *auditing event* `os.kill` with arguments `pid`, `sig`.

Availability: Unix, Windows, not WASI, not iOS.

Changed in version 3.2: Added Windows support.

os.**killpg**(*pgid*, *sig*, /)

Send the signal *sig* to the process group *pgid*.

Raises an *auditing event* `os.killpg` with arguments `pgid`, `sig`.

Availability: Unix, not WASI, not iOS.

os.**nice**(*increment*, /)

Add *increment* to the process's "niceness". Return the new niceness.

Availability: Unix, not WASI.

os.**pidfd_open**(*pid*, *flags=0*)

Return a file descriptor referring to the process *pid* with *flags* set. This descriptor can be used to perform process management without races and signals.

See the *pidfd_open(2)* man page for more details.

Availability: Linux >= 5.3

Added in version 3.9.

os.**PIDFD_NONBLOCK**

This flag indicates that the file descriptor will be non-blocking. If the process referred to by the file descriptor has not yet terminated, then an attempt to wait on the file descriptor using *waitid(2)* will immediately return the error *EAGAIN* rather than blocking.

Availability: Linux >= 5.10

Added in version 3.12.

os.**plock**(*op*, /)

Lock program segments into memory. The value of *op* (defined in <sys/lock.h>) determines which segments are locked.

Availability: Unix, not WASI, not iOS.

os.**popen**(*cmd*, *mode='r'*, *buffering=-1*)

Open a pipe to or from command *cmd*. The return value is an open file object connected to the pipe, which can be read or written depending on whether *mode* is `'r'` (default) or `'w'`. The *buffering* argument have the same meaning as the corresponding argument to the built-in *open()* function. The returned file object reads or writes text strings rather than bytes.

The `close` method returns *None* if the subprocess exited successfully, or the subprocess's return code if there was an error. On POSIX systems, if the return code is positive it represents the return value of the process left-shifted by one byte. If the return code is negative, the process was terminated by the signal given by the negated value of the return code. (For example, the return value might be `- signal.SIGKILL` if the subprocess was killed.) On Windows systems, the return value contains the signed integer return code from the child process.

On Unix, *waitstatus_to_exitcode()* can be used to convert the `close` method result (exit status) into an exit code if it is not `None`. On Windows, the `close` method result is directly the exit code (or `None`).

This is implemented using *subprocess.Popen*; see that class's documentation for more powerful ways to manage and communicate with subprocesses.

Availability: not WASI, not iOS.

> **Note**
>
> The *Python UTF-8 Mode* affects encodings used for *cmd* and pipe contents.
>
> *popen()* is a simple wrapper around *subprocess.Popen*. Use *subprocess.Popen* or *subprocess.run()* to control options like encodings.

os.**posix_spawn**(*path*, *argv*, *env*, *, *file_actions=None*, *setpgroup=None*, *resetids=False*, *setsid=False*, *setsigmask=()*, *setsigdef=()*, *scheduler=None*)

Wraps the `posix_spawn()` C library API for use from Python.

Most users should use *subprocess.run()* instead of *posix_spawn()*.

The positional-only arguments *path*, *args*, and *env* are similar to *execve()*. *env* is allowed to be `None`, in which case current process' environment is used.

The *path* parameter is the path to the executable file. The *path* should contain a directory. Use *posix_spawnp()* to pass an executable file without directory.

The *file_actions* argument may be a sequence of tuples describing actions to take on specific file descriptors in the child process between the C library implementation's `fork()` and `exec()` steps. The first item in each tuple must be one of the three type indicator listed below describing the remaining tuple elements:

os.**POSIX_SPAWN_OPEN**

(`os.POSIX_SPAWN_OPEN`, *fd*, *path*, *flags*, *mode*)

Performs `os.dup2(os.open(path, flags, mode), fd)`.

os.**POSIX_SPAWN_CLOSE**

(`os.POSIX_SPAWN_CLOSE`, *fd*)

Performs `os.close(fd)`.

os.**POSIX_SPAWN_DUP2**

(`os.POSIX_SPAWN_DUP2`, *fd*, *new_fd*)

Performs `os.dup2(fd, new_fd)`.

os.**POSIX_SPAWN_CLOSEFROM**

(`os.POSIX_SPAWN_CLOSEFROM`, *fd*)

Performs `os.closerange(fd, INF)`.

These tuples correspond to the C library `posix_spawn_file_actions_addopen()`, `posix_spawn_file_actions_addclose()`, `posix_spawn_file_actions_adddup2()`, and `posix_spawn_file_actions_addclosefrom_np()` API calls used to prepare for the `posix_spawn()` call itself.

The *setpgroup* argument will set the process group of the child to the value specified. If the value specified is 0, the child's process group ID will be made the same as its process ID. If the value of *setpgroup* is not set, the child will inherit the parent's process group ID. This argument corresponds to the C library `POSIX_SPAWN_SETPGROUP` flag.

If the *resetids* argument is `True` it will reset the effective UID and GID of the child to the real UID and GID of the parent process. If the argument is `False`, then the child retains the effective UID and GID of the parent. In either case, if the set-user-ID and set-group-ID permission bits are enabled on the executable file, their effect will override the setting of the effective UID and GID. This argument corresponds to the C library `POSIX_SPAWN_RESETIDS` flag.

If the *setsid* argument is `True`, it will create a new session ID for `posix_spawn`. *setsid* requires `POSIX_SPAWN_SETSID` or `POSIX_SPAWN_SETSID_NP` flag. Otherwise, *NotImplementedError* is raised.

The *setsigmask* argument will set the signal mask to the signal set specified. If the parameter is not used, then the child inherits the parent's signal mask. This argument corresponds to the C library `POSIX_SPAWN_SETSIGMASK` flag.

The *sigdef* argument will reset the disposition of all signals in the set specified. This argument corresponds to the C library `POSIX_SPAWN_SETSIGDEF` flag.

The *scheduler* argument must be a tuple containing the (optional) scheduler policy and an instance of *sched_param* with the scheduler parameters. A value of `None` in the place of the scheduler policy indicates that is not being provided. This argument is a combination of the C library `POSIX_SPAWN_SETSCHEDPARAM` and `POSIX_SPAWN_SETSCHEDULER` flags.

Raises an *auditing event* `os.posix_spawn` with arguments `path`, `argv`, `env`.

Added in version 3.8.

Changed in version 3.13: *env* parameter accepts `None`. `os.POSIX_SPAWN_CLOSEFROM` is available on platforms where `posix_spawn_file_actions_addclosefrom_np()` exists.

Availability: Unix, not WASI, not iOS.

`os.`**`posix_spawnp`**(*path*, *argv*, *env*, ***, *file_actions=None*, *setpgroup=None*, *resetids=False*, *setsid=False*, *setsigmask=()*, *setsigdef=()*, *scheduler=None*)

Wraps the `posix_spawnp()` C library API for use from Python.

Similar to *posix_spawn()* except that the system searches for the *executable* file in the list of directories specified by the `PATH` environment variable (in the same way as for `execvp(3)`).

Raises an *auditing event* `os.posix_spawn` with arguments `path`, `argv`, `env`.

Added in version 3.8.

Availability: POSIX, not WASI, not iOS.

See *posix_spawn()* documentation.

`os.`**`register_at_fork`**(***, *before=None*, *after_in_parent=None*, *after_in_child=None*)

Register callables to be executed when a new child process is forked using *os.fork()* or similar process cloning APIs. The parameters are optional and keyword-only. Each specifies a different call point.

- *before* is a function called before forking a child process.
- *after_in_parent* is a function called from the parent process after forking a child process.
- *after_in_child* is a function called from the child process.

These calls are only made if control is expected to return to the Python interpreter. A typical *subprocess* launch will not trigger them as the child is not going to re-enter the interpreter.

Functions registered for execution before forking are called in reverse registration order. Functions registered for execution after forking (either in the parent or in the child) are called in registration order.

Note that `fork()` calls made by third-party C code may not call those functions, unless it explicitly calls `PyOS_BeforeFork()`, `PyOS_AfterFork_Parent()` and `PyOS_AfterFork_Child()`.

There is no way to unregister a function.

Availability: Unix, not WASI, not iOS.

Added in version 3.7.

os.**spawnl**(*mode*, *path*, *...*)

os.**spawnle**(*mode*, *path*, *...*, *env*)

os.**spawnlp**(*mode*, *file*, *...*)

os.**spawnlpe**(*mode*, *file*, *...*, *env*)

os.**spawnv**(*mode*, *path*, *args*)

os.**spawnve**(*mode*, *path*, *args*, *env*)

os.**spawnvp**(*mode*, *file*, *args*)

os.**spawnvpe**(*mode*, *file*, *args*, *env*)

Execute the program *path* in a new process.

(Note that the *subprocess* module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using these functions. Check especially the *Replacing Older Functions with the subprocess Module* section.)

If *mode* is *P_NOWAIT*, this function returns the process id of the new process; if *mode* is *P_WAIT*, returns the process's exit code if it exits normally, or `-signal`, where *signal* is the signal that killed the process. On Windows, the process id will actually be the process handle, so can be used with the *waitpid()* function.

Note on VxWorks, this function doesn't return `-signal` when the new process is killed. Instead it raises OSError exception.

The "l" and "v" variants of the *spawn** functions differ in how command-line arguments are passed. The "l" variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the `spawnl*()` functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the *args* parameter. In either case, the arguments to the child process must start with the name of the command being run.

The variants which include a second "p" near the end (*spawnlp()*, *spawnlpe()*, *spawnvp()*, and *spawnvpe()*) will use the `PATH` environment variable to locate the program *file*. When the environment is being replaced (using one of the *spawn*e* variants, discussed in the next paragraph), the new environment is used as the source of the `PATH` variable. The other variants, *spawnl()*, *spawnle()*, *spawnv()*, and *spawnve()*, will not use the `PATH` variable to locate the executable; *path* must contain an appropriate absolute or relative path.

For *spawnle()*, *spawnlpe()*, *spawnve()*, and *spawnvpe()* (note that these all end in "e"), the *env* parameter must be a mapping which is used to define the environment variables for the new process (they are used instead of the current process' environment); the functions *spawnl()*, *spawnlp()*, *spawnv()*, and *spawnvp()* all cause the new process to inherit the environment of the current process. Note that keys and values in the *env* dictionary must be strings; invalid keys or values will cause the function to fail, with a return value of `127`.

As an example, the following calls to *spawnlp()* and *spawnvpe()* are equivalent:

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

Raises an *auditing event* `os.spawn` with arguments `mode`, `path`, `args`, `env`.

Availability: Unix, Windows, not WASI, not iOS.

spawnlp(), *spawnlpe()*, *spawnvp()* and *spawnvpe()* are not available on Windows. *spawnle()* and *spawnve()* are not thread-safe on Windows; we advise you to use the *subprocess* module instead.

Changed in version 3.6: Accepts a *path-like object*.

os.**P_NOWAIT**

os.**P_NOWAITO**

Possible values for the *mode* parameter to the *spawn** family of functions. If either of these values is given, the *spawn** functions will return as soon as the new process has been created, with the process id as the return value.

Availability: Unix, Windows.

os.**P_WAIT**

Possible value for the *mode* parameter to the *spawn** family of functions. If this is given as *mode*, the *spawn** functions will not return until the new process has run to completion and will return the exit code of the process the run is successful, or `-signal` if a signal kills the process.

Availability: Unix, Windows.

os.**P_DETACH**

os.**P_OVERLAY**

Possible values for the *mode* parameter to the *spawn** family of functions. These are less portable than those listed above. *P_DETACH* is similar to *P_NOWAIT*, but the new process is detached from the console of the calling process. If *P_OVERLAY* is used, the current process will be replaced; the *spawn** function will not return.

Availability: Windows.

os.**startfile**(*path*[, *operation*][, *arguments*][, *cwd*][, *show_cmd*])

Start a file with its associated application.

When *operation* is not specified, this acts like double-clicking the file in Windows Explorer, or giving the file name as an argument to the **start** command from the interactive command shell: the file is opened with whatever application (if any) its extension is associated.

When another *operation* is given, it must be a "command verb" that specifies what should be done with the file. Common verbs documented by Microsoft are `'open'`, `'print'` and `'edit'` (to be used on files) as well as `'explore'` and `'find'` (to be used on directories).

When launching an application, specify *arguments* to be passed as a single string. This argument may have no effect when using this function to launch a document.

The default working directory is inherited, but may be overridden by the *cwd* argument. This should be an absolute path. A relative *path* will be resolved against this argument.

Use *show_cmd* to override the default window style. Whether this has any effect will depend on the application being launched. Values are integers as supported by the Win32 `ShellExecute()` function.

startfile() returns as soon as the associated application is launched. There is no option to wait for the application to close, and no way to retrieve the application's exit status. The *path* parameter is relative to the current directory or *cwd*. If you want to use an absolute path, make sure the first character is not a slash (`'/'`) Use *pathlib* or the *os.path.normpath()* function to ensure that paths are properly encoded for Win32.

To reduce interpreter startup overhead, the Win32 `ShellExecute()` function is not resolved until this function is first called. If the function cannot be resolved, *NotImplementedError* will be raised.

Raises an *auditing event* `os.startfile` with arguments `path`, `operation`.

Raises an *auditing event* `os.startfile/2` with arguments `path`, `operation`, `arguments`, `cwd`, `show_cmd`.

Availability: Windows.

Changed in version 3.10: Added the *arguments*, *cwd* and *show_cmd* arguments, and the `os.startfile/2` audit event.

os.**system**(*command*)

Execute the command (a string) in a subshell. This is implemented by calling the Standard C function `system()`, and has the same limitations. Changes to *sys.stdin*, etc. are not reflected in the environment of the executed command. If *command* generates any output, it will be sent to the interpreter standard output stream. The C standard does not specify the meaning of the return value of the C function, so the return value of the Python function is system-dependent.

On Unix, the return value is the exit status of the process encoded in the format specified for *wait()*.

On Windows, the return value is that returned by the system shell after running *command*. The shell is given by the Windows environment variable `COMSPEC`: it is usually **cmd.exe**, which returns the exit status of the command run; on systems using a non-native shell, consult your shell documentation.

The *subprocess* module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using this function. See the *Replacing Older Functions with the subprocess Module* section in the *subprocess* documentation for some helpful recipes.

On Unix, *waitstatus_to_exitcode()* can be used to convert the result (exit status) into an exit code. On Windows, the result is directly the exit code.

Raises an *auditing event* `os.system` with argument `command`.

Availability: Unix, Windows, not WASI, not iOS.

os.**times**()

Returns the current global process times. The return value is an object with five attributes:

- `user` - user time
- `system` - system time
- `children_user` - user time of all child processes
- `children_system` - system time of all child processes
- `elapsed` - elapsed real time since a fixed point in the past

For backwards compatibility, this object also behaves like a five-tuple containing `user`, `system`, `children_user`, `children_system`, and `elapsed` in that order.

See the Unix manual page *times(2)* and times(3) manual page on Unix or the GetProcessTimes MSDN on Windows. On Windows, only `user` and `system` are known; the other attributes are zero.

Availability: Unix, Windows.

Changed in version 3.3: Return type changed from a tuple to a tuple-like object with named attributes.

os.**wait**()

Wait for completion of a child process, and return a tuple containing its pid and exit status indication: a 16-bit number, whose low byte is the signal number that killed the process, and whose high byte is the exit status (if the signal number is zero); the high bit of the low byte is set if a core file was produced.

If there are no children that could be waited for, *ChildProcessError* is raised.

waitstatus_to_exitcode() can be used to convert the exit status into an exit code.

Availability: Unix, not WASI, not iOS.

See also

The other `wait*()` functions documented below can be used to wait for the completion of a specific child process and have more options. *waitpid()* is the only one also available on Windows.

os.**waitid**(*idtype*, *id*, *options*, /)

Wait for the completion of a child process.

idtype can be *P_PID*, *P_PGID*, *P_ALL*, or (on Linux) *P_PIDFD*. The interpretation of *id* depends on it; see their individual descriptions.

options is an OR combination of flags. At least one of *WEXITED*, *WSTOPPED* or *WCONTINUED* is required; *WNOHANG* and *WNOWAIT* are additional optional flags.

The return value is an object representing the data contained in the `siginfo_t` structure with the following attributes:

- `si_pid` (process ID)
- `si_uid` (real user ID of the child)
- `si_signo` (always *SIGCHLD*)
- `si_status` (the exit status or signal number, depending on `si_code`)
- `si_code` (see *CLD_EXITED* for possible values)

If *WNOHANG* is specified and there are no matching children in the requested state, `None` is returned. Otherwise, if there are no matching children that could be waited for, *ChildProcessError* is raised.

Availability: Unix, not WASI, not iOS.

Added in version 3.3.

Changed in version 3.13: This function is now available on macOS as well.

os.**waitpid**(*pid*, *options*, /)

The details of this function differ on Unix and Windows.

On Unix: Wait for completion of a child process given by process id *pid*, and return a tuple containing its process id and exit status indication (encoded as for *wait()*). The semantics of the call are affected by the value of the integer *options*, which should be `0` for normal operation.

If *pid* is greater than `0`, *waitpid()* requests status information for that specific process. If *pid* is `0`, the request is for the status of any child in the process group of the current process. If *pid* is `-1`, the request pertains to any child of the current process. If *pid* is less than `-1`, status is requested for any process in the process group `-pid` (the absolute value of *pid*).

options is an OR combination of flags. If it contains *WNOHANG* and there are no matching children in the requested state, `(0, 0)` is returned. Otherwise, if there are no matching children that could be waited for, *ChildProcessError* is raised. Other options that can be used are *WUNTRACED* and *WCONTINUED*.

On Windows: Wait for completion of a process given by process handle *pid*, and return a tuple containing *pid*, and its exit status shifted left by 8 bits (shifting makes cross-platform use of the function easier). A *pid* less than or equal to `0` has no special meaning on Windows, and raises an exception. The value of integer *options* has no effect. *pid* can refer to any process whose id is known, not necessarily a child process. The *spawn** functions called with *P_NOWAIT* return suitable process handles.

waitstatus_to_exitcode() can be used to convert the exit status into an exit code.

Availability: Unix, Windows, not WASI, not iOS.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

os.**wait3**(*options*)

Similar to *waitpid()*, except no process id argument is given and a 3-element tuple containing the child's process id, exit status indication, and resource usage information is returned. Refer to *resource.getrusage()* for details on resource usage information. The *options* argument is the same as that provided to *waitpid()* and *wait4()*.

waitstatus_to_exitcode() can be used to convert the exit status into an exitcode.

Availability: Unix, not WASI, not iOS.

os.**wait4**(*pid*, *options*)

Similar to *waitpid()*, except a 3-element tuple, containing the child's process id, exit status indication, and resource usage information is returned. Refer to *resource.getrusage()* for details on resource usage information. The arguments to *wait4()* are the same as those provided to *waitpid()*.

waitstatus_to_exitcode() can be used to convert the exit status into an exitcode.

Availability: Unix, not WASI, not iOS.

os.**P_PID**

os.**P_PGID**

os.**P_ALL**

os.**P_PIDFD**

These are the possible values for *idtype* in *waitid()*. They affect how *id* is interpreted:

- P_PID - wait for the child whose PID is *id*.
- P_PGID - wait for any child whose progress group ID is *id*.
- P_ALL - wait for any child; *id* is ignored.
- P_PIDFD - wait for the child identified by the file descriptor *id* (a process file descriptor created with *pidfd_open()*).

Availability: Unix, not WASI, not iOS.

> **Note**
>
> P_PIDFD is only available on Linux >= 5.4.

Added in version 3.3.

Added in version 3.9: The P_PIDFD constant.

os.**WCONTINUED**

This *options* flag for *waitpid()*, *wait3()*, *wait4()*, and *waitid()* causes child processes to be reported if they have been continued from a job control stop since they were last reported.

Availability: Unix, not WASI, not iOS.

os.**WEXITED**

This *options* flag for *waitid()* causes child processes that have terminated to be reported.

The other wait* functions always report children that have terminated, so this option is not available for them.

Availability: Unix, not WASI, not iOS.

Added in version 3.3.

os.**WSTOPPED**

This *options* flag for *waitid()* causes child processes that have been stopped by the delivery of a signal to be reported.

This option is not available for the other `wait*` functions.

Availability: Unix, not WASI, not iOS.

Added in version 3.3.

os.**WUNTRACED**

This *options* flag for *waitpid()*, *wait3()*, and *wait4()* causes child processes to also be reported if they have been stopped but their current state has not been reported since they were stopped.

This option is not available for *waitid()*.

Availability: Unix, not WASI, not iOS.

os.**WNOHANG**

This *options* flag causes *waitpid()*, *wait3()*, *wait4()*, and *waitid()* to return right away if no child process status is available immediately.

Availability: Unix, not WASI, not iOS.

os.**WNOWAIT**

This *options* flag causes *waitid()* to leave the child in a waitable state, so that a later `wait*()` call can be used to retrieve the child status information again.

This option is not available for the other `wait*` functions.

Availability: Unix, not WASI, not iOS.

os.**CLD_EXITED**

os.**CLD_KILLED**

os.**CLD_DUMPED**

os.**CLD_TRAPPED**

os.**CLD_STOPPED**

os.**CLD_CONTINUED**

These are the possible values for `si_code` in the result returned by *waitid()*.

Availability: Unix, not WASI, not iOS.

Added in version 3.3.

Changed in version 3.9: Added *CLD_KILLED* and *CLD_STOPPED* values.

os.**waitstatus_to_exitcode**(*status*)

Convert a wait status to an exit code.

On Unix:

- If the process exited normally (if `WIFEXITED(status)` is true), return the process exit status (return `WEXITSTATUS(status)`): result greater than or equal to 0.
- If the process was terminated by a signal (if `WIFSIGNALED(status)` is true), return `-signum` where *signum* is the number of the signal that caused the process to terminate (return `-WTERMSIG(status)`): result less than 0.
- Otherwise, raise a *ValueError*.

On Windows, return *status* shifted right by 8 bits.

On Unix, if the process is being traced or if *waitpid()* was called with *WUNTRACED* option, the caller must first check if `WIFSTOPPED(status)` is true. This function must not be called if `WIFSTOPPED(status)` is true.

See also

WIFEXITED(), *WEXITSTATUS()*, *WIFSIGNALED()*, *WTERMSIG()*, *WIFSTOPPED()*, *WSTOPSIG()* functions.

Availability: Unix, Windows, not WASI, not iOS.

Added in version 3.9.

The following functions take a process status code as returned by *system()*, *wait()*, or *waitpid()* as a parameter. They may be used to determine the disposition of a process.

`os.`**`WCOREDUMP`**(*status*, `/`)

Return `True` if a core dump was generated for the process, otherwise return `False`.

This function should be employed only if *WIFSIGNALED()* is true.

Availability: Unix, not WASI, not iOS.

`os.`**`WIFCONTINUED`**(*status*)

Return `True` if a stopped child has been resumed by delivery of *SIGCONT* (if the process has been continued from a job control stop), otherwise return `False`.

See *WCONTINUED* option.

Availability: Unix, not WASI, not iOS.

`os.`**`WIFSTOPPED`**(*status*)

Return `True` if the process was stopped by delivery of a signal, otherwise return `False`.

WIFSTOPPED() only returns `True` if the *waitpid()* call was done using *WUNTRACED* option or when the process is being traced (see *ptrace(2)*).

Availability: Unix, not WASI, not iOS.

`os.`**`WIFSIGNALED`**(*status*)

Return `True` if the process was terminated by a signal, otherwise return `False`.

Availability: Unix, not WASI, not iOS.

`os.`**`WIFEXITED`**(*status*)

Return `True` if the process exited terminated normally, that is, by calling `exit()` or `_exit()`, or by returning from `main()`; otherwise return `False`.

Availability: Unix, not WASI, not iOS.

`os.`**`WEXITSTATUS`**(*status*)

Return the process exit status.

This function should be employed only if *WIFEXITED()* is true.

Availability: Unix, not WASI, not iOS.

os.**WSTOPSIG**(*status*)

Return the signal which caused the process to stop.

This function should be employed only if *WIFSTOPPED()* is true.

Availability: Unix, not WASI, not iOS.

os.**WTERMSIG**(*status*)

Return the number of the signal that caused the process to terminate.

This function should be employed only if *WIFSIGNALED()* is true.

Availability: Unix, not WASI, not iOS.

16.1.8 Interface to the scheduler

These functions control how a process is allocated CPU time by the operating system. They are only available on some Unix platforms. For more detailed information, consult your Unix manpages.

Added in version 3.3.

The following scheduling policies are exposed if they are supported by the operating system.

os.**SCHED_OTHER**

The default scheduling policy.

os.**SCHED_BATCH**

Scheduling policy for CPU-intensive processes that tries to preserve interactivity on the rest of the computer.

os.**SCHED_IDLE**

Scheduling policy for extremely low priority background tasks.

os.**SCHED_SPORADIC**

Scheduling policy for sporadic server programs.

os.**SCHED_FIFO**

A First In First Out scheduling policy.

os.**SCHED_RR**

A round-robin scheduling policy.

os.**SCHED_RESET_ON_FORK**

This flag can be OR'ed with any other scheduling policy. When a process with this flag set forks, its child's scheduling policy and priority are reset to the default.

class os.**sched_param**(*sched_priority*)

This class represents tunable scheduling parameters used in *sched_setparam()*, *sched_setscheduler()*, and *sched_getparam()*. It is immutable.

At the moment, there is only one possible parameter:

sched_priority

The scheduling priority for a scheduling policy.

os.**sched_get_priority_min**(*policy*)

Get the minimum priority value for *policy*. *policy* is one of the scheduling policy constants above.

os.**sched_get_priority_max**(*policy*)

Get the maximum priority value for *policy*. *policy* is one of the scheduling policy constants above.

os.**sched_setscheduler**(*pid*, *policy*, *param*, /)

Set the scheduling policy for the process with PID *pid*. A *pid* of 0 means the calling process. *policy* is one of the scheduling policy constants above. *param* is a *sched_param* instance.

os.**sched_getscheduler**(*pid*, /)

Return the scheduling policy for the process with PID *pid*. A *pid* of 0 means the calling process. The result is one of the scheduling policy constants above.

os.**sched_setparam**(*pid*, *param*, /)

Set the scheduling parameters for the process with PID *pid*. A *pid* of 0 means the calling process. *param* is a *sched_param* instance.

os.**sched_getparam**(*pid*, /)

Return the scheduling parameters as a *sched_param* instance for the process with PID *pid*. A *pid* of 0 means the calling process.

os.**sched_rr_get_interval**(*pid*, /)

Return the round-robin quantum in seconds for the process with PID *pid*. A *pid* of 0 means the calling process.

os.**sched_yield**()

Voluntarily relinquish the CPU.

os.**sched_setaffinity**(*pid*, *mask*, /)

Restrict the process with PID *pid* (or the current process if zero) to a set of CPUs. *mask* is an iterable of integers representing the set of CPUs to which the process should be restricted.

os.**sched_getaffinity**(*pid*, /)

Return the set of CPUs the process with PID *pid* is restricted to.

If *pid* is zero, return the set of CPUs the calling thread of the current process is restricted to.

See also the *process_cpu_count()* function.

16.1.9 Miscellaneous System Information

os.**confstr**(*name*, /)

Return string-valued system configuration values. *name* specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given as the keys of the `confstr_names` dictionary. For configuration variables not included in that mapping, passing an integer for *name* is also accepted.

If the configuration value specified by *name* isn't defined, `None` is returned.

If *name* is a string and is not known, *ValueError* is raised. If a specific value for *name* is not supported by the host system, even if it is included in `confstr_names`, an *OSError* is raised with *errno.EINVAL* for the error number.

Availability: Unix.

os.**confstr_names**

Dictionary mapping names accepted by *confstr()* to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

Availability: Unix.

os.**cpu_count**()

Return the number of logical CPUs in the **system**. Returns None if undetermined.

The *process_cpu_count()* function can be used to get the number of logical CPUs usable by the calling thread of the **current process**.

Added in version 3.4.

Changed in version 3.13: If -X cpu_count is given or PYTHON_CPU_COUNT is set, *cpu_count()* returns the overridden value *n*.

os.**getloadavg**()

Return the number of processes in the system run queue averaged over the last 1, 5, and 15 minutes or raises *OSError* if the load average was unobtainable.

Availability: Unix.

os.**process_cpu_count**()

Get the number of logical CPUs usable by the calling thread of the **current process**. Returns None if undetermined. It can be less than *cpu_count()* depending on the CPU affinity.

The *cpu_count()* function can be used to get the number of logical CPUs in the **system**.

If -X cpu_count is given or PYTHON_CPU_COUNT is set, *process_cpu_count()* returns the overridden value *n*.

See also the *sched_getaffinity()* functions.

Added in version 3.13.

os.**sysconf**(*name*, /)

Return integer-valued system configuration values. If the configuration value specified by *name* isn't defined, -1 is returned. The comments regarding the *name* parameter for *confstr()* apply here as well; the dictionary that provides information on the known names is given by sysconf_names.

Availability: Unix.

os.**sysconf_names**

Dictionary mapping names accepted by *sysconf()* to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

Availability: Unix.

Changed in version 3.11: Add 'SC_MINSIGSTKSZ' name.

The following data values are used to support path manipulation operations. These are defined for all platforms.

Higher-level operations on pathnames are defined in the *os.path* module.

os.**curdir**

The constant string used by the operating system to refer to the current directory. This is '.' for Windows and POSIX. Also available via *os.path*.

os.**pardir**

The constant string used by the operating system to refer to the parent directory. This is '..' for Windows and POSIX. Also available via *os.path*.

os.**sep**

The character used by the operating system to separate pathname components. This is '/' for POSIX and '\\' for Windows. Note that knowing this is not sufficient to be able to parse or concatenate pathnames — use *os.path.split()* and *os.path.join()* — but it is occasionally useful. Also available via *os.path*.

os.**altsep**

An alternative character used by the operating system to separate pathname components, or `None` if only one separator character exists. This is set to `'/'` on Windows systems where `sep` is a backslash. Also available via *os.path*.

os.**extsep**

The character which separates the base filename from the extension; for example, the `'.'` in `os.py`. Also available via *os.path*.

os.**pathsep**

The character conventionally used by the operating system to separate search path components (as in `PATH`), such as `':'` for POSIX or `';'` for Windows. Also available via *os.path*.

os.**defpath**

The default search path used by *exec*p** and *spawn*p** if the environment doesn't have a `'PATH'` key. Also available via *os.path*.

os.**linesep**

The string used to separate (or, rather, terminate) lines on the current platform. This may be a single character, such as `'\n'` for POSIX, or multiple characters, for example, `'\r\n'` for Windows. Do not use *os.linesep* as a line terminator when writing files opened in text mode (the default); use a single `'\n'` instead, on all platforms.

os.**devnull**

The file path of the null device. For example: `'/dev/null'` for POSIX, `'nul'` for Windows. Also available via *os.path*.

os.**RTLD_LAZY**

os.**RTLD_NOW**

os.**RTLD_GLOBAL**

os.**RTLD_LOCAL**

os.**RTLD_NODELETE**

os.**RTLD_NOLOAD**

os.**RTLD_DEEPBIND**

Flags for use with the *setdlopenflags()* and *getdlopenflags()* functions. See the Unix manual page *dlopen(3)* for what the different flags mean.

Added in version 3.3.

16.1.10 Random numbers

os.**getrandom**(*size*, *flags=0*)

Get up to *size* random bytes. The function can return less bytes than requested.

These bytes can be used to seed user-space random number generators or for cryptographic purposes.

`getrandom()` relies on entropy gathered from device drivers and other sources of environmental noise. Unnecessarily reading large quantities of data will have a negative impact on other users of the `/dev/random` and `/dev/urandom` devices.

The flags argument is a bit mask that can contain zero or more of the following values ORed together: *os.GRND_RANDOM* and *GRND_NONBLOCK*.

See also the Linux getrandom() manual page.

Availability: Linux >= 3.17.

Added in version 3.6.

os.**urandom**(*size*, /)

Return a bytestring of *size* random bytes suitable for cryptographic use.

This function returns random bytes from an OS-specific randomness source. The returned data should be unpredictable enough for cryptographic applications, though its exact quality depends on the OS implementation.

On Linux, if the `getrandom()` syscall is available, it is used in blocking mode: block until the system urandom entropy pool is initialized (128 bits of entropy are collected by the kernel). See the **PEP 524** for the rationale. On Linux, the *getrandom()* function can be used to get random bytes in non-blocking mode (using the *GRND_NONBLOCK* flag) or to poll until the system urandom entropy pool is initialized.

On a Unix-like system, random bytes are read from the `/dev/urandom` device. If the `/dev/urandom` device is not available or not readable, the *NotImplementedError* exception is raised.

On Windows, it will use `BCryptGenRandom()`.

See also

The *secrets* module provides higher level functions. For an easy-to-use interface to the random number generator provided by your platform, please see *random.SystemRandom*.

Changed in version 3.5: On Linux 3.17 and newer, the `getrandom()` syscall is now used when available. On OpenBSD 5.6 and newer, the C `getentropy()` function is now used. These functions avoid the usage of an internal file descriptor.

Changed in version 3.5.2: On Linux, if the `getrandom()` syscall blocks (the urandom entropy pool is not initialized yet), fall back on reading `/dev/urandom`.

Changed in version 3.6: On Linux, `getrandom()` is now used in blocking mode to increase the security.

Changed in version 3.11: On Windows, `BCryptGenRandom()` is used instead of `CryptGenRandom()` which is deprecated.

os.**GRND_NONBLOCK**

By default, when reading from `/dev/random`, *getrandom()* blocks if no random bytes are available, and when reading from `/dev/urandom`, it blocks if the entropy pool has not yet been initialized.

If the *GRND_NONBLOCK* flag is set, then *getrandom()* does not block in these cases, but instead immediately raises *BlockingIOError*.

Added in version 3.6.

os.**GRND_RANDOM**

If this bit is set, then random bytes are drawn from the `/dev/random` pool instead of the `/dev/urandom` pool.

Added in version 3.6.

16.2 io — Core tools for working with streams

Source code: Lib/io.py

16.2.1 Overview

The *io* module provides Python's main facilities for dealing with various types of I/O. There are three main types of I/O: *text I/O*, *binary I/O* and *raw I/O*. These are generic categories, and various backing stores can be used for each of them. A concrete object belonging to any of these categories is called a *file object*. Other common terms are *stream* and *file-like object*.

Independent of its category, each concrete stream object will also have various capabilities: it can be read-only, write-only, or read-write. It can also allow arbitrary random access (seeking forwards or backwards to any location), or only sequential access (for example in the case of a socket or pipe).

All streams are careful about the type of data you give to them. For example giving a *str* object to the `write()` method of a binary stream will raise a *TypeError*. So will giving a *bytes* object to the `write()` method of a text stream.

Changed in version 3.3: Operations that used to raise *IOError* now raise *OSError*, since *IOError* is now an alias of *OSError*.

Text I/O

Text I/O expects and produces *str* objects. This means that whenever the backing store is natively made of bytes (such as in the case of a file), encoding and decoding of data is made transparently as well as optional translation of platform-specific newline characters.

The easiest way to create a text stream is with *open()*, optionally specifying an encoding:

```
f = open("myfile.txt", "r", encoding="utf-8")
```

In-memory text streams are also available as *StringIO* objects:

```
f = io.StringIO("some initial text data")
```

The text stream API is described in detail in the documentation of *TextIOBase*.

Binary I/O

Binary I/O (also called *buffered I/O*) expects *bytes-like objects* and produces *bytes* objects. No encoding, decoding, or newline translation is performed. This category of streams can be used for all kinds of non-text data, and also when manual control over the handling of text data is desired.

The easiest way to create a binary stream is with *open()* with `'b'` in the mode string:

```
f = open("myfile.jpg", "rb")
```

In-memory binary streams are also available as *BytesIO* objects:

```
f = io.BytesIO(b"some initial binary data: \x00\x01")
```

The binary stream API is described in detail in the docs of *BufferedIOBase*.

Other library modules may provide additional ways to create text or binary streams. See *socket.socket.makefile()* for example.

Raw I/O

Raw I/O (also called *unbuffered I/O*) is generally used as a low-level building-block for binary and text streams; it is rarely useful to directly manipulate a raw stream from user code. Nevertheless, you can create a raw stream by opening a file in binary mode with buffering disabled:

```
f = open("myfile.jpg", "rb", buffering=0)
```

The raw stream API is described in detail in the docs of *RawIOBase*.

16.2.2 Text Encoding

The default encoding of *TextIOWrapper* and *open()* is locale-specific (*locale.getencoding()*).

However, many developers forget to specify the encoding when opening text files encoded in UTF-8 (e.g. JSON, TOML, Markdown, etc…) since most Unix platforms use UTF-8 locale by default. This causes bugs because the locale encoding is not UTF-8 for most Windows users. For example:

```
# May not work on Windows when non-ASCII characters in the file.
with open("README.md") as f:
    long_description = f.read()
```

Accordingly, it is highly recommended that you specify the encoding explicitly when opening text files. If you want to use UTF-8, pass `encoding="utf-8"`. To use the current locale encoding, `encoding="locale"` is supported since Python 3.10.

See also

Python UTF-8 Mode
Python UTF-8 Mode can be used to change the default encoding to UTF-8 from locale-specific encoding.

PEP 686
Python 3.15 will make *Python UTF-8 Mode* default.

Opt-in EncodingWarning

Added in version 3.10: See **PEP 597** for more details.

To find where the default locale encoding is used, you can enable the `-X warn_default_encoding` command line option or set the `PYTHONWARNDEFAULTENCODING` environment variable, which will emit an *EncodingWarning* when the default encoding is used.

If you are providing an API that uses *open()* or *TextIOWrapper* and passes `encoding=None` as a parameter, you can use *text_encoding()* so that callers of the API will emit an *EncodingWarning* if they don't pass an `encoding`. However, please consider using UTF-8 by default (i.e. `encoding="utf-8"`) for new APIs.

16.2.3 High-level Module Interface

io.**DEFAULT_BUFFER_SIZE**

An int containing the default buffer size used by the module's buffered I/O classes. *open()* uses the file's blksize (as obtained by *os.stat()*) if possible.

io.**open**(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)

This is an alias for the builtin *open()* function.

This function raises an *auditing event* `open` with arguments *path*, *mode* and *flags*. The *mode* and *flags* arguments may have been modified or inferred from the original call.

io.**open_code**(*path*)

Opens the provided file with mode `'rb'`. This function should be used when the intent is to treat the contents as executable code.

path should be a *str* and an absolute path.

The behavior of this function may be overridden by an earlier call to the `PyFile_SetOpenCodeHook()`. However, assuming that *path* is a *str* and an absolute path, `open_code(path)` should always behave the same as `open(path, 'rb')`. Overriding the behavior is intended for additional validation or preprocessing of the file.

Added in version 3.8.

io.**text_encoding**(*encoding*, *stacklevel=2*, */*)

This is a helper function for callables that use *open()* or *TextIOWrapper* and have an `encoding=None` parameter.

This function returns *encoding* if it is not `None`. Otherwise, it returns `"locale"` or `"utf-8"` depending on *UTF-8 Mode*.

This function emits an *EncodingWarning* if *sys.flags.warn_default_encoding* is true and *encoding* is `None`. *stacklevel* specifies where the warning is emitted. For example:

```
def read_text(path, encoding=None):
    encoding = io.text_encoding(encoding)  # stacklevel=2
    with open(path, encoding) as f:
        return f.read()
```

In this example, an *EncodingWarning* is emitted for the caller of `read_text()`.

See *Text Encoding* for more information.

Added in version 3.10.

Changed in version 3.11: *text_encoding()* returns "utf-8" when UTF-8 mode is enabled and *encoding* is `None`.

exception io.**BlockingIOError**

This is a compatibility alias for the builtin *BlockingIOError* exception.

exception io.**UnsupportedOperation**

An exception inheriting *OSError* and *ValueError* that is raised when an unsupported operation is called on a stream.

See also

sys
: contains the standard IO streams: *sys.stdin*, *sys.stdout*, and *sys.stderr*.

16.2.4 Class hierarchy

The implementation of I/O streams is organized as a hierarchy of classes. First *abstract base classes* (ABCs), which are used to specify the various categories of streams, then concrete classes providing the standard stream implementations.

Note

The abstract base classes also provide default implementations of some methods in order to help implementation of concrete stream classes. For example, *BufferedIOBase* provides unoptimized implementations of `readinto()` and `readline()`.

At the top of the I/O hierarchy is the abstract base class *IOBase*. It defines the basic interface to a stream. Note, however, that there is no separation between reading and writing to streams; implementations are allowed to raise *UnsupportedOperation* if they do not support a given operation.

The *RawIOBase* ABC extends *IOBase*. It deals with the reading and writing of bytes to a stream. *FileIO* subclasses *RawIOBase* to provide an interface to files in the machine's file system.

The *BufferedIOBase* ABC extends *IOBase*. It deals with buffering on a raw binary stream (*RawIOBase*). Its subclasses, *BufferedWriter*, *BufferedReader*, and *BufferedRWPair* buffer raw binary streams that are writable, readable, and both readable and writable, respectively. *BufferedRandom* provides a buffered interface to seekable streams. Another *BufferedIOBase* subclass, *BytesIO*, is a stream of in-memory bytes.

The *TextIOBase* ABC extends *IOBase*. It deals with streams whose bytes represent text, and handles encoding and decoding to and from strings. *TextIOWrapper*, which extends *TextIOBase*, is a buffered text interface to a buffered raw stream (*BufferedIOBase*). Finally, *StringIO* is an in-memory stream for text.

Argument names are not part of the specification, and only the arguments of *open()* are intended to be used as keyword arguments.

The following table summarizes the ABCs provided by the *io* module:

ABC	Inherits	Stub Methods	Mixin Methods and Properties
IOBase		`fileno`, `seek`, and `truncate`	`close`, `closed`, `__enter__`, `__exit__`, `flush`, `isatty`, `__iter__`, `__next__`, `readable`, `readline`, `readlines`, `seekable`, `tell`, `writable`, and `writelines`
RawIOBase	*IOBase*	`readinto` and `write`	Inherited *IOBase* methods, `read`, and `readall`
BufferedIOBase	*IOBase*	`detach`, `read`, `read1`, and `write`	Inherited *IOBase* methods, `readinto`, and `readinto1`
TextIOBase	*IOBase*	`detach`, `read`, `readline`, and `write`	Inherited *IOBase* methods, `encoding`, `errors`, and `newlines`

I/O Base Classes

class io.**IOBase**

The abstract base class for all I/O classes.

This class provides empty abstract implementations for many methods that derived classes can override selectively; the default implementations represent a file that cannot be read, written or seeked.

Even though *IOBase* does not declare `read()` or `write()` because their signatures will vary, implementations and clients should consider those methods part of the interface. Also, implementations may raise a *ValueError* (or *UnsupportedOperation*) when operations they do not support are called.

The basic type used for binary data read from or written to a file is *bytes*. Other *bytes-like objects* are accepted as method arguments too. Text I/O classes work with *str* data.

Note that calling any method (even inquiries) on a closed stream is undefined. Implementations may raise *ValueError* in this case.

IOBase (and its subclasses) supports the iterator protocol, meaning that an *IOBase* object can be iterated over yielding the lines in a stream. Lines are defined slightly differently depending on whether the stream is a binary stream (yielding bytes), or a text stream (yielding character strings). See *readline()* below.

IOBase is also a context manager and therefore supports the `with` statement. In this example, *file* is closed after the `with` statement's suite is finished—even if an exception occurs:

```
with open('spam.txt', 'w') as file:
    file.write('Spam and eggs!')
```

IOBase provides these data attributes and methods:

close()

Flush and close this stream. This method has no effect if the file is already closed. Once the file is closed, any operation on the file (e.g. reading or writing) will raise a *ValueError*.

As a convenience, it is allowed to call this method more than once; only the first call, however, will have an effect.

closed

`True` if the stream is closed.

fileno()

Return the underlying file descriptor (an integer) of the stream if it exists. An *OSError* is raised if the IO object does not use a file descriptor.

flush()

Flush the write buffers of the stream if applicable. This does nothing for read-only and non-blocking streams.

isatty()

Return `True` if the stream is interactive (i.e., connected to a terminal/tty device).

readable()

Return `True` if the stream can be read from. If `False`, `read()` will raise *OSError*.

readline(*size=-1*, */*)

Read and return one line from the stream. If *size* is specified, at most *size* bytes will be read.

The line terminator is always `b'\n'` for binary files; for text files, the *newline* argument to *open()* can be used to select the line terminator(s) recognized.

readlines(*hint=-1*, /)

Read and return a list of lines from the stream. *hint* can be specified to control the number of lines read: no more lines will be read if the total size (in bytes/characters) of all lines so far exceeds *hint*.

hint values of `0` or less, as well as `None`, are treated as no hint.

Note that it's already possible to iterate on file objects using `for line in file: ...` without calling `file.readlines()`.

seek(*offset*, *whence=os.SEEK_SET*, /)

Change the stream position to the given byte *offset*, interpreted relative to the position indicated by *whence*, and return the new absolute position. Values for *whence* are:

- *os.SEEK_SET* or `0` – start of the stream (the default); *offset* should be zero or positive
- *os.SEEK_CUR* or `1` – current stream position; *offset* may be negative
- *os.SEEK_END* or `2` – end of the stream; *offset* is usually negative

Added in version 3.1: The `SEEK_*` constants.

Added in version 3.3: Some operating systems could support additional values, like *os.SEEK_HOLE* or *os.SEEK_DATA*. The valid values for a file could depend on it being open in text or binary mode.

seekable()

Return `True` if the stream supports random access. If `False`, *seek()*, *tell()* and *truncate()* will raise *OSError*.

tell()

Return the current stream position.

truncate(*size=None*, /)

Resize the stream to the given *size* in bytes (or the current position if *size* is not specified). The current stream position isn't changed. This resizing can extend or reduce the current file size. In case of extension, the contents of the new file area depend on the platform (on most systems, additional bytes are zero-filled). The new file size is returned.

Changed in version 3.5: Windows will now zero-fill files when extending.

writable()

Return `True` if the stream supports writing. If `False`, `write()` and *truncate()* will raise *OSError*.

writelines(*lines*, /)

Write a list of lines to the stream. Line separators are not added, so it is usual for each of the lines provided to have a line separator at the end.

__del__()

Prepare for object destruction. *IOBase* provides a default implementation of this method that calls the instance's *close()* method.

class io.**RawIOBase**

Base class for raw binary streams. It inherits from *IOBase*.

Raw binary streams typically provide low-level access to an underlying OS device or API, and do not try to encapsulate it in high-level primitives (this functionality is done at a higher-level in buffered binary streams and text streams, described later in this page).

RawIOBase provides these methods in addition to those from *IOBase*:

read(*size=-1*, */*)

Read up to *size* bytes from the object and return them. As a convenience, if *size* is unspecified or -1, all bytes until EOF are returned. Otherwise, only one system call is ever made. Fewer than *size* bytes may be returned if the operating system call returns fewer than *size* bytes.

If 0 bytes are returned, and *size* was not 0, this indicates end of file. If the object is in non-blocking mode and no bytes are available, `None` is returned.

The default implementation defers to *readall()* and *readinto()*.

readall()

Read and return all the bytes from the stream until EOF, using multiple calls to the stream if necessary.

readinto(*b*, */*)

Read bytes into a pre-allocated, writable *bytes-like object* *b*, and return the number of bytes read. For example, *b* might be a *bytearray*. If the object is in non-blocking mode and no bytes are available, `None` is returned.

write(*b*, */*)

Write the given *bytes-like object*, *b*, to the underlying raw stream, and return the number of bytes written. This can be less than the length of *b* in bytes, depending on specifics of the underlying raw stream, and especially if it is in non-blocking mode. `None` is returned if the raw stream is set not to block and no single byte could be readily written to it. The caller may release or mutate *b* after this method returns, so the implementation should only access *b* during the method call.

class io.**BufferedIOBase**

Base class for binary streams that support some kind of buffering. It inherits from *IOBase*.

The main difference with *RawIOBase* is that methods *read()*, *readinto()* and *write()* will try (respectively) to read as much input as requested or to consume all given output, at the expense of making perhaps more than one system call.

In addition, those methods can raise *BlockingIOError* if the underlying raw stream is in non-blocking mode and cannot take or give enough data; unlike their *RawIOBase* counterparts, they will never return `None`.

Besides, the *read()* method does not have a default implementation that defers to *readinto()*.

A typical *BufferedIOBase* implementation should not inherit from a *RawIOBase* implementation, but wrap one, like *BufferedWriter* and *BufferedReader* do.

BufferedIOBase provides or overrides these data attributes and methods in addition to those from *IOBase*:

raw

The underlying raw stream (a *RawIOBase* instance) that *BufferedIOBase* deals with. This is not part of the *BufferedIOBase* API and may not exist on some implementations.

detach()

Separate the underlying raw stream from the buffer and return it.

After the raw stream has been detached, the buffer is in an unusable state.

Some buffers, like *BytesIO*, do not have the concept of a single raw stream to return from this method. They raise *UnsupportedOperation*.

Added in version 3.1.

read(*size=-1*, */*)

Read and return up to *size* bytes. If the argument is omitted, `None`, or negative, data is read and returned until EOF is reached. An empty *bytes* object is returned if the stream is already at EOF.

If the argument is positive, and the underlying raw stream is not interactive, multiple raw reads may be issued to satisfy the byte count (unless EOF is reached first). But for interactive raw streams, at most one raw read will be issued, and a short result does not imply that EOF is imminent.

A `BlockingIOError` is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

read1(*size=-1*, */*)

Read and return up to *size* bytes, with at most one call to the underlying raw stream's `read()` (or `readinto()`) method. This can be useful if you are implementing your own buffering on top of a `BufferedIOBase` object.

If *size* is `-1` (the default), an arbitrary number of bytes are returned (more than zero unless EOF is reached).

readinto(*b*, */*)

Read bytes into a pre-allocated, writable *bytes-like object* *b* and return the number of bytes read. For example, *b* might be a `bytearray`.

Like `read()`, multiple reads may be issued to the underlying raw stream, unless the latter is interactive.

A `BlockingIOError` is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

readinto1(*b*, */*)

Read bytes into a pre-allocated, writable *bytes-like object* *b*, using at most one call to the underlying raw stream's `read()` (or `readinto()`) method. Return the number of bytes read.

A `BlockingIOError` is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

Added in version 3.5.

write(*b*, */*)

Write the given *bytes-like object*, *b*, and return the number of bytes written (always equal to the length of *b* in bytes, since if the write fails an `OSError` will be raised). Depending on the actual implementation, these bytes may be readily written to the underlying stream, or held in a buffer for performance and latency reasons.

When in non-blocking mode, a `BlockingIOError` is raised if the data needed to be written to the raw stream but it couldn't accept all the data without blocking.

The caller may release or mutate *b* after this method returns, so the implementation should only access *b* during the method call.

Raw File I/O

class io.**FileIO**(*name*, *mode='r'*, *closefd=True*, *opener=None*)

A raw binary stream representing an OS-level file containing bytes data. It inherits from `RawIOBase`.

The *name* can be one of two things:

- a character string or `bytes` object representing the path to the file which will be opened. In this case closefd must be `True` (the default) otherwise an error will be raised.
- an integer representing the number of an existing OS-level file descriptor to which the resulting `FileIO` object will give access. When the FileIO object is closed this fd will be closed as well, unless *closefd* is set to `False`.

The *mode* can be `'r'`, `'w'`, `'x'` or `'a'` for reading (default), writing, exclusive creation or appending. The file will be created if it doesn't exist when opened for writing or appending; it will be truncated when opened for writing. `FileExistsError` will be raised if it already exists when opened for creating. Opening a file

for creating implies writing, so this mode behaves in a similar way to `'w'`. Add a `'+'` to the mode to allow simultaneous reading and writing.

The *read()* (when called with a positive argument), *readinto()* and *write()* methods on this class will only make one system call.

A custom opener can be used by passing a callable as *opener*. The underlying file descriptor for the file object is then obtained by calling *opener* with (*name*, *flags*). *opener* must return an open file descriptor (passing *os.open* as *opener* results in functionality similar to passing `None`).

The newly created file is *non-inheritable*.

See the *open()* built-in function for examples on using the *opener* parameter.

Changed in version 3.3: The *opener* parameter was added. The `'x'` mode was added.

Changed in version 3.4: The file is now non-inheritable.

FileIO provides these data attributes in addition to those from *RawIOBase* and *IOBase*:

mode

The mode as given in the constructor.

name

The file name. This is the file descriptor of the file when no name is given in the constructor.

Buffered Streams

Buffered I/O streams provide a higher-level interface to an I/O device than raw I/O does.

class io.**BytesIO**(*initial_bytes=b''*)

A binary stream using an in-memory bytes buffer. It inherits from *BufferedIOBase*. The buffer is discarded when the *close()* method is called.

The optional argument *initial_bytes* is a *bytes-like object* that contains initial data.

BytesIO provides or overrides these methods in addition to those from *BufferedIOBase* and *IOBase*:

getbuffer()

Return a readable and writable view over the contents of the buffer without copying them. Also, mutating the view will transparently update the contents of the buffer:

```
>>> b = io.BytesIO(b"abcdef")
>>> view = b.getbuffer()
>>> view[2:4] = b"56"
>>> b.getvalue()
b'ab56ef'
```

> **Note**
>
> As long as the view exists, the *BytesIO* object cannot be resized or closed.

Added in version 3.2.

getvalue()

Return *bytes* containing the entire contents of the buffer.

read1(*size=-1*, */*)

In *BytesIO*, this is the same as *read()*.

Changed in version 3.7: The *size* argument is now optional.

readinto1(*b*, */*)

In *BytesIO*, this is the same as *readinto()*.

Added in version 3.5.

class io.**BufferedReader**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

A buffered binary stream providing higher-level access to a readable, non seekable *RawIOBase* raw binary stream. It inherits from *BufferedIOBase*.

When reading data from this object, a larger amount of data may be requested from the underlying raw stream, and kept in an internal buffer. The buffered data can then be returned directly on subsequent reads.

The constructor creates a *BufferedReader* for the given readable *raw* stream and *buffer_size*. If *buffer_size* is omitted, *DEFAULT_BUFFER_SIZE* is used.

BufferedReader provides or overrides these methods in addition to those from *BufferedIOBase* and *IOBase*:

peek(*size=0*, */*)

Return bytes from the stream without advancing the position. At most one single read on the raw stream is done to satisfy the call. The number of bytes returned may be less or more than requested.

read(*size=-1*, */*)

Read and return *size* bytes, or if *size* is not given or negative, until EOF or if the read call would block in non-blocking mode.

read1(*size=-1*, */*)

Read and return up to *size* bytes with only one call on the raw stream. If at least one byte is buffered, only buffered bytes are returned. Otherwise, one raw stream read call is made.

Changed in version 3.7: The *size* argument is now optional.

class io.**BufferedWriter**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

A buffered binary stream providing higher-level access to a writeable, non seekable *RawIOBase* raw binary stream. It inherits from *BufferedIOBase*.

When writing to this object, data is normally placed into an internal buffer. The buffer will be written out to the underlying *RawIOBase* object under various conditions, including:

- when the buffer gets too small for all pending data;
- when *flush()* is called;
- when a *seek()* is requested (for *BufferedRandom* objects);
- when the *BufferedWriter* object is closed or destroyed.

The constructor creates a *BufferedWriter* for the given writeable *raw* stream. If the *buffer_size* is not given, it defaults to *DEFAULT_BUFFER_SIZE*.

BufferedWriter provides or overrides these methods in addition to those from *BufferedIOBase* and *IOBase*:

flush()

Force bytes held in the buffer into the raw stream. A *BlockingIOError* should be raised if the raw stream blocks.

write(*b*, /)

Write the *bytes-like object*, *b*, and return the number of bytes written. When in non-blocking mode, a `BlockingIOError` is raised if the buffer needs to be written out but the raw stream blocks.

class io.**BufferedRandom**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

A buffered binary stream providing higher-level access to a seekable `RawIOBase` raw binary stream. It inherits from `BufferedReader` and `BufferedWriter`.

The constructor creates a reader and writer for a seekable raw stream, given in the first argument. If the *buffer_size* is omitted it defaults to `DEFAULT_BUFFER_SIZE`.

`BufferedRandom` is capable of anything `BufferedReader` or `BufferedWriter` can do. In addition, `seek()` and `tell()` are guaranteed to be implemented.

class io.**BufferedRWPair**(*reader*, *writer*, *buffer_size=DEFAULT_BUFFER_SIZE*, /)

A buffered binary stream providing higher-level access to two non seekable `RawIOBase` raw binary streams—one readable, the other writeable. It inherits from `BufferedIOBase`.

reader and *writer* are `RawIOBase` objects that are readable and writeable respectively. If the *buffer_size* is omitted it defaults to `DEFAULT_BUFFER_SIZE`.

`BufferedRWPair` implements all of `BufferedIOBase`'s methods except for `detach()`, which raises `UnsupportedOperation`.

> **Warning**
>
> `BufferedRWPair` does not attempt to synchronize accesses to its underlying raw streams. You should not pass it the same object as reader and writer; use `BufferedRandom` instead.

Text I/O

class io.**TextIOBase**

Base class for text streams. This class provides a character and line based interface to stream I/O. It inherits from `IOBase`.

`TextIOBase` provides or overrides these data attributes and methods in addition to those from `IOBase`:

encoding

The name of the encoding used to decode the stream's bytes into strings, and to encode strings into bytes.

errors

The error setting of the decoder or encoder.

newlines

A string, a tuple of strings, or `None`, indicating the newlines translated so far. Depending on the implementation and the initial constructor flags, this may not be available.

buffer

The underlying binary buffer (a `BufferedIOBase` instance) that `TextIOBase` deals with. This is not part of the `TextIOBase` API and may not exist in some implementations.

detach()

Separate the underlying binary buffer from the `TextIOBase` and return it.

After the underlying buffer has been detached, the `TextIOBase` is in an unusable state.

Some *TextIOBase* implementations, like *StringIO*, may not have the concept of an underlying buffer and calling this method will raise *UnsupportedOperation*.

Added in version 3.1.

read(*size=-1*, */*)

Read and return at most *size* characters from the stream as a single *str*. If *size* is negative or None, reads until EOF.

readline(*size=-1*, */*)

Read until newline or EOF and return a single *str*. If the stream is already at EOF, an empty string is returned.

If *size* is specified, at most *size* characters will be read.

seek(*offset*, *whence=SEEK_SET*, */*)

Change the stream position to the given *offset*. Behaviour depends on the *whence* parameter. The default value for *whence* is SEEK_SET.

- SEEK_SET or 0: seek from the start of the stream (the default); *offset* must either be a number returned by *TextIOBase.tell()*, or zero. Any other *offset* value produces undefined behaviour.
- SEEK_CUR or 1: "seek" to the current position; *offset* must be zero, which is a no-operation (all other values are unsupported).
- SEEK_END or 2: seek to the end of the stream; *offset* must be zero (all other values are unsupported).

Return the new absolute position as an opaque number.

Added in version 3.1: The SEEK_* constants.

tell()

Return the current stream position as an opaque number. The number does not usually represent a number of bytes in the underlying binary storage.

write(*s*, */*)

Write the string *s* to the stream and return the number of characters written.

class io.**TextIOWrapper**(*buffer*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*, *write_through=False*)

A buffered text stream providing higher-level access to a *BufferedIOBase* buffered binary stream. It inherits from *TextIOBase*.

encoding gives the name of the encoding that the stream will be decoded or encoded with. It defaults to *locale.getencoding()*. encoding="locale" can be used to specify the current locale's encoding explicitly. See *Text Encoding* for more information.

errors is an optional string that specifies how encoding and decoding errors are to be handled. Pass 'strict' to raise a *ValueError* exception if there is an encoding error (the default of None has the same effect), or pass 'ignore' to ignore errors. (Note that ignoring encoding errors can lead to data loss.) 'replace' causes a replacement marker (such as '?') to be inserted where there is malformed data. 'backslashreplace' causes malformed data to be replaced by a backslashed escape sequence. When writing, 'xmlcharrefreplace' (replace with the appropriate XML character reference) or 'namereplace' (replace with \N{...} escape sequences) can be used. Any other error handling name that has been registered with *codecs.register_error()* is also valid.

newline controls how line endings are handled. It can be None, '', '\n', '\r', and '\r\n'. It works as follows:

- When reading input from the stream, if *newline* is None, *universal newlines* mode is enabled. Lines in the input can end in '\n', '\r', or '\r\n', and these are translated into '\n' before being returned to

the caller. If *newline* is `''`, universal newlines mode is enabled, but line endings are returned to the caller untranslated. If *newline* has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.

- When writing output to the stream, if *newline* is `None`, any `'\n'` characters written are translated to the system default line separator, *os.linesep*. If *newline* is `''` or `'\n'`, no translation takes place. If *newline* is any of the other legal values, any `'\n'` characters written are translated to the given string.

If *line_buffering* is `True`, *flush()* is implied when a call to write contains a newline character or a carriage return.

If *write_through* is `True`, calls to *write()* are guaranteed not to be buffered: any data written on the *TextIOWrapper* object is immediately handled to its underlying binary *buffer*.

Changed in version 3.3: The *write_through* argument has been added.

Changed in version 3.3: The default *encoding* is now `locale.getpreferredencoding(False)` instead of `locale.getpreferredencoding()`. Don't change temporary the locale encoding using *locale.setlocale()*, use the current locale encoding instead of the user preferred encoding.

Changed in version 3.10: The *encoding* argument now supports the `"locale"` dummy encoding name.

TextIOWrapper provides these data attributes and methods in addition to those from *TextIOBase* and *IOBase*:

line_buffering

Whether line buffering is enabled.

write_through

Whether writes are passed immediately to the underlying binary buffer.

Added in version 3.7.

reconfigure(*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=None*, *write_through=None*)

Reconfigure this text stream using new settings for *encoding*, *errors*, *newline*, *line_buffering* and *write_through*.

Parameters not specified keep current settings, except `errors='strict'` is used when *encoding* is specified but *errors* is not specified.

It is not possible to change the encoding or newline if some data has already been read from the stream. On the other hand, changing encoding after write is possible.

This method does an implicit stream flush before setting the new parameters.

Added in version 3.7.

Changed in version 3.11: The method supports `encoding="locale"` option.

seek(*cookie*, *whence=os.SEEK_SET*, */*)

Set the stream position. Return the new stream position as an *int*.

Four operations are supported, given by the following argument combinations:

- `seek(0, SEEK_SET)`: Rewind to the start of the stream.
- `seek(cookie, SEEK_SET)`: Restore a previous position; *cookie* **must be** a number returned by *tell()*.
- `seek(0, SEEK_END)`: Fast-forward to the end of the stream.
- `seek(0, SEEK_CUR)`: Leave the current stream position unchanged.

Any other argument combinations are invalid, and may raise exceptions.

> **See also**
>
> *os.SEEK_SET*, *os.SEEK_CUR*, and *os.SEEK_END*.

tell()

Return the stream position as an opaque number. The return value of `tell()` can be given as input to *seek()*, to restore a previous stream position.

class io.**StringIO**(*initial_value=''*, *newline='\n'*)

A text stream using an in-memory text buffer. It inherits from *TextIOBase*.

The text buffer is discarded when the *close()* method is called.

The initial value of the buffer can be set by providing *initial_value*. If newline translation is enabled, newlines will be encoded as if by *write()*. The stream is positioned at the start of the buffer which emulates opening an existing file in a `w+` mode, making it ready for an immediate write from the beginning or for a write that would overwrite the initial value. To emulate opening a file in an `a+` mode ready for appending, use `f.seek(0, io.SEEK_END)` to reposition the stream at the end of the buffer.

The *newline* argument works like that of *TextIOWrapper*, except that when writing output to the stream, if *newline* is `None`, newlines are written as `\n` on all platforms.

StringIO provides this method in addition to those from *TextIOBase* and *IOBase*:

getvalue()

Return a *str* containing the entire contents of the buffer. Newlines are decoded as if by *read()*, although the stream position is not changed.

Example usage:

```
import io

output = io.StringIO()
output.write('First line.\n')
print('Second line.', file=output)

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

class io.**IncrementalNewlineDecoder**

A helper codec that decodes newlines for *universal newlines* mode. It inherits from *codecs.IncrementalDecoder*.

16.2.5 Performance

This section discusses the performance of the provided concrete I/O implementations.

Binary I/O

By reading and writing only large chunks of data even when the user asks for a single byte, buffered I/O hides any inefficiency in calling and executing the operating system's unbuffered I/O routines. The gain depends on the OS and the kind of I/O which is performed. For example, on some modern OSes such as Linux, unbuffered disk I/O can be as fast as buffered I/O. The bottom line, however, is that buffered I/O offers predictable performance regardless of the platform and the backing device. Therefore, it is almost always preferable to use buffered I/O rather than unbuffered I/O for binary data.

Text I/O

Text I/O over a binary storage (such as a file) is significantly slower than binary I/O over the same storage, because it requires conversions between unicode and binary data using a character codec. This can become noticeable handling huge amounts of text data like large log files. Also, *tell()* and *seek()* are both quite slow due to the reconstruction algorithm used.

StringIO, however, is a native in-memory unicode container and will exhibit similar speed to *BytesIO*.

Multi-threading

FileIO objects are thread-safe to the extent that the operating system calls (such as *read(2)* under Unix) they wrap are thread-safe too.

Binary buffered objects (instances of *BufferedReader*, *BufferedWriter*, *BufferedRandom* and *BufferedRWPair*) protect their internal structures using a lock; it is therefore safe to call them from multiple threads at once.

TextIOWrapper objects are not thread-safe.

Reentrancy

Binary buffered objects (instances of *BufferedReader*, *BufferedWriter*, *BufferedRandom* and *BufferedRWPair*) are not reentrant. While reentrant calls will not happen in normal situations, they can arise from doing I/O in a *signal* handler. If a thread tries to re-enter a buffered object which it is already accessing, a *RuntimeError* is raised. Note this doesn't prohibit a different thread from entering the buffered object.

The above implicitly extends to text files, since the *open()* function will wrap a buffered object inside a *TextIOWrapper*. This includes standard streams and therefore affects the built-in *print()* function as well.

16.3 `time` — Time access and conversions

This module provides various time-related functions. For related functionality, see also the *datetime* and *calendar* modules.

Although this module is always available, not all functions are available on all platforms. Most of the functions defined in this module call platform C library functions with the same name. It may sometimes be helpful to consult the platform documentation, because the semantics of these functions varies among platforms.

An explanation of some terminology and conventions is in order.

- The *epoch* is the point where the time starts, the return value of `time.gmtime(0)`. It is January 1, 1970, 00:00:00 (UTC) on all platforms.
- The term *seconds since the epoch* refers to the total number of elapsed seconds since the epoch, typically excluding leap seconds. Leap seconds are excluded from this total on all POSIX-compliant platforms.
- The functions in this module may not handle dates and times before the *epoch* or far in the future. The cut-off point in the future is determined by the C library; for 32-bit systems, it is typically in 2038.
- Function *strptime()* can parse 2-digit years when given `%y` format code. When 2-digit years are parsed, they are converted according to the POSIX and ISO C standards: values 69–99 are mapped to 1969–1999, and values 0–68 are mapped to 2000–2068.
- UTC is Coordinated Universal Time (formerly known as Greenwich Mean Time, or GMT). The acronym UTC is not a mistake but a compromise between English and French.
- DST is Daylight Saving Time, an adjustment of the timezone by (usually) one hour during part of the year. DST rules are magic (determined by local law) and can change from year to year. The C library has a table containing the local rules (often it is read from a system file for flexibility) and is the only source of True Wisdom in this respect.
- The precision of the various real-time functions may be less than suggested by the units in which their value or argument is expressed. E.g. on most Unix systems, the clock "ticks" only 50 or 100 times a second.
- On the other hand, the precision of *time()* and *sleep()* is better than their Unix equivalents: times are expressed as floating-point numbers, *time()* returns the most accurate time available (using Unix `gettimeofday()` where available), and *sleep()* will accept a time with a nonzero fraction (Unix `select()` is used to implement this, where available).
- The time value as returned by *gmtime()*, *localtime()*, and *strptime()*, and accepted by *asctime()*, *mktime()* and *strftime()*, is a sequence of 9 integers. The return values of *gmtime()*, *localtime()*, and *strptime()* also offer attribute names for individual fields.

  See *struct_time* for a description of these objects.

  Changed in version 3.3: The *struct_time* type was extended to provide the *tm_gmtoff* and *tm_zone* attributes when platform supports corresponding `struct tm` members.

  Changed in version 3.6: The *struct_time* attributes *tm_gmtoff* and *tm_zone* are now available on all platforms.

- Use the following functions to convert between time representations:

From	To	Use
seconds since the epoch	*struct_time* in UTC	*gmtime()*
seconds since the epoch	*struct_time* in local time	*localtime()*
struct_time in UTC	seconds since the epoch	*calendar.timegm()*
struct_time in local time	seconds since the epoch	*mktime()*

16.3.1 Functions

time.**asctime**([*t*])

Convert a tuple or *struct_time* representing a time as returned by *gmtime()* or *localtime()* to a string of the following form: `'Sun Jun 20 23:21:05 1993'`. The day field is two characters long and is space padded if the day is a single digit, e.g.: `'Wed Jun 9 04:26:40 1993'`.

If *t* is not provided, the current time as returned by *localtime()* is used. Locale information is not used by *asctime()*.

> **Note**
>
> Unlike the C function of the same name, *asctime()* does not add a trailing newline.

time.**pthread_getcpuclockid**(*thread_id*)

Return the *clk_id* of the thread-specific CPU-time clock for the specified *thread_id*.

Use *threading.get_ident()* or the *ident* attribute of *threading.Thread* objects to get a suitable value for *thread_id*.

> **Warning**
>
> Passing an invalid or expired *thread_id* may result in undefined behavior, such as segmentation fault.

Availability: Unix

See the man page for *pthread_getcpuclockid(3)* for further information.

Added in version 3.7.

time.**clock_getres**(*clk_id*)

Return the resolution (precision) of the specified clock *clk_id*. Refer to *Clock ID Constants* for a list of accepted values for *clk_id*.

Availability: Unix.

Added in version 3.3.

time.**clock_gettime**(*clk_id*) → *float*

Return the time of the specified clock *clk_id*. Refer to *Clock ID Constants* for a list of accepted values for *clk_id*.

Use *clock_gettime_ns()* to avoid the precision loss caused by the *float* type.

Availability: Unix.

Added in version 3.3.

time.**clock_gettime_ns**(*clk_id*) → *int*

Similar to *clock_gettime()* but return time as nanoseconds.

Availability: Unix.

Added in version 3.7.

time.**clock_settime**(*clk_id*, *time:* float)

Set the time of the specified clock *clk_id*. Currently, *CLOCK_REALTIME* is the only accepted value for *clk_id*.

Use *clock_settime_ns()* to avoid the precision loss caused by the *float* type.

Availability: Unix.

Added in version 3.3.

time.**clock_settime_ns**(*clk_id*, *time:* int)

Similar to *clock_settime()* but set time with nanoseconds.

Availability: Unix.

Added in version 3.7.

time.**ctime**([*secs*])

Convert a time expressed in seconds since the *epoch* to a string of a form: `'Sun Jun 20 23:21:05 1993'` representing local time. The day field is two characters long and is space padded if the day is a single digit, e.g.: `'Wed Jun 9 04:26:40 1993'`.

If *secs* is not provided or *None*, the current time as returned by *time()* is used. `ctime(secs)` is equivalent to `asctime(localtime(secs))`. Locale information is not used by *ctime()*.

time.**get_clock_info**(*name*)

Get information on the specified clock as a namespace object. Supported clock names and the corresponding functions to read their value are:

- `'monotonic'`: *time.monotonic()*
- `'perf_counter'`: *time.perf_counter()*
- `'process_time'`: *time.process_time()*
- `'thread_time'`: *time.thread_time()*
- `'time'`: *time.time()*

The result has the following attributes:

- *adjustable*: `True` if the clock can be changed automatically (e.g. by a NTP daemon) or manually by the system administrator, `False` otherwise
- *implementation*: The name of the underlying C function used to get the clock value. Refer to *Clock ID Constants* for possible values.
- *monotonic*: `True` if the clock cannot go backward, `False` otherwise
- *resolution*: The resolution of the clock in seconds (*float*)

Added in version 3.3.

time.**gmtime**([*secs*])

Convert a time expressed in seconds since the *epoch* to a *struct_time* in UTC in which the dst flag is always zero. If *secs* is not provided or *None*, the current time as returned by *time()* is used. Fractions of a second are ignored. See above for a description of the *struct_time* object. See *calendar.timegm()* for the inverse of this function.

time.**localtime**([*secs*])

Like *gmtime()* but converts to local time. If *secs* is not provided or *None*, the current time as returned by *time()* is used. The dst flag is set to `1` when DST applies to the given time.

localtime() may raise *OverflowError*, if the timestamp is outside the range of values supported by the platform C `localtime()` or `gmtime()` functions, and *OSError* on `localtime()` or `gmtime()` failure. It's common for this to be restricted to years between 1970 and 2038.

time.**mktime**(*t*)

This is the inverse function of *localtime()*. Its argument is the *struct_time* or full 9-tuple (since the dst flag is needed; use `-1` as the dst flag if it is unknown) which expresses the time in *local* time, not UTC. It returns a floating-point number, for compatibility with *time()*. If the input value cannot be represented as a valid time, either *OverflowError* or *ValueError* will be raised (which depends on whether the invalid value is caught by Python or the underlying C libraries). The earliest date for which it can generate a time is platform-dependent.

time.**monotonic**() → *float*

Return the value (in fractional seconds) of a monotonic clock, i.e. a clock that cannot go backwards. The clock is not affected by system clock updates. The reference point of the returned value is undefined, so that only the difference between the results of two calls is valid.

Clock:

- On Windows, call `QueryPerformanceCounter()` and `QueryPerformanceFrequency()`.
- On macOS, call `mach_absolute_time()` and `mach_timebase_info()`.
- On HP-UX, call `gethrtime()`.
- Call `clock_gettime(CLOCK_HIGHRES)` if available.
- Otherwise, call `clock_gettime(CLOCK_MONOTONIC)`.

Use *monotonic_ns()* to avoid the precision loss caused by the *float* type.

Added in version 3.3.

Changed in version 3.5: The function is now always available and always system-wide.

Changed in version 3.10: On macOS, the function is now system-wide.

time.**monotonic_ns**() → *int*

Similar to *monotonic()*, but return time as nanoseconds.

Added in version 3.7.

time.**perf_counter**() → *float*

Return the value (in fractional seconds) of a performance counter, i.e. a clock with the highest available resolution to measure a short duration. It does include time elapsed during sleep and is system-wide. The reference point of the returned value is undefined, so that only the difference between the results of two calls is valid.

CPython implementation detail: On CPython, use the same clock than *time.monotonic()* and is a monotonic clock, i.e. a clock that cannot go backwards.

Use *perf_counter_ns()* to avoid the precision loss caused by the *float* type.

Added in version 3.3.

Changed in version 3.10: On Windows, the function is now system-wide.

Changed in version 3.13: Use the same clock than *time.monotonic()*.

time.**perf_counter_ns**() → *int*

Similar to *perf_counter()*, but return time as nanoseconds.

Added in version 3.7.

time.**process_time**() → *float*

Return the value (in fractional seconds) of the sum of the system and user CPU time of the current process. It does not include time elapsed during sleep. It is process-wide by definition. The reference point of the returned value is undefined, so that only the difference between the results of two calls is valid.

Use *process_time_ns()* to avoid the precision loss caused by the *float* type.

Added in version 3.3.

time.**process_time_ns**() → *int*

Similar to *process_time()* but return time as nanoseconds.

Added in version 3.7.

time.**sleep**(*secs*)

Suspend execution of the calling thread for the given number of seconds. The argument may be a floating-point number to indicate a more precise sleep time.

If the sleep is interrupted by a signal and no exception is raised by the signal handler, the sleep is restarted with a recomputed timeout.

The suspension time may be longer than requested by an arbitrary amount, because of the scheduling of other activity in the system.

On Windows, if *secs* is zero, the thread relinquishes the remainder of its time slice to any other thread that is ready to run. If there are no other threads ready to run, the function returns immediately, and the thread continues execution. On Windows 8.1 and newer the implementation uses a high-resolution timer which provides resolution of 100 nanoseconds. If *secs* is zero, `Sleep(0)` is used.

Unix implementation:

- Use `clock_nanosleep()` if available (resolution: 1 nanosecond);
- Or use `nanosleep()` if available (resolution: 1 nanosecond);
- Or use `select()` (resolution: 1 microsecond).

Raises an *auditing event* `time.sleep` with argument `secs`.

Changed in version 3.5: The function now sleeps at least *secs* even if the sleep is interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale).

Changed in version 3.11: On Unix, the `clock_nanosleep()` and `nanosleep()` functions are now used if available. On Windows, a waitable timer is now used.

Changed in version 3.13: Raises an auditing event.

time.**strftime**(*format*[, *t*])

Convert a tuple or *struct_time* representing a time as returned by *gmtime()* or *localtime()* to a string as specified by the *format* argument. If *t* is not provided, the current time as returned by *localtime()* is used. *format* must be a string. *ValueError* is raised if any field in *t* is outside of the allowed range.

0 is a legal argument for any position in the time tuple; if it is normally illegal the value is forced to a correct one.

The following directives can be embedded in the *format* string. They are shown without the optional field width and precision specification, and are replaced by the indicated characters in the *strftime()* result:

Directive	Meaning	Notes
%a	Locale's abbreviated weekday name.	
%A	Locale's full weekday name.	
%b	Locale's abbreviated month name.	
%B	Locale's full month name.	
%c	Locale's appropriate date and time representation.	
%d	Day of the month as a decimal number [01,31].	
%f	**Microseconds as a decimal number** [000000,999999].	(1)
%H	Hour (24-hour clock) as a decimal number [00,23].	
%I	Hour (12-hour clock) as a decimal number [01,12].	
%j	Day of the year as a decimal number [001,366].	
%m	Month as a decimal number [01,12].	
%M	Minute as a decimal number [00,59].	
%p	Locale's equivalent of either AM or PM.	(2)
%S	Second as a decimal number [00,61].	(3)
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.	(4)
%w	Weekday as a decimal number [0(Sunday),6].	
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.	(4)
%x	Locale's appropriate date representation.	
%X	Locale's appropriate time representation.	
%y	Year without century as a decimal number [00,99].	
%Y	Year with century as a decimal number.	
%z	Time zone offset indicating a positive or negative time difference from UTC/GMT of the form +HHMM or -HHMM, where H represents decimal hour digits and M represents decimal minute digits [-23:59, +23:59].[1]	
%Z	Time zone name (no characters	

Notes:

(1) The `%f` format directive only applies to *strptime()*, not to *strftime()*. However, see also *datetime.datetime.strptime()* and *datetime.datetime.strftime()* where the `%f` format directive *applies to microseconds*.

(2) When used with the *strptime()* function, the `%p` directive only affects the output hour field if the `%I` directive is used to parse the hour.

(3) The range really is `0` to `61`; value `60` is valid in timestamps representing leap seconds and value `61` is supported for historical reasons.

(4) When used with the *strptime()* function, `%U` and `%W` are only used in calculations when the day of the week and the year are specified.

Here is an example, a format for dates compatible with that specified in the **RFC 2822** Internet email standard.[1]

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

Additional directives may be supported on certain platforms, but only the ones listed here have a meaning standardized by ANSI C. To see the full set of format codes supported on your platform, consult the *strftime(3)* documentation.

On some platforms, an optional field width and precision specification can immediately follow the initial `'%'` of a directive in the following order; this is also not portable. The field width is normally 2 except for `%j` where it is 3.

`time.`**`strptime`**(*string*[, *format*])

Parse a string representing a time according to a format. The return value is a *struct_time* as returned by *gmtime()* or *localtime()*.

The *format* parameter uses the same directives as those used by *strftime()*; it defaults to `"%a %b %d %H:%M:%S %Y"` which matches the formatting returned by *ctime()*. If *string* cannot be parsed according to *format*, or if it has excess data after parsing, *ValueError* is raised. The default values used to fill in any missing data when more accurate values cannot be inferred are `(1900, 1, 1, 0, 0, 0, 0, 1, -1)`. Both *string* and *format* must be strings.

For example:

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")
time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0,
                 tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1)
```

Support for the `%Z` directive is based on the values contained in `tzname` and whether `daylight` is true. Because of this, it is platform-specific except for recognizing UTC and GMT which are always known (and are considered to be non-daylight savings timezones).

Only the directives specified in the documentation are supported. Because `strftime()` is implemented per platform it can sometimes offer more directives than those listed. But `strptime()` is independent of any platform and thus does not necessarily support all directives available that are not documented as supported.

[1] The use of `%Z` is now deprecated, but the `%z` escape that expands to the preferred hour/minute offset is not supported by all ANSI C libraries. Also, a strict reading of the original 1982 **RFC 822** standard calls for a two-digit year (`%y` rather than `%Y`), but practice moved to 4-digit years long before the year 2000. After that, **RFC 822** became obsolete and the 4-digit year has been first recommended by **RFC 1123** and then mandated by **RFC 2822**.

class time.**struct_time**

The type of the time value sequence returned by *gmtime()*, *localtime()*, and *strptime()*. It is an object with a *named tuple* interface: values can be accessed by index and by attribute name. The following values are present:

Index	Attribute	Values
0	**tm_year**	(for example, 1993)
1	**tm_mon**	range [1, 12]
2	**tm_mday**	range [1, 31]
3	**tm_hour**	range [0, 23]
4	**tm_min**	range [0, 59]
5	**tm_sec**	range [0, 61]; see *Note (2)* in *strftime()*
6	**tm_wday**	range [0, 6]; Monday is 0
7	**tm_yday**	range [1, 366]
8	**tm_isdst**	0, 1 or -1; see below
N/A	**tm_zone**	abbreviation of timezone name
N/A	**tm_gmtoff**	offset east of UTC in seconds

Note that unlike the C structure, the month value is a range of [1, 12], not [0, 11].

In calls to *mktime()*, *tm_isdst* may be set to 1 when daylight savings time is in effect, and 0 when it is not. A value of -1 indicates that this is not known, and will usually result in the correct state being filled in.

When a tuple with an incorrect length is passed to a function expecting a *struct_time*, or having elements of the wrong type, a *TypeError* is raised.

time.**time**() → *float*

Return the time in seconds since the *epoch* as a floating-point number. The handling of leap seconds is platform dependent. On Windows and most Unix systems, the leap seconds are not counted towards the time in seconds since the *epoch*. This is commonly referred to as Unix time.

Note that even though the time is always returned as a floating-point number, not all systems provide time with a better precision than 1 second. While this function normally returns non-decreasing values, it can return a lower

value than a previous call if the system clock has been set back between the two calls.

The number returned by *time()* may be converted into a more common time format (i.e. year, month, day, hour, etc…) in UTC by passing it to *gmtime()* function or in local time by passing it to the *localtime()* function. In both cases a *struct_time* object is returned, from which the components of the calendar date may be accessed as attributes.

Clock:

- On Windows, call `GetSystemTimeAsFileTime()`.
- Call `clock_gettime(CLOCK_REALTIME)` if available.
- Otherwise, call `gettimeofday()`.

Use *time_ns()* to avoid the precision loss caused by the *float* type.

`time.`**`time_ns`**`()` → *int*

Similar to *time()* but returns time as an integer number of nanoseconds since the *epoch*.

Added in version 3.7.

`time.`**`thread_time`**`()` → *float*

Return the value (in fractional seconds) of the sum of the system and user CPU time of the current thread. It does not include time elapsed during sleep. It is thread-specific by definition. The reference point of the returned value is undefined, so that only the difference between the results of two calls in the same thread is valid.

Use *thread_time_ns()* to avoid the precision loss caused by the *float* type.

Availability: Linux, Unix, Windows.

Unix systems supporting `CLOCK_THREAD_CPUTIME_ID`.

Added in version 3.7.

`time.`**`thread_time_ns`**`()` → *int*

Similar to *thread_time()* but return time as nanoseconds.

Added in version 3.7.

`time.`**`tzset`**`()`

Reset the time conversion rules used by the library routines. The environment variable `TZ` specifies how this is done. It will also set the variables `tzname` (from the `TZ` environment variable), `timezone` (non-DST seconds West of UTC), `altzone` (DST seconds west of UTC) and `daylight` (to 0 if this timezone does not have any daylight saving time rules, or to nonzero if there is a time, past, present or future when daylight saving time applies).

Availability: Unix.

Note

Although in many cases, changing the `TZ` environment variable may affect the output of functions like *localtime()* without calling *tzset()*, this behavior should not be relied on.

The `TZ` environment variable should contain no whitespace.

The standard format of the `TZ` environment variable is (whitespace added for clarity):

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

Where the components are:

std and dst
Three or more alphanumerics giving the timezone abbreviations. These will be propagated into time.tzname

offset
The offset has the form: `± hh[:mm[:ss]]`. This indicates the value added the local time to arrive at UTC. If preceded by a '-', the timezone is east of the Prime Meridian; otherwise, it is west. If no offset follows dst, summer time is assumed to be one hour ahead of standard time.

start[/time], end[/time]
Indicates when to change to and back from DST. The format of the start and end dates are one of the following:

Jn
The Julian day *n* (1 <= *n* <= 365). Leap days are not counted, so in all years February 28 is day 59 and March 1 is day 60.

n
The zero-based Julian day (0 <= *n* <= 365). Leap days are counted, and it is possible to refer to February 29.

Mm.n.d
The *d*'th day (0 <= *d* <= 6) of week *n* of month *m* of the year (1 <= *n* <= 5, 1 <= *m* <= 12, where week 5 means "the last *d* day in month *m*" which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*'th day occurs. Day zero is a Sunday.

`time` has the same format as `offset` except that no leading sign ('-' or '+') is allowed. The default, if time is not given, is 02:00:00.

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

On many Unix systems (including *BSD, Linux, Solaris, and Darwin), it is more convenient to use the system's zoneinfo (*tzfile(5)*) database to specify the timezone rules. To do this, set the `TZ` environment variable to the path of the required timezone datafile, relative to the root of the systems 'zoneinfo' timezone database, usually located at `/usr/share/zoneinfo`. For example, `'US/Eastern'`, `'Australia/Melbourne'`, `'Egypt'` or `'Europe/Amsterdam'`.

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

16.3.2 Clock ID Constants

These constants are used as parameters for *clock_getres()* and *clock_gettime()*.

time.**CLOCK_BOOTTIME**

Identical to *CLOCK_MONOTONIC*, except it also includes any time that the system is suspended.

This allows applications to get a suspend-aware monotonic clock without having to deal with the complications of *CLOCK_REALTIME*, which may have discontinuities if the time is changed using `settimeofday()` or similar.

Availability: Linux >= 2.6.39.

Added in version 3.7.

time.**CLOCK_HIGHRES**

The Solaris OS has a `CLOCK_HIGHRES` timer that attempts to use an optimal hardware source, and may give close to nanosecond resolution. `CLOCK_HIGHRES` is the nonadjustable, high-resolution clock.

Availability: Solaris.

Added in version 3.3.

time.**CLOCK_MONOTONIC**

Clock that cannot be set and represents monotonic time since some unspecified starting point.

Availability: Unix.

Added in version 3.3.

time.**CLOCK_MONOTONIC_RAW**

Similar to *CLOCK_MONOTONIC*, but provides access to a raw hardware-based time that is not subject to NTP adjustments.

Availability: Linux >= 2.6.28, macOS >= 10.12.

Added in version 3.3.

time.**CLOCK_MONOTONIC_RAW_APPROX**

Similar to *CLOCK_MONOTONIC_RAW*, but reads a value cached by the system at context switch and hence has less accuracy.

Availability: macOS >= 10.12.

Added in version 3.13.

time.**CLOCK_PROCESS_CPUTIME_ID**

High-resolution per-process timer from the CPU.

Availability: Unix.

Added in version 3.3.

time.**CLOCK_PROF**

High-resolution per-process timer from the CPU.

Availability: FreeBSD, NetBSD >= 7, OpenBSD.

Added in version 3.7.

time.**CLOCK_TAI**

International Atomic Time

The system must have a current leap second table in order for this to give the correct answer. PTP or NTP software can maintain a leap second table.

Availability: Linux.

Added in version 3.9.

time.**CLOCK_THREAD_CPUTIME_ID**

Thread-specific CPU-time clock.

Availability: Unix.

Added in version 3.3.

time.**CLOCK_UPTIME**

Time whose absolute value is the time the system has been running and not suspended, providing accurate uptime measurement, both absolute and interval.

Availability: FreeBSD, OpenBSD >= 5.5.

Added in version 3.7.

time.**CLOCK_UPTIME_RAW**

Clock that increments monotonically, tracking the time since an arbitrary point, unaffected by frequency or time adjustments and not incremented while the system is asleep.

Availability: macOS >= 10.12.

Added in version 3.8.

time.**CLOCK_UPTIME_RAW_APPROX**

Like *CLOCK_UPTIME_RAW*, but the value is cached by the system at context switches and therefore has less accuracy.

Availability: macOS >= 10.12.

Added in version 3.13.

The following constant is the only parameter that can be sent to *clock_settime()*.

time.**CLOCK_REALTIME**

System-wide real-time clock. Setting this clock requires appropriate privileges.

Availability: Unix.

Added in version 3.3.

16.3.3 Timezone Constants

time.**altzone**

The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if `daylight` is nonzero. See note below.

time.**daylight**

Nonzero if a DST timezone is defined. See note below.

time.**timezone**

The offset of the local (non-DST) timezone, in seconds west of UTC (negative in most of Western Europe, positive in the US, zero in the UK). See note below.

time.**tzname**

A tuple of two strings: the first is the name of the local non-DST timezone, the second is the name of the local DST timezone. If no DST timezone is defined, the second string should not be used. See note below.

Note

For the above Timezone constants (*altzone*, *daylight*, *timezone*, and *tzname*), the value is determined by the timezone rules in effect at module load time or the last time *tzset()* is called and may be incorrect for times in the past. It is recommended to use the *tm_gmtoff* and *tm_zone* results from *localtime()* to obtain timezone information.

See also

Module *datetime*
More object-oriented interface to dates and times.

Module *locale*
Internationalization services. The locale setting affects the interpretation of many format specifiers in *strftime()* and *strptime()*.

Module *calendar*
General calendar-related functions. *timegm()* is the inverse of *gmtime()* from this module.

16.4 argparse — Parser for command-line options, arguments and sub-commands

Added in version 3.2.

Source code: Lib/argparse.py

Tutorial

This page contains the API reference information. For a more gentle introduction to Python command-line parsing, have a look at the argparse tutorial.

The *argparse* module makes it easy to write user-friendly command-line interfaces. The program defines what arguments it requires, and *argparse* will figure out how to parse those out of *sys.argv*. The *argparse* module also automatically generates help and usage messages. The module will also issue errors when users give the program invalid arguments.

16.4.1 Core Functionality

The *argparse* module's support for command-line interfaces is built around an instance of *argparse.ArgumentParser*. It is a container for argument specifications and has options that apply to the parser as whole:

```
parser = argparse.ArgumentParser(
                    prog='ProgramName',
                    description='What the program does',
                    epilog='Text at the bottom of help')
```

The *ArgumentParser.add_argument()* method attaches individual argument specifications to the parser. It supports positional arguments, options that accept values, and on/off flags:

```
parser.add_argument('filename')           # positional argument
parser.add_argument('-c', '--count')      # option that takes a value
parser.add_argument('-v', '--verbose',
                    action='store_true')  # on/off flag
```

The *ArgumentParser.parse_args()* method runs the parser and places the extracted data in a *argparse.Namespace* object:

```
args = parser.parse_args()
print(args.filename, args.count, args.verbose)
```

16.4.2 Quick Links for add_argument()

Name	Description	Values
action	Specify how an argument should be handled	`'store'`, `'store_const'`, `'store_true'`, `'append'`, `'append_const'`, `'count'`, `'help'`, `'version'`
choices	Limit values to a specific set of choices	`['foo', 'bar']`, `range(1, 10)`, or *Container* instance
const	Store a constant value	
default	Default value used when an argument is not provided	Defaults to `None`
dest	Specify the attribute name used in the result namespace	
help	Help message for an argument	
metavar	Alternate display name for the argument as shown in help	
nargs	Number of times the argument can be used	*int*, `'?'`, `'*'`, or `'+'`
required	Indicate whether an argument is required or optional	`True` or `False`
type	Automatically convert an argument to the given type	*int*, *float*, `argparse.FileType('w')`, or callable function

16.4.3 Example

The following code is a Python program that takes a list of integers and produces either the sum or the max:

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

Assuming the above Python code is saved into a file called `prog.py`, it can be run at the command line and it provides useful help messages:

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
 N           an integer for the accumulator

options:
 -h, --help  show this help message and exit
 --sum       sum the integers (default: find the max)
```

When run with the appropriate arguments, it prints either the sum or the max of the command-line integers:

```
$ python prog.py 1 2 3 4
4

$ python prog.py 1 2 3 4 --sum
10
```

If invalid arguments are passed in, an error will be displayed:

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

The following sections walk you through this example.

Creating a parser

The first step in using the *argparse* is creating an *ArgumentParser* object:

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

The *ArgumentParser* object will hold all the information necessary to parse the command line into Python data types.

Adding arguments

Filling an *ArgumentParser* with information about program arguments is done by making calls to the *add_argument()* method. Generally, these calls tell the *ArgumentParser* how to take the strings on the command line and turn them into objects. This information is stored and used when *parse_args()* is called. For example:

```
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
...                     help='an integer for the accumulator')
>>> parser.add_argument('--sum', dest='accumulate', action='store_const',
...                     const=sum, default=max,
...                     help='sum the integers (default: find the max)')
```

Later, calling *parse_args()* will return an object with two attributes, `integers` and `accumulate`. The `integers` attribute will be a list of one or more integers, and the `accumulate` attribute will be either the *sum()* function, if `--sum` was specified at the command line, or the *max()* function if it was not.

Parsing arguments

ArgumentParser parses arguments through the *parse_args()* method. This will inspect the command line, convert each argument to the appropriate type and then invoke the appropriate action. In most cases, this means a simple *Namespace* object will be built up from attributes parsed out of the command line:

```
>>> parser.parse_args(['--sum', '7', '-1', '42'])
Namespace(accumulate=<built-in function sum>, integers=[7, -1, 42])
```

In a script, *parse_args()* will typically be called with no arguments, and the *ArgumentParser* will automatically determine the command-line arguments from *sys.argv*.

16.4.4 ArgumentParser objects

class argparse.**ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=argparse.HelpFormatter*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*, *exit_on_error=True*)

Create a new *ArgumentParser* object. All parameters should be passed as keyword arguments. Each parameter has its own more detailed description below, but in short they are:

- *prog* - The name of the program (default: `os.path.basename(sys.argv[0])`)
- *usage* - The string describing the program usage (default: generated from arguments added to parser)
- *description* - Text to display before the argument help (by default, no text)
- *epilog* - Text to display after the argument help (by default, no text)
- *parents* - A list of *ArgumentParser* objects whose arguments should also be included

- *formatter_class* - A class for customizing the help output
- *prefix_chars* - The set of characters that prefix optional arguments (default: '-')
- *fromfile_prefix_chars* - The set of characters that prefix files from which additional arguments should be read (default: `None`)
- *argument_default* - The global default value for arguments (default: `None`)
- *conflict_handler* - The strategy for resolving conflicting optionals (usually unnecessary)
- *add_help* - Add a `-h/--help` option to the parser (default: `True`)
- *allow_abbrev* - Allows long options to be abbreviated if the abbreviation is unambiguous. (default: `True`)
- *exit_on_error* - Determines whether or not ArgumentParser exits with error info when an error occurs. (default: `True`)

Changed in version 3.5: *allow_abbrev* parameter was added.

Changed in version 3.8: In previous versions, *allow_abbrev* also disabled grouping of short flags such as `-vv` to mean `-v -v`.

Changed in version 3.9: *exit_on_error* parameter was added.

The following sections describe how each of these are used.

prog

By default, *ArgumentParser* objects use `sys.argv[0]` to determine how to display the name of the program in help messages. This default is almost always desirable because it will make the help messages match how the program was invoked on the command line. For example, consider a file named `myprogram.py` with the following code:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

The help for this program will display `myprogram.py` as the program name (regardless of where the program was invoked from):

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

options:
 -h, --help  show this help message and exit
 --foo FOO   foo help
$ cd ..
$ python subdir/myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

options:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

To change this default behavior, another value can be supplied using the `prog=` argument to *ArgumentParser*:

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]
```

(continues on next page)

(continued from previous page)

```
options:
 -h, --help  show this help message and exit
```

Note that the program name, whether determined from `sys.argv[0]` or from the `prog=` argument, is available to help messages using the `%(prog)s` format specifier.

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

options:
 -h, --help  show this help message and exit
 --foo FOO   foo of the myprogram program
```

usage

By default, *ArgumentParser* calculates the usage message from the arguments it contains:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
 bar          bar help

options:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

The default message can be overridden with the `usage=` keyword argument:

```
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [options]

positional arguments:
 bar          bar help

options:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

The `%(prog)s` format specifier is available to fill in the program name in your usage messages.

description

Most calls to the *ArgumentParser* constructor will use the `description=` keyword argument. This argument gives a brief description of what the program does and how it works. In help messages, the description is displayed between the command-line usage string and the help messages for the various arguments:

```
>>> parser = argparse.ArgumentParser(description='A foo that bars')
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

options:
 -h, --help  show this help message and exit
```

By default, the description will be line-wrapped so that it fits within the given space. To change this behavior, see the *formatter_class* argument.

epilog

Some programs like to display additional description of the program after the description of the arguments. Such text can be specified using the `epilog=` argument to *ArgumentParser*:

```
>>> parser = argparse.ArgumentParser(
...     description='A foo that bars',
...     epilog="And that's how you'd foo a bar")
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

options:
 -h, --help  show this help message and exit

And that's how you'd foo a bar
```

As with the *description* argument, the `epilog=` text is by default line-wrapped, but this behavior can be adjusted with the *formatter_class* argument to *ArgumentParser*.

parents

Sometimes, several parsers share a common set of arguments. Rather than repeating the definitions of these arguments, a single parser with all the shared arguments and passed to `parents=` argument to *ArgumentParser* can be used. The `parents=` argument takes a list of *ArgumentParser* objects, collects all the positional and optional actions from them, and adds these actions to the *ArgumentParser* object being constructed:

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
```

(continues on next page)

(continued from previous page)

```
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

Note that most parent parsers will specify `add_help=False`. Otherwise, the *ArgumentParser* will see two `-h/--help` options (one in the parent and one in the child) and raise an error.

> **Note**
>
> You must fully initialize the parsers before passing them via `parents=`. If you change the parent parsers after the child parser, those changes will not be reflected in the child.

formatter_class

ArgumentParser objects allow the help formatting to be customized by specifying an alternate formatting class. Currently, there are four such classes:

class argparse.**RawDescriptionHelpFormatter**

class argparse.**RawTextHelpFormatter**

class argparse.**ArgumentDefaultsHelpFormatter**

class argparse.**MetavarTypeHelpFormatter**

RawDescriptionHelpFormatter and *RawTextHelpFormatter* give more control over how textual descriptions are displayed. By default, *ArgumentParser* objects line-wrap the *description* and *epilog* texts in command-line help messages:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
...         was indented weird
...             but that is okay''',
...     epilog='''
...             likewise for this epilog whose whitespace will
...         be cleaned up and whose words will be wrapped
...         across a couple lines''')
>>> parser.print_help()
usage: PROG [-h]

this description was indented weird but that is okay

options:
 -h, --help  show this help message and exit

likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines
```

Passing *RawDescriptionHelpFormatter* as `formatter_class=` indicates that *description* and *epilog* are already correctly formatted and should not be line-wrapped:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''\
```

(continues on next page)

(continued from previous page)

```
...         Please do not mess up this text!
...         --------------------------------
...             I have indented it
...             exactly the way
...             I want it
...         '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
--------------------------------
   I have indented it
   exactly the way
   I want it

options:
 -h, --help  show this help message and exit
```

RawTextHelpFormatter maintains whitespace for all sorts of help text, including argument descriptions. However, multiple new lines are replaced with one. If you wish to preserve multiple blank lines, add spaces between the newlines.

ArgumentDefaultsHelpFormatter automatically adds information about default values to each of the argument help messages:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar ...]

positional arguments:
 bar         BAR! (default: [1, 2, 3])

options:
 -h, --help  show this help message and exit
 --foo FOO   FOO! (default: 42)
```

MetavarTypeHelpFormatter uses the name of the *type* argument for each argument as the display name for its values (rather than using the *dest* as the regular formatter does):

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.MetavarTypeHelpFormatter)
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', type=float)
>>> parser.print_help()
usage: PROG [-h] [--foo int] float

positional arguments:
  float

options:
  -h, --help  show this help message and exit
  --foo int
```

prefix_chars

Most command-line options will use - as the prefix, e.g. `-f/--foo`. Parsers that need to support different or additional prefix characters, e.g. for options like `+f` or `/foo`, may specify them using the `prefix_chars=` argument to the ArgumentParser constructor:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='-+')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

The `prefix_chars=` argument defaults to `'-'`. Supplying a set of characters that does not include - will cause `-f/--foo` options to be disallowed.

fromfile_prefix_chars

Sometimes, when dealing with a particularly long argument list, it may make sense to keep the list of arguments in a file rather than typing it out at the command line. If the `fromfile_prefix_chars=` argument is given to the *ArgumentParser* constructor, then arguments that start with any of the specified characters will be treated as files, and will be replaced by the arguments they contain. For example:

```
>>> with open('args.txt', 'w', encoding=sys.getfilesystemencoding()) as fp:
...     fp.write('-f\nbar')
...
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

Arguments read from a file must by default be one per line (but see also *convert_arg_line_to_args()*) and are treated as if they were in the same place as the original file referencing argument on the command line. So in the example above, the expression `['-f', 'foo', '@args.txt']` is considered equivalent to the expression `['-f', 'foo', '-f', 'bar']`.

ArgumentParser uses *filesystem encoding and error handler* to read the file containing arguments.

The `fromfile_prefix_chars=` argument defaults to `None`, meaning that arguments will never be treated as file references.

Changed in version 3.12: *ArgumentParser* changed encoding and errors to read arguments files from default (e.g. *locale.getpreferredencoding(False)* and `"strict"`) to *filesystem encoding and error handler*. Arguments file should be encoded in UTF-8 instead of ANSI Codepage on Windows.

argument_default

Generally, argument defaults are specified either by passing a default to *add_argument()* or by calling the *set_defaults()* methods with a specific set of name-value pairs. Sometimes however, it may be useful to specify a single parser-wide default for arguments. This can be accomplished by passing the `argument_default=` keyword argument to *ArgumentParser*. For example, to globally suppress attribute creation on *parse_args()* calls, we supply `argument_default=SUPPRESS`:

```
>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
```

(continues on next page)

(continued from previous page)

```
>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()
```

allow_abbrev

Normally, when you pass an argument list to the *parse_args()* method of an *ArgumentParser*, it *recognizes abbreviations* of long options.

This feature can be disabled by setting `allow_abbrev` to `False`:

```
>>> parser = argparse.ArgumentParser(prog='PROG', allow_abbrev=False)
>>> parser.add_argument('--foobar', action='store_true')
>>> parser.add_argument('--foonley', action='store_false')
>>> parser.parse_args(['--foon'])
usage: PROG [-h] [--foobar] [--foonley]
PROG: error: unrecognized arguments: --foon
```

Added in version 3.5.

conflict_handler

ArgumentParser objects do not allow two actions with the same option string. By default, *ArgumentParser* objects raise an exception if an attempt is made to create an argument with an option string that is already in use:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
 ..
ArgumentError: argument --foo: conflicting option string(s): --foo
```

Sometimes (e.g. when using *parents*) it may be useful to simply override any older arguments with the same option string. To get this behavior, the value `'resolve'` can be supplied to the `conflict_handler=` argument of *ArgumentParser*:

```
>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]

options:
 -h, --help  show this help message and exit
 -f FOO      old foo help
 --foo FOO   new foo help
```

Note that *ArgumentParser* objects only remove an action if all of its option strings are overridden. So, in the example above, the old `-f/--foo` action is retained as the `-f` action, because only the `--foo` option string was overridden.

add_help

By default, ArgumentParser objects add an option which simply displays the parser's help message. For example, consider a file named `myprogram.py` containing the following code:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

If `-h` or `--help` is supplied at the command line, the ArgumentParser help will be printed:

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

options:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

Occasionally, it may be useful to disable the addition of this help option. This can be achieved by passing `False` as the `add_help=` argument to *ArgumentParser*:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]

options:
 --foo FOO  foo help
```

The help option is typically `-h/--help`. The exception to this is if the `prefix_chars=` is specified and does not include `-`, in which case `-h` and `--help` are not valid options. In this case, the first character in `prefix_chars` is used to prefix the help options:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]

options:
  +h, ++help  show this help message and exit
```

exit_on_error

Normally, when you pass an invalid argument list to the *parse_args()* method of an *ArgumentParser*, it will exit with error info.

If the user would like to catch errors manually, the feature can be enabled by setting `exit_on_error` to `False`:

```
>>> parser = argparse.ArgumentParser(exit_on_error=False)
>>> parser.add_argument('--integers', type=int)
_StoreAction(option_strings=['--integers'], dest='integers', nargs=None, const=None,␣
↪default=None, type=<class 'int'>, choices=None, help=None, metavar=None)
>>> try:
...     parser.parse_args('--integers a'.split())
... except argparse.ArgumentError:
...     print('Catching an argumentError')
```

(continues on next page)

(continued from previous page)

```
...
Catching an argumentError
```

Added in version 3.9.

16.4.5 The add_argument() method

ArgumentParser.**add_argument**(*name or flags...*[, *action*][, *nargs*][, *const*][, *default*][, *type*][, *choices*][, *required*][, *help*][, *metavar*][, *dest*][, *deprecated*])

Define how a single command-line argument should be parsed. Each parameter has its own more detailed description below, but in short they are:

- *name or flags* - Either a name or a list of option strings, e.g. `foo` or `-f, --foo`.
- *action* - The basic type of action to be taken when this argument is encountered at the command line.
- *nargs* - The number of command-line arguments that should be consumed.
- *const* - A constant value required by some *action* and *nargs* selections.
- *default* - The value produced if the argument is absent from the command line and if it is absent from the namespace object.
- *type* - The type to which the command-line argument should be converted.
- *choices* - A sequence of the allowable values for the argument.
- *required* - Whether or not the command-line option may be omitted (optionals only).
- *help* - A brief description of what the argument does.
- *metavar* - A name for the argument in usage messages.
- *dest* - The name of the attribute to be added to the object returned by *parse_args()*.
- *deprecated* - Whether or not use of the argument is deprecated.

The following sections describe how each of these are used.

name or flags

The *add_argument()* method must know whether an optional argument, like `-f` or `--foo`, or a positional argument, like a list of filenames, is expected. The first arguments passed to *add_argument()* must therefore be either a series of flags, or a simple argument name.

For example, an optional argument could be created like:

```
>>> parser.add_argument('-f', '--foo')
```

while a positional argument could be created like:

```
>>> parser.add_argument('bar')
```

When *parse_args()* is called, optional arguments will be identified by the - prefix, and the remaining arguments will be assumed to be positional:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
usage: PROG [-h] [-f FOO] bar
PROG: error: the following arguments are required: bar
```

action

ArgumentParser objects associate command-line arguments with actions. These actions can do just about anything with the command-line arguments associated with them, though most actions simply add an attribute to the object returned by *parse_args()*. The `action` keyword argument specifies how the command-line arguments should be handled. The supplied actions are:

- `'store'` - This just stores the argument's value. This is the default action. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args('--foo 1'.split())
Namespace(foo='1')
```

- `'store_const'` - This stores the value specified by the *const* keyword argument; note that the *const* keyword argument defaults to `None`. The `'store_const'` action is most commonly used with optional arguments that specify some sort of flag. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_const', const=42)
>>> parser.parse_args(['--foo'])
Namespace(foo=42)
```

- `'store_true'` and `'store_false'` - These are special cases of `'store_const'` used for storing the values `True` and `False` respectively. In addition, they create default values of `False` and `True` respectively. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('--bar', action='store_false')
>>> parser.add_argument('--baz', action='store_false')
>>> parser.parse_args('--foo --bar'.split())
Namespace(foo=True, bar=False, baz=True)
```

- `'append'` - This stores a list, and appends each argument value to the list. It is useful to allow an option to be specified multiple times. If the default value is non-empty, the default elements will be present in the parsed value for the option, with any values from the command line appended after those default values. Example usage:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])
```

- `'append_const'` - This stores a list, and appends the value specified by the *const* keyword argument to the list; note that the *const* keyword argument defaults to `None`. The `'append_const'` action is typically useful when

multiple arguments need to store constants to the same list. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const', const=str)
>>> parser.add_argument('--int', dest='types', action='append_const', const=int)
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<class 'str'>, <class 'int'>])
```

- `'count'` - This counts the number of times a keyword argument occurs. For example, this is useful for increasing verbosity levels:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count', default=0)
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```

  Note, the *default* will be `None` unless explicitly set to *0*.

- `'help'` - This prints a complete help message for all the options in the current parser and then exits. By default a help action is automatically added to the parser. See *ArgumentParser* for details of how the output is created.

- `'version'` - This expects a `version=` keyword argument in the *add_argument()* call, and prints version information and exits when invoked:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--version', action='version', version='%(prog)s 2.0')
>>> parser.parse_args(['--version'])
PROG 2.0
```

- `'extend'` - This stores a list, and extends each argument value to the list. Example usage:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument("--foo", action="extend", nargs="+", type=str)
>>> parser.parse_args(["--foo", "f1", "--foo", "f2", "f3", "f4"])
Namespace(foo=['f1', 'f2', 'f3', 'f4'])
```

  Added in version 3.8.

You may also specify an arbitrary action by passing an Action subclass or other object that implements the same interface. The `BooleanOptionalAction` is available in `argparse` and adds support for boolean actions such as `--foo` and `--no-foo`:

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=argparse.BooleanOptionalAction)
>>> parser.parse_args(['--no-foo'])
Namespace(foo=False)
```

Added in version 3.9.

The recommended way to create a custom action is to extend *Action*, overriding the `__call__` method and optionally the `__init__` and `format_usage` methods.

An example of a custom action:

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
```

(continues on next page)

(continued from previous page)

```
...             raise ValueError("nargs not allowed")
...         super().__init__(option_strings, dest, **kwargs)
...     def __call__(self, parser, namespace, values, option_string=None):
...         print('%r %r %r' % (namespace, values, option_string))
...         setattr(namespace, self.dest, values)
...
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
>>> args = parser.parse_args('1 --foo 2'.split())
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

For more details, see *Action*.

nargs

ArgumentParser objects usually associate a single command-line argument with a single action to be taken. The `nargs` keyword argument associates a different number of command-line arguments with a single action. See also specifying-ambiguous-arguments. The supported values are:

- `N` (an integer). `N` arguments from the command line will be gathered together into a list. For example:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs=2)
  >>> parser.add_argument('bar', nargs=1)
  >>> parser.parse_args('c --foo a b'.split())
  Namespace(bar=['c'], foo=['a', 'b'])
  ```

  Note that `nargs=1` produces a list of one item. This is different from the default, in which the item is produced by itself.

- `'?'`. One argument will be consumed from the command line if possible, and produced as a single item. If no command-line argument is present, the value from *default* will be produced. Note that for optional arguments, there is an additional case - the option string is present but not followed by a command-line argument. In this case the value from *const* will be produced. Some examples to illustrate this:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs='?', const='c', default='d')
  >>> parser.add_argument('bar', nargs='?', default='d')
  >>> parser.parse_args(['XX', '--foo', 'YY'])
  Namespace(bar='XX', foo='YY')
  >>> parser.parse_args(['XX', '--foo'])
  Namespace(bar='XX', foo='c')
  >>> parser.parse_args([])
  Namespace(bar='d', foo='d')
  ```

  One of the more common uses of `nargs='?'` is to allow optional input and output files:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('infile', nargs='?', type=argparse.FileType('r'),
  ...                     default=sys.stdin)
  >>> parser.add_argument('outfile', nargs='?', type=argparse.FileType('w'),
  ```

(continues on next page)

(continued from previous page)

```
...                     default=sys.stdout)
>>> parser.parse_args(['input.txt', 'output.txt'])
Namespace(infile=<_io.TextIOWrapper name='input.txt' encoding='UTF-8'>,
          outfile=<_io.TextIOWrapper name='output.txt' encoding='UTF-8'>)
>>> parser.parse_args([])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>,
          outfile=<_io.TextIOWrapper name='<stdout>' encoding='UTF-8'>)
```

- `'*'`. All command-line arguments present are gathered into a list. Note that it generally doesn't make much sense to have more than one positional argument with `nargs='*'`, but multiple optional arguments with `nargs='*'` is possible. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args('a b --foo x y --bar 1 2'.split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
```

- `'+'`. Just like `'*'`, all command-line args present are gathered into a list. Additionally, an error message will be generated if there wasn't at least one command-line argument present. For example:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args(['a', 'b'])
Namespace(foo=['a', 'b'])
>>> parser.parse_args([])
usage: PROG [-h] foo [foo ...]
PROG: error: the following arguments are required: foo
```

If the `nargs` keyword argument is not provided, the number of arguments consumed is determined by the *action*. Generally this means a single command-line argument will be consumed and a single item (not a list) will be produced.

const

The `const` argument of *`add_argument()`* is used to hold constant values that are not read from the command line but are required for the various *`ArgumentParser`* actions. The two most common uses of it are:

- When *`add_argument()`* is called with `action='store_const'` or `action='append_const'`. These actions add the `const` value to one of the attributes of the object returned by *`parse_args()`*. See the *action* description for examples. If `const` is not provided to *`add_argument()`*, it will receive a default value of `None`.
- When *`add_argument()`* is called with option strings (like `-f` or `--foo`) and `nargs='?'`. This creates an optional argument that can be followed by zero or one command-line arguments. When parsing the command line, if the option string is encountered with no command-line argument following it, the value of `const` will be assumed to be `None` instead. See the *nargs* description for examples.

Changed in version 3.11: `const=None` by default, including when `action='append_const'` or `action='store_const'`.

default

All optional arguments and some positional arguments may be omitted at the command line. The `default` keyword argument of *`add_argument()`*, whose value defaults to `None`, specifies what value should be used if the command-line argument is not present. For optional arguments, the `default` value is used when the option string was not present at the command line:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

If the target namespace already has an attribute set, the action *default* will not over write it:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args([], namespace=argparse.Namespace(foo=101))
Namespace(foo=101)
```

If the `default` value is a string, the parser parses the value as if it were a command-line argument. In particular, the parser applies any *type* conversion argument, if provided, before setting the attribute on the *`Namespace`* return value. Otherwise, the parser uses the value as is:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

For positional arguments with *nargs* equal to `?` or `*`, the `default` value is used when no command-line argument was present:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

Providing `default=argparse.SUPPRESS` causes no attribute to be added if the command-line argument was not present:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

type

By default, the parser reads command-line arguments in as simple strings. However, quite often the command-line string should instead be interpreted as another type, such as a *float* or *int*. The `type` keyword for *add_argument()* allows any necessary type-checking and type conversions to be performed.

If the *type* keyword is used with the *default* keyword, the type converter is only applied if the default is a string.

The argument to `type` can be any callable that accepts a single string. If the function raises *ArgumentTypeError*, *TypeError*, or *ValueError*, the exception is caught and a nicely formatted error message is displayed. No other exception types are handled.

Common built-in types and functions can be used as type converters:

```
import argparse
import pathlib

parser = argparse.ArgumentParser()
parser.add_argument('count', type=int)
parser.add_argument('distance', type=float)
parser.add_argument('street', type=ascii)
parser.add_argument('code_point', type=ord)
parser.add_argument('source_file', type=open)
parser.add_argument('dest_file', type=argparse.FileType('w', encoding='latin-1'))
parser.add_argument('datapath', type=pathlib.Path)
```

User defined functions can be used as well:

```
>>> def hyphenated(string):
...     return '-'.join([word[:4] for word in string.casefold().split()])
...
>>> parser = argparse.ArgumentParser()
>>> _ = parser.add_argument('short_title', type=hyphenated)
>>> parser.parse_args(['"The Tale of Two Cities"'])
Namespace(short_title='"the-tale-of-two-citi')
```

The *bool()* function is not recommended as a type converter. All it does is convert empty strings to `False` and non-empty strings to `True`. This is usually not what is desired.

In general, the `type` keyword is a convenience that should only be used for simple conversions that can only raise one of the three supported exceptions. Anything with more interesting error-handling or resource management should be done downstream after the arguments are parsed.

For example, JSON or YAML conversions have complex error cases that require better reporting than can be given by the `type` keyword. A *JSONDecodeError* would not be well formatted and a *FileNotFoundError* exception would not be handled at all.

Even *FileType* has its limitations for use with the `type` keyword. If one argument uses *FileType* and then a subsequent argument fails, an error is reported but the file is not automatically closed. In this case, it would be better to wait until after the parser has run and then use the `with`-statement to manage the files.

For type checkers that simply check against a fixed set of values, consider using the *choices* keyword instead.

choices

Some command-line arguments should be selected from a restricted set of values. These can be handled by passing a sequence object as the *choices* keyword argument to *add_argument()*. When the command line is parsed, argument values will be checked, and an error message will be displayed if the argument was not one of the acceptable values:

```
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock',
'paper', 'scissors')
```

Note that inclusion in the *choices* sequence is checked after any *type* conversions have been performed, so the type of the objects in the *choices* sequence should match the *type* specified:

```
>>> parser = argparse.ArgumentParser(prog='doors.py')
>>> parser.add_argument('door', type=int, choices=range(1, 4))
>>> print(parser.parse_args(['3']))
Namespace(door=3)
>>> parser.parse_args(['4'])
usage: doors.py [-h] {1,2,3}
doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)
```

Any sequence can be passed as the *choices* value, so *list* objects, *tuple* objects, and custom sequences are all supported.

Use of *enum.Enum* is not recommended because it is difficult to control its appearance in usage, help, and error messages.

Formatted choices override the default *metavar* which is normally derived from *dest*. This is usually what you want because the user never sees the *dest* parameter. If this display isn't desirable (perhaps because there are many choices), just specify an explicit *metavar*.

required

In general, the *argparse* module assumes that flags like `-f` and `--bar` indicate *optional* arguments, which can always be omitted at the command line. To make an option *required*, `True` can be specified for the `required=` keyword argument to *add_argument()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: [-h] --foo FOO
: error: the following arguments are required: --foo
```

As the example shows, if an option is marked as `required`, *parse_args()* will report an error if that option is not present at the command line.

> **Note**
>
> Required options are generally considered bad form because users expect *options* to be *optional*, and thus they should be avoided when possible.

help

The `help` value is a string containing a brief description of the argument. When a user requests help (usually by using `-h` or `--help` at the command line), these `help` descriptions will be displayed with each argument:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', action='store_true',
...                     help='foo the bars before frobbling')
>>> parser.add_argument('bar', nargs='+',
...                     help='one of the bars to be frobbled')
>>> parser.parse_args(['-h'])
usage: frobble [-h] [--foo] bar [bar ...]

positional arguments:
 bar     one of the bars to be frobbled

options:
 -h, --help  show this help message and exit
 --foo   foo the bars before frobbling
```

The `help` strings can include various format specifiers to avoid repetition of things like the program name or the argument *default*. The available specifiers include the program name, `%(prog)s` and most keyword arguments to *add_argument()*, e.g. `%(default)s`, `%(type)s`, etc.:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...                     help='the bar to %(prog)s (default: %(default)s)')
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
 bar     the bar to frobble (default: 42)

options:
 -h, --help  show this help message and exit
```

As the help string supports %-formatting, if you want a literal `%` to appear in the help string, you must escape it as `%%`.

argparse supports silencing the help entry for certain options, by setting the `help` value to `argparse.SUPPRESS`:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]

options:
  -h, --help  show this help message and exit
```

metavar

When *ArgumentParser* generates help messages, it needs some way to refer to each expected argument. By default, ArgumentParser objects use the *dest* value as the "name" of each object. By default, for positional argument actions, the *dest* value is used directly, and for optional argument actions, the *dest* value is uppercased. So, a single positional argument with `dest='bar'` will be referred to as `bar`. A single optional argument `--foo` that should be followed by a single command-line argument will be referred to as `FOO`. An example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo FOO] bar

positional arguments:
 bar

options:
 -h, --help  show this help message and exit
 --foo FOO
```

An alternative name can be specified with `metavar`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo YYY] XXX

positional arguments:
 XXX

options:
 -h, --help  show this help message and exit
 --foo YYY
```

Note that `metavar` only changes the *displayed* name - the name of the attribute on the *parse_args()* object is still determined by the *dest* value.

Different values of `nargs` may cause the metavar to be used multiple times. Providing a tuple to `metavar` specifies a different display for each of the arguments:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
>>> parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
>>> parser.print_help()
usage: PROG [-h] [-x X X] [--foo bar baz]

options:
 -h, --help     show this help message and exit
 -x X X
 --foo bar baz
```

dest

Most *ArgumentParser* actions add some value as an attribute of the object returned by *parse_args()*. The name of this attribute is determined by the `dest` keyword argument of *add_argument()*. For positional argument actions, `dest` is normally supplied as the first argument to *add_argument()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

For optional argument actions, the value of `dest` is normally inferred from the option strings. *ArgumentParser* generates the value of `dest` by taking the first long option string and stripping away the initial `--` string. If no long option strings were supplied, `dest` will be derived from the first short option string by stripping the initial `-` character. Any internal `-` characters will be converted to `_` characters to make sure the string is a valid attribute name. The examples below illustrate this behavior:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
>>> parser.parse_args('--foo 1 -y 2'.split())
Namespace(foo_bar='1', x='2')
```

`dest` allows a custom attribute name to be provided:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

deprecated

During a project's lifetime, some arguments may need to be removed from the command line. Before removing them, you should inform your users that the arguments are deprecated and will be removed. The `deprecated` keyword argument of *add_argument()*, which defaults to `False`, specifies if the argument is deprecated and will be removed in the future. For arguments, if `deprecated` is `True`, then a warning will be printed to standard error when the argument is used:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='snake.py')
>>> parser.add_argument('--legs', default=0, type=int, deprecated=True)
>>> parser.parse_args([])
Namespace(legs=0)
>>> parser.parse_args(['--legs', '4'])
snake.py: warning: option '--legs' is deprecated
Namespace(legs=4)
```

Added in version 3.13.

Action classes

Action classes implement the Action API, a callable which returns a callable which processes arguments from the command-line. Any object which follows this API may be passed as the `action` parameter to *`add_argument()`*.

class argparse.**Action**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Action objects are used by an ArgumentParser to represent the information needed to parse a single argument from one or more strings from the command line. The Action class must accept the two positional arguments plus any keyword arguments passed to *`ArgumentParser.add_argument()`* except for the `action` itself.

Instances of Action (or return value of any callable to the `action` parameter) should have attributes "dest", "option_strings", "default", "type", "required", "help", etc. defined. The easiest way to ensure these attributes are defined is to call `Action.__init__`.

Action instances should be callable, so subclasses must override the `__call__` method, which should accept four parameters:

- `parser` - The ArgumentParser object which contains this action.
- `namespace` - The *`Namespace`* object that will be returned by *`parse_args()`*. Most actions add an attribute to this object using *`setattr()`*.
- `values` - The associated command-line arguments, with any type conversions applied. Type conversions are specified with the *type* keyword argument to *`add_argument()`*.
- `option_string` - The option string that was used to invoke this action. The `option_string` argument is optional, and will be absent if the action is associated with a positional argument.

The `__call__` method may perform arbitrary actions, but will typically set attributes on the `namespace` based on `dest` and `values`.

Action subclasses can define a `format_usage` method that takes no argument and return a string which will be used when printing the usage of the program. If such method is not provided, a sensible default will be used.

16.4.6 The parse_args() method

ArgumentParser.**parse_args**(*args=None*, *namespace=None*)

Convert argument strings to objects and assign them as attributes of the namespace. Return the populated namespace.

Previous calls to *`add_argument()`* determine exactly what objects are created and how they are assigned. See the documentation for *`add_argument()`* for details.

- *args* - List of strings to parse. The default is taken from *`sys.argv`*.
- *namespace* - An object to take the attributes. The default is a new empty *`Namespace`* object.

Option value syntax

The *parse_args()* method supports several ways of specifying the value of an option (if it takes one). In the simplest case, the option and its value are passed as two separate arguments:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo', 'FOO'])
Namespace(foo='FOO', x=None)
```

For long options (options with names longer than a single character), the option and value can also be passed as a single command-line argument, using = to separate them:

```
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

For short options (options only one character long), the option and its value can be concatenated:

```
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

Several short options can be joined together, using only a single - prefix, as long as only the last option (or none of them) requires a value:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
>>> parser.add_argument('-z')
>>> parser.parse_args(['-xyzZ'])
Namespace(x=True, y=True, z='Z')
```

Invalid arguments

While parsing the command line, *parse_args()* checks for a variety of errors, including ambiguous options, invalid types, invalid options, wrong number of positional arguments, etc. When it encounters such an error, it exits and prints the error along with a usage message:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

>>> # invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'

>>> # invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

>>> # wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
```

(continues on next page)

(continued from previous page)

```
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```

Arguments containing –

The *parse_args()* method attempts to give errors whenever the user has clearly made a mistake, but some situations are inherently ambiguous. For example, the command-line argument `-1` could either be an attempt to specify an option or an attempt to provide a positional argument. The *parse_args()* method is cautious here: positional arguments may only begin with `-` if they look like negative numbers and there are no options in the parser that look like negative numbers:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

If you have positional arguments that must begin with `-` and don't look like negative numbers, you can insert the pseudo-argument `'--'` which tells *parse_args()* that everything after that is a positional argument:

```
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

See also the argparse howto on ambiguous arguments for more details.

Argument abbreviations (prefix matching)

The *parse_args()* method *by default* allows long options to be abbreviated to a prefix, if the abbreviation is unambiguous (the prefix matches a unique option):

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args('-bac MMM'.split())
Namespace(bacon='MMM', badger=None)
>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args('-ba BA'.split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

An error is produced for arguments that could produce more than one options. This feature can be disabled by setting *allow_abbrev* to `False`.

Beyond `sys.argv`

Sometimes it may be useful to have an ArgumentParser parse arguments other than those of *sys.argv*. This can be accomplished by passing a list of strings to *parse_args()*. This is useful for testing at the interactive prompt:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=range(10),
...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(
...     '--sum', dest='accumulate', action='store_const', const=sum,
...     default=max, help='sum the integers (default: find the max)')
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args(['1', '2', '3', '4', '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

The Namespace object

class argparse.**Namespace**

Simple class used by default by *parse_args()* to create an object holding attributes and return it.

This class is deliberately simple, just an *object* subclass with a readable string representation. If you prefer to have dict-like view of the attributes, you can use the standard Python idiom, *vars()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

It may also be useful to have an *ArgumentParser* assign attributes to an already existing object, rather than a new *Namespace* object. This can be achieved by specifying the `namespace=` keyword argument:

```
>>> class C:
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

16.4.7 Other utilities

Sub-commands

ArgumentParser.**add_subparsers**([*title*][, *description*][, *prog*][, *parser_class*][, *action*][, *option_strings*][, *dest*][, *required*][, *help*][, *metavar*])

Many programs split up their functionality into a number of sub-commands, for example, the `svn` program can invoke sub-commands like `svn checkout`, `svn update`, and `svn commit`. Splitting up functionality this way can be a particularly good idea when a program performs several different functions which require different kinds of command-line arguments. *ArgumentParser* supports the creation of such sub-commands with the *add_subparsers()* method. The *add_subparsers()* method is normally called with no arguments and returns a special action object. This object has a single method, `add_parser()`, which takes a command name and any *ArgumentParser* constructor arguments, and returns an *ArgumentParser* object that can be modified as usual.

Description of parameters:

- title - title for the sub-parser group in help output; by default "subcommands" if description is provided, otherwise uses title for positional arguments
- description - description for the sub-parser group in help output, by default `None`
- prog - usage information that will be displayed with sub-command help, by default the name of the program and any positional arguments before the subparser argument
- parser_class - class which will be used to create sub-parser instances, by default the class of the current parser (e.g. ArgumentParser)
- *action* - the basic type of action to be taken when this argument is encountered at the command line
- *dest* - name of the attribute under which sub-command name will be stored; by default `None` and no value is stored
- *required* - Whether or not a subcommand must be provided, by default `False` (added in 3.7)
- *help* - help for sub-parser group in help output, by default `None`
- *metavar* - string presenting available sub-commands in help; by default it is `None` and presents sub-commands in form {cmd1, cmd2, ..}

Some example usage:

```
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='sub-command help')
>>>
```

(continues on next page)

(continued from previous page)

```
>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')
>>>
>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices='XYZ', help='baz help')
>>>
>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

Note that the object returned by *parse_args()* will only contain attributes for the main parser and the subparser that was selected by the command line (and not any other subparsers). So in the example above, when the `a` command is specified, only the `foo` and `bar` attributes are present, and when the `b` command is specified, only the `foo` and `baz` attributes are present.

Similarly, when a help message is requested from a subparser, only the help for that particular parser will be printed. The help message will not include parent parser or sibling parser messages. (A help message for each subparser command, however, can be given by supplying the `help=` argument to `add_parser()` as above.)

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}   sub-command help
    a     a help
    b     b help

options:
  -h, --help  show this help message and exit
  --foo   foo help

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar     bar help

options:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

options:
  -h, --help     show this help message and exit
  --baz {X,Y,Z}  baz help
```

The *add_subparsers()* method also supports `title` and `description` keyword arguments. When either is present, the subparser's commands will appear in their own group in the help output. For example:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
```

(continues on next page)

(continued from previous page)

```
...                                     description='valid subcommands',
...                                     help='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage:  [-h] {foo,bar} ...

options:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands

  {foo,bar}   additional help
```

Furthermore, `add_parser()` supports an additional *aliases* argument, which allows multiple strings to refer to the same subparser. This example, like `svn`, aliases `co` as a shorthand for `checkout`:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>> checkout = subparsers.add_parser('checkout', aliases=['co'])
>>> checkout.add_argument('foo')
>>> parser.parse_args(['co', 'bar'])
Namespace(foo='bar')
```

`add_parser()` supports also an additional *deprecated* argument, which allows to deprecate the subparser.

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='chicken.py')
>>> subparsers = parser.add_subparsers()
>>> run = subparsers.add_parser('run')
>>> fly = subparsers.add_parser('fly', deprecated=True)
>>> parser.parse_args(['fly'])
chicken.py: warning: command 'fly' is deprecated
Namespace()
```

Added in version 3.13.

One particularly effective way of handling sub-commands is to combine the use of the *`add_subparsers()`* method with calls to *`set_defaults()`* so that each subparser knows which Python function it should execute. For example:

```
>>> # sub-command functions
>>> def foo(args):
...     print(args.x * args.y)
...
>>> def bar(args):
...     print('((%s))' % args.z)
...
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(required=True)
>>>
>>> # create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
```

(continues on next page)

(continued from previous page)

```
>>> parser_foo.set_defaults(func=foo)
>>>
>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
2.0
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))
```

This way, you can let *parse_args()* do the job of calling the appropriate function after argument parsing is complete. Associating functions with actions like this is typically the easiest way to handle the different actions for each of your subparsers. However, if it is necessary to check the name of the subparser that was invoked, the `dest` keyword argument to the *add_subparsers()* call will work:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

Changed in version 3.7: New *required* keyword argument.

FileType objects

class argparse.**FileType**(*mode='r'*, *bufsize=-1*, *encoding=None*, *errors=None*)

The *FileType* factory creates objects that can be passed to the type argument of *ArgumentParser.add_argument()*. Arguments that have *FileType* objects as their type will open command-line arguments as files with the requested modes, buffer sizes, encodings and error handling (see the *open()* function for more details):

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--raw', type=argparse.FileType('wb', 0))
>>> parser.add_argument('out', type=argparse.FileType('w', encoding='UTF-8'))
>>> parser.parse_args(['--raw', 'raw.dat', 'file.txt'])
Namespace(out=<_io.TextIOWrapper name='file.txt' mode='w' encoding='UTF-8'>, raw=
↪<_io.FileIO name='raw.dat' mode='wb'>)
```

FileType objects understand the pseudo-argument `'-'` and automatically convert this into *sys.stdin* for readable *FileType* objects and *sys.stdout* for writable *FileType* objects:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>)
```

Changed in version 3.4: Added the *encodings* and *errors* parameters.

Argument groups

ArgumentParser.**add_argument_group**(*title=None*, *description=None*)

By default, *ArgumentParser* groups command-line arguments into "positional arguments" and "options" when displaying help messages. When there is a better conceptual grouping of arguments than this default one, appropriate groups can be created using the *add_argument_group()* method:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
  bar    bar help
  --foo FOO  foo help
```

The *add_argument_group()* method returns an argument group object which has an *add_argument()* method just like a regular *ArgumentParser*. When an argument is added to the group, the parser treats it just like a normal argument, but displays the argument in a separate group for help messages. The *add_argument_group()* method accepts *title* and *description* arguments which can be used to customize this display:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo

group1:
  group1 description

  foo    foo help

group2:
  group2 description

  --bar BAR  bar help
```

Note that any arguments not in your user-defined groups will end up back in the usual "positional arguments" and "optional arguments" sections.

Changed in version 3.11: Calling *add_argument_group()* on an argument group is deprecated. This feature was never supported and does not always work correctly. The function exists on the API by accident through inheritance and will be removed in the future.

Mutual exclusion

ArgumentParser.**add_mutually_exclusive_group**(*required=False*)

Create a mutually exclusive group. *argparse* will make sure that only one of the arguments in the mutually exclusive group was present on the command line:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

The *add_mutually_exclusive_group()* method also accepts a *required* argument, to indicate that at least one of the mutually exclusive arguments is required:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

Note that currently mutually exclusive argument groups do not support the *title* and *description* arguments of *add_argument_group()*. However, a mutually exclusive group can be added to an argument group that has a title and description. For example:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_argument_group('Group title', 'Group description')
>>> exclusive_group = group.add_mutually_exclusive_group(required=True)
>>> exclusive_group.add_argument('--foo', help='foo help')
>>> exclusive_group.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [-h] (--foo FOO | --bar BAR)

options:
  -h, --help  show this help message and exit

Group title:
  Group description

  --foo FOO   foo help
  --bar BAR   bar help
```

Changed in version 3.11: Calling *add_argument_group()* or *add_mutually_exclusive_group()* on a mutually exclusive group is deprecated. These features were never supported and do not always work correctly. The functions exist on the API by accident through inheritance and will be removed in the future.

Parser defaults

ArgumentParser.**set_defaults**(***kwargs*)

Most of the time, the attributes of the object returned by *parse_args()* will be fully determined by inspecting the command-line arguments and the argument actions. *set_defaults()* allows some additional attributes that are determined without any inspection of the command line to be added:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

Note that parser-level defaults always override argument-level defaults:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```

Parser-level defaults can be particularly useful when working with multiple parsers. See the *add_subparsers()* method for an example of this type.

ArgumentParser.**get_default**(*dest*)

Get the default value for a namespace attribute, as set by either *add_argument()* or by *set_defaults()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'
```

Printing help

In most typical applications, *parse_args()* will take care of formatting and printing any usage or error messages. However, several formatting methods are available:

ArgumentParser.**print_usage**(*file=None*)

Print a brief description of how the *ArgumentParser* should be invoked on the command line. If *file* is `None`, *sys.stdout* is assumed.

ArgumentParser.**print_help**(*file=None*)

Print a help message, including the program usage and information about the arguments registered with the *ArgumentParser*. If *file* is `None`, *sys.stdout* is assumed.

There are also variants of these methods that simply return a string instead of printing it:

ArgumentParser.**format_usage**()

Return a string containing a brief description of how the *ArgumentParser* should be invoked on the command line.

ArgumentParser.**format_help**()

Return a string containing a help message, including the program usage and information about the arguments registered with the *ArgumentParser*.

Partial parsing

ArgumentParser.**parse_known_args**(*args=None*, *namespace=None*)

Sometimes a script may only parse a few of the command-line arguments, passing the remaining arguments on to another script or program. In these cases, the *parse_known_args()* method can be useful. It works much like *parse_args()* except that it does not produce an error when extra arguments are present. Instead, it returns a two item tuple containing the populated namespace and the list of remaining argument strings.

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

Warning

Prefix matching rules apply to *parse_known_args()*. The parser may consume an option even if it's just a prefix of one of its known options, instead of leaving it in the remaining arguments list.

Customizing file parsing

ArgumentParser.**convert_arg_line_to_args**(*arg_line*)

Arguments that are read from a file (see the *fromfile_prefix_chars* keyword argument to the *ArgumentParser* constructor) are read one argument per line. *convert_arg_line_to_args()* can be overridden for fancier reading.

This method takes a single argument *arg_line* which is a string read from the argument file. It returns a list of arguments parsed from this string. The method is called once per line read from the argument file, in order.

A useful override of this method is one that treats each space-separated word as an argument. The following example demonstrates how to do this:

```
class MyArgumentParser(argparse.ArgumentParser):
    def convert_arg_line_to_args(self, arg_line):
        return arg_line.split()
```

Exiting methods

ArgumentParser.**exit**(*status=0*, *message=None*)

This method terminates the program, exiting with the specified *status* and, if given, it prints a *message* before that. The user can override this method to handle these steps differently:

```
class ErrorCatchingArgumentParser(argparse.ArgumentParser):
    def exit(self, status=0, message=None):
        if status:
            raise Exception(f'Exiting because of an error: {message}')
        exit(status)
```

ArgumentParser.**error**(*message*)

This method prints a usage message including the *message* to the standard error and terminates the program with a status code of 2.

Intermixed parsing

ArgumentParser.**parse_intermixed_args**(*args=None*, *namespace=None*)

ArgumentParser.**parse_known_intermixed_args**(*args=None*, *namespace=None*)

A number of Unix commands allow the user to intermix optional arguments with positional arguments. The *parse_intermixed_args()* and *parse_known_intermixed_args()* methods support this parsing style.

These parsers do not support all the argparse features, and will raise exceptions if unsupported features are used. In particular, subparsers, and mutually exclusive groups that include both optionals and positionals are not supported.

The following example shows the difference between *parse_known_args()* and *parse_intermixed_args()*: the former returns `['2', '3']` as unparsed arguments, while the latter collects all the positionals into `rest`.

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('cmd')
>>> parser.add_argument('rest', nargs='*', type=int)
>>> parser.parse_known_args('doit 1 --foo bar 2 3'.split())
(Namespace(cmd='doit', foo='bar', rest=[1]), ['2', '3'])
>>> parser.parse_intermixed_args('doit 1 --foo bar 2 3'.split())
Namespace(cmd='doit', foo='bar', rest=[1, 2, 3])
```

parse_known_intermixed_args() returns a two item tuple containing the populated namespace and the list of remaining argument strings. *parse_intermixed_args()* raises an error if there are any remaining unparsed argument strings.

Added in version 3.7.

16.4.8 Upgrading optparse code

Originally, the *argparse* module had attempted to maintain compatibility with *optparse*. However, *optparse* was difficult to extend transparently, particularly with the changes required to support the new `nargs=` specifiers and better usage messages. When most everything in *optparse* had either been copy-pasted over or monkey-patched, it no longer seemed practical to try to maintain the backwards compatibility.

The *argparse* module improves on the standard library *optparse* module in a number of ways including:

- Handling positional arguments.
- Supporting sub-commands.
- Allowing alternative option prefixes like + and `/`.
- Handling zero-or-more and one-or-more style arguments.
- Producing more informative usage messages.
- Providing a much simpler interface for custom `type` and `action`.

A partial upgrade path from *optparse* to *argparse*:

- Replace all *optparse.OptionParser.add_option()* calls with *ArgumentParser.add_argument()* calls.
- Replace `(options, args) = parser.parse_args()` with `args = parser.parse_args()` and add additional *ArgumentParser.add_argument()* calls for the positional arguments. Keep in mind that what was previously called `options`, now in the *argparse* context is called `args`.

- Replace *optparse.OptionParser.disable_interspersed_args()* by using *parse_intermixed_args()* instead of *parse_args()*.
- Replace callback actions and the `callback_*` keyword arguments with `type` or `action` arguments.
- Replace string names for `type` keyword arguments with the corresponding type objects (e.g. int, float, complex, etc).
- Replace *optparse.Values* with *Namespace* and *optparse.OptionError* and *optparse.OptionValueError* with *ArgumentError*.
- Replace strings with implicit arguments such as `%default` or `%prog` with the standard Python syntax to use dictionaries to format strings, that is, `%(default)s` and `%(prog)s`.
- Replace the OptionParser constructor `version` argument with a call to `parser.add_argument('--version', action='version', version='<the version>')`.

16.4.9 Exceptions

exception argparse.**ArgumentError**

An error from creating or using an argument (optional or positional).

The string value of this exception is the message, augmented with information about the argument that caused it.

exception argparse.**ArgumentTypeError**

Raised when something goes wrong converting a command line string to a type.

16.5 `logging` — Logging facility for Python

Source code: Lib/logging/__init__.py

Important

This page contains the API reference information. For tutorial information and discussion of more advanced topics, see

- Basic Tutorial
- Advanced Tutorial
- Logging Cookbook

This module defines functions and classes which implement a flexible event logging system for applications and libraries.

The key benefit of having the logging API provided by a standard library module is that all Python modules can participate in logging, so your application log can include your own messages integrated with messages from third-party modules.

Here's a simple example of idiomatic usage:

```
# myapp.py
import logging
import mylib
logger = logging.getLogger(__name__)
```

(continues on next page)

(continued from previous page)

```
def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logger.info('Started')
    mylib.do_something()
    logger.info('Finished')

if __name__ == '__main__':
    main()
```

```
# mylib.py
import logging
logger = logging.getLogger(__name__)

def do_something():
    logger.info('Doing something')
```

If you run *myapp.py*, you should see this in *myapp.log*:

```
INFO:__main__:Started
INFO:mylib:Doing something
INFO:__main__:Finished
```

The key feature of this idiomatic usage is that the majority of code is simply creating a module level logger with `getLogger(__name__)`, and using that logger to do any needed logging. This is concise, while allowing downstream code fine-grained control if needed. Logged messages to the module-level logger get forwarded to handlers of loggers in higher-level modules, all the way up to the highest-level logger known as the root logger; this approach is known as hierarchical logging.

For logging to be useful, it needs to be configured: setting the levels and destinations for each logger, potentially changing how specific modules log, often based on command-line arguments or application configuration. In most cases, like the one above, only the root logger needs to be so configured, since all the lower level loggers at module level eventually forward their messages to its handlers. *basicConfig()* provides a quick way to configure the root logger that handles many use cases.

The module provides a lot of functionality and flexibility. If you are unfamiliar with logging, the best way to get to grips with it is to view the tutorials (**see the links above and on the right**).

The basic classes defined by the module, together with their attributes and methods, are listed in the sections below.

- Loggers expose the interface that application code directly uses.
- Handlers send the log records (created by loggers) to the appropriate destination.
- Filters provide a finer grained facility for determining which log records to output.
- Formatters specify the layout of log records in the final output.

16.5.1 Logger Objects

Loggers have the following attributes and methods. Note that Loggers should *NEVER* be instantiated directly, but always through the module-level function `logging.getLogger(name)`. Multiple calls to *getLogger()* with the same name will always return a reference to the same Logger object.

The `name` is potentially a period-separated hierarchical value, like `foo.bar.baz` (though it could also be just plain `foo`, for example). Loggers that are further down in the hierarchical list are children of loggers higher up in the list. For example, given a logger with a name of `foo`, loggers with names of `foo.bar`, `foo.bar.baz`, and `foo.bam` are all descendants of `foo`. In addition, all loggers are descendants of the root logger. The logger name hierarchy is analogous to the Python package hierarchy, and identical to it if you organise your loggers on a per-module basis using the recommended construction `logging.getLogger(__name__)`. That's because in a module, `__name__` is the module's name in the Python package namespace.

class logging.**Logger**

name

This is the logger's name, and is the value that was passed to *getLogger()* to obtain the logger.

> **Note**
>
> This attribute should be treated as read-only.

level

The threshold of this logger, as set by the *setLevel()* method.

> **Note**
>
> Do not set this attribute directly - always use *setLevel()*, which has checks for the level passed to it.

parent

The parent logger of this logger. It may change based on later instantiation of loggers which are higher up in the namespace hierarchy.

> **Note**
>
> This value should be treated as read-only.

propagate

If this attribute evaluates to true, events logged to this logger will be passed to the handlers of higher level (ancestor) loggers, in addition to any handlers attached to this logger. Messages are passed directly to the ancestor loggers' handlers - neither the level nor filters of the ancestor loggers in question are considered.

If this evaluates to false, logging messages are not passed to the handlers of ancestor loggers.

Spelling it out with an example: If the propagate attribute of the logger named `A.B.C` evaluates to true, any event logged to `A.B.C` via a method call such as `logging.getLogger('A.B.C').error(...)` will [subject to passing that logger's level and filter settings] be passed in turn to any handlers attached to loggers named `A.B`, `A` and the root logger, after first being passed to any handlers attached to `A.B.C`. If any logger in the chain `A.B.C`, `A.B`, `A` has its `propagate` attribute set to false, then that is the last logger whose handlers are offered the event to handle, and propagation stops at that point.

The constructor sets this attribute to `True`.

> **Note**
>
> If you attach a handler to a logger *and* one or more of its ancestors, it may emit the same record multiple times. In general, you should not need to attach a handler to more than one logger - if you just attach it to the appropriate logger which is highest in the logger hierarchy, then it will see all events logged by all descendant loggers, provided that their propagate setting is left set to `True`. A common scenario is to attach handlers only to the root logger, and to let propagation take care of the rest.

handlers

The list of handlers directly attached to this logger instance.

> **Note**
>
> This attribute should be treated as read-only; it is normally changed via the *addHandler()* and *removeHandler()* methods, which use locks to ensure thread-safe operation.

disabled

This attribute disables handling of any events. It is set to `False` in the initializer, and only changed by logging configuration code.

> **Note**
>
> This attribute should be treated as read-only.

setLevel(*level*)

Sets the threshold for this logger to *level*. Logging messages which are less severe than *level* will be ignored; logging messages which have severity *level* or higher will be emitted by whichever handler or handlers service this logger, unless a handler's level has been set to a higher severity level than *level*.

When a logger is created, the level is set to *NOTSET* (which causes all messages to be processed when the logger is the root logger, or delegation to the parent when the logger is a non-root logger). Note that the root logger is created with level *WARNING*.

The term 'delegation to the parent' means that if a logger has a level of NOTSET, its chain of ancestor loggers is traversed until either an ancestor with a level other than NOTSET is found, or the root is reached.

If an ancestor is found with a level other than NOTSET, then that ancestor's level is treated as the effective level of the logger where the ancestor search began, and is used to determine how a logging event is handled.

If the root is reached, and it has a level of NOTSET, then all messages will be processed. Otherwise, the root's level will be used as the effective level.

See *Logging Levels* for a list of levels.

Changed in version 3.2: The *level* parameter now accepts a string representation of the level such as 'INFO' as an alternative to the integer constants such as *INFO*. Note, however, that levels are internally stored as integers, and methods such as e.g. *getEffectiveLevel()* and *isEnabledFor()* will return/expect to be passed integers.

isEnabledFor(*level*)

Indicates if a message of severity *level* would be processed by this logger. This method checks first the module-level level set by `logging.disable(level)` and then the logger's effective level as determined by *getEffectiveLevel()*.

getEffectiveLevel()

Indicates the effective level for this logger. If a value other than *NOTSET* has been set using *setLevel()*, it is returned. Otherwise, the hierarchy is traversed towards the root until a value other than *NOTSET* is found, and that value is returned. The value returned is an integer, typically one of *logging.DEBUG*, *logging.INFO* etc.

getChild(*suffix*)

Returns a logger which is a descendant to this logger, as determined by the suffix. Thus, `logging.getLogger('abc').getChild('def.ghi')` would return the same logger as would be returned by `logging.getLogger('abc.def.ghi')`. This is a convenience method, useful when the parent logger is named using e.g. `__name__` rather than a literal string.

Added in version 3.2.

getChildren()

Returns a set of loggers which are immediate children of this logger. So for example `logging.getLogger().getChildren()` might return a set containing loggers named `foo` and `bar`, but a logger named `foo.bar` wouldn't be included in the set. Likewise, `logging.getLogger('foo').getChildren()` might return a set including a logger named `foo.bar`, but it wouldn't include one named `foo.bar.baz`.

Added in version 3.12.

debug(*msg*, **args*, ***kwargs*)

Logs a message with level *DEBUG* on this logger. The *msg* is the message format string, and the *args* are the arguments which are merged into *msg* using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.) No % formatting operation is performed on *msg* when no *args* are supplied.

There are four keyword arguments in *kwargs* which are inspected: *exc_info*, *stack_info*, *stacklevel* and *extra*.

If *exc_info* does not evaluate as false, it causes exception information to be added to the logging message. If an exception tuple (in the format returned by *sys.exc_info()*) or an exception instance is provided, it is used; otherwise, *sys.exc_info()* is called to get the exception information.

The second optional keyword argument is *stack_info*, which defaults to `False`. If true, stack information is added to the logging message, including the actual logging call. Note that this is not the same stack information as that displayed through specifying *exc_info*: The former is stack frames from the bottom of the stack up to the logging call in the current thread, whereas the latter is information about stack frames which have been unwound, following an exception, while searching for exception handlers.

You can specify *stack_info* independently of *exc_info*, e.g. to just show how you got to a certain point in your code, even when no exceptions were raised. The stack frames are printed following a header line which says:

```
Stack (most recent call last):
```

This mimics the `Traceback (most recent call last):` which is used when displaying exception frames.

The third optional keyword argument is *stacklevel*, which defaults to `1`. If greater than 1, the corresponding number of stack frames are skipped when computing the line number and function name set in the *LogRecord* created for the logging event. This can be used in logging helpers so that the function name, filename and line number recorded are not the information for the helper function/method, but rather its caller. The name of this parameter mirrors the equivalent one in the *warnings* module.

The fourth keyword argument is *extra* which can be used to pass a dictionary which is used to populate the __dict__ of the *LogRecord* created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example:

```
FORMAT = '%(asctime)s %(clientip)-15s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset', extra=d)
```

would print something like

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection␣
↪reset
```

The keys in the dictionary passed in *extra* should not clash with the keys used by the logging system. (See the section on *LogRecord attributes* for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the `Formatter` has been set up with a format string which expects 'clientip' and 'user' in the attribute dictionary of the `LogRecord`. If these are missing, the message will not be logged because a string formatting exception will occur. So in this case, you always need to pass the *extra* dictionary with these keys.

While this might be annoying, this feature is intended for use in specialized circumstances, such as multi-threaded servers where the same code executes in many contexts, and interesting conditions which arise are dependent on this context (such as remote client IP address and authenticated user name, in the above example). In such circumstances, it is likely that specialized `Formatter`s would be used with particular `Handler`s.

If no handler is attached to this logger (or any of its ancestors, taking into account the relevant `Logger.propagate` attributes), the message will be sent to the handler set on `lastResort`.

Changed in version 3.2: The *stack_info* parameter was added.

Changed in version 3.5: The *exc_info* parameter can now accept exception instances.

Changed in version 3.8: The *stacklevel* parameter was added.

Changed in version 3.13: Remove the undocumented `warn()` method which was an alias to the `warning()` method.

info(*msg*, **args*, ***kwargs*)

Logs a message with level `INFO` on this logger. The arguments are interpreted as for `debug()`.

warning(*msg*, **args*, ***kwargs*)

Logs a message with level `WARNING` on this logger. The arguments are interpreted as for `debug()`.

error(*msg*, **args*, ***kwargs*)

Logs a message with level `ERROR` on this logger. The arguments are interpreted as for `debug()`.

critical(*msg*, **args*, ***kwargs*)

Logs a message with level `CRITICAL` on this logger. The arguments are interpreted as for `debug()`.

log(*level*, *msg*, **args*, ***kwargs*)

Logs a message with integer level *level* on this logger. The other arguments are interpreted as for `debug()`.

exception(*msg*, **args*, ***kwargs*)

Logs a message with level `ERROR` on this logger. The arguments are interpreted as for `debug()`. Exception info is added to the logging message. This method should only be called from an exception handler.

addFilter(*filter*)

Adds the specified filter *filter* to this logger.

removeFilter(*filter*)

Removes the specified filter *filter* from this logger.

filter(*record*)

Apply this logger's filters to the record and return `True` if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be processed (passed to handlers). If one returns a false value, no further processing of the record occurs.

addHandler(*hdlr*)

Adds the specified handler *hdlr* to this logger.

removeHandler(*hdlr*)

Removes the specified handler *hdlr* from this logger.

findCaller(*stack_info=False*, *stacklevel=1*)

Finds the caller's source filename and line number. Returns the filename, line number, function name and stack information as a 4-element tuple. The stack information is returned as `None` unless *stack_info* is `True`.

The *stacklevel* parameter is passed from code calling the *debug()* and other APIs. If greater than 1, the excess is used to skip stack frames before determining the values to be returned. This will generally be useful when calling logging APIs from helper/wrapper code, so that the information in the event log refers not to the helper/wrapper code, but to the code that calls it.

handle(*record*)

Handles a record by passing it to all handlers associated with this logger and its ancestors (until a false value of *propagate* is found). This method is used for unpickled records received from a socket, as well as those created locally. Logger-level filtering is applied using *filter()*.

makeRecord(*name*, *level*, *fn*, *lno*, *msg*, *args*, *exc_info*, *func=None*, *extra=None*, *sinfo=None*)

This is a factory method which can be overridden in subclasses to create specialized *LogRecord* instances.

hasHandlers()

Checks to see if this logger has any handlers configured. This is done by looking for handlers in this logger and its parents in the logger hierarchy. Returns `True` if a handler was found, else `False`. The method stops searching up the hierarchy whenever a logger with the 'propagate' attribute set to false is found - that will be the last logger which is checked for the existence of handlers.

Added in version 3.2.

Changed in version 3.7: Loggers can now be pickled and unpickled.

16.5.2 Logging Levels

The numeric values of logging levels are given in the following table. These are primarily of interest if you want to define your own levels, and need them to have specific values relative to the predefined levels. If you define a level with the same numeric value, it overwrites the predefined value; the predefined name is lost.

Level	Numeric value	What it means / When to use it
`logging.`**`NOTSET`**	0	When set on a logger, indicates that ancestor loggers are to be consulted to determine the effective level. If that still resolves to `NOTSET`, then all events are logged. When set on a handler, all events are handled.
`logging.`**`DEBUG`**	10	Detailed information, typically only of interest to a developer trying to diagnose a problem.
`logging.`**`INFO`**	20	Confirmation that things are working as expected.
`logging.`**`WARNING`**	30	An indication that something unexpected happened, or that a problem might occur in the near future (e.g. 'disk space low'). The software is still working as expected.
`logging.`**`ERROR`**	40	Due to a more serious problem, the software has not been able to perform some function.
`logging.`**`CRITICAL`**	50	A serious error, indicating that the program itself may be unable to continue running.

16.5.3 Handler Objects

Handlers have the following attributes and methods. Note that *Handler* is never instantiated directly; this class acts as a base for more useful subclasses. However, the `__init__()` method in subclasses needs to call *Handler.__init__()*.

class logging.**Handler**

__init__(*level=NOTSET*)

Initializes the *Handler* instance by setting its level, setting the list of filters to the empty list and creating a lock (using *createLock()*) for serializing access to an I/O mechanism.

createLock()

Initializes a thread lock which can be used to serialize access to underlying I/O functionality which may not be threadsafe.

acquire()

Acquires the thread lock created with *createLock()*.

release()

Releases the thread lock acquired with *acquire()*.

setLevel(*level*)

Sets the threshold for this handler to *level*. Logging messages which are less severe than *level* will be ignored. When a handler is created, the level is set to *NOTSET* (which causes all messages to be processed).

See *Logging Levels* for a list of levels.

Changed in version 3.2: The *level* parameter now accepts a string representation of the level such as 'INFO' as an alternative to the integer constants such as `INFO`.

setFormatter(*fmt*)

Sets the `Formatter` for this handler to *fmt*.

addFilter(*filter*)

Adds the specified filter *filter* to this handler.

removeFilter(*filter*)

Removes the specified filter *filter* from this handler.

filter(*record*)

Apply this handler's filters to the record and return `True` if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be emitted. If one returns a false value, the handler will not emit the record.

flush()

Ensure all logging output has been flushed. This version does nothing and is intended to be implemented by subclasses.

close()

Tidy up any resources used by the handler. This version does no output but removes the handler from an internal list of handlers which is closed when `shutdown()` is called. Subclasses should ensure that this gets called from overridden `close()` methods.

handle(*record*)

Conditionally emits the specified logging record, depending on filters which may have been added to the handler. Wraps the actual emission of the record with acquisition/release of the I/O thread lock.

handleError(*record*)

This method should be called from handlers when an exception is encountered during an `emit()` call. If the module-level attribute `raiseExceptions` is `False`, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The specified record is the one which was being processed when the exception occurred. (The default value of `raiseExceptions` is `True`, as that is more useful during development).

format(*record*)

Do formatting for a record - if a formatter is set, use it. Otherwise, use the default formatter for the module.

emit(*record*)

Do whatever it takes to actually log the specified logging record. This version is intended to be implemented by subclasses and so raises a `NotImplementedError`.

> **Warning**
>
> This method is called after a handler-level lock is acquired, which is released after this method returns. When you override this method, note that you should be careful when calling anything that invokes other parts of the logging API which might do locking, because that might result in a deadlock. Specifically:
>
> - Logging configuration APIs acquire the module-level lock, and then individual handler-level locks as those handlers are configured.
> - Many logging APIs lock the module-level lock. If such an API is called from this method, it could cause a deadlock if a configuration call is made on another thread, because that thread will try to acquire the module-level lock *before* the handler-level lock, whereas this thread tries to acquire the

module-level lock *after* the handler-level lock (because in this method, the handler-level lock has already been acquired).

For a list of handlers included as standard, see *logging.handlers*.

16.5.4 Formatter Objects

class logging.**Formatter**(*fmt=None*, *datefmt=None*, *style='%'*, *validate=True*, *, *defaults=None*)

Responsible for converting a *LogRecord* to an output string to be interpreted by a human or external system.

Parameters

- **fmt** (str) – A format string in the given *style* for the logged output as a whole. The possible mapping keys are drawn from the *LogRecord* object's *LogRecord attributes*. If not specified, '%(message)s' is used, which is just the logged message.
- **datefmt** (str) – A format string in the given *style* for the date/time portion of the logged output. If not specified, the default described in *formatTime()* is used.
- **style** (str) – Can be one of '%', '{' or '$' and determines how the format string will be merged with its data: using one of *printf-style String Formatting* (%), *str.format()* ({) or *string.Template* ($). This only applies to *fmt* and *datefmt* (e.g. '%(message)s' versus '{message}'), not to the actual log messages passed to the logging methods. However, there are other ways to use {- and $-formatting for log messages.
- **validate** (bool) – If True (the default), incorrect or mismatched *fmt* and *style* will raise a *ValueError*; for example, logging.Formatter('%(asctime)s - %(message)s', style='{').
- **defaults** (dict[str, Any]) – A dictionary with default values to use in custom fields. For example, logging.Formatter('%(ip)s %(message)s', defaults={"ip": None})

Changed in version 3.2: Added the *style* parameter.

Changed in version 3.8: Added the *validate* parameter.

Changed in version 3.10: Added the *defaults* parameter.

format(*record*)

The record's attribute dictionary is used as the operand to a string formatting operation. Returns the resulting string. Before formatting the dictionary, a couple of preparatory steps are carried out. The *message* attribute of the record is computed using *msg % args*. If the formatting string contains '(asctime)', *formatTime()* is called to format the event time. If there is exception information, it is formatted using *formatException()* and appended to the message. Note that the formatted exception information is cached in attribute *exc_text*. This is useful because the exception information can be pickled and sent across the wire, but you should be careful if you have more than one *Formatter* subclass which customizes the formatting of exception information. In this case, you will have to clear the cached value (by setting the *exc_text* attribute to None) after a formatter has done its formatting, so that the next formatter to handle the event doesn't use the cached value, but recalculates it afresh.

If stack information is available, it's appended after the exception information, using *formatStack()* to transform it if necessary.

formatTime(*record*, *datefmt=None*)

This method should be called from *format()* by a formatter which wants to make use of a formatted time. This method can be overridden in formatters to provide for any specific requirement, but the basic behavior

is as follows: if *datefmt* (a string) is specified, it is used with `time.strftime()` to format the creation time of the record. Otherwise, the format '%Y-%m-%d %H:%M:%S,uuu' is used, where the uuu part is a millisecond value and the other letters are as per the `time.strftime()` documentation. An example time in this format is `2003-01-23 00:29:50,411`. The resulting string is returned.

This function uses a user-configurable function to convert the creation time to a tuple. By default, `time.localtime()` is used; to change this for a particular formatter instance, set the `converter` attribute to a function with the same signature as `time.localtime()` or `time.gmtime()`. To change it for all formatters, for example if you want all logging times to be shown in GMT, set the `converter` attribute in the `Formatter` class.

Changed in version 3.3: Previously, the default format was hard-coded as in this example: `2010-09-06 22:38:15,292` where the part before the comma is handled by a strptime format string (`'%Y-%m-%d %H:%M:%S'`), and the part after the comma is a millisecond value. Because strptime does not have a format placeholder for milliseconds, the millisecond value is appended using another format string, `'%s,%03d'` — and both of these format strings have been hardcoded into this method. With the change, these strings are defined as class-level attributes which can be overridden at the instance level when desired. The names of the attributes are `default_time_format` (for the strptime format string) and `default_msec_format` (for appending the millisecond value).

Changed in version 3.9: The `default_msec_format` can be `None`.

formatException(*exc_info*)

Formats the specified exception information (a standard exception tuple as returned by `sys.exc_info()`) as a string. This default implementation just uses `traceback.print_exception()`. The resulting string is returned.

formatStack(*stack_info*)

Formats the specified stack information (a string as returned by `traceback.print_stack()`, but with the last newline removed) as a string. This default implementation just returns the input value.

class logging.**BufferingFormatter**(*linefmt=None*)

A base formatter class suitable for subclassing when you want to format a number of records. You can pass a `Formatter` instance which you want to use to format each line (that corresponds to a single record). If not specified, the default formatter (which just outputs the event message) is used as the line formatter.

formatHeader(*records*)

Return a header for a list of *records*. The base implementation just returns the empty string. You will need to override this method if you want specific behaviour, e.g. to show the count of records, a title or a separator line.

formatFooter(*records*)

Return a footer for a list of *records*. The base implementation just returns the empty string. You will need to override this method if you want specific behaviour, e.g. to show the count of records or a separator line.

format(*records*)

Return formatted text for a list of *records*. The base implementation just returns the empty string if there are no records; otherwise, it returns the concatenation of the header, each record formatted with the line formatter, and the footer.

16.5.5 Filter Objects

`Filters` can be used by `Handlers` and `Loggers` for more sophisticated filtering than is provided by levels. The base filter class only allows events which are below a certain point in the logger hierarchy. For example, a filter initialized with 'A.B' will allow events logged by loggers 'A.B', 'A.B.C', 'A.B.C.D', 'A.B.D' etc. but not 'A.BB', 'B.A.B' etc. If initialized with the empty string, all events are passed.

class logging.**Filter**(*name=''*)

Returns an instance of the *Filter* class. If *name* is specified, it names a logger which, together with its children, will have its events allowed through the filter. If *name* is the empty string, allows every event.

filter(*record*)

Is the specified record to be logged? Returns false for no, true for yes. Filters can either modify log records in-place or return a completely different record instance which will replace the original log record in any future processing of the event.

Note that filters attached to handlers are consulted before an event is emitted by the handler, whereas filters attached to loggers are consulted whenever an event is logged (using *debug()*, *info()*, etc.), before sending an event to handlers. This means that events which have been generated by descendant loggers will not be filtered by a logger's filter setting, unless the filter has also been applied to those descendant loggers.

You don't actually need to subclass `Filter`: you can pass any instance which has a `filter` method with the same semantics.

Changed in version 3.2: You don't need to create specialized `Filter` classes, or use other classes with a `filter` method: you can use a function (or other callable) as a filter. The filtering logic will check to see if the filter object has a `filter` attribute: if it does, it's assumed to be a `Filter` and its *filter()* method is called. Otherwise, it's assumed to be a callable and called with the record as the single parameter. The returned value should conform to that returned by *filter()*.

Changed in version 3.12: You can now return a *LogRecord* instance from filters to replace the log record rather than modifying it in place. This allows filters attached to a *Handler* to modify the log record before it is emitted, without having side effects on other handlers.

Although filters are used primarily to filter records based on more sophisticated criteria than levels, they get to see every record which is processed by the handler or logger they're attached to: this can be useful if you want to do things like counting how many records were processed by a particular logger or handler, or adding, changing or removing attributes in the *LogRecord* being processed. Obviously changing the LogRecord needs to be done with some care, but it does allow the injection of contextual information into logs (see filters-contextual).

16.5.6 LogRecord Objects

LogRecord instances are created automatically by the *Logger* every time something is logged, and can be created manually via *makeLogRecord()* (for example, from a pickled event received over the wire).

class logging.**LogRecord**(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*, *sinfo=None*)

Contains all the information pertinent to the event being logged.

The primary information is passed in *msg* and *args*, which are combined using `msg % args` to create the `message` attribute of the record.

Parameters

- **name** (`str`) – The name of the logger used to log the event represented by this `LogRecord`. Note that the logger name in the `LogRecord` will always have this value, even though it may be emitted by a handler attached to a different (ancestor) logger.

- **level** (int) – The *numeric level* of the logging event (such as 10 for DEBUG, 20 for INFO, etc). Note that this is converted to *two* attributes of the LogRecord: levelno for the numeric value and levelname for the corresponding level name.
- **pathname** (str) – The full string path of the source file where the logging call was made.
- **lineno** (int) – The line number in the source file where the logging call was made.
- **msg** (Any) – The event description message, which can be a %-format string with placeholders for variable data, or an arbitrary object (see arbitrary-object-messages).
- **args** (tuple | dict[str, Any]) – Variable data to merge into the *msg* argument to obtain the event description.
- **exc_info** (tuple[type[BaseException], BaseException, types.TracebackType] | *None*) – An exception tuple with the current exception information, as returned by *sys.exc_info()*, or None if no exception information is available.
- **func** (str | *None*) – The name of the function or method from which the logging call was invoked.
- **sinfo** (str | *None*) – A text string representing stack information from the base of the stack in the current thread, up to the logging call.

getMessage()

Returns the message for this *LogRecord* instance after merging any user-supplied arguments with the message. If the user-supplied message argument to the logging call is not a string, *str()* is called on it to convert it to a string. This allows use of user-defined classes as messages, whose __str__ method can return the actual format string to be used.

Changed in version 3.2: The creation of a *LogRecord* has been made more configurable by providing a factory which is used to create the record. The factory can be set using *getLogRecordFactory()* and *setLogRecordFactory()* (see this for the factory's signature).

This functionality can be used to inject your own values into a *LogRecord* at creation time. You can use the following pattern:

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

logging.setLogRecordFactory(record_factory)
```

With this pattern, multiple factories could be chained, and as long as they don't overwrite each other's attributes or unintentionally overwrite the standard attributes listed above, there should be no surprises.

16.5.7 LogRecord attributes

The LogRecord has a number of attributes, most of which are derived from the parameters to the constructor. (Note that the names do not always correspond exactly between the LogRecord constructor parameters and the LogRecord attributes.) These attributes can be used to merge data from the record into the format string. The following table lists (in alphabetical order) the attribute names, their meanings and the corresponding placeholder in a %-style format string.

If you are using {}-formatting (*str.format()*), you can use `{attrname}` as the placeholder in the format string. If you are using $-formatting (*string.Template*), use the form `${attrname}`. In both cases, of course, replace `attrname` with the actual attribute name you want to use.

In the case of {}-formatting, you can specify formatting flags by placing them after the attribute name, separated from it with a colon. For example: a placeholder of `{msecs:03.0f}` would format a millisecond value of `4` as `004`. Refer to the *str.format()* documentation for full details on the options available to you.

Attribute name	Format	Description
args	You shouldn't need to format this yourself.	The tuple of arguments merged into `msg` to produce `message`, or a dict whose values are used for the merge (when there is only one argument, and it is a dictionary).
asctime	`%(asctime)s`	Human-readable time when the *LogRecord* was created. By default this is of the form '2003-07-08 16:49:45,896' (the numbers after the comma are millisecond portion of the time).
created	`%(created)f`	Time when the *LogRecord* was created (as returned by *time.time_ns()* / 1e9).
exc_info	You shouldn't need to format this yourself.	Exception tuple (à la `sys.exc_info`) or, if no exception has occurred, `None`.
filename	`%(filename)s`	Filename portion of `pathname`.
funcName	`%(funcName)s`	Name of function containing the logging call.
levelname	`%(levelname)s`	Text logging level for the message (`'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, `'CRITICAL'`).
levelno	`%(levelno)s`	Numeric logging level for the message (*DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*).
lineno	`%(lineno)d`	Source line number where the logging call was issued (if available).
message	`%(message)s`	The logged message, computed as `msg % args`. This is set when *Formatter.format()* is invoked.
module	`%(module)s`	Module (name portion of `filename`).
msecs	`%(msecs)d`	Millisecond portion of the time when the *LogRecord* was created.
msg	You shouldn't need to format this yourself.	The format string passed in the original logging call. Merged with `args` to produce `message`, or an arbitrary object (see arbitrary-object-messages).
name	`%(name)s`	Name of the logger used to log the call.
pathname	`%(pathname)s`	Full pathname of the source file where the logging call was issued (if available).
process	`%(process)d`	Process ID (if available).
processName	`%(processName)s`	Process name (if available).
relativeCreated	`%(relativeCreated)d`	Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded.
stack_info	You shouldn't need to format this yourself.	Stack frame information (where available) from the bottom of the stack in the current thread, up to and including the stack frame of the logging call which resulted in the creation of this record.
thread	`%(thread)d`	Thread ID (if available).
threadName	`%(threadName)s`	Thread name (if available).
taskName	`%(taskName)s`	*asyncio.Task* name (if available).

Changed in version 3.1: *processName* was added.

Changed in version 3.12: *taskName* was added.

16.5.8 LoggerAdapter Objects

LoggerAdapter instances are used to conveniently pass contextual information into logging calls. For a usage example, see the section on adding contextual information to your logging output.

class logging.**LoggerAdapter**(*logger*, *extra*, *merge_extra=False*)

Returns an instance of *LoggerAdapter* initialized with an underlying *Logger* instance, a dict-like object (*extra*), and a boolean (*merge_extra*) indicating whether or not the *extra* argument of individual log calls should be merged with the *LoggerAdapter* extra. The default behavior is to ignore the *extra* argument of individual log calls and only use the one of the *LoggerAdapter* instance

process(*msg*, *kwargs*)

Modifies the message and/or keyword arguments passed to a logging call in order to insert contextual information. This implementation takes the object passed as *extra* to the constructor and adds it to *kwargs* using key 'extra'. The return value is a (*msg*, *kwargs*) tuple which has the (possibly modified) versions of the arguments passed in.

manager

Delegates to the underlying `manager` on *logger*.

_log

Delegates to the underlying `_log`() method on *logger*.

In addition to the above, *LoggerAdapter* supports the following methods of *Logger*: *debug()*, *info()*, *warning()*, *error()*, *exception()*, *critical()*, *log()*, *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()* and *hasHandlers()*. These methods have the same signatures as their counterparts in *Logger*, so you can use the two types of instances interchangeably.

Changed in version 3.2: The *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()* and *hasHandlers()* methods were added to *LoggerAdapter*. These methods delegate to the underlying logger.

Changed in version 3.6: Attribute `manager` and method `_log()` were added, which delegate to the underlying logger and allow adapters to be nested.

Changed in version 3.13: Remove the undocumented `warn`() method which was an alias to the `warning()` method.

Changed in version 3.13: The *merge_extra* argument was added.

16.5.9 Thread Safety

The logging module is intended to be thread-safe without any special work needing to be done by its clients. It achieves this though using threading locks; there is one lock to serialize access to the module's shared data, and each handler also creates a lock to serialize access to its underlying I/O.

If you are implementing asynchronous signal handlers using the *signal* module, you may not be able to use logging from within such handlers. This is because lock implementations in the *threading* module are not always re-entrant, and so cannot be invoked from such signal handlers.

16.5.10 Module-Level Functions

In addition to the classes described above, there are a number of module-level functions.

`logging.`**`getLogger`**(*name=None*)

Return a logger with the specified name or, if name is `None`, return the root logger of the hierarchy. If specified, the name is typically a dot-separated hierarchical name like *'a'*, *'a.b'* or *'a.b.c.d'*. Choice of these names is entirely up to the developer who is using logging, though it is recommended that `__name__` be used unless you have a specific reason for not doing that, as mentioned in *Logger Objects*.

All calls to this function with a given name return the same logger instance. This means that logger instances never need to be passed between different parts of an application.

`logging.`**`getLoggerClass`**()

Return either the standard *Logger* class, or the last class passed to *setLoggerClass()*. This function may be called from within a new class definition, to ensure that installing a customized *Logger* class will not undo customizations already applied by other code. For example:

```
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

`logging.`**`getLogRecordFactory`**()

Return a callable which is used to create a *LogRecord*.

Added in version 3.2: This function has been provided, along with *setLogRecordFactory()*, to allow developers more control over how the *LogRecord* representing a logging event is constructed.

See *setLogRecordFactory()* for more information about the how the factory is called.

`logging.`**`debug`**(*msg*, **args*, ***kwargs*)

This is a convenience function that calls *Logger.debug()*, on the root logger. The handling of the arguments is in every way identical to what is described in that method.

The only difference is that if the root logger has no handlers, then *basicConfig()* is called, prior to calling `debug` on the root logger.

For very short scripts or quick demonstrations of `logging` facilities, `debug` and the other module-level functions may be convenient. However, most programs will want to carefully and explicitly control the logging configuration, and should therefore prefer creating a module-level logger and calling *Logger.debug()* (or other level-specific methods) on it, as described at the beginnning of this documentation.

`logging.`**`info`**(*msg*, **args*, ***kwargs*)

Logs a message with level *INFO* on the root logger. The arguments and behavior are otherwise the same as for *debug()*.

`logging.`**`warning`**(*msg*, **args*, ***kwargs*)

Logs a message with level *WARNING* on the root logger. The arguments and behavior are otherwise the same as for *debug()*.

> **Note**
>
> There is an obsolete function `warn` which is functionally identical to `warning`. As `warn` is deprecated, please do not use it - use `warning` instead.

Changed in version 3.13: Remove the undocumented `warn()` function which was an alias to the *warning()* function.

logging.**error**(*msg*, **args*, ***kwargs*)

Logs a message with level *ERROR* on the root logger. The arguments and behavior are otherwise the same as for *debug()*.

logging.**critical**(*msg*, **args*, ***kwargs*)

Logs a message with level *CRITICAL* on the root logger. The arguments and behavior are otherwise the same as for *debug()*.

logging.**exception**(*msg*, **args*, ***kwargs*)

Logs a message with level *ERROR* on the root logger. The arguments and behavior are otherwise the same as for *debug()*. Exception info is added to the logging message. This function should only be called from an exception handler.

logging.**log**(*level*, *msg*, **args*, ***kwargs*)

Logs a message with level *level* on the root logger. The arguments and behavior are otherwise the same as for *debug()*.

logging.**disable**(*level=CRITICAL*)

Provides an overriding level *level* for all loggers which takes precedence over the logger's own level. When the need arises to temporarily throttle logging output down across the whole application, this function can be useful. Its effect is to disable all logging calls of severity *level* and below, so that if you call it with a value of INFO, then all INFO and DEBUG events would be discarded, whereas those of severity WARNING and above would be processed according to the logger's effective level. If `logging.disable(logging.NOTSET)` is called, it effectively removes this overriding level, so that logging output again depends on the effective levels of individual loggers.

Note that if you have defined any custom logging level higher than `CRITICAL` (this is not recommended), you won't be able to rely on the default value for the *level* parameter, but will have to explicitly supply a suitable value.

Changed in version 3.7: The *level* parameter was defaulted to level `CRITICAL`. See bpo-28524 for more information about this change.

logging.**addLevelName**(*level*, *levelName*)

Associates level *level* with text *levelName* in an internal dictionary, which is used to map numeric levels to a textual representation, for example when a *Formatter* formats a message. This function can also be used to define your own levels. The only constraints are that all levels used must be registered using this function, levels should be positive integers and they should increase in increasing order of severity.

> **Note**
>
> If you are thinking of defining your own levels, please see the section on custom-levels.

logging.**getLevelNamesMapping**()

Returns a mapping from level names to their corresponding logging levels. For example, the string "CRITICAL" maps to *CRITICAL*. The returned mapping is copied from an internal mapping on each call to this function.

Added in version 3.11.

logging.**getLevelName**(*level*)

Returns the textual or numeric representation of logging level *level*.

If *level* is one of the predefined levels *CRITICAL*, *ERROR*, *WARNING*, *INFO* or *DEBUG* then you get the corresponding string. If you have associated levels with names using *addLevelName()* then the name you have associated with *level* is returned. If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned.

The *level* parameter also accepts a string representation of the level such as 'INFO'. In such cases, this functions returns the corresponding numeric value of the level.

If no matching numeric or string value is passed in, the string 'Level %s' % level is returned.

> **Note**
>
> Levels are internally integers (as they need to be compared in the logging logic). This function is used to convert between an integer level and the level name displayed in the formatted log output by means of the `%(levelname)s` format specifier (see *LogRecord attributes*), and vice versa.

Changed in version 3.4: In Python versions earlier than 3.4, this function could also be passed a text level, and would return the corresponding numeric value of the level. This undocumented behaviour was considered a mistake, and was removed in Python 3.4, but reinstated in 3.4.2 due to retain backward compatibility.

`logging.`**`getHandlerByName`**`(`*name*`)`

Returns a handler with the specified *name*, or `None` if there is no handler with that name.

Added in version 3.12.

`logging.`**`getHandlerNames`**`()`

Returns an immutable set of all known handler names.

Added in version 3.12.

`logging.`**`makeLogRecord`**`(`*attrdict*`)`

Creates and returns a new *`LogRecord`* instance whose attributes are defined by *attrdict*. This function is useful for taking a pickled *`LogRecord`* attribute dictionary, sent over a socket, and reconstituting it as a *`LogRecord`* instance at the receiving end.

`logging.`**`basicConfig`**`(`***kwargs*`)`

Does basic configuration for the logging system by creating a *`StreamHandler`* with a default *`Formatter`* and adding it to the root logger. The functions *`debug()`*, *`info()`*, *`warning()`*, *`error()`* and *`critical()`* will call *`basicConfig()`* automatically if no handlers are defined for the root logger.

This function does nothing if the root logger already has handlers configured, unless the keyword argument *force* is set to `True`.

> **Note**
>
> This function should be called from the main thread before other threads are started. In versions of Python prior to 2.7.1 and 3.2, if this function is called from multiple threads, it is possible (in rare circumstances) that a handler will be added to the root logger more than once, leading to unexpected results such as messages being duplicated in the log.

The following keyword arguments are supported.

Format	Description
filename	Specifies that a `FileHandler` be created, using the specified filename, rather than a `StreamHandler`.
filemode	If *filename* is specified, open the file in this *mode*. Defaults to `'a'`.
format	Use the specified format string for the handler. Defaults to attributes `levelname`, `name` and `message` separated by colons.
datefmt	Use the specified date/time format, as accepted by `time.strftime()`.
style	If *format* is specified, use this style for the format string. One of `'%'`, `'{'` or `'$'` for *printf-style*, `str.format()` or `string.Template` respectively. Defaults to `'%'`.
level	Set the root logger level to the specified *level*.
stream	Use the specified stream to initialize the `StreamHandler`. Note that this argument is incompatible with *filename* - if both are present, a `ValueError` is raised.
handlers	If specified, this should be an iterable of already created handlers to add to the root logger. Any handlers which don't already have a formatter set will be assigned the default formatter created in this function. Note that this argument is incompatible with *filename* or *stream* - if both are present, a `ValueError` is raised.
force	If this keyword argument is specified as true, any existing handlers attached to the root logger are removed and closed, before carrying out the configuration as specified by the other arguments.
encoding	If this keyword argument is specified along with *filename*, its value is used when the `FileHandler` is created, and thus used when opening the output file.
errors	If this keyword argument is specified along with *filename*, its value is used when the `FileHandler` is created, and thus used when opening the output file. If not specified, the value 'backslashreplace' is used. Note that if `None` is specified, it will be passed as such to `open()`, which means that it will be treated the same as passing 'errors'.

Changed in version 3.2: The *style* argument was added.

Changed in version 3.3: The *handlers* argument was added. Additional checks were added to catch situations where incompatible arguments are specified (e.g. *handlers* together with *stream* or *filename*, or *stream* together with *filename*).

Changed in version 3.8: The *force* argument was added.

Changed in version 3.9: The *encoding* and *errors* arguments were added.

`logging.`**`shutdown`**`()`

Informs the logging system to perform an orderly shutdown by flushing and closing all handlers. This should be called at application exit and no further use of the logging system should be made after this call.

When the logging module is imported, it registers this function as an exit handler (see `atexit`), so normally there's no need to do that manually.

`logging.`**`setLoggerClass`**`(`*klass*`)`

Tells the logging system to use the class *klass* when instantiating a logger. The class should define `__init__()` such that only a name argument is required, and the `__init__()` should call `Logger.__init__()`. This function is typically called before any loggers are instantiated by applications which need to use custom logger behavior. After this call, as at any other time, do not instantiate loggers directly using the subclass: continue to use the `logging.getLogger()` API to get your loggers.

`logging.`**`setLogRecordFactory`**`(`*factory*`)`

Set a callable which is used to create a `LogRecord`.

Parameters

`factory` – The factory callable to be used to instantiate a log record.

Added in version 3.2: This function has been provided, along with *getLogRecordFactory()*, to allow developers more control over how the *LogRecord* representing a logging event is constructed.

The factory has the following signature:

`factory(name, level, fn, lno, msg, args, exc_info, func=None, sinfo=None, **kwargs)`

name
The logger name.

level
The logging level (numeric).

fn
The full pathname of the file where the logging call was made.

lno
The line number in the file where the logging call was made.

msg
The logging message.

args
The arguments for the logging message.

exc_info
An exception tuple, or `None`.

func
The name of the function or method which invoked the logging call.

sinfo
A stack traceback such as is provided by *traceback.print_stack()*, showing the call hierarchy.

kwargs
Additional keyword arguments.

16.5.11 Module-Level Attributes

`logging.`**`lastResort`**

A "handler of last resort" is available through this attribute. This is a *StreamHandler* writing to `sys.stderr` with a level of `WARNING`, and is used to handle logging events in the absence of any logging configuration. The end result is to just print the message to `sys.stderr`. This replaces the earlier error message saying that "no handlers could be found for logger XYZ". If you need the earlier behaviour for some reason, `lastResort` can be set to `None`.

Added in version 3.2.

`logging.`**`raiseExceptions`**

Used to see if exceptions during handling should be propagated.

Default: `True`.

If *raiseExceptions* is `False`, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors.

16.5.12 Integration with the warnings module

The *captureWarnings()* function can be used to integrate *logging* with the *warnings* module.

logging.**captureWarnings**(*capture*)

This function is used to turn the capture of warnings by logging on and off.

If *capture* is `True`, warnings issued by the *warnings* module will be redirected to the logging system. Specifically, a warning will be formatted using *warnings.formatwarning()* and the resulting string logged to a logger named `'py.warnings'` with a severity of *WARNING*.

If *capture* is `False`, the redirection of warnings to the logging system will stop, and warnings will be redirected to their original destinations (i.e. those in effect before `captureWarnings(True)` was called).

See also

Module *logging.config*
Configuration API for the logging module.

Module *logging.handlers*
Useful handlers included with the logging module.

PEP 282 - A Logging System
The proposal which described this feature for inclusion in the Python standard library.

Original Python logging package
This is the original source for the *logging* package. The version of the package available from this site is suitable for use with Python 1.5.2, 2.1.x and 2.2.x, which do not include the *logging* package in the standard library.

16.6 logging.config — Logging configuration

Source code: Lib/logging/config.py

Important

This page contains only reference information. For tutorials, please see

- Basic Tutorial
- Advanced Tutorial
- Logging Cookbook

This section describes the API for configuring the logging module.

16.6.1 Configuration functions

The following functions configure the logging module. They are located in the *logging.config* module. Their use is optional — you can configure the logging module using these functions or by making calls to the main API (defined in *logging* itself) and defining handlers which are declared either in *logging* or *logging.handlers*.

logging.config.**dictConfig**(*config*)

Takes the logging configuration from a dictionary. The contents of this dictionary are described in *Configuration dictionary schema* below.

If an error is encountered during configuration, this function will raise a *ValueError*, *TypeError*, *AttributeError* or *ImportError* with a suitably descriptive message. The following is a (possibly incomplete) list of conditions which will raise an error:

- A `level` which is not a string or which is a string not corresponding to an actual logging level.
- A `propagate` value which is not a boolean.
- An id which does not have a corresponding destination.
- A non-existent handler id found during an incremental call.
- An invalid logger name.
- Inability to resolve to an internal or external object.

Parsing is performed by the `DictConfigurator` class, whose constructor is passed the dictionary used for configuration, and has a `configure()` method. The *logging.config* module has a callable attribute `dictConfigClass` which is initially set to `DictConfigurator`. You can replace the value of `dictConfigClass` with a suitable implementation of your own.

dictConfig() calls `dictConfigClass` passing the specified dictionary, and then calls the `configure()` method on the returned object to put the configuration into effect:

```
def dictConfig(config):
    dictConfigClass(config).configure()
```

For example, a subclass of `DictConfigurator` could call `DictConfigurator.__init__()` in its own `__init__()`, then set up custom prefixes which would be usable in the subsequent `configure()` call. `dictConfigClass` would be bound to this new subclass, and then *dictConfig()* could be called exactly as in the default, uncustomized state.

Added in version 3.2.

logging.config.**fileConfig**(*fname*, *defaults=None*, *disable_existing_loggers=True*, *encoding=None*)

Reads the logging configuration from a *configparser*-format file. The format of the file should be as described in *Configuration file format*. This function can be called several times from an application, allowing an end user to select from various pre-canned configurations (if the developer provides a mechanism to present the choices and load the chosen configuration).

It will raise *FileNotFoundError* if the file doesn't exist and *RuntimeError* if the file is invalid or empty.

Parameters

- **fname** – A filename, or a file-like object, or an instance derived from *RawConfigParser*. If a `RawConfigParser`-derived instance is passed, it is used as is. Otherwise, a *ConfigParser* is instantiated, and the configuration read by it from the object passed in `fname`. If that has a *readline()* method, it is assumed to be a file-like object and read using *read_file()*; otherwise, it is assumed to be a filename and passed to *read()*.
- **defaults** – Defaults to be passed to the `ConfigParser` can be specified in this argument.

- **disable_existing_loggers** – If specified as `False`, loggers which exist when this call is made are left enabled. The default is `True` because this enables old behaviour in a backward-compatible way. This behaviour is to disable any existing non-root loggers unless they or their ancestors are explicitly named in the logging configuration.
- **encoding** – The encoding used to open file when *fname* is filename.

Changed in version 3.4: An instance of a subclass of *RawConfigParser* is now accepted as a value for `fname`. This facilitates:

- Use of a configuration file where logging configuration is just part of the overall application configuration.
- Use of a configuration read from a file, and then modified by the using application (e.g. based on command-line parameters or other aspects of the runtime environment) before being passed to `fileConfig`.

Changed in version 3.10: Added the *encoding* parameter.

Changed in version 3.12: An exception will be thrown if the provided file doesn't exist or is invalid or empty.

`logging.config.`**listen**(*port=DEFAULT_LOGGING_CONFIG_PORT*, *verify=None*)

Starts up a socket server on the specified port, and listens for new configurations. If no port is specified, the module's default `DEFAULT_LOGGING_CONFIG_PORT` is used. Logging configurations will be sent as a file suitable for processing by *dictConfig()* or *fileConfig()*. Returns a *Thread* instance on which you can call *start()* to start the server, and which you can *join()* when appropriate. To stop the server, call *stopListening()*.

The `verify` argument, if specified, should be a callable which should verify whether bytes received across the socket are valid and should be processed. This could be done by encrypting and/or signing what is sent across the socket, such that the `verify` callable can perform signature verification and/or decryption. The `verify` callable is called with a single argument - the bytes received across the socket - and should return the bytes to be processed, or `None` to indicate that the bytes should be discarded. The returned bytes could be the same as the passed in bytes (e.g. when only verification is done), or they could be completely different (perhaps if decryption were performed).

To send a configuration to the socket, read in the configuration file and send it to the socket as a sequence of bytes preceded by a four-byte length string packed in binary using `struct.pack('>L', n)`.

Note

Because portions of the configuration are passed through *eval()*, use of this function may open its users to a security risk. While the function only binds to a socket on `localhost`, and so does not accept connections from remote machines, there are scenarios where untrusted code could be run under the account of the process which calls *listen()*. Specifically, if the process calling *listen()* runs on a multi-user machine where users cannot trust each other, then a malicious user could arrange to run essentially arbitrary code in a victim user's process, simply by connecting to the victim's *listen()* socket and sending a configuration which runs whatever code the attacker wants to have executed in the victim's process. This is especially easy to do if the default port is used, but not hard even if a different port is used. To avoid the risk of this happening, use the `verify` argument to *listen()* to prevent unrecognised configurations from being applied.

Changed in version 3.4: The `verify` argument was added.

Note

If you want to send configurations to the listener which don't disable existing loggers, you will need to use a JSON format for the configuration, which will use *dictConfig()* for configuration. This method allows you to specify `disable_existing_loggers` as `False` in the configuration you send.

logging.config.**stopListening**()

Stops the listening server which was created with a call to *listen()*. This is typically called before calling join() on the return value from *listen()*.

16.6.2 Security considerations

The logging configuration functionality tries to offer convenience, and in part this is done by offering the ability to convert text in configuration files into Python objects used in logging configuration - for example, as described in *User-defined objects*. However, these same mechanisms (importing callables from user-defined modules and calling them with parameters from the configuration) could be used to invoke any code you like, and for this reason you should treat configuration files from untrusted sources with *extreme caution* and satisfy yourself that nothing bad can happen if you load them, before actually loading them.

16.6.3 Configuration dictionary schema

Describing a logging configuration requires listing the various objects to create and the connections between them; for example, you may create a handler named 'console' and then say that the logger named 'startup' will send its messages to the 'console' handler. These objects aren't limited to those provided by the *logging* module because you might write your own formatter or handler class. The parameters to these classes may also need to include external objects such as sys.stderr. The syntax for describing these objects and connections is defined in *Object connections* below.

Dictionary Schema Details

The dictionary passed to *dictConfig()* must contain the following keys:

- *version* - to be set to an integer value representing the schema version. The only valid value at present is 1, but having this key allows the schema to evolve while still preserving backwards compatibility.

All other keys are optional, but if present they will be interpreted as described below. In all cases below where a 'configuring dict' is mentioned, it will be checked for the special '()' key to see if a custom instantiation is required. If so, the mechanism described in *User-defined objects* below is used to create an instance; otherwise, the context is used to determine what to instantiate.

- *formatters* - the corresponding value will be a dict in which each key is a formatter id and each value is a dict describing how to configure the corresponding *Formatter* instance.

  The configuring dict is searched for the following optional keys which correspond to the arguments passed to create a *Formatter* object:

  - format
  - datefmt
  - style
  - validate (since version >=3.8)
  - defaults (since version >=3.12)

  An optional class key indicates the name of the formatter's class (as a dotted module and class name). The instantiation arguments are as for *Formatter*, thus this key is most useful for instantiating a customised subclass of *Formatter*. For example, the alternative class might present exception tracebacks in an expanded or condensed format. If your formatter requires different or extra configuration keys, you should use *User-defined objects*.

- *filters* - the corresponding value will be a dict in which each key is a filter id and each value is a dict describing how to configure the corresponding Filter instance.

The configuring dict is searched for the key `name` (defaulting to the empty string) and this is used to construct a *logging.Filter* instance.

- *handlers* - the corresponding value will be a dict in which each key is a handler id and each value is a dict describing how to configure the corresponding Handler instance.

  The configuring dict is searched for the following keys:

  - `class` (mandatory). This is the fully qualified name of the handler class.
  - `level` (optional). The level of the handler.
  - `formatter` (optional). The id of the formatter for this handler.
  - `filters` (optional). A list of ids of the filters for this handler.

    Changed in version 3.11: `filters` can take filter instances in addition to ids.

  All *other* keys are passed through as keyword arguments to the handler's constructor. For example, given the snippet:

```
handlers:
  console:
    class : logging.StreamHandler
    formatter: brief
    level   : INFO
    filters: [allow_foo]
    stream  : ext://sys.stdout
  file:
    class : logging.handlers.RotatingFileHandler
    formatter: precise
    filename: logconfig.log
    maxBytes: 1024
    backupCount: 3
```

  the handler with id `console` is instantiated as a *logging.StreamHandler*, using `sys.stdout` as the underlying stream. The handler with id `file` is instantiated as a *logging.handlers.RotatingFileHandler* with the keyword arguments `filename='logconfig.log', maxBytes=1024, backupCount=3`.

- *loggers* - the corresponding value will be a dict in which each key is a logger name and each value is a dict describing how to configure the corresponding Logger instance.

  The configuring dict is searched for the following keys:

  - `level` (optional). The level of the logger.
  - `propagate` (optional). The propagation setting of the logger.
  - `filters` (optional). A list of ids of the filters for this logger.

    Changed in version 3.11: `filters` can take filter instances in addition to ids.

  - `handlers` (optional). A list of ids of the handlers for this logger.

  The specified loggers will be configured according to the level, propagation, filters and handlers specified.

- *root* - this will be the configuration for the root logger. Processing of the configuration will be as for any logger, except that the `propagate` setting will not be applicable.

- *incremental* - whether the configuration is to be interpreted as incremental to the existing configuration. This value defaults to `False`, which means that the specified configuration replaces the existing configuration with the same semantics as used by the existing *fileConfig()* API.

If the specified value is `True`, the configuration is processed as described in the section on *Incremental Configuration*.

- *disable_existing_loggers* - whether any existing non-root loggers are to be disabled. This setting mirrors the parameter of the same name in `fileConfig()`. If absent, this parameter defaults to `True`. This value is ignored if *incremental* is `True`.

Incremental Configuration

It is difficult to provide complete flexibility for incremental configuration. For example, because objects such as filters and formatters are anonymous, once a configuration is set up, it is not possible to refer to such anonymous objects when augmenting a configuration.

Furthermore, there is not a compelling case for arbitrarily altering the object graph of loggers, handlers, filters, formatters at run-time, once a configuration is set up; the verbosity of loggers and handlers can be controlled just by setting levels (and, in the case of loggers, propagation flags). Changing the object graph arbitrarily in a safe way is problematic in a multi-threaded environment; while not impossible, the benefits are not worth the complexity it adds to the implementation.

Thus, when the `incremental` key of a configuration dict is present and is `True`, the system will completely ignore any `formatters` and `filters` entries, and process only the `level` settings in the `handlers` entries, and the `level` and `propagate` settings in the `loggers` and `root` entries.

Using a value in the configuration dict lets configurations to be sent over the wire as pickled dicts to a socket listener. Thus, the logging verbosity of a long-running application can be altered over time with no need to stop and restart the application.

Object connections

The schema describes a set of logging objects - loggers, handlers, formatters, filters - which are connected to each other in an object graph. Thus, the schema needs to represent connections between the objects. For example, say that, once configured, a particular logger has attached to it a particular handler. For the purposes of this discussion, we can say that the logger represents the source, and the handler the destination, of a connection between the two. Of course in the configured objects this is represented by the logger holding a reference to the handler. In the configuration dict, this is done by giving each destination object an id which identifies it unambiguously, and then using the id in the source object's configuration to indicate that a connection exists between the source and the destination object with that id.

So, for example, consider the following YAML snippet:

```
formatters:
  brief:
    # configuration for formatter with id 'brief' goes here
  precise:
    # configuration for formatter with id 'precise' goes here
handlers:
  h1: #This is an id
   # configuration of handler with id 'h1' goes here
   formatter: brief
  h2: #This is another id
   # configuration of handler with id 'h2' goes here
   formatter: precise
loggers:
  foo.bar.baz:
    # other configuration for logger 'foo.bar.baz'
    handlers: [h1, h2]
```

(Note: YAML used here because it's a little more readable than the equivalent Python source form for the dictionary.)

The ids for loggers are the logger names which would be used programmatically to obtain a reference to those loggers, e.g. `foo.bar.baz`. The ids for Formatters and Filters can be any string value (such as `brief`, `precise` above) and they are transient, in that they are only meaningful for processing the configuration dictionary and used to determine connections between objects, and are not persisted anywhere when the configuration call is complete.

The above snippet indicates that logger named `foo.bar.baz` should have two handlers attached to it, which are described by the handler ids `h1` and `h2`. The formatter for `h1` is that described by id `brief`, and the formatter for `h2` is that described by id `precise`.

User-defined objects

The schema supports user-defined objects for handlers, filters and formatters. (Loggers do not need to have different types for different instances, so there is no support in this configuration schema for user-defined logger classes.)

Objects to be configured are described by dictionaries which detail their configuration. In some places, the logging system will be able to infer from the context how an object is to be instantiated, but when a user-defined object is to be instantiated, the system will not know how to do this. In order to provide complete flexibility for user-defined object instantiation, the user needs to provide a 'factory' - a callable which is called with a configuration dictionary and which returns the instantiated object. This is signalled by an absolute import path to the factory being made available under the special key `'()'`. Here's a concrete example:

```
formatters:
  brief:
    format: '%(message)s'
  default:
    format: '%(asctime)s %(levelname)-8s %(name)-15s %(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
  custom:
      (): my.package.customFormatterFactory
      bar: baz
      spam: 99.9
      answer: 42
```

The above YAML snippet defines three formatters. The first, with id `brief`, is a standard *`logging.Formatter`* instance with the specified format string. The second, with id `default`, has a longer format and also defines the time format explicitly, and will result in a *`logging.Formatter`* initialized with those two format strings. Shown in Python source form, the `brief` and `default` formatters have configuration sub-dictionaries:

```
{
  'format' : '%(message)s'
}
```

and:

```
{
  'format' : '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
  'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

respectively, and as these dictionaries do not contain the special key `'()'`, the instantiation is inferred from the context: as a result, standard *`logging.Formatter`* instances are created. The configuration sub-dictionary for the third formatter, with id `custom`, is:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
```

(continues on next page)

(continued from previous page)

```
  'spam' : 99.9,
  'answer' : 42
}
```

and this contains the special key `'()'`, which means that user-defined instantiation is wanted. In this case, the specified factory callable will be used. If it is an actual callable it will be used directly - otherwise, if you specify a string (as in the example) the actual callable will be located using normal import mechanisms. The callable will be called with the **remaining** items in the configuration sub-dictionary as keyword arguments. In the above example, the formatter with id `custom` will be assumed to be returned by the call:

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

Warning

The values for keys such as `bar`, `spam` and `answer` in the above example should not be configuration dictionaries or references such as `cfg://foo` or `ext://bar`, because they will not be processed by the configuration machinery, but passed to the callable as-is.

The key `'()'` has been used as the special key because it is not a valid keyword parameter name, and so will not clash with the names of the keyword arguments used in the call. The `'()'` also serves as a mnemonic that the corresponding value is a callable.

Changed in version 3.11: The `filters` member of `handlers` and `loggers` can take filter instances in addition to ids.

You can also specify a special key `'.'` whose value is a dictionary is a mapping of attribute names to values. If found, the specified attributes will be set on the user-defined object before it is returned. Thus, with the following configuration:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42,
  '.' {
    'foo': 'bar',
    'baz': 'bozz'
  }
}
```

the returned formatter will have attribute `foo` set to `'bar'` and attribute `baz` set to `'bozz'`.

Warning

The values for attributes such as `foo` and `baz` in the above example should not be configuration dictionaries or references such as `cfg://foo` or `ext://bar`, because they will not be processed by the configuration machinery, but set as attribute values as-is.

Handler configuration order

Handlers are configured in alphabetical order of their keys, and a configured handler replaces the configuration dictionary in (a working copy of) the `handlers` dictionary in the schema. If you use a construct such as `cfg://handlers.foo`, then initially `handlers['foo']` points to the configuration dictionary for the handler named `foo`, and later (once that handler has been configured) it points to the configured handler instance. Thus, `cfg://handlers.foo` could resolve to either a dictionary or a handler instance. In general, it is wise to name handlers in a way such that dependent handlers are configured _after_ any handlers they depend on; that allows something like `cfg://handlers.foo` to be used in configuring a handler that depends on handler `foo`. If that dependent handler were named `bar`, problems would result, because the configuration of `bar` would be attempted before that of `foo`, and `foo` would not yet have been configured. However, if the dependent handler were named `foobar`, it would be configured after `foo`, with the result that `cfg://handlers.foo` would resolve to configured handler `foo`, and not its configuration dictionary.

Access to external objects

There are times where a configuration needs to refer to objects external to the configuration, for example `sys.stderr`. If the configuration dict is constructed using Python code, this is straightforward, but a problem arises when the configuration is provided via a text file (e.g. JSON, YAML). In a text file, there is no standard way to distinguish `sys.stderr` from the literal string `'sys.stderr'`. To facilitate this distinction, the configuration system looks for certain special prefixes in string values and treat them specially. For example, if the literal string `'ext://sys.stderr'` is provided as a value in the configuration, then the `ext://` will be stripped off and the remainder of the value processed using normal import mechanisms.

The handling of such prefixes is done in a way analogous to protocol handling: there is a generic mechanism to look for prefixes which match the regular expression `^(?P<prefix>[a-z]+)://(?P<suffix>.*)$` whereby, if the `prefix` is recognised, the `suffix` is processed in a prefix-dependent manner and the result of the processing replaces the string value. If the prefix is not recognised, then the string value will be left as-is.

Access to internal objects

As well as external objects, there is sometimes also a need to refer to objects in the configuration. This will be done implicitly by the configuration system for things that it knows about. For example, the string value `'DEBUG'` for a `level` in a logger or handler will automatically be converted to the value `logging.DEBUG`, and the `handlers`, `filters` and `formatter` entries will take an object id and resolve to the appropriate destination object.

However, a more generic mechanism is needed for user-defined objects which are not known to the *logging* module. For example, consider *logging.handlers.MemoryHandler*, which takes a `target` argument which is another handler to delegate to. Since the system already knows about this class, then in the configuration, the given `target` just needs to be the object id of the relevant target handler, and the system will resolve to the handler from the id. If, however, a user defines a `my.package.MyHandler` which has an `alternate` handler, the configuration system would not know that the `alternate` referred to a handler. To cater for this, a generic resolution system allows the user to specify:

```
handlers:
  file:
    # configuration of file handler goes here

  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

The literal string `'cfg://handlers.file'` will be resolved in an analogous way to strings with the `ext://` prefix, but looking in the configuration itself rather than the import namespace. The mechanism allows access by dot or by index, in a similar way to that provided by `str.format`. Thus, given the following snippet:

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
    fromaddr: my_app@domain.tld
    toaddrs:
      - support_team@domain.tld
      - dev_team@domain.tld
    subject: Houston, we have a problem.
```

in the configuration, the string `'cfg://handlers'` would resolve to the dict with key `handlers`, the string `'cfg://handlers.email` would resolve to the dict with key `email` in the `handlers` dict, and so on. The string `'cfg://handlers.email.toaddrs[1]` would resolve to `'dev_team@domain.tld'` and the string `'cfg://handlers.email.toaddrs[0]'` would resolve to the value `'support_team@domain.tld'`. The `subject` value could be accessed using either `'cfg://handlers.email.subject'` or, equivalently, `'cfg://handlers.email[subject]'`. The latter form only needs to be used if the key contains spaces or non-alphanumeric characters. Please note that the characters `[` and `]` are not allowed in the keys. If an index value consists only of decimal digits, access will be attempted using the corresponding integer value, falling back to the string value if needed.

Given a string `cfg://handlers.myhandler.mykey.123`, this will resolve to `config_dict['handlers']['myhandler']['mykey']['123']`. If the string is specified as `cfg://handlers.myhandler.mykey[123]`, the system will attempt to retrieve the value from `config_dict['handlers']['myhandler']['mykey'][123]`, and fall back to `config_dict['handlers']['myhandler']['mykey']['123']` if that fails.

Import resolution and custom importers

Import resolution, by default, uses the builtin *`__import__()`* function to do its importing. You may want to replace this with your own importing mechanism: if so, you can replace the `importer` attribute of the `DictConfigurator` or its superclass, the `BaseConfigurator` class. However, you need to be careful because of the way functions are accessed from classes via descriptors. If you are using a Python callable to do your imports, and you want to define it at class level rather than instance level, you need to wrap it with *`staticmethod()`*. For example:

```
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

You don't need to wrap with *`staticmethod()`* if you're setting the import callable on a configurator *instance*.

Configuring QueueHandler and QueueListener

If you want to configure a *`QueueHandler`*, noting that this is normally used in conjunction with a *`QueueListener`*, you can configure both together. After the configuration, the `QueueListener` instance will be available as the *`listener`* attribute of the created handler, and that in turn will be available to you using *`getHandlerByName()`* and passing the name you have used for the `QueueHandler` in your configuration. The dictionary schema for configuring the pair is shown in the example YAML snippet below.

```
handlers:
  qhand:
    class: logging.handlers.QueueHandler
    queue: my.module.queue_factory
```

(continues on next page)

(continued from previous page)

```
    listener: my.package.CustomListener
    handlers:
      - hand_name_1
      - hand_name_2
      ...
```

The `queue` and `listener` keys are optional.

If the `queue` key is present, the corresponding value can be one of the following:

- An actual instance of *`queue.Queue`* or a subclass thereof. This is of course only possible if you are constructing or modifying the configuration dictionary in code.
- A string that resolves to a callable which, when called with no arguments, returns the *`queue.Queue`* instance to use. That callable could be a *`queue.Queue`* subclass or a function which returns a suitable queue instance, such as `my.module.queue_factory()`.
- A dict with a `'()'` key which is constructed in the usual way as discussed in *User-defined objects*. The result of this construction should be a *`queue.Queue`* instance.

If the `queue` key is absent, a standard unbounded *`queue.Queue`* instance is created and used.

If the `listener` key is present, the corresponding value can be one of the following:

- A subclass of *`logging.handlers.QueueListener`*. This is of course only possible if you are constructing or modifying the configuration dictionary in code.
- A string which resolves to a class which is a subclass of `QueueListener`, such as `'my.package.CustomListener'`.
- A dict with a `'()'` key which is constructed in the usual way as discussed in *User-defined objects*. The result of this construction should be a callable with the same signature as the `QueueListener` initializer.

If the `listener` key is absent, *`logging.handlers.QueueListener`* is used.

The values under the `handlers` key are the names of other handlers in the configuration (not shown in the above snippet) which will be passed to the queue listener.

Any custom queue handler and listener classes will need to be defined with the same initialization signatures as *`QueueHandler`* and *`QueueListener`*.

Added in version 3.12.

16.6.4 Configuration file format

The configuration file format understood by *`fileConfig()`* is based on *`configparser`* functionality. The file must contain sections called `[loggers]`, `[handlers]` and `[formatters]` which identify by name the entities of each type which are defined in the file. For each such entity, there is a separate section which identifies how that entity is configured. Thus, for a logger named `log01` in the `[loggers]` section, the relevant configuration details are held in a section `[logger_log01]`. Similarly, a handler called `hand01` in the `[handlers]` section will have its configuration held in a section called `[handler_hand01]`, while a formatter called `form01` in the `[formatters]` section will have its configuration specified in a section called `[formatter_form01]`. The root logger configuration must be specified in a section called `[logger_root]`.

Note

The *`fileConfig()`* API is older than the *`dictConfig()`* API and does not provide functionality to cover certain aspects of logging. For example, you cannot configure *`Filter`* objects, which provide for filtering of messages

beyond simple integer levels, using *fileConfig()*. If you need to have instances of *Filter* in your logging configuration, you will need to use *dictConfig()*. Note that future enhancements to configuration functionality will be added to *dictConfig()*, so it's worth considering transitioning to this newer API when it's convenient to do so.

Examples of these sections in the file are given below.

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

The root logger must specify a level and a list of handlers. An example of a root logger section is given below.

```
[logger_root]
level=NOTSET
handlers=hand01
```

The `level` entry can be one of `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL` or `NOTSET`. For the root logger only, `NOTSET` means that all messages will be logged. Level values are *evaluated* in the context of the `logging` package's namespace.

The `handlers` entry is a comma-separated list of handler names, which must appear in the `[handlers]` section. These names must appear in the `[handlers]` section and have corresponding sections in the configuration file.

For loggers other than the root logger, some additional information is required. This is illustrated by the following example.

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

The `level` and `handlers` entries are interpreted as for the root logger, except that if a non-root logger's level is specified as `NOTSET`, the system consults loggers higher up the hierarchy to determine the effective level of the logger. The `propagate` entry is set to 1 to indicate that messages must propagate to handlers higher up the logger hierarchy from this logger, or 0 to indicate that messages are **not** propagated to handlers up the hierarchy. The `qualname` entry is the hierarchical channel name of the logger, that is to say the name used by the application to get the logger.

Sections which specify handler configuration are exemplified by the following.

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

The `class` entry indicates the handler's class (as determined by *eval()* in the `logging` package's namespace). The `level` is interpreted as for loggers, and `NOTSET` is taken to mean 'log everything'.

The `formatter` entry indicates the key name of the formatter for this handler. If blank, a default formatter (`logging._defaultFormatter`) is used. If a name is specified, it must appear in the `[formatters]` section and have a corresponding section in the configuration file.

The `args` entry, when *evaluated* in the context of the `logging` package's namespace, is the list of arguments to the constructor for the handler class. Refer to the constructors for the relevant handlers, or to the examples below, to see how typical entries are constructed. If not provided, it defaults to `()`.

The optional `kwargs` entry, when *evaluated* in the context of the `logging` package's namespace, is the keyword argument dict to the constructor for the handler class. If not provided, it defaults to `{}`.

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=(('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')
kwargs={'timeout': 10.0}

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
kwargs={'secure': True}
```

Sections which specify formatter configuration are typified by the following.

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s %(customfield)s
datefmt=
style=%
validate=True
defaults={'customfield': 'defaultvalue'}
class=logging.Formatter
```

The arguments for the formatter configuration are the same as the keys in the dictionary schema *formatters section*.

The `defaults` entry, when *evaluated* in the context of the `logging` package's namespace, is a dictionary of default values for custom formatting fields. If not provided, it defaults to `None`.

Note

Due to the use of `eval()` as described above, there are potential security risks which result from using the `listen()` to send and receive configurations via sockets. The risks are limited to where multiple users with no mutual trust run code on the same machine; see the `listen()` documentation for more information.

See also

Module `logging`
API reference for the logging module.

Module `logging.handlers`
Useful handlers included with the logging module.

16.7 `logging.handlers` — Logging handlers

Source code: Lib/logging/handlers.py

Important

This page contains only reference information. For tutorials, please see

- Basic Tutorial
- Advanced Tutorial
- Logging Cookbook

The following useful handlers are provided in the package. Note that three of the handlers (`StreamHandler`, `FileHandler` and `NullHandler`) are actually defined in the `logging` module itself, but have been documented here along with the other handlers.

16.7.1 StreamHandler

The *StreamHandler* class, located in the core *logging* package, sends logging output to streams such as *sys.stdout*, *sys.stderr* or any file-like object (or, more precisely, any object which supports `write()` and `flush()` methods).

class logging.**StreamHandler**(*stream=None*)

Returns a new instance of the *StreamHandler* class. If *stream* is specified, the instance will use it for logging output; otherwise, *sys.stderr* will be used.

emit(*record*)

If a formatter is specified, it is used to format the record. The record is then written to the stream followed by *terminator*. If exception information is present, it is formatted using *traceback.print_exception()* and appended to the stream.

flush()

Flushes the stream by calling its *flush()* method. Note that the `close()` method is inherited from *Handler* and so does no output, so an explicit *flush()* call may be needed at times.

setStream(*stream*)

Sets the instance's stream to the specified value, if it is different. The old stream is flushed before the new stream is set.

Parameters
stream – The stream that the handler should use.

Returns
the old stream, if the stream was changed, or `None` if it wasn't.

Added in version 3.7.

terminator

String used as the terminator when writing a formatted record to a stream. Default value is `'\n'`.

If you don't want a newline termination, you can set the handler instance's `terminator` attribute to the empty string.

In earlier versions, the terminator was hardcoded as `'\n'`.

Added in version 3.2.

16.7.2 FileHandler

The *FileHandler* class, located in the core *logging* package, sends logging output to a disk file. It inherits the output functionality from *StreamHandler*.

class logging.**FileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*)

Returns a new instance of the *FileHandler* class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to *emit()*. By default, the file grows indefinitely. If *errors* is specified, it's used to determine how encoding errors are handled.

Changed in version 3.6: As well as string values, *Path* objects are also accepted for the *filename* argument.

Changed in version 3.9: The *errors* parameter was added.

close()

Closes the file.

emit(*record*)

Outputs the record to the file.

Note that if the file was closed due to logging shutdown at exit and the file mode is 'w', the record will not be emitted (see bpo-42378).

16.7.3 NullHandler

Added in version 3.1.

The *NullHandler* class, located in the core *logging* package, does not do any formatting or output. It is essentially a 'no-op' handler for use by library developers.

class logging.**NullHandler**

Returns a new instance of the *NullHandler* class.

emit(*record*)

This method does nothing.

handle(*record*)

This method does nothing.

createLock()

This method returns None for the lock, since there is no underlying I/O to which access needs to be serialized.

See library-config for more information on how to use *NullHandler*.

16.7.4 WatchedFileHandler

The *WatchedFileHandler* class, located in the *logging.handlers* module, is a FileHandler which watches the file it is logging to. If the file changes, it is closed and reopened using the file name.

A file change can happen because of usage of programs such as *newsyslog* and *logrotate* which perform log file rotation. This handler, intended for use under Unix/Linux, watches the file to see if it has changed since the last emit. (A file is deemed to have changed if its device or inode have changed.) If the file has changed, the old file stream is closed, and the file opened to get a new stream.

This handler is not appropriate for use under Windows, because under Windows open log files cannot be moved or renamed - logging opens the files with exclusive locks - and so there is no need for such a handler. Furthermore, *ST_INO* is not supported under Windows; *stat()* always returns zero for this value.

class logging.handlers.**WatchedFileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*)

Returns a new instance of the *WatchedFileHandler* class. The specified file is opened and used as the stream for logging. If *mode* is not specified, 'a' is used. If *encoding* is not None, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to *emit()*. By default, the file grows indefinitely. If *errors* is provided, it determines how encoding errors are handled.

Changed in version 3.6: As well as string values, *Path* objects are also accepted for the *filename* argument.

Changed in version 3.9: The *errors* parameter was added.

reopenIfNeeded()

Checks to see if the file has changed. If it has, the existing stream is flushed and closed and the file opened again, typically as a precursor to outputting the record to the file.

Added in version 3.6.

emit(*record*)

Outputs the record to the file, but first calls `reopenIfNeeded()` to reopen the file if it has changed.

16.7.5 BaseRotatingHandler

The `BaseRotatingHandler` class, located in the `logging.handlers` module, is the base class for the rotating file handlers, `RotatingFileHandler` and `TimedRotatingFileHandler`. You should not need to instantiate this class, but it has attributes and methods you may need to override.

class logging.handlers.**BaseRotatingHandler**(*filename*, *mode*, *encoding=None*, *delay=False*, *errors=None*)

The parameters are as for `FileHandler`. The attributes are:

namer

If this attribute is set to a callable, the `rotation_filename()` method delegates to this callable. The parameters passed to the callable are those passed to `rotation_filename()`.

> **Note**
>
> The namer function is called quite a few times during rollover, so it should be as simple and as fast as possible. It should also return the same output every time for a given input, otherwise the rollover behaviour may not work as expected.
>
> It's also worth noting that care should be taken when using a namer to preserve certain attributes in the filename which are used during rotation. For example, `RotatingFileHandler` expects to have a set of log files whose names contain successive integers, so that rotation works as expected, and `TimedRotatingFileHandler` deletes old log files (based on the `backupCount` parameter passed to the handler's initializer) by determining the oldest files to delete. For this to happen, the filenames should be sortable using the date/time portion of the filename, and a namer needs to respect this. (If a namer is wanted that doesn't respect this scheme, it will need to be used in a subclass of `TimedRotatingFileHandler` which overrides the `getFilesToDelete()` method to fit in with the custom naming scheme.)

Added in version 3.3.

rotator

If this attribute is set to a callable, the `rotate()` method delegates to this callable. The parameters passed to the callable are those passed to `rotate()`.

Added in version 3.3.

rotation_filename(*default_name*)

Modify the filename of a log file when rotating.

This is provided so that a custom filename can be provided.

The default implementation calls the 'namer' attribute of the handler, if it's callable, passing the default name to it. If the attribute isn't callable (the default is `None`), the name is returned unchanged.

Parameters

default_name – The default name for the log file.

Added in version 3.3.

rotate(*source*, *dest*)

When rotating, rotate the current log.

The default implementation calls the 'rotator' attribute of the handler, if it's callable, passing the source and dest arguments to it. If the attribute isn't callable (the default is `None`), the source is simply renamed to the destination.

Parameters

- **source** – The source filename. This is normally the base filename, e.g. 'test.log'.
- **dest** – The destination filename. This is normally what the source is rotated to, e.g. 'test.log.1'.

Added in version 3.3.

The reason the attributes exist is to save you having to subclass - you can use the same callables for instances of *RotatingFileHandler* and *TimedRotatingFileHandler*. If either the namer or rotator callable raises an exception, this will be handled in the same way as any other exception during an `emit()` call, i.e. via the `handleError()` method of the handler.

If you need to make more significant changes to rotation processing, you can override the methods.

For an example, see cookbook-rotator-namer.

16.7.6 RotatingFileHandler

The *RotatingFileHandler* class, located in the *logging.handlers* module, supports rotation of disk log files.

class logging.handlers.**RotatingFileHandler**(*filename*, *mode='a'*, *maxBytes=0*, *backupCount=0*, *encoding=None*, *delay=False*, *errors=None*)

Returns a new instance of the *RotatingFileHandler* class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to *emit()*. By default, the file grows indefinitely. If *errors* is provided, it determines how encoding errors are handled.

You can use the *maxBytes* and *backupCount* values to allow the file to *rollover* at a predetermined size. When the size is about to be exceeded, the file is closed and a new file is silently opened for output. Rollover occurs whenever the current log file is nearly *maxBytes* in length; but if either of *maxBytes* or *backupCount* is zero, rollover never occurs, so you generally want to set *backupCount* to at least 1, and have a non-zero *maxBytes*. When *backupCount* is non-zero, the system will save old log files by appending the extensions '.1', '.2' etc., to the filename. For example, with a *backupCount* of 5 and a base file name of `app.log`, you would get `app.log`, `app.log.1`, `app.log.2`, up to `app.log.5`. The file being written to is always `app.log`. When this file is filled, it is closed and renamed to `app.log.1`, and if files `app.log.1`, `app.log.2`, etc. exist, then they are renamed to `app.log.2`, `app.log.3` etc. respectively.

Changed in version 3.6: As well as string values, *Path* objects are also accepted for the *filename* argument.

Changed in version 3.9: The *errors* parameter was added.

doRollover()

Does a rollover, as described above.

emit(*record*)

Outputs the record to the file, catering for rollover as described previously.

16.7.7 TimedRotatingFileHandler

The *TimedRotatingFileHandler* class, located in the *logging.handlers* module, supports rotation of disk log files at certain timed intervals.

class logging.handlers.**TimedRotatingFileHandler**(*filename*, *when='h'*, *interval=1*, *backupCount=0*, *encoding=None*, *delay=False*, *utc=False*, *atTime=None*, *errors=None*)

Returns a new instance of the *TimedRotatingFileHandler* class. The specified file is opened and used as the stream for logging. On rotating it also sets the filename suffix. Rotating happens based on the product of *when* and *interval*.

You can use the *when* to specify the type of *interval*. The list of possible values is below. Note that they are not case sensitive.

Value	Type of interval	If/how *atTime* is used
`'S'`	Seconds	Ignored
`'M'`	Minutes	Ignored
`'H'`	Hours	Ignored
`'D'`	Days	Ignored
`'W0'-'W6'`	Weekday (0=Monday)	Used to compute initial rollover time
`'midnight'`	Roll over at midnight, if *atTime* not specified, else at time *atTime*	Used to compute initial rollover time

When using weekday-based rotation, specify 'W0' for Monday, 'W1' for Tuesday, and so on up to 'W6' for Sunday. In this case, the value passed for *interval* isn't used.

The system will save old log files by appending extensions to the filename. The extensions are date-and-time based, using the strftime format `%Y-%m-%d_%H-%M-%S` or a leading portion thereof, depending on the rollover interval.

When computing the next rollover time for the first time (when the handler is created), the last modification time of an existing log file, or else the current time, is used to compute when the next rotation will occur.

If the *utc* argument is true, times in UTC will be used; otherwise local time is used.

If *backupCount* is nonzero, at most *backupCount* files will be kept, and if more would be created when rollover occurs, the oldest one is deleted. The deletion logic uses the interval to determine which files to delete, so changing the interval may leave old files lying around.

If *delay* is true, then file opening is deferred until the first call to *emit()*.

If *atTime* is not `None`, it must be a `datetime.time` instance which specifies the time of day when rollover occurs, for the cases where rollover is set to happen "at midnight" or "on a particular weekday". Note that in these cases, the *atTime* value is effectively used to compute the *initial* rollover, and subsequent rollovers would be calculated via the normal interval calculation.

If *errors* is specified, it's used to determine how encoding errors are handled.

Note

Calculation of the initial rollover time is done when the handler is initialised. Calculation of subsequent rollover times is done only when rollover occurs, and rollover occurs only when emitting output. If this is not kept in mind, it might lead to some confusion. For example, if an interval of "every minute" is set, that does not mean you will always see log files with times (in the filename) separated by a minute; if, during application execution,

logging output is generated more frequently than once a minute, *then* you can expect to see log files with times separated by a minute. If, on the other hand, logging messages are only output once every five minutes (say), then there will be gaps in the file times corresponding to the minutes where no output (and hence no rollover) occurred.

Changed in version 3.4: *atTime* parameter was added.

Changed in version 3.6: As well as string values, `Path` objects are also accepted for the *filename* argument.

Changed in version 3.9: The *errors* parameter was added.

doRollover()

Does a rollover, as described above.

emit(*record*)

Outputs the record to the file, catering for rollover as described above.

getFilesToDelete()

Returns a list of filenames which should be deleted as part of rollover. These are the absolute paths of the oldest backup log files written by the handler.

16.7.8 SocketHandler

The `SocketHandler` class, located in the `logging.handlers` module, sends logging output to a network socket. The base class uses a TCP socket.

class logging.handlers.**SocketHandler**(*host*, *port*)

Returns a new instance of the `SocketHandler` class intended to communicate with a remote machine whose address is given by *host* and *port*.

Changed in version 3.4: If `port` is specified as `None`, a Unix domain socket is created using the value in `host` - otherwise, a TCP socket is created.

close()

Closes the socket.

emit()

Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. If the connection was previously lost, re-establishes the connection. To unpickle the record at the receiving end into a `LogRecord`, use the `makeLogRecord()` function.

handleError()

Handles an error which has occurred during `emit()`. The most likely cause is a lost connection. Closes the socket so that we can retry on the next event.

makeSocket()

This is a factory method which allows subclasses to define the precise type of socket they want. The default implementation creates a TCP socket (`socket.SOCK_STREAM`).

makePickle(*record*)

Pickles the record's attribute dictionary in binary format with a length prefix, and returns it ready for transmission across the socket. The details of this operation are equivalent to:

```
data = pickle.dumps(record_attr_dict, 1)
datalen = struct.pack('>L', len(data))
return datalen + data
```

Note that pickles aren't completely secure. If you are concerned about security, you may want to override this method to implement a more secure mechanism. For example, you can sign pickles using HMAC and then verify them on the receiving end, or alternatively you can disable unpickling of global objects on the receiving end.

send(*packet*)

Send a pickled byte-string *packet* to the socket. The format of the sent byte-string is as described in the documentation for *makePickle()*.

This function allows for partial sends, which can happen when the network is busy.

createSocket()

Tries to create a socket; on failure, uses an exponential back-off algorithm. On initial failure, the handler will drop the message it was trying to send. When subsequent messages are handled by the same instance, it will not try connecting until some time has passed. The default parameters are such that the initial delay is one second, and if after that delay the connection still can't be made, the handler will double the delay each time up to a maximum of 30 seconds.

This behaviour is controlled by the following handler attributes:

- `retryStart` (initial delay, defaulting to 1.0 seconds).
- `retryFactor` (multiplier, defaulting to 2.0).
- `retryMax` (maximum delay, defaulting to 30.0 seconds).

This means that if the remote listener starts up *after* the handler has been used, you could lose messages (since the handler won't even attempt a connection until the delay has elapsed, but just silently drop messages during the delay period).

16.7.9 DatagramHandler

The *DatagramHandler* class, located in the *logging.handlers* module, inherits from *SocketHandler* to support sending logging messages over UDP sockets.

class logging.handlers.**DatagramHandler**(*host*, *port*)

Returns a new instance of the *DatagramHandler* class intended to communicate with a remote machine whose address is given by *host* and *port*.

> **Note**
>
> As UDP is not a streaming protocol, there is no persistent connection between an instance of this handler and *host*. For this reason, when using a network socket, a DNS lookup might have to be made each time an event is logged, which can introduce some latency into the system. If this affects you, you can do a lookup yourself and initialize this handler using the looked-up IP address rather than the hostname.

Changed in version 3.4: If `port` is specified as `None`, a Unix domain socket is created using the value in `host` - otherwise, a UDP socket is created.

emit()

Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. To unpickle the record at the receiving end into a *LogRecord*, use the *makeLogRecord()* function.

makeSocket()

The factory method of *SocketHandler* is here overridden to create a UDP socket (*socket.SOCK_DGRAM*).

send(*s*)

Send a pickled byte-string to a socket. The format of the sent byte-string is as described in the documentation for *SocketHandler.makePickle()*.

16.7.10 SysLogHandler

The *SysLogHandler* class, located in the *logging.handlers* module, supports sending logging messages to a remote or local Unix syslog.

class logging.handlers.**SysLogHandler**(*address=('localhost', SYSLOG_UDP_PORT)*, *facility=LOG_USER*, *socktype=socket.SOCK_DGRAM*)

Returns a new instance of the *SysLogHandler* class intended to communicate with a remote Unix machine whose address is given by *address* in the form of a `(host, port)` tuple. If *address* is not specified, `('localhost', 514)` is used. The address is used to open a socket. An alternative to providing a `(host, port)` tuple is providing an address as a string, for example '/dev/log'. In this case, a Unix domain socket is used to send the message to the syslog. If *facility* is not specified, `LOG_USER` is used. The type of socket opened depends on the *socktype* argument, which defaults to *socket.SOCK_DGRAM* and thus opens a UDP socket. To open a TCP socket (for use with the newer syslog daemons such as rsyslog), specify a value of *socket.SOCK_STREAM*.

Note that if your server is not listening on UDP port 514, *SysLogHandler* may appear not to work. In that case, check what address you should be using for a domain socket - it's system dependent. For example, on Linux it's usually '/dev/log' but on OS/X it's '/var/run/syslog'. You'll need to check your platform and use the appropriate address (you may need to do this check at runtime if your application needs to run on several platforms). On Windows, you pretty much have to use the UDP option.

> **Note**
>
> On macOS 12.x (Monterey), Apple has changed the behaviour of their syslog daemon - it no longer listens on a domain socket. Therefore, you cannot expect *SysLogHandler* to work on this system.
>
> See gh-91070 for more information.

Changed in version 3.2: *socktype* was added.

close()

Closes the socket to the remote host.

createSocket()

Tries to create a socket and, if it's not a datagram socket, connect it to the other end. This method is called during handler initialization, but it's not regarded as an error if the other end isn't listening at this point - the method will be called again when emitting an event, if there is no socket at that point.

Added in version 3.11.

emit(*record*)

The record is formatted, and then sent to the syslog server. If exception information is present, it is *not* sent to the server.

Changed in version 3.2.1: (See: bpo-12168.) In earlier versions, the message sent to the syslog daemons was always terminated with a NUL byte, because early versions of these daemons expected a NUL terminated

message - even though it's not in the relevant specification (**RFC 5424**). More recent versions of these daemons don't expect the NUL byte but strip it off if it's there, and even more recent daemons (which adhere more closely to RFC 5424) pass the NUL byte on as part of the message.

To enable easier handling of syslog messages in the face of all these differing daemon behaviours, the appending of the NUL byte has been made configurable, through the use of a class-level attribute, `append_nul`. This defaults to `True` (preserving the existing behaviour) but can be set to `False` on a `SysLogHandler` instance in order for that instance to *not* append the NUL terminator.

Changed in version 3.3: (See: bpo-12419.) In earlier versions, there was no facility for an "ident" or "tag" prefix to identify the source of the message. This can now be specified using a class-level attribute, defaulting to `""` to preserve existing behaviour, but which can be overridden on a `SysLogHandler` instance in order for that instance to prepend the ident to every message handled. Note that the provided ident must be text, not bytes, and is prepended to the message exactly as is.

encodePriority(*facility*, *priority*)

Encodes the facility and priority into an integer. You can pass in strings or integers - if strings are passed, internal mapping dictionaries are used to convert them to integers.

The symbolic `LOG_` values are defined in *SysLogHandler* and mirror the values defined in the `sys/syslog.h` header file.

Priorities

Name (string)	Symbolic value
`alert`	LOG_ALERT
`crit` or `critical`	LOG_CRIT
`debug`	LOG_DEBUG
`emerg` or `panic`	LOG_EMERG
`err` or `error`	LOG_ERR
`info`	LOG_INFO
`notice`	LOG_NOTICE
`warn` or `warning`	LOG_WARNING

Facilities

Name (string)	Symbolic value
auth	LOG_AUTH
authpriv	LOG_AUTHPRIV
cron	LOG_CRON
daemon	LOG_DAEMON
ftp	LOG_FTP
kern	LOG_KERN
lpr	LOG_LPR
mail	LOG_MAIL
news	LOG_NEWS
syslog	LOG_SYSLOG
user	LOG_USER
uucp	LOG_UUCP
local0	LOG_LOCAL0
local1	LOG_LOCAL1
local2	LOG_LOCAL2
local3	LOG_LOCAL3
local4	LOG_LOCAL4
local5	LOG_LOCAL5
local6	LOG_LOCAL6
local7	LOG_LOCAL7

mapPriority(*levelname*)

Maps a logging level name to a syslog priority name. You may need to override this if you are using custom levels, or if the default algorithm is not suitable for your needs. The default algorithm maps `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL` to the equivalent syslog names, and all other level names to 'warning'.

16.7.11 NTEventLogHandler

The *NTEventLogHandler* class, located in the *logging.handlers* module, supports sending logging messages to a local Windows NT, Windows 2000 or Windows XP event log. Before you can use it, you need Mark Hammond's Win32 extensions for Python installed.

class logging.handlers.**NTEventLogHandler**(*appname*, *dllname=None*, *logtype='Application'*)

Returns a new instance of the *NTEventLogHandler* class. The *appname* is used to define the application name as it appears in the event log. An appropriate registry entry is created using this name. The *dllname* should give the fully qualified pathname of a .dll or .exe which contains message definitions to hold in the log (if not specified, `'win32service.pyd'` is used - this is installed with the Win32 extensions and contains some basic placeholder message definitions. Note that use of these placeholders will make your event logs big, as the entire message source is held in the log. If you want slimmer logs, you have to pass in the name of your own .dll or .exe which contains the message definitions you want to use in the event log). The *logtype* is one of `'Application'`, `'System'` or `'Security'`, and defaults to `'Application'`.

close()

At this point, you can remove the application name from the registry as a source of event log entries. However, if you do this, you will not be able to see the events as you intended in the Event Log Viewer - it needs to be able to access the registry to get the .dll name. The current version does not do this.

emit(*record*)

Determines the message ID, event category and event type, and then logs the message in the NT event log.

getEventCategory(*record*)

Returns the event category for the record. Override this if you want to specify your own categories. This version returns 0.

getEventType(*record*)

Returns the event type for the record. Override this if you want to specify your own types. This version does a mapping using the handler's typemap attribute, which is set up in `__init__()` to a dictionary which contains mappings for `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL`. If you are using your own levels, you will either need to override this method or place a suitable dictionary in the handler's *typemap* attribute.

getMessageID(*record*)

Returns the message ID for the record. If you are using your own messages, you could do this by having the *msg* passed to the logger being an ID rather than a format string. Then, in here, you could use a dictionary lookup to get the message ID. This version returns 1, which is the base message ID in `win32service.pyd`.

16.7.12 SMTPHandler

The *SMTPHandler* class, located in the *logging.handlers* module, supports sending logging messages to an email address via SMTP.

class logging.handlers.**SMTPHandler**(*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*, *timeout=1.0*)

Returns a new instance of the *SMTPHandler* class. The instance is initialized with the from and to addresses and subject line of the email. The *toaddrs* should be a list of strings. To specify a non-standard SMTP port, use the (host, port) tuple format for the *mailhost* argument. If you use a string, the standard SMTP port is used. If your SMTP server requires authentication, you can specify a (username, password) tuple for the *credentials* argument.

To specify the use of a secure protocol (TLS), pass in a tuple to the *secure* argument. This will only be used when authentication credentials are supplied. The tuple should be either an empty tuple, or a single-value tuple with the name of a keyfile, or a 2-value tuple with the names of the keyfile and certificate file. (This tuple is passed to the *smtplib.SMTP.starttls()* method.)

A timeout can be specified for communication with the SMTP server using the *timeout* argument.

Changed in version 3.3: Added the *timeout* parameter.

emit(*record*)

Formats the record and sends it to the specified addressees.

getSubject(*record*)

If you want to specify a subject line which is record-dependent, override this method.

16.7.13 MemoryHandler

The *MemoryHandler* class, located in the *logging.handlers* module, supports buffering of logging records in memory, periodically flushing them to a *target* handler. Flushing occurs whenever the buffer is full, or when an event of a certain severity or greater is seen.

MemoryHandler is a subclass of the more general *BufferingHandler*, which is an abstract class. This buffers logging records in memory. Whenever each record is added to the buffer, a check is made by calling `shouldFlush()` to see if the buffer should be flushed. If it should, then `flush()` is expected to do the flushing.

class logging.handlers.**BufferingHandler**(*capacity*)

Initializes the handler with a buffer of the specified capacity. Here, *capacity* means the number of logging records buffered.

emit(*record*)

Append the record to the buffer. If `shouldFlush()` returns true, call `flush()` to process the buffer.

flush()

For a `BufferingHandler` instance, flushing means that it sets the buffer to an empty list. This method can be overwritten to implement more useful flushing behavior.

shouldFlush(*record*)

Return `True` if the buffer is up to capacity. This method can be overridden to implement custom flushing strategies.

class logging.handlers.**MemoryHandler**(*capacity*, *flushLevel=ERROR*, *target=None*, *flushOnClose=True*)

Returns a new instance of the `MemoryHandler` class. The instance is initialized with a buffer size of *capacity* (number of records buffered). If *flushLevel* is not specified, `ERROR` is used. If no *target* is specified, the target will need to be set using `setTarget()` before this handler does anything useful. If *flushOnClose* is specified as `False`, then the buffer is *not* flushed when the handler is closed. If not specified or specified as `True`, the previous behaviour of flushing the buffer will occur when the handler is closed.

Changed in version 3.6: The *flushOnClose* parameter was added.

close()

Calls `flush()`, sets the target to `None` and clears the buffer.

flush()

For a `MemoryHandler` instance, flushing means just sending the buffered records to the target, if there is one. The buffer is also cleared when buffered records are sent to the target. Override if you want different behavior.

setTarget(*target*)

Sets the target handler for this handler.

shouldFlush(*record*)

Checks for buffer full or a record at the *flushLevel* or higher.

16.7.14 HTTPHandler

The `HTTPHandler` class, located in the `logging.handlers` module, supports sending logging messages to a web server, using either `GET` or `POST` semantics.

class logging.handlers.**HTTPHandler**(*host*, *url*, *method='GET'*, *secure=False*, *credentials=None*, *context=None*)

Returns a new instance of the `HTTPHandler` class. The *host* can be of the form `host:port`, should you need to use a specific port number. If no *method* is specified, `GET` is used. If *secure* is true, a HTTPS connection will be used. The *context* parameter may be set to a `ssl.SSLContext` instance to configure the SSL settings used for the HTTPS connection. If *credentials* is specified, it should be a 2-tuple consisting of userid and password, which will be placed in a HTTP 'Authorization' header using Basic authentication. If you specify credentials, you should also specify secure=True so that your userid and password are not passed in cleartext across the wire.

Changed in version 3.5: The *context* parameter was added.

mapLogRecord(*record*)

Provides a dictionary, based on `record`, which is to be URL-encoded and sent to the web server. The default implementation just returns `record.__dict__`. This method can be overridden if e.g. only a subset of *LogRecord* is to be sent to the web server, or if more specific customization of what's sent to the server is required.

emit(*record*)

Sends the record to the web server as a URL-encoded dictionary. The *mapLogRecord()* method is used to convert the record to the dictionary to be sent.

> **Note**
>
> Since preparing a record for sending it to a web server is not the same as a generic formatting operation, using *setFormatter()* to specify a *Formatter* for a *HTTPHandler* has no effect. Instead of calling *format()*, this handler calls *mapLogRecord()* and then *urllib.parse.urlencode()* to encode the dictionary in a form suitable for sending to a web server.

16.7.15 QueueHandler

Added in version 3.2.

The *QueueHandler* class, located in the *logging.handlers* module, supports sending logging messages to a queue, such as those implemented in the *queue* or *multiprocessing* modules.

Along with the *QueueListener* class, *QueueHandler* can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via *SMTPHandler*) are done on a separate thread.

class logging.handlers.**QueueHandler**(*queue*)

Returns a new instance of the *QueueHandler* class. The instance is initialized with the queue to send messages to. The *queue* can be any queue-like object; it's used as-is by the *enqueue()* method, which needs to know how to send messages to it. The queue is not *required* to have the task tracking API, which means that you can use *SimpleQueue* instances for *queue*.

> **Note**
>
> If you are using *multiprocessing*, you should avoid using *SimpleQueue* and instead use *multiprocessing.Queue*.

emit(*record*)

Enqueues the result of preparing the LogRecord. Should an exception occur (e.g. because a bounded queue has filled up), the *handleError()* method is called to handle the error. This can result in the record silently being dropped (if *logging.raiseExceptions* is `False`) or a message printed to `sys.stderr` (if *logging.raiseExceptions* is `True`).

prepare(*record*)

Prepares a record for queuing. The object returned by this method is enqueued.

The base implementation formats the record to merge the message, arguments, exception and stack information, if present. It also removes unpickleable items from the record in-place. Specifically, it overwrites the record's `msg` and `message` attributes with the merged message (obtained by calling the handler's *format()* method), and sets the `args`, `exc_info` and `exc_text` attributes to `None`.

You might want to override this method if you want to convert the record to a dict or JSON string, or send a modified copy of the record while leaving the original intact.

> **Note**
>
> The base implementation formats the message with arguments, sets the `message` and `msg` attributes to the formatted message and sets the `args` and `exc_text` attributes to `None` to allow pickling and to prevent further attempts at formatting. This means that a handler on the *QueueListener* side won't have the information to do custom formatting, e.g. of exceptions. You may wish to subclass `QueueHandler` and override this method to e.g. avoid setting `exc_text` to `None`. Note that the `message` / `msg` / `args` changes are related to ensuring the record is pickleable, and you might or might not be able to avoid doing that depending on whether your `args` are pickleable. (Note that you may have to consider not only your own code but also code in any libraries that you use.)

enqueue(*record*)

Enqueues the record on the queue using `put_nowait()`; you may want to override this if you want to use blocking behaviour, or a timeout, or a customized queue implementation.

listener

When created via configuration using *dictConfig()*, this attribute will contain a *QueueListener* instance for use with this handler. Otherwise, it will be `None`.

Added in version 3.12.

16.7.16 QueueListener

Added in version 3.2.

The *QueueListener* class, located in the *logging.handlers* module, supports receiving logging messages from a queue, such as those implemented in the *queue* or *multiprocessing* modules. The messages are received from a queue in an internal thread and passed, on the same thread, to one or more handlers for processing. While *QueueListener* is not itself a handler, it is documented here because it works hand-in-hand with *QueueHandler*.

Along with the *QueueHandler* class, *QueueListener* can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via *SMTPHandler*) are done on a separate thread.

class logging.handlers.**QueueListener**(*queue*, **handlers*, *respect_handler_level=False*)

Returns a new instance of the *QueueListener* class. The instance is initialized with the queue to send messages to and a list of handlers which will handle entries placed on the queue. The queue can be any queue-like object; it's passed as-is to the *dequeue()* method, which needs to know how to get messages from it. The queue is not *required* to have the task tracking API (though it's used if available), which means that you can use *SimpleQueue* instances for *queue*.

> **Note**
>
> If you are using *multiprocessing*, you should avoid using *SimpleQueue* and instead use *multiprocessing.Queue*.

If `respect_handler_level` is `True`, a handler's level is respected (compared with the level for the message) when deciding whether to pass messages to that handler; otherwise, the behaviour is as in previous Python versions - to always pass each message to each handler.

Changed in version 3.5: The `respect_handler_level` argument was added.

dequeue(*block*)

Dequeues a record and return it, optionally blocking.

The base implementation uses `get()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

prepare(*record*)

Prepare a record for handling.

This implementation just returns the passed-in record. You may want to override this method if you need to do any custom marshalling or manipulation of the record before passing it to the handlers.

handle(*record*)

Handle a record.

This just loops through the handlers offering them the record to handle. The actual object passed to the handlers is that which is returned from *prepare()*.

start()

Starts the listener.

This starts up a background thread to monitor the queue for LogRecords to process.

stop()

Stops the listener.

This asks the thread to terminate, and then waits for it to do so. Note that if you don't call this before your application exits, there may be some records still left on the queue, which won't be processed.

enqueue_sentinel()

Writes a sentinel to the queue to tell the listener to quit. This implementation uses `put_nowait()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

Added in version 3.3.

See also

Module *logging*
API reference for the logging module.

Module *logging.config*
Configuration API for the logging module.

16.8 `getpass` — Portable password input

Source code: Lib/getpass.py

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

The *getpass* module provides two functions:

getpass.**getpass**(*prompt='Password: '*, *stream=None*)

Prompt the user for a password without echoing. The user is prompted using the string *prompt*, which defaults to `'Password: '`. On Unix, the prompt is written to the file-like object *stream* using the replace error handler if needed. *stream* defaults to the controlling terminal (`/dev/tty`) or if that is unavailable to `sys.stderr` (this argument is ignored on Windows).

If echo free input is unavailable getpass() falls back to printing a warning message to *stream* and reading from `sys.stdin` and issuing a *GetPassWarning*.

> **Note**
>
> If you call getpass from within IDLE, the input may be done in the terminal you launched IDLE from rather than the idle window itself.

exception getpass.**GetPassWarning**

A *UserWarning* subclass issued when password input may be echoed.

getpass.**getuser**()

Return the "login name" of the user.

This function checks the environment variables `LOGNAME`, `USER`, `LNAME` and `USERNAME`, in order, and returns the value of the first one which is set to a non-empty string. If none are set, the login name from the password database is returned on systems which support the *pwd* module, otherwise, an *OSError* is raised.

In general, this function should be preferred over *os.getlogin()*.

Changed in version 3.13: Previously, various exceptions beyond just *OSError* were raised.

16.9 `curses` — Terminal handling for character-cell displays

Source code: Lib/curses

The *curses* module provides an interface to the curses library, the de-facto standard for portable advanced terminal handling.

While curses is most widely used in the Unix environment, versions are available for Windows, DOS, and possibly other systems as well. This extension module is designed to match the API of ncurses, an open-source curses library hosted on Linux and the BSD variants of Unix.

Availability: not WASI, not iOS.

This module does not work or is not available on WebAssembly platforms, or on iOS. See *WebAssembly platforms* for more information on WASM availability; see *iOS* for more information on iOS availability.

> **Note**
>
> Whenever the documentation mentions a *character* it can be specified as an integer, a one-character Unicode string or a one-byte byte string.
>
> Whenever the documentation mentions a *character string* it can be specified as a Unicode string or a byte string.

See also

Module *curses.ascii*
Utilities for working with ASCII characters, regardless of your locale settings.

Module *curses.panel*
A panel stack extension that adds depth to curses windows.

Module *curses.textpad*
Editable text widget for curses supporting **Emacs**-like bindings.

curses-howto
Tutorial material on using curses with Python, by Andrew Kuchling and Eric Raymond.

16.9.1 Functions

The module *curses* defines the following exception:

exception curses.**error**
Exception raised when a curses library function returns an error.

Note

Whenever *x* or *y* arguments to a function or a method are optional, they default to the current cursor location. Whenever *attr* is optional, it defaults to *A_NORMAL*.

The module *curses* defines the following functions:

curses.**baudrate**()
Return the output speed of the terminal in bits per second. On software terminal emulators it will have a fixed high value. Included for historical reasons; in former times, it was used to write output loops for time delays and occasionally to change interfaces depending on the line speed.

curses.**beep**()
Emit a short attention sound.

curses.**can_change_color**()
Return `True` or `False`, depending on whether the programmer can change the colors displayed by the terminal.

curses.**cbreak**()
Enter cbreak mode. In cbreak mode (sometimes called "rare" mode) normal tty line buffering is turned off and characters are available to be read one by one. However, unlike raw mode, special characters (interrupt, quit, suspend, and flow control) retain their effects on the tty driver and calling program. Calling first *raw()* then *cbreak()* leaves the terminal in cbreak mode.

curses.**color_content**(*color_number*)
Return the intensity of the red, green, and blue (RGB) components in the color *color_number*, which must be between `0` and `COLORS - 1`. Return a 3-tuple, containing the R,G,B values for the given color, which will be between `0` (no component) and `1000` (maximum amount of component).

curses.**color_pair**(*pair_number*)
Return the attribute value for displaying text in the specified color pair. Only the first 256 color pairs are supported. This attribute value can be combined with *A_STANDOUT*, *A_REVERSE*, and the other `A_*` attributes. *pair_number()* is the counterpart to this function.

curses.**curs_set**(*visibility*)

Set the cursor state. *visibility* can be set to 0, 1, or 2, for invisible, normal, or very visible. If the terminal supports the visibility requested, return the previous cursor state; otherwise raise an exception. On many terminals, the "visible" mode is an underline cursor and the "very visible" mode is a block cursor.

curses.**def_prog_mode**()

Save the current terminal mode as the "program" mode, the mode when the running program is using curses. (Its counterpart is the "shell" mode, for when the program is not in curses.) Subsequent calls to *reset_prog_mode()* will restore this mode.

curses.**def_shell_mode**()

Save the current terminal mode as the "shell" mode, the mode when the running program is not using curses. (Its counterpart is the "program" mode, when the program is using curses capabilities.) Subsequent calls to *reset_shell_mode()* will restore this mode.

curses.**delay_output**(*ms*)

Insert an *ms* millisecond pause in output.

curses.**doupdate**()

Update the physical screen. The curses library keeps two data structures, one representing the current physical screen contents and a virtual screen representing the desired next state. The *doupdate()* ground updates the physical screen to match the virtual screen.

The virtual screen may be updated by a *noutrefresh()* call after write operations such as *addstr()* have been performed on a window. The normal *refresh()* call is simply `noutrefresh()` followed by `doupdate()`; if you have to update multiple windows, you can speed performance and perhaps reduce screen flicker by issuing `noutrefresh()` calls on all windows, followed by a single `doupdate()`.

curses.**echo**()

Enter echo mode. In echo mode, each character input is echoed to the screen as it is entered.

curses.**endwin**()

De-initialize the library, and return terminal to normal status.

curses.**erasechar**()

Return the user's current erase character as a one-byte bytes object. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses.**filter**()

The *filter()* routine, if used, must be called before *initscr()* is called. The effect is that, during those calls, `LINES` is set to `1`; the capabilities `clear`, `cup`, `cud`, `cud1`, `cuu1`, `cuu`, `vpa` are disabled; and the `home` string is set to the value of `cr`. The effect is that the cursor is confined to the current line, and so are screen updates. This may be used for enabling character-at-a-time line editing without touching the rest of the screen.

curses.**flash**()

Flash the screen. That is, change it to reverse-video and then change it back in a short interval. Some people prefer such as 'visible bell' to the audible attention signal produced by *beep()*.

curses.**flushinp**()

Flush all input buffers. This throws away any typeahead that has been typed by the user and has not yet been processed by the program.

curses.**getmouse**()

After *getch()* returns *KEY_MOUSE* to signal a mouse event, this method should be called to retrieve the queued mouse event, represented as a 5-tuple `(id, x, y, z, bstate)`. *id* is an ID value used to distinguish multiple devices, and *x*, *y*, *z* are the event's coordinates. (*z* is currently unused.) *bstate* is an integer value whose bits will be set to indicate the type of event, and will be the bitwise OR of one or more of the following constants, where

n is the button number from 1 to 5: *BUTTONn_PRESSED*, *BUTTONn_RELEASED*, *BUTTONn_CLICKED*, *BUTTONn_DOUBLE_CLICKED*, *BUTTONn_TRIPLE_CLICKED*, *BUTTON_SHIFT*, *BUTTON_CTRL*, *BUTTON_ALT*.

Changed in version 3.10: The `BUTTON5_*` constants are now exposed if they are provided by the underlying curses library.

`curses.`**`getsyx`**`()`

Return the current coordinates of the virtual screen cursor as a tuple `(y, x)`. If *leaveok* is currently `True`, then return `(-1, -1)`.

`curses.`**`getwin`**`(`*file*`)`

Read window related data stored in the file by an earlier *window.putwin()* call. The routine then creates and initializes a new window using that data, returning the new window object.

`curses.`**`has_colors`**`()`

Return `True` if the terminal can display colors; otherwise, return `False`.

`curses.`**`has_extended_color_support`**`()`

Return `True` if the module supports extended colors; otherwise, return `False`. Extended color support allows more than 256 color pairs for terminals that support more than 16 colors (e.g. xterm-256color).

Extended color support requires ncurses version 6.1 or later.

Added in version 3.10.

`curses.`**`has_ic`**`()`

Return `True` if the terminal has insert- and delete-character capabilities. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

`curses.`**`has_il`**`()`

Return `True` if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

`curses.`**`has_key`**`(`*ch*`)`

Take a key value *ch*, and return `True` if the current terminal type recognizes a key with that value.

`curses.`**`halfdelay`**`(`*tenths*`)`

Used for half-delay mode, which is similar to cbreak mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, raise an exception if nothing has been typed. The value of *tenths* must be a number between `1` and `255`. Use *nocbreak()* to leave half-delay mode.

`curses.`**`init_color`**`(`*color_number*, *r*, *g*, *b*`)`

Change the definition of a color, taking the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of *color_number* must be between `0` and `COLORS - 1`. Each of *r*, *g*, *b*, must be a value between `0` and `1000`. When *init_color()* is used, all occurrences of that color on the screen immediately change to the new definition. This function is a no-op on most terminals; it is active only if *can_change_color()* returns `True`.

`curses.`**`init_pair`**`(`*pair_number*, *fg*, *bg*`)`

Change the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of *pair_number* must be between `1` and `COLOR_PAIRS - 1` (the `0` color pair is wired to white on black and cannot be changed). The value of *fg* and *bg* arguments must be between `0` and `COLORS - 1`, or, after calling *use_default_colors()*, `-1`. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair are changed to the new definition.

curses.**initscr**()

Initialize the library. Return a *window* object which represents the whole screen.

> **Note**
>
> If there is an error opening the terminal, the underlying curses library may cause the interpreter to exit.

curses.**is_term_resized**(*nlines*, *ncols*)

Return `True` if *resize_term()* would modify the window structure, `False` otherwise.

curses.**isendwin**()

Return `True` if *endwin()* has been called (that is, the curses library has been deinitialized).

curses.**keyname**(*k*)

Return the name of the key numbered *k* as a bytes object. The name of a key generating printable ASCII character is the key's character. The name of a control-key combination is a two-byte bytes object consisting of a caret (`b'^'`) followed by the corresponding printable ASCII character. The name of an alt-key combination (128–255) is a bytes object consisting of the prefix `b'M-'` followed by the name of the corresponding ASCII character.

curses.**killchar**()

Return the user's current line kill character as a one-byte bytes object. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses.**longname**()

Return a bytes object containing the terminfo long name field describing the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to *initscr()*.

curses.**meta**(*flag*)

If *flag* is `True`, allow 8-bit characters to be input. If *flag* is `False`, allow only 7-bit chars.

curses.**mouseinterval**(*interval*)

Set the maximum time in milliseconds that can elapse between press and release events in order for them to be recognized as a click, and return the previous interval value. The default value is 200 milliseconds, or one fifth of a second.

curses.**mousemask**(*mousemask*)

Set the mouse events to be reported, and return a tuple `(availmask, oldmask)`. *availmask* indicates which of the specified mouse events can be reported; on complete failure it returns `0`. *oldmask* is the previous value of the given window's mouse event mask. If this function is never called, no mouse events are ever reported.

curses.**napms**(*ms*)

Sleep for *ms* milliseconds.

curses.**newpad**(*nlines*, *ncols*)

Create and return a pointer to a new pad data structure with the given number of lines and columns. Return a pad as a window object.

A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (such as from scrolling or echoing of input) do not occur. The *refresh()* and *noutrefresh()* methods of a pad require 6 arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display. The arguments are *pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*; the *p* arguments refer to the upper left corner of the pad region to be displayed and the *s* arguments define a clipping box on the screen within which the pad region is to be displayed.

curses.**newwin**(*nlines*, *ncols*)

curses.**newwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)

Return a new *window*, whose left-upper corner is at `(begin_y, begin_x)`, and whose height/width is *nlines*/*ncols*.

By default, the window will extend from the specified position to the lower right corner of the screen.

curses.**nl**()

Enter newline mode. This mode translates the return key into newline on input, and translates newline into return and line-feed on output. Newline mode is initially on.

curses.**nocbreak**()

Leave cbreak mode. Return to normal "cooked" mode with line buffering.

curses.**noecho**()

Leave echo mode. Echoing of input characters is turned off.

curses.**nonl**()

Leave newline mode. Disable translation of return into newline on input, and disable low-level translation of newline into newline/return on output (but this does not change the behavior of `addch('\n')`, which always does the equivalent of return and line feed on the virtual screen). With translation off, curses can sometimes speed up vertical motion a little; also, it will be able to detect the return key on input.

curses.**noqiflush**()

When the `noqiflush()` routine is used, normal flush of input and output queues associated with the `INTR`, `QUIT` and `SUSP` characters will not be done. You may want to call `noqiflush()` in a signal handler if you want output to continue as though the interrupt had not occurred, after the handler exits.

curses.**noraw**()

Leave raw mode. Return to normal "cooked" mode with line buffering.

curses.**pair_content**(*pair_number*)

Return a tuple `(fg, bg)` containing the colors for the requested color pair. The value of *pair_number* must be between `0` and `COLOR_PAIRS - 1`.

curses.**pair_number**(*attr*)

Return the number of the color-pair set by the attribute value *attr*. *color_pair()* is the counterpart to this function.

curses.**putp**(*str*)

Equivalent to `tputs(str, 1, putchar)`; emit the value of a specified terminfo capability for the current terminal. Note that the output of *putp()* always goes to standard output.

curses.**qiflush**([*flag*])

If *flag* is `False`, the effect is the same as calling *noqiflush()*. If *flag* is `True`, or no argument is provided, the queues will be flushed when these control characters are read.

curses.**raw**()

Enter raw mode. In raw mode, normal line buffering and processing of interrupt, quit, suspend, and flow control keys are turned off; characters are presented to curses input functions one by one.

curses.**reset_prog_mode**()

Restore the terminal to "program" mode, as previously saved by *def_prog_mode()*.

curses.**reset_shell_mode**()

Restore the terminal to "shell" mode, as previously saved by *def_shell_mode()*.

curses.**resetty**()

Restore the state of the terminal modes to what it was at the last call to *savetty()*.

curses.**resize_term**(*nlines*, *ncols*)

Backend function used by *resizeterm()*, performing most of the work; when resizing the windows, *resize_term()* blank-fills the areas that are extended. The calling application should fill in these areas with appropriate data. The `resize_term()` function attempts to resize all windows. However, due to the calling convention of pads, it is not possible to resize these without additional interaction with the application.

curses.**resizeterm**(*nlines*, *ncols*)

Resize the standard and current windows to the specified dimensions, and adjusts other bookkeeping data used by the curses library that record the window dimensions (in particular the SIGWINCH handler).

curses.**savetty**()

Save the current state of the terminal modes in a buffer, usable by *resetty()*.

curses.**get_escdelay**()

Retrieves the value set by *set_escdelay()*.

Added in version 3.9.

curses.**set_escdelay**(*ms*)

Sets the number of milliseconds to wait after reading an escape character, to distinguish between an individual escape character entered on the keyboard from escape sequences sent by cursor and function keys.

Added in version 3.9.

curses.**get_tabsize**()

Retrieves the value set by *set_tabsize()*.

Added in version 3.9.

curses.**set_tabsize**(*size*)

Sets the number of columns used by the curses library when converting a tab character to spaces as it adds the tab to a window.

Added in version 3.9.

curses.**setsyx**(*y*, *x*)

Set the virtual screen cursor to *y*, *x*. If *y* and *x* are both `-1`, then *leaveok* is set `True`.

curses.**setupterm**(*term=None*, *fd=-1*)

Initialize the terminal. *term* is a string giving the terminal name, or `None`; if omitted or `None`, the value of the `TERM` environment variable will be used. *fd* is the file descriptor to which any initialization sequences will be sent; if not supplied or `-1`, the file descriptor for `sys.stdout` will be used.

curses.**start_color**()

Must be called if the programmer wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after *initscr()*.

start_color() initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white), and two global variables in the *curses* module, *COLORS* and *COLOR_PAIRS*, containing the maximum number of colors and color-pairs the terminal can support. It also restores the colors on the terminal to the values they had when the terminal was just turned on.

curses.**termattrs**()

Return a logical OR of all video attributes supported by the terminal. This information is useful when a curses program needs complete control over the appearance of the screen.

curses.**termname**()

Return the value of the environment variable `TERM`, as a bytes object, truncated to 14 characters.

curses.**tigetflag**(*capname*)

Return the value of the Boolean capability corresponding to the terminfo capability name *capname* as an integer. Return the value `-1` if *capname* is not a Boolean capability, or `0` if it is canceled or absent from the terminal description.

curses.**tigetnum**(*capname*)

Return the value of the numeric capability corresponding to the terminfo capability name *capname* as an integer. Return the value `-2` if *capname* is not a numeric capability, or `-1` if it is canceled or absent from the terminal description.

curses.**tigetstr**(*capname*)

Return the value of the string capability corresponding to the terminfo capability name *capname* as a bytes object. Return `None` if *capname* is not a terminfo "string capability", or is canceled or absent from the terminal description.

curses.**tparm**(*str*[, ...])

Instantiate the bytes object *str* with the supplied parameters, where *str* should be a parameterized string obtained from the terminfo database. E.g. `tparm(tigetstr("cup"), 5, 3)` could result in `b'\033[6;4H'`, the exact result depending on terminal type.

curses.**typeahead**(*fd*)

Specify that the file descriptor *fd* be used for typeahead checking. If *fd* is `-1`, then no typeahead checking is done.

The curses library does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until refresh or doupdate is called again, allowing faster response to commands typed in advance. This function allows specifying a different file descriptor for typeahead checking.

curses.**unctrl**(*ch*)

Return a bytes object which is a printable representation of the character *ch*. Control characters are represented as a caret followed by the character, for example as `b'^C'`. Printing characters are left as they are.

curses.**ungetch**(*ch*)

Push *ch* so the next *getch()* will return it.

> **Note**
>
> Only one *ch* can be pushed before `getch()` is called.

curses.**update_lines_cols**()

Update the *LINES* and *COLS* module variables. Useful for detecting manual screen resize.

Added in version 3.5.

curses.**unget_wch**(*ch*)

Push *ch* so the next *get_wch()* will return it.

> **Note**
>
> Only one *ch* can be pushed before `get_wch()` is called.

Added in version 3.3.

curses.**ungetmouse**(*id*, *x*, *y*, *z*, *bstate*)

Push a *KEY_MOUSE* event onto the input queue, associating the given state data with it.

curses.**use_env**(*flag*)

If used, this function should be called before *initscr()* or newterm are called. When *flag* is False, the values of lines and columns specified in the terminfo database will be used, even if environment variables LINES and COLUMNS (used by default) are set, or if curses is running in a window (in which case default behavior would be to use the window size if LINES and COLUMNS are not set).

curses.**use_default_colors**()

Allow use of default values for colors on terminals supporting this feature. Use this to support transparency in your application. The default color is assigned to the color number -1. After calling this function, init_pair(x, curses.COLOR_RED, -1) initializes, for instance, color pair *x* to a red foreground color on the default background.

curses.**wrapper**(*func*, /, **args*, ***kwargs*)

Initialize curses and call another callable object, *func*, which should be the rest of your curses-using application. If the application raises an exception, this function will restore the terminal to a sane state before re-raising the exception and generating a traceback. The callable object *func* is then passed the main window 'stdscr' as its first argument, followed by any other arguments passed to wrapper(). Before calling *func*, wrapper() turns on cbreak mode, turns off echo, enables the terminal keypad, and initializes colors if the terminal has color support. On exit (whether normally or by exception) it restores cooked mode, turns on echo, and disables the terminal keypad.

16.9.2 Window Objects

Window objects, as returned by *initscr()* and *newwin()* above, have the following methods and attributes:

window.**addch**(*ch*[, *attr*])

window.**addch**(*y*, *x*, *ch*[, *attr*])

Paint character *ch* at (y, x) with attributes *attr*, overwriting any character previously painted at that location. By default, the character position and attributes are the current settings for the window object.

> **Note**
>
> Writing outside the window, subwindow, or pad raises a *curses.error*. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the character is printed.

window.**addnstr**(*str*, *n*[, *attr*])

window.**addnstr**(*y*, *x*, *str*, *n*[, *attr*])

Paint at most *n* characters of the character string *str* at (y, x) with attributes *attr*, overwriting anything previously on the display.

window.**addstr**(*str*[, *attr*])

window.**addstr**(*y*, *x*, *str*[, *attr*])

Paint the character string *str* at (y, x) with attributes *attr*, overwriting anything previously on the display.

> **Note**
>
> - Writing outside the window, subwindow, or pad raises *curses.error*. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the string is printed.

- A bug in ncurses, the backend for this Python module, can cause SegFaults when resizing windows. This is fixed in ncurses-6.1-20190511. If you are stuck with an earlier ncurses, you can avoid triggering this if you do not call `addstr()` with a *str* that has embedded newlines. Instead, call `addstr()` separately for each line.

`window.`**`attroff`**`(`*attr*`)`

Remove attribute *attr* from the "background" set applied to all writes to the current window.

`window.`**`attron`**`(`*attr*`)`

Add attribute *attr* from the "background" set applied to all writes to the current window.

`window.`**`attrset`**`(`*attr*`)`

Set the "background" set of attributes to *attr*. This set is initially `0` (no attributes).

`window.`**`bkgd`**`(`*ch*[, *attr*]`)`

Set the background property of the window to the character *ch*, with attributes *attr*. The change is then applied to every character position in that window:

- The attribute of every character in the window is changed to the new background attribute.
- Wherever the former background character appears, it is changed to the new background character.

`window.`**`bkgdset`**`(`*ch*[, *attr*]`)`

Set the window's background. A window's background consists of a character and any combination of attributes. The attribute part of the background is combined (OR'ed) with all non-blank characters that are written into the window. Both the character and attribute parts of the background are combined with the blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations.

`window.`**`border`**`(`[*ls*[, *rs*[, *ts*[, *bs*[, *tl*[, *tr*[, *bl*[, *br*]]]]]]]]`)`

Draw a border around the edges of the window. Each parameter specifies the character to use for a specific part of the border; see the table below for more details.

Note

A `0` value for any parameter will cause the default character to be used for that parameter. Keyword parameters can *not* be used. The defaults are listed in this table:

Parameter	Description	Default value
ls	Left side	`ACS_VLINE`
rs	Right side	`ACS_VLINE`
ts	Top	`ACS_HLINE`
bs	Bottom	`ACS_HLINE`
tl	Upper-left corner	`ACS_ULCORNER`
tr	Upper-right corner	`ACS_URCORNER`
bl	Bottom-left corner	`ACS_LLCORNER`
br	Bottom-right corner	`ACS_LRCORNER`

`window.`**`box`**`(`[*vertch*, *horch*]`)`

Similar to `border()`, but both *ls* and *rs* are *vertch* and both *ts* and *bs* are *horch*. The default corner characters are always used by this function.

`window.`**`chgat`**`(`*attr*`)`

window.**chgat**(*num*, *attr*)

window.**chgat**(*y*, *x*, *attr*)

window.**chgat**(*y*, *x*, *num*, *attr*)

Set the attributes of *num* characters at the current cursor position, or at position `(y, x)` if supplied. If *num* is not given or is `-1`, the attribute will be set on all the characters to the end of the line. This function moves cursor to position `(y, x)` if supplied. The changed line will be touched using the *touchline()* method so that the contents will be redisplayed by the next window refresh.

window.**clear**()

Like *erase()*, but also cause the whole window to be repainted upon next call to *refresh()*.

window.**clearok**(*flag*)

If *flag* is `True`, the next call to *refresh()* will clear the window completely.

window.**clrtobot**()

Erase from cursor to the end of the window: all lines below the cursor are deleted, and then the equivalent of *clrtoeol()* is performed.

window.**clrtoeol**()

Erase from cursor to the end of the line.

window.**cursyncup**()

Update the current cursor position of all the ancestors of the window to reflect the current cursor position of the window.

window.**delch**([*y*, *x*])

Delete any character at `(y, x)`.

window.**deleteln**()

Delete the line under the cursor. All following lines are moved up by one line.

window.**derwin**(*begin_y*, *begin_x*)

window.**derwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)

An abbreviation for "derive window", *derwin()* is the same as calling *subwin()*, except that *begin_y* and *begin_x* are relative to the origin of the window, rather than relative to the entire screen. Return a window object for the derived window.

window.**echochar**(*ch*[, *attr*])

Add character *ch* with attribute *attr*, and immediately call *refresh()* on the window.

window.**enclose**(*y*, *x*)

Test whether the given pair of screen-relative character-cell coordinates are enclosed by the given window, returning `True` or `False`. It is useful for determining what subset of the screen windows enclose the location of a mouse event.

Changed in version 3.10: Previously it returned `1` or `0` instead of `True` or `False`.

window.**encoding**

Encoding used to encode method arguments (Unicode strings and characters). The encoding attribute is inherited from the parent window when a subwindow is created, for example with *window.subwin()*. By default, current locale encoding is used (see *locale.getencoding()*).

Added in version 3.3.

window.**erase**()

Clear the window.

window.**getbegyx**()

Return a tuple `(y, x)` of coordinates of upper-left corner.

window.**getbkgd**()

Return the given window's current background character/attribute pair.

window.**getch**([*y*, *x*])

Get a character. Note that the integer returned does *not* have to be in ASCII range: function keys, keypad keys and so on are represented by numbers higher than 255. In no-delay mode, return `-1` if there is no input, otherwise wait until a key is pressed.

window.**get_wch**([*y*, *x*])

Get a wide character. Return a character for most keys, or an integer for function keys, keypad keys, and other special keys. In no-delay mode, raise an exception if there is no input.

Added in version 3.3.

window.**getkey**([*y*, *x*])

Get a character, returning a string instead of an integer, as *getch()* does. Function keys, keypad keys and other special keys return a multibyte string containing the key name. In no-delay mode, raise an exception if there is no input.

window.**getmaxyx**()

Return a tuple `(y, x)` of the height and width of the window.

window.**getparyx**()

Return the beginning coordinates of this window relative to its parent window as a tuple `(y, x)`. Return `(-1, -1)` if this window has no parent.

window.**getstr**()

window.**getstr**(*n*)

window.**getstr**(*y*, *x*)

window.**getstr**(*y*, *x*, *n*)

Read a bytes object from the user, with primitive line editing capacity.

window.**getyx**()

Return a tuple `(y, x)` of current cursor position relative to the window's upper-left corner.

window.**hline**(*ch*, *n*)

window.**hline**(*y*, *x*, *ch*, *n*)

Display a horizontal line starting at `(y, x)` with length *n* consisting of the character *ch*.

window.**idcok**(*flag*)

If *flag* is `False`, curses no longer considers using the hardware insert/delete character feature of the terminal; if *flag* is `True`, use of character insertion and deletion is enabled. When curses is first initialized, use of character insert/delete is enabled by default.

window.**idlok**(*flag*)

If *flag* is `True`, *curses* will try and use hardware line editing facilities. Otherwise, line insertion/deletion are disabled.

window.**immedok**(*flag*)

If *flag* is `True`, any change in the window image automatically causes the window to be refreshed; you no longer have to call *refresh()* yourself. However, it may degrade performance considerably, due to repeated calls to wrefresh. This option is disabled by default.

window.**inch**([*y*, *x*])

Return the character at the given position in the window. The bottom 8 bits are the character proper, and upper bits are the attributes.

window.**insch**(*ch*[, *attr*])

window.**insch**(*y*, *x*, *ch*[, *attr*])

Paint character *ch* at `(y, x)` with attributes *attr*, moving the line from position *x* right by one character.

window.**insdelln**(*nlines*)

Insert *nlines* lines into the specified window above the current line. The *nlines* bottom lines are lost. For negative *nlines*, delete *nlines* lines starting with the one under the cursor, and move the remaining lines up. The bottom *nlines* lines are cleared. The current cursor position remains the same.

window.**insertln**()

Insert a blank line under the cursor. All following lines are moved down by one line.

window.**insnstr**(*str*, *n*[, *attr*])

window.**insnstr**(*y*, *x*, *str*, *n*[, *attr*])

Insert a character string (as many characters as will fit on the line) before the character under the cursor, up to *n* characters. If *n* is zero or negative, the entire string is inserted. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

window.**insstr**(*str*[, *attr*])

window.**insstr**(*y*, *x*, *str*[, *attr*])

Insert a character string (as many characters as will fit on the line) before the character under the cursor. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

window.**instr**([*n*])

window.**instr**(*y*, *x*[, *n*])

Return a bytes object of characters, extracted from the window starting at the current cursor position, or at *y*, *x* if specified. Attributes are stripped from the characters. If *n* is specified, *instr()* returns a string at most *n* characters long (exclusive of the trailing NUL).

window.**is_linetouched**(*line*)

Return `True` if the specified line was modified since the last call to *refresh()*; otherwise return `False`. Raise a *curses.error* exception if *line* is not valid for the given window.

window.**is_wintouched**()

Return `True` if the specified window was modified since the last call to *refresh()*; otherwise return `False`.

window.**keypad**(*flag*)

If *flag* is `True`, escape sequences generated by some keys (keypad, function keys) will be interpreted by *curses*. If *flag* is `False`, escape sequences will be left as is in the input stream.

window.**leaveok**(*flag*)

If *flag* is `True`, cursor is left where it is on update, instead of being at "cursor position." This reduces cursor movement where possible. If possible the cursor will be made invisible.

If *flag* is `False`, cursor will always be at "cursor position" after an update.

window.**move**(*new_y*, *new_x*)

Move cursor to `(new_y, new_x)`.

window.**mvderwin**(*y*, *x*)

Move the window inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

window.**mvwin**(*new_y*, *new_x*)

Move the window so its upper-left corner is at `(new_y, new_x)`.

window.**nodelay**(*flag*)

If *flag* is `True`, *getch()* will be non-blocking.

window.**notimeout**(*flag*)

If *flag* is `True`, escape sequences will not be timed out.

If *flag* is `False`, after a few milliseconds, an escape sequence will not be interpreted, and will be left in the input stream as is.

window.**noutrefresh**()

Mark for refresh but wait. This function updates the data structure representing the desired state of the window, but does not force an update of the physical screen. To accomplish that, call *doupdate()*.

window.**overlay**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])

Overlay the window on top of *destwin*. The windows need not be the same size, only the overlapping region is copied. This copy is non-destructive, which means that the current background character does not overwrite the old contents of *destwin*.

To get fine-grained control over the copied region, the second form of *overlay()* can be used. *sminrow* and *smincol* are the upper-left coordinates of the source window, and the other variables mark a rectangle in the destination window.

window.**overwrite**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])

Overwrite the window on top of *destwin*. The windows need not be the same size, in which case only the overlapping region is copied. This copy is destructive, which means that the current background character overwrites the old contents of *destwin*.

To get fine-grained control over the copied region, the second form of *overwrite()* can be used. *sminrow* and *smincol* are the upper-left coordinates of the source window, the other variables mark a rectangle in the destination window.

window.**putwin**(*file*)

Write all data associated with the window into the provided file object. This information can be later retrieved using the *getwin()* function.

window.**redrawln**(*beg*, *num*)

Indicate that the *num* screen lines, starting at line *beg*, are corrupted and should be completely redrawn on the next *refresh()* call.

window.**redrawwin**()

Touch the entire window, causing it to be completely redrawn on the next *refresh()* call.

window.**refresh**([*pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*])

Update the display immediately (sync actual screen with previous drawing/deleting methods).

The 6 optional arguments can only be specified when the window is a pad created with *newpad()*. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left-hand corner of the rectangle to be displayed in the pad. *sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow*, or *smincol* are treated as if they were zero.

window.**resize**(*nlines*, *ncols*)

Reallocate storage for a curses window to adjust its dimensions to the specified values. If either dimension is larger than the current values, the window's data is filled with blanks that have the current background rendition (as set by *bkgdset()*) merged into them.

window.**scroll**([*lines=1*])

Scroll the screen or scrolling region upward by *lines* lines.

window.**scrollok**(*flag*)

Control what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action on the bottom line, or typing the last character of the last line. If *flag* is `False`, the cursor is left on the bottom line. If *flag* is `True`, the window is scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call *idlok()*.

window.**setscrreg**(*top*, *bottom*)

Set the scrolling region from line *top* to line *bottom*. All scrolling actions will take place in this region.

window.**standend**()

Turn off the standout attribute. On some terminals this has the side effect of turning off all attributes.

window.**standout**()

Turn on attribute *A_STANDOUT*.

window.**subpad**(*begin_y*, *begin_x*)

window.**subpad**(*nlines*, *ncols*, *begin_y*, *begin_x*)

Return a sub-window, whose upper-left corner is at `(begin_y, begin_x)`, and whose width/height is *ncols*/*nlines*.

window.**subwin**(*begin_y*, *begin_x*)

window.**subwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)

Return a sub-window, whose upper-left corner is at `(begin_y, begin_x)`, and whose width/height is *ncols*/*nlines*.

By default, the sub-window will extend from the specified position to the lower right corner of the window.

window.**syncdown**()

Touch each location in the window that has been touched in any of its ancestor windows. This routine is called by *refresh()*, so it should almost never be necessary to call it manually.

window.**syncok**(*flag*)

If *flag* is `True`, then *syncup()* is called automatically whenever there is a change in the window.

window.**syncup**()

Touch all locations in ancestors of the window that have been changed in the window.

window.**timeout**(*delay*)

Set blocking or non-blocking read behavior for the window. If *delay* is negative, blocking read is used (which will wait indefinitely for input). If *delay* is zero, then non-blocking read is used, and *getch()* will return `-1` if no input is waiting. If *delay* is positive, then *getch()* will block for *delay* milliseconds, and return `-1` if there is still no input at the end of that time.

window.**touchline**(*start*, *count*[, *changed*])

Pretend *count* lines have been changed, starting with line *start*. If *changed* is supplied, it specifies whether the affected lines are marked as having been changed (*changed*=`True`) or unchanged (*changed*=`False`).

window.**touchwin**()

Pretend the whole window has been changed, for purposes of drawing optimizations.

window.**untouchwin**()

Mark all lines in the window as unchanged since the last call to *refresh()*.

window.**vline**(*ch*, *n*[, *attr*])

window.**vline**(*y*, *x*, *ch*, *n*[, *attr*])

Display a vertical line starting at `(y, x)` with length *n* consisting of the character *ch* with attributes *attr*.

16.9.3 Constants

The *curses* module defines the following data members:

curses.**ERR**

Some curses routines that return an integer, such as *getch()*, return *ERR* upon failure.

curses.**OK**

Some curses routines that return an integer, such as *napms()*, return *OK* upon success.

curses.**version**

curses.**__version__**

A bytes object representing the current version of the module.

curses.**ncurses_version**

A named tuple containing the three components of the ncurses library version: *major*, *minor*, and *patch*. All values are integers. The components can also be accessed by name, so `curses.ncurses_version[0]` is equivalent to `curses.ncurses_version.major` and so on.

Availability: if the ncurses library is used.

Added in version 3.8.

curses.**COLORS**

The maximum number of colors the terminal can support. It is defined only after the call to *start_color()*.

curses.**COLOR_PAIRS**

The maximum number of color pairs the terminal can support. It is defined only after the call to *start_color()*.

curses.**COLS**

The width of the screen, i.e., the number of columns. It is defined only after the call to *initscr()*. Updated by *update_lines_cols()*, *resizeterm()* and *resize_term()*.

curses.**LINES**

The height of the screen, i.e., the number of lines. It is defined only after the call to *initscr()*. Updated by *update_lines_cols()*, *resizeterm()* and *resize_term()*.

Some constants are available to specify character cell attributes. The exact constants available are system dependent.

Attribute	Meaning
curses.**A_ALTCHARSET**	Alternate character set mode
curses.**A_BLINK**	Blink mode
curses.**A_BOLD**	Bold mode
curses.**A_DIM**	Dim mode
curses.**A_INVIS**	Invisible or blank mode
curses.**A_ITALIC**	Italic mode
curses.**A_NORMAL**	Normal attribute
curses.**A_PROTECT**	Protected mode
curses.**A_REVERSE**	Reverse background and foreground colors
curses.**A_STANDOUT**	Standout mode
curses.**A_UNDERLINE**	Underline mode
curses.**A_HORIZONTAL**	Horizontal highlight
curses.**A_LEFT**	Left highlight
curses.**A_LOW**	Low highlight
curses.**A_RIGHT**	Right highlight
curses.**A_TOP**	Top highlight
curses.**A_VERTICAL**	Vertical highlight

Added in version 3.7: A_ITALIC was added.

Several constants are available to extract corresponding attributes returned by some methods.

Bit-mask	Meaning
curses.**A_ATTRIBUTES**	Bit-mask to extract attributes
curses.**A_CHARTEXT**	Bit-mask to extract a character
curses.**A_COLOR**	Bit-mask to extract color-pair field information

Keys are referred to by integer constants with names starting with KEY_. The exact keycaps available are system dependent.

Key constant	Key
curses.**KEY_MIN**	Minimum key value
curses.**KEY_BREAK**	Break key (unreliable)
curses.**KEY_DOWN**	Down-arrow
curses.**KEY_UP**	Up-arrow
curses.**KEY_LEFT**	Left-arrow
curses.**KEY_RIGHT**	Right-arrow
curses.**KEY_HOME**	Home key (upward+left arrow)
curses.**KEY_BACKSPACE**	Backspace (unreliable)
curses.**KEY_F0**	Function keys. Up to 64 function keys are supported.
curses.**KEY_Fn**	Value of function key *n*

continues on next page

Table 1 – continued from previous page

Key constant	Key
curses.**KEY_DL**	Delete line
curses.**KEY_IL**	Insert line
curses.**KEY_DC**	Delete character
curses.**KEY_IC**	Insert char or enter insert mode
curses.**KEY_EIC**	Exit insert char mode
curses.**KEY_CLEAR**	Clear screen
curses.**KEY_EOS**	Clear to end of screen
curses.**KEY_EOL**	Clear to end of line
curses.**KEY_SF**	Scroll 1 line forward
curses.**KEY_SR**	Scroll 1 line backward (reverse)
curses.**KEY_NPAGE**	Next page
curses.**KEY_PPAGE**	Previous page
curses.**KEY_STAB**	Set tab
curses.**KEY_CTAB**	Clear tab
curses.**KEY_CATAB**	Clear all tabs

continues on next page

Table 1 – continued from previous page

Key constant	Key
curses.**KEY_ENTER**	Enter or send (unreliable)
curses.**KEY_SRESET**	Soft (partial) reset (unreliable)
curses.**KEY_RESET**	Reset or hard reset (unreliable)
curses.**KEY_PRINT**	Print
curses.**KEY_LL**	Home down or bottom (lower left)
curses.**KEY_A1**	Upper left of keypad
curses.**KEY_A3**	Upper right of keypad
curses.**KEY_B2**	Center of keypad
curses.**KEY_C1**	Lower left of keypad
curses.**KEY_C3**	Lower right of keypad
curses.**KEY_BTAB**	Back tab
curses.**KEY_BEG**	Beg (beginning)
curses.**KEY_CANCEL**	Cancel
curses.**KEY_CLOSE**	Close
curses.**KEY_COMMAND**	Cmd (command)

continues on next page

Table 1 – continued from previous page

Key constant	Key
curses.**KEY_COPY**	Copy
curses.**KEY_CREATE**	Create
curses.**KEY_END**	End
curses.**KEY_EXIT**	Exit
curses.**KEY_FIND**	Find
curses.**KEY_HELP**	Help
curses.**KEY_MARK**	Mark
curses.**KEY_MESSAGE**	Message
curses.**KEY_MOVE**	Move
curses.**KEY_NEXT**	Next
curses.**KEY_OPEN**	Open
curses.**KEY_OPTIONS**	Options
curses.**KEY_PREVIOUS**	Prev (previous)
curses.**KEY_REDO**	Redo
curses.**KEY_REFERENCE**	Ref (reference)

continues on next page

Table 1 – continued from previous page

Key constant	Key
curses.**KEY_REFRESH**	Refresh
curses.**KEY_REPLACE**	Replace
curses.**KEY_RESTART**	Restart
curses.**KEY_RESUME**	Resume
curses.**KEY_SAVE**	Save
curses.**KEY_SBEG**	Shifted Beg (beginning)
curses.**KEY_SCANCEL**	Shifted Cancel
curses.**KEY_SCOMMAND**	Shifted Command
curses.**KEY_SCOPY**	Shifted Copy
curses.**KEY_SCREATE**	Shifted Create
curses.**KEY_SDC**	Shifted Delete char
curses.**KEY_SDL**	Shifted Delete line
curses.**KEY_SELECT**	Select
curses.**KEY_SEND**	Shifted End
curses.**KEY_SEOL**	Shifted Clear line

continues on next page

Table 1 – continued from previous page

Key constant	Key
curses.**KEY_SEXIT**	Shifted Exit
curses.**KEY_SFIND**	Shifted Find
curses.**KEY_SHELP**	Shifted Help
curses.**KEY_SHOME**	Shifted Home
curses.**KEY_SIC**	Shifted Input
curses.**KEY_SLEFT**	Shifted Left arrow
curses.**KEY_SMESSAGE**	Shifted Message
curses.**KEY_SMOVE**	Shifted Move
curses.**KEY_SNEXT**	Shifted Next
curses.**KEY_SOPTIONS**	Shifted Options
curses.**KEY_SPREVIOUS**	Shifted Prev
curses.**KEY_SPRINT**	Shifted Print
curses.**KEY_SREDO**	Shifted Redo
curses.**KEY_SREPLACE**	Shifted Replace
curses.**KEY_SRIGHT**	Shifted Right arrow

continues on next page

Table 1 – continued from previous page

Key constant	Key
curses.**KEY_SRSUME**	Shifted Resume
curses.**KEY_SSAVE**	Shifted Save
curses.**KEY_SSUSPEND**	Shifted Suspend
curses.**KEY_SUNDO**	Shifted Undo
curses.**KEY_SUSPEND**	Suspend
curses.**KEY_UNDO**	Undo
curses.**KEY_MOUSE**	Mouse event has occurred
curses.**KEY_RESIZE**	Terminal resize event
curses.**KEY_MAX**	Maximum key value

On VT100s and their software emulations, such as X terminal emulators, there are normally at least four function keys (*KEY_F1*, *KEY_F2*, *KEY_F3*, *KEY_F4*) available, and the arrow keys mapped to *KEY_UP*, *KEY_DOWN*, *KEY_LEFT* and *KEY_RIGHT* in the obvious way. If your machine has a PC keyboard, it is safe to expect arrow keys and twelve function keys (older PC keyboards may have only ten function keys); also, the following keypad mappings are standard:

Keycap	Constant
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_PPAGE
Page Down	KEY_NPAGE

The following table lists characters from the alternate character set. These are inherited from the VT100 terminal, and will generally be available on software emulations such as X terminals. When there is no graphic available, curses falls back on a crude printable ASCII approximation.

Note

These are available only after *initscr()* has been called.

ACS code	Meaning
curses.**ACS_BBSS**	alternate name for upper right corner
curses.**ACS_BLOCK**	solid square block
curses.**ACS_BOARD**	board of squares
curses.**ACS_BSBS**	alternate name for horizontal line
curses.**ACS_BSSB**	alternate name for upper left corner
curses.**ACS_BSSS**	alternate name for top tee
curses.**ACS_BTEE**	bottom tee
curses.**ACS_BULLET**	bullet
curses.**ACS_CKBOARD**	checker board (stipple)
curses.**ACS_DARROW**	arrow pointing down
curses.**ACS_DEGREE**	degree symbol
curses.**ACS_DIAMOND**	diamond
curses.**ACS_GEQUAL**	greater-than-or-equal-to
curses.**ACS_HLINE**	horizontal line

continues on next page

Table 2 – continued from previous page

ACS code	Meaning
curses.**ACS_LANTERN**	lantern symbol
curses.**ACS_LARROW**	left arrow
curses.**ACS_LEQUAL**	less-than-or-equal-to
curses.**ACS_LLCORNER**	lower left-hand corner
curses.**ACS_LRCORNER**	lower right-hand corner
curses.**ACS_LTEE**	left tee
curses.**ACS_NEQUAL**	not-equal sign
curses.**ACS_PI**	letter pi
curses.**ACS_PLMINUS**	plus-or-minus sign
curses.**ACS_PLUS**	big plus sign
curses.**ACS_RARROW**	right arrow
curses.**ACS_RTEE**	right tee
curses.**ACS_S1**	scan line 1
curses.**ACS_S3**	scan line 3
curses.**ACS_S7**	scan line 7

continues on next page

Table 2 – continued from previous page

ACS code	Meaning
curses.**ACS_S9**	scan line 9
curses.**ACS_SBBS**	alternate name for lower right corner
curses.**ACS_SBSB**	alternate name for vertical line
curses.**ACS_SBSS**	alternate name for right tee
curses.**ACS_SSBB**	alternate name for lower left corner
curses.**ACS_SSBS**	alternate name for bottom tee
curses.**ACS_SSSB**	alternate name for left tee
curses.**ACS_SSSS**	alternate name for crossover or big plus
curses.**ACS_STERLING**	pound sterling
curses.**ACS_TTEE**	top tee
curses.**ACS_UARROW**	up arrow
curses.**ACS_ULCORNER**	upper left corner
curses.**ACS_URCORNER**	upper right corner
curses.**ACS_VLINE**	vertical line

The following table lists mouse button constants used by *getmouse()*:

Mouse button constant	Meaning
curses.**BUTTONn_PRESSED**	Mouse button *n* pressed
curses.**BUTTONn_RELEASED**	Mouse button *n* released
curses.**BUTTONn_CLICKED**	Mouse button *n* clicked
curses.**BUTTONn_DOUBLE_CLICKED**	Mouse button *n* double clicked
curses.**BUTTONn_TRIPLE_CLICKED**	Mouse button *n* triple clicked
curses.**BUTTON_SHIFT**	Shift was down during button state change
curses.**BUTTON_CTRL**	Control was down during button state change
curses.**BUTTON_ALT**	Control was down during button state change

Changed in version 3.10: The `BUTTON5_*` constants are now exposed if they are provided by the underlying curses library.

The following table lists the predefined colors:

Constant	Color
curses.**COLOR_BLACK**	Black
curses.**COLOR_BLUE**	Blue
curses.**COLOR_CYAN**	Cyan (light greenish blue)
curses.**COLOR_GREEN**	Green
curses.**COLOR_MAGENTA**	Magenta (purplish red)
curses.**COLOR_RED**	Red
curses.**COLOR_WHITE**	White
curses.**COLOR_YELLOW**	Yellow

16.10 curses.textpad — Text input widget for curses programs

The *curses.textpad* module provides a *Textbox* class that handles elementary text editing in a curses window, supporting a set of keybindings resembling those of Emacs (thus, also of Netscape Navigator, BBedit 6.x, FrameMaker, and many other programs). The module also provides a rectangle-drawing function useful for framing text boxes or for other purposes.

The module *curses.textpad* defines the following function:

curses.textpad.**rectangle**(*win*, *uly*, *ulx*, *lry*, *lrx*)

Draw a rectangle. The first argument must be a window object; the remaining arguments are coordinates relative to that window. The second and third arguments are the y and x coordinates of the upper left hand corner of the rectangle to be drawn; the fourth and fifth arguments are the y and x coordinates of the lower right hand corner. The rectangle will be drawn using VT100/IBM PC forms characters on terminals that make this possible (including xterm and most other software terminal emulators). Otherwise it will be drawn with ASCII dashes, vertical bars, and plus signs.

16.10.1 Textbox objects

You can instantiate a *Textbox* object as follows:

class curses.textpad.**Textbox**(*win*)

Return a textbox widget object. The *win* argument should be a curses *window* object in which the textbox is to be contained. The edit cursor of the textbox is initially located at the upper left hand corner of the containing window, with coordinates `(0, 0)`. The instance's *stripspaces* flag is initially on.

Textbox objects have the following methods:

edit([*validator*])

This is the entry point you will normally use. It accepts editing keystrokes until one of the termination keystrokes is entered. If *validator* is supplied, it must be a function. It will be called for each keystroke entered with the keystroke as a parameter; command dispatch is done on the result. This method returns the window contents as a string; whether blanks in the window are included is affected by the *stripspaces* attribute.

do_command(*ch*)

Process a single command keystroke. Here are the supported special keystrokes:

Keystroke	Action
`Control-A`	Go to left edge of window.
`Control-B`	Cursor left, wrapping to previous line if appropriate.
`Control-D`	Delete character under cursor.
`Control-E`	Go to right edge (stripspaces off) or end of line (stripspaces on).
`Control-F`	Cursor right, wrapping to next line when appropriate.
`Control-G`	Terminate, returning the window contents.
`Control-H`	Delete character backward.
`Control-J`	Terminate if the window is 1 line, otherwise insert newline.
`Control-K`	If line is blank, delete it, otherwise clear to end of line.
`Control-L`	Refresh screen.
`Control-N`	Cursor down; move down one line.
`Control-O`	Insert a blank line at cursor location.
`Control-P`	Cursor up; move up one line.

Move operations do nothing if the cursor is at an edge where the movement is not possible. The following synonyms are supported where possible:

Constant	Keystroke
KEY_LEFT	`Control-B`
KEY_RIGHT	`Control-F`
KEY_UP	`Control-P`
KEY_DOWN	`Control-N`
KEY_BACKSPACE	`Control-h`

All other keystrokes are treated as a command to insert the given character and move right (with line wrapping).

gather()

Return the window contents as a string; whether blanks in the window are included is affected by the *stripspaces* member.

stripspaces

This attribute is a flag which controls the interpretation of blanks in the window. When it is on, trailing blanks on each line are ignored; any cursor motion that would land the cursor on a trailing blank goes to the end of that line instead, and trailing blanks are stripped when the window contents are gathered.

16.11 `curses.ascii` — Utilities for ASCII characters

Source code: Lib/curses/ascii.py

The *curses.ascii* module supplies name constants for ASCII characters and functions to test membership in various ASCII character classes. The constants supplied are names for control characters as follows:

Name	Meaning
curses.ascii.**NUL**	
curses.ascii.**SOH**	Start of heading, console interrupt
curses.ascii.**STX**	Start of text
curses.ascii.**ETX**	End of text
curses.ascii.**EOT**	End of transmission
curses.ascii.**ENQ**	Enquiry, goes with *ACK* flow control
curses.ascii.**ACK**	Acknowledgement
curses.ascii.**BEL**	Bell
curses.ascii.**BS**	Backspace
curses.ascii.**TAB**	Tab
curses.ascii.**HT**	Alias for *TAB*: “Horizontal tab”

continues on next page

Table 3 – continued from previous page

Name	Meaning
curses.ascii.**LF**	Line feed
curses.ascii.**NL**	Alias for *LF*: “New line”
curses.ascii.**VT**	Vertical tab
curses.ascii.**FF**	Form feed
curses.ascii.**CR**	Carriage return
curses.ascii.**SO**	Shift-out, begin alternate character set
curses.ascii.**SI**	Shift-in, resume default character set
curses.ascii.**DLE**	Data-link escape
curses.ascii.**DC1**	XON, for flow control
curses.ascii.**DC2**	Device control 2, block-mode flow control
curses.ascii.**DC3**	XOFF, for flow control
curses.ascii.**DC4**	Device control 4
curses.ascii.**NAK**	Negative acknowledgement
curses.ascii.**SYN**	Synchronous idle
curses.ascii.**ETB**	End transmission block

continues on next page

Table 3 – continued from previous page

Name	Meaning
curses.ascii.**CAN**	Cancel
curses.ascii.**EM**	End of medium
curses.ascii.**SUB**	Substitute
curses.ascii.**ESC**	Escape
curses.ascii.**FS**	File separator
curses.ascii.**GS**	Group separator
curses.ascii.**RS**	Record separator, block-mode terminator
curses.ascii.**US**	Unit separator
curses.ascii.**SP**	Space
curses.ascii.**DEL**	Delete

Note that many of these have little practical significance in modern usage. The mnemonics derive from teleprinter conventions that predate digital computers.

The module supplies the following functions, patterned on those in the standard C library:

curses.ascii.**isalnum**(*c*)

Checks for an ASCII alphanumeric character; it is equivalent to `isalpha(c) or isdigit(c)`.

curses.ascii.**isalpha**(*c*)

Checks for an ASCII alphabetic character; it is equivalent to `isupper(c) or islower(c)`.

curses.ascii.**isascii**(*c*)

Checks for a character value that fits in the 7-bit ASCII set.

curses.ascii.**isblank**(*c*)

Checks for an ASCII whitespace character; space or horizontal tab.

curses.ascii.**iscntrl**(*c*)

Checks for an ASCII control character (in the range 0x00 to 0x1f or 0x7f).

curses.ascii.**isdigit**(*c*)

Checks for an ASCII decimal digit, `'0'` through `'9'`. This is equivalent to `c in string.digits`.

curses.ascii.**isgraph**(*c*)

Checks for ASCII any printable character except space.

curses.ascii.**islower**(*c*)

Checks for an ASCII lower-case character.

curses.ascii.**isprint**(*c*)

Checks for any ASCII printable character including space.

curses.ascii.**ispunct**(*c*)

Checks for any printable ASCII character which is not a space or an alphanumeric character.

curses.ascii.**isspace**(*c*)

Checks for ASCII white-space characters; space, line feed, carriage return, form feed, horizontal tab, vertical tab.

curses.ascii.**isupper**(*c*)

Checks for an ASCII uppercase letter.

curses.ascii.**isxdigit**(*c*)

Checks for an ASCII hexadecimal digit. This is equivalent to `c in string.hexdigits`.

curses.ascii.**isctrl**(*c*)

Checks for an ASCII control character (ordinal values 0 to 31).

curses.ascii.**ismeta**(*c*)

Checks for a non-ASCII character (ordinal values 0x80 and above).

These functions accept either integers or single-character strings; when the argument is a string, it is first converted using the built-in function *ord()*.

Note that all these functions check ordinal bit values derived from the character of the string you pass in; they do not actually know anything about the host machine's character encoding.

The following two functions take either a single-character string or integer byte value; they return a value of the same type.

curses.ascii.**ascii**(*c*)

Return the ASCII value corresponding to the low 7 bits of *c*.

curses.ascii.**ctrl**(*c*)

Return the control character corresponding to the given character (the character bit value is bitwise-anded with 0x1f).

curses.ascii.**alt**(*c*)

Return the 8-bit character corresponding to the given ASCII character (the character bit value is bitwise-ored with 0x80).

The following function takes either a single-character string or integer value; it returns a string.

curses.ascii.**unctrl**(*c*)

Return a string representation of the ASCII character *c*. If *c* is printable, this string is the character itself. If the character is a control character (0x00–0x1f) the string consists of a caret (`'^'`) followed by the corresponding uppercase letter. If the character is an ASCII delete (0x7f) the string is `'^?'`. If the character has its meta bit (0x80) set, the meta bit is stripped, the preceding rules applied, and `'!'` prepended to the result.

curses.ascii.**controlnames**

A 33-element string array that contains the ASCII mnemonics for the thirty-two ASCII control characters from 0 (NUL) to 0x1f (US), in order, plus the mnemonic `SP` for the space character.

16.12 `curses.panel` — A panel stack extension for curses

Panels are windows with the added feature of depth, so they can be stacked on top of each other, and only the visible portions of each window will be displayed. Panels can be added, moved up or down in the stack, and removed.

16.12.1 Functions

The module *curses.panel* defines the following functions:

curses.panel.**bottom_panel**()

Returns the bottom panel in the panel stack.

curses.panel.**new_panel**(*win*)

Returns a panel object, associating it with the given window *win*. Be aware that you need to keep the returned panel object referenced explicitly. If you don't, the panel object is garbage collected and removed from the panel stack.

curses.panel.**top_panel**()

Returns the top panel in the panel stack.

curses.panel.**update_panels**()

Updates the virtual screen after changes in the panel stack. This does not call *curses.doupdate()*, so you'll have to do this yourself.

16.12.2 Panel Objects

Panel objects, as returned by *new_panel()* above, are windows with a stacking order. There's always a window associated with a panel which determines the content, while the panel methods are responsible for the window's depth in the panel stack.

Panel objects have the following methods:

Panel.**above**()

Returns the panel above the current panel.

Panel.**below**()

Returns the panel below the current panel.

Panel.**bottom**()

Push the panel to the bottom of the stack.

Panel.**hidden**()

Returns `True` if the panel is hidden (not visible), `False` otherwise.

Panel.**hide**()

Hide the panel. This does not delete the object, it just makes the window on screen invisible.

Panel.**move**(*y*, *x*)

Move the panel to the screen coordinates `(y, x)`.

Panel.**replace**(*win*)

Change the window associated with the panel to the window *win*.

Panel.**set_userptr**(*obj*)

Set the panel's user pointer to *obj*. This is used to associate an arbitrary piece of data with the panel, and can be any Python object.

Panel.**show**()

Display the panel (which might have been hidden).

Panel.**top**()

Push panel to the top of the stack.

Panel.**userptr**()

Returns the user pointer for the panel. This might be any Python object.

Panel.**window**()

Returns the window object associated with the panel.

16.13 `platform` — Access to underlying platform's identifying data

Source code: Lib/platform.py

Note

Specific platforms listed alphabetically, with Linux included in the Unix section.

16.13.1 Cross Platform

platform.**architecture**(*executable=sys.executable*, *bits=''*, *linkage=''*)

Queries the given executable (defaults to the Python interpreter binary) for various architecture information.

Returns a tuple `(bits, linkage)` which contain information about the bit architecture and the linkage format used for the executable. Both values are returned as strings.

Values that cannot be determined are returned as given by the parameter presets. If bits is given as `''`, the `sizeof(pointer)` (or `sizeof(long)` on Python version < 1.5.2) is used as indicator for the supported pointer size.

The function relies on the system's `file` command to do the actual work. This is available on most if not all Unix platforms and some non-Unix platforms and then only if the executable points to the Python interpreter. Reasonable defaults are used when the above needs are not met.

Note

On macOS (and perhaps other platforms), executable files may be universal files containing multiple architectures.

To get at the "64-bitness" of the current interpreter, it is more reliable to query the *sys.maxsize* attribute:

```
is_64bits = sys.maxsize > 2**32
```

platform.**machine**()

Returns the machine type, e.g. `'AMD64'`. An empty string is returned if the value cannot be determined.

platform.**node**()

Returns the computer's network name (may not be fully qualified!). An empty string is returned if the value cannot be determined.

platform.**platform**(*aliased=False*, *terse=False*)

Returns a single string identifying the underlying platform with as much useful information as possible.

The output is intended to be *human readable* rather than machine parseable. It may look different on different platforms and this is intended.

If *aliased* is true, the function will use aliases for various platforms that report system names which differ from their common names, for example SunOS will be reported as Solaris. The *system_alias()* function is used to implement this.

Setting *terse* to true causes the function to return only the absolute minimum information needed to identify the platform.

Changed in version 3.8: On macOS, the function now uses *mac_ver()*, if it returns a non-empty release string, to get the macOS version rather than the darwin version.

platform.**processor**()

Returns the (real) processor name, e.g. `'amdk6'`.

An empty string is returned if the value cannot be determined. Note that many platforms do not provide this information or simply return the same value as for *machine()*. NetBSD does this.

platform.**python_build**()

Returns a tuple `(buildno, builddate)` stating the Python build number and date as strings.

platform.**python_compiler**()

Returns a string identifying the compiler used for compiling Python.

platform.**python_branch**()

Returns a string identifying the Python implementation SCM branch.

platform.**python_implementation**()

Returns a string identifying the Python implementation. Possible return values are: 'CPython', 'IronPython', 'Jython', 'PyPy'.

platform.**python_revision**()

Returns a string identifying the Python implementation SCM revision.

platform.**python_version**()

Returns the Python version as string `'major.minor.patchlevel'`.

Note that unlike the Python `sys.version`, the returned value will always include the patchlevel (it defaults to 0).

platform.**python_version_tuple**()

Returns the Python version as tuple `(major, minor, patchlevel)` of strings.

Note that unlike the Python `sys.version`, the returned value will always include the patchlevel (it defaults to `'0'`).

platform.**release**()

Returns the system's release, e.g. `'2.2.0'` or `'NT'`. An empty string is returned if the value cannot be determined.

platform.**system**()

Returns the system/OS name, such as `'Linux'`, `'Darwin'`, `'Java'`, `'Windows'`. An empty string is returned if the value cannot be determined.

On iOS and Android, this returns the user-facing OS name (i.e, `'iOS`, `'iPadOS'` or `'Android'`). To obtain the kernel name (`'Darwin'` or `'Linux'`), use *os.uname()*.

platform.**system_alias**(*system*, *release*, *version*)

Returns `(system, release, version)` aliased to common marketing names used for some systems. It also does some reordering of the information in some cases where it would otherwise cause confusion.

platform.**version**()

Returns the system's release version, e.g. `'#3 on degas'`. An empty string is returned if the value cannot be determined.

On iOS and Android, this is the user-facing OS version. To obtain the Darwin or Linux kernel version, use *os.uname()*.

platform.**uname**()

Fairly portable uname interface. Returns a *namedtuple()* containing six attributes: *system*, *node*, *release*, *version*, *machine*, and *processor*.

processor is resolved late, on demand.

Note: the first two attribute names differ from the names presented by *os.uname()*, where they are named `sysname` and `nodename`.

Entries which cannot be determined are set to `''`.

Changed in version 3.3: Result changed from a tuple to a *namedtuple()*.

Changed in version 3.9: *processor* is resolved late instead of immediately.

16.13.2 Java Platform

platform.**java_ver**(*release=''*, *vendor=''*, *vminfo=('', '', '')*, *osinfo=('', '', '')*)

Version interface for Jython.

Returns a tuple `(release, vendor, vminfo, osinfo)` with *vminfo* being a tuple `(vm_name, vm_release, vm_vendor)` and *osinfo* being a tuple `(os_name, os_version, os_arch)`. Values which cannot be determined are set to the defaults given as parameters (which all default to `''`).

Deprecated since version 3.13, will be removed in version 3.15: It was largely untested, had a confusing API, and was only useful for Jython support.

16.13.3 Windows Platform

platform.**win32_ver**(*release=''*, *version=''*, *csd=''*, *ptype=''*)

Get additional version information from the Windows Registry and return a tuple `(release, version, csd, ptype)` referring to OS release, version number, CSD level (service pack) and OS type (multi/single processor). Values which cannot be determined are set to the defaults given as parameters (which all default to an empty string).

As a hint: *ptype* is `'Uniprocessor Free'` on single processor NT machines and `'Multiprocessor Free'` on multi processor machines. The `'Free'` refers to the OS version being free of debugging code. It could also state `'Checked'` which means the OS version uses debugging code, i.e. code that checks arguments, ranges, etc.

platform.**win32_edition**()

Returns a string representing the current Windows edition, or `None` if the value cannot be determined. Possible values include but are not limited to `'Enterprise'`, `'IoTUAP'`, `'ServerStandard'`, and `'nanoserver'`.

Added in version 3.8.

platform.**win32_is_iot**()

Return `True` if the Windows edition returned by *win32_edition()* is recognized as an IoT edition.

Added in version 3.8.

16.13.4 macOS Platform

platform.**mac_ver**(*release=''*, *versioninfo=('', '', '')*, *machine=''*)

Get macOS version information and return it as tuple `(release, versioninfo, machine)` with *versioninfo* being a tuple `(version, dev_stage, non_release_version)`.

Entries which cannot be determined are set to `''`. All tuple entries are strings.

16.13.5 iOS Platform

platform.**ios_ver**(*system=''*, *release=''*, *model=''*, *is_simulator=False*)

Get iOS version information and return it as a *namedtuple()* with the following attributes:

- `system` is the OS name; either `'iOS'` or `'iPadOS'`.
- `release` is the iOS version number as a string (e.g., `'17.2'`).
- `model` is the device model identifier; this will be a string like `'iPhone13,2'` for a physical device, or `'iPhone'` on a simulator.
- `is_simulator` is a boolean describing if the app is running on a simulator or a physical device.

Entries which cannot be determined are set to the defaults given as parameters.

16.13.6 Unix Platforms

platform.**libc_ver**(*executable=sys.executable*, *lib=''*, *version=''*, *chunksize=16384*)

Tries to determine the libc version against which the file executable (defaults to the Python interpreter) is linked. Returns a tuple of strings `(lib, version)` which default to the given parameters in case the lookup fails.

Note that this function has intimate knowledge of how different libc versions add symbols to the executable is probably only usable for executables compiled using **gcc**.

The file is read and scanned in chunks of *chunksize* bytes.

16.13.7 Linux Platforms

platform.**freedesktop_os_release**()

Get operating system identification from `os-release` file and return it as a dict. The `os-release` file is a freedesktop.org standard and is available in most Linux distributions. A noticeable exception is Android and Android-based distributions.

Raises *OSError* or subclass when neither `/etc/os-release` nor `/usr/lib/os-release` can be read.

On success, the function returns a dictionary where keys and values are strings. Values have their special characters like `"` and `$` unquoted. The fields `NAME`, `ID`, and `PRETTY_NAME` are always defined according to the standard. All other fields are optional. Vendors may include additional fields.

Note that fields like `NAME`, `VERSION`, and `VARIANT` are strings suitable for presentation to users. Programs should use fields like `ID`, `ID_LIKE`, `VERSION_ID`, or `VARIANT_ID` to identify Linux distributions.

Example:

```
def get_like_distro():
    info = platform.freedesktop_os_release()
    ids = [info["ID"]]
    if "ID_LIKE" in info:
        # ids are space separated and ordered by precedence
        ids.extend(info["ID_LIKE"].split())
    return ids
```

Added in version 3.10.

16.13.8 Android Platform

platform.**android_ver**(*release=''*, *api_level=0*, *manufacturer=''*, *model=''*, *device=''*, *is_emulator=False*)

Get Android device information. Returns a *namedtuple()* with the following attributes. Values which cannot be determined are set to the defaults given as parameters.

- `release` - Android version, as a string (e.g. `"14"`).
- `api_level` - API level of the running device, as an integer (e.g. `34` for Android 14). To get the API level which Python was built against, see *sys.getandroidapilevel()*.
- `manufacturer` - Manufacturer name.
- `model` - Model name – typically the marketing name or model number.
- `device` - Device name – typically the model number or a codename.
- `is_emulator` - `True` if the device is an emulator; `False` if it's a physical device.

Google maintains a list of known model and device names.

Added in version 3.13.

16.14 errno — Standard errno system symbols

This module makes available standard `errno` system symbols. The value of each symbol is the corresponding integer value. The names and descriptions are borrowed from `linux/include/errno.h`, which should be all-inclusive.

`errno.`**`errorcode`**

Dictionary providing a mapping from the errno value to the string name in the underlying system. For instance, `errno.errorcode[errno.EPERM]` maps to `'EPERM'`.

To translate a numeric error code to an error message, use *os.strerror()*.

Of the following list, symbols that are not used on the current platform are not defined by the module. The specific list of defined symbols is available as `errno.errorcode.keys()`. Symbols available can include:

`errno.`**`EPERM`**

Operation not permitted. This error is mapped to the exception *PermissionError*.

`errno.`**`ENOENT`**

No such file or directory. This error is mapped to the exception *FileNotFoundError*.

`errno.`**`ESRCH`**

No such process. This error is mapped to the exception *ProcessLookupError*.

`errno.`**`EINTR`**

Interrupted system call. This error is mapped to the exception *InterruptedError*.

`errno.`**`EIO`**

I/O error

`errno.`**`ENXIO`**

No such device or address

`errno.`**`E2BIG`**

Arg list too long

`errno.`**`ENOEXEC`**

Exec format error

`errno.`**`EBADF`**

Bad file number

`errno.`**`ECHILD`**

No child processes. This error is mapped to the exception *ChildProcessError*.

`errno.`**`EAGAIN`**

Try again. This error is mapped to the exception *BlockingIOError*.

`errno.`**`ENOMEM`**

Out of memory

errno.**EACCES**
Permission denied. This error is mapped to the exception *PermissionError*.

errno.**EFAULT**
Bad address

errno.**ENOTBLK**
Block device required

errno.**EBUSY**
Device or resource busy

errno.**EEXIST**
File exists. This error is mapped to the exception *FileExistsError*.

errno.**EXDEV**
Cross-device link

errno.**ENODEV**
No such device

errno.**ENOTDIR**
Not a directory. This error is mapped to the exception *NotADirectoryError*.

errno.**EISDIR**
Is a directory. This error is mapped to the exception *IsADirectoryError*.

errno.**EINVAL**
Invalid argument

errno.**ENFILE**
File table overflow

errno.**EMFILE**
Too many open files

errno.**ENOTTY**
Not a typewriter

errno.**ETXTBSY**
Text file busy

errno.**EFBIG**
File too large

errno.**ENOSPC**
No space left on device

errno.**ESPIPE**
Illegal seek

errno.**EROFS**
Read-only file system

errno.**EMLINK**
Too many links

errno.**EPIPE**
Broken pipe. This error is mapped to the exception *BrokenPipeError*.

errno.**EDOM**
Math argument out of domain of func

errno.**ERANGE**
Math result not representable

errno.**EDEADLK**
Resource deadlock would occur

errno.**ENAMETOOLONG**
File name too long

errno.**ENOLCK**
No record locks available

errno.**ENOSYS**
Function not implemented

errno.**ENOTEMPTY**
Directory not empty

errno.**ELOOP**
Too many symbolic links encountered

errno.**EWOULDBLOCK**
Operation would block. This error is mapped to the exception *BlockingIOError*.

errno.**ENOMSG**
No message of desired type

errno.**EIDRM**
Identifier removed

errno.**ECHRNG**
Channel number out of range

errno.**EL2NSYNC**
Level 2 not synchronized

errno.**EL3HLT**
Level 3 halted

errno.**EL3RST**
Level 3 reset

errno.**ELNRNG**
Link number out of range

errno.**EUNATCH**
Protocol driver not attached

errno.**ENOCSI**
No CSI structure available

errno.**EL2HLT**
Level 2 halted

errno.**EBADE**
Invalid exchange

errno.**EBADR**
Invalid request descriptor

errno.**EXFULL**
Exchange full

errno.**ENOANO**
No anode

errno.**EBADRQC**
Invalid request code

errno.**EBADSLT**
Invalid slot

errno.**EDEADLOCK**
File locking deadlock error

errno.**EBFONT**
Bad font file format

errno.**ENOSTR**
Device not a stream

errno.**ENODATA**
No data available

errno.**ETIME**
Timer expired

errno.**ENOSR**
Out of streams resources

errno.**ENONET**
Machine is not on the network

errno.**ENOPKG**
Package not installed

errno.**EREMOTE**
Object is remote

errno.**ENOLINK**
Link has been severed

errno.**EADV**
Advertise error

errno.**ESRMNT**
Srmount error

errno.**ECOMM**
Communication error on send

errno.**EPROTO**
Protocol error

errno.**EMULTIHOP**
Multihop attempted

errno.**EDOTDOT**
RFS specific error

errno.**EBADMSG**
Not a data message

errno.**EOVERFLOW**
Value too large for defined data type

errno.**ENOTUNIQ**
Name not unique on network

errno.**EBADFD**
File descriptor in bad state

errno.**EREMCHG**
Remote address changed

errno.**ELIBACC**
Can not access a needed shared library

errno.**ELIBBAD**
Accessing a corrupted shared library

errno.**ELIBSCN**
.lib section in a.out corrupted

errno.**ELIBMAX**
Attempting to link in too many shared libraries

errno.**ELIBEXEC**
Cannot exec a shared library directly

errno.**EILSEQ**
Illegal byte sequence

errno.**ERESTART**
Interrupted system call should be restarted

errno.**ESTRPIPE**
Streams pipe error

errno.**EUSERS**
Too many users

errno.**ENOTSOCK**
Socket operation on non-socket

errno.**EDESTADDRREQ**
Destination address required

errno.**EMSGSIZE**
Message too long

errno.**EPROTOTYPE**
Protocol wrong type for socket

errno.**ENOPROTOOPT**
Protocol not available

errno.**EPROTONOSUPPORT**
Protocol not supported

errno.**ESOCKTNOSUPPORT**
Socket type not supported

errno.**EOPNOTSUPP**
Operation not supported on transport endpoint

errno.**ENOTSUP**
Operation not supported

Added in version 3.2.

errno.**EPFNOSUPPORT**
Protocol family not supported

errno.**EAFNOSUPPORT**
Address family not supported by protocol

errno.**EADDRINUSE**
Address already in use

errno.**EADDRNOTAVAIL**
Cannot assign requested address

errno.**ENETDOWN**
Network is down

errno.**ENETUNREACH**
Network is unreachable

errno.**ENETRESET**
Network dropped connection because of reset

errno.**ECONNABORTED**
Software caused connection abort. This error is mapped to the exception *ConnectionAbortedError*.

errno.**ECONNRESET**
Connection reset by peer. This error is mapped to the exception *ConnectionResetError*.

errno.**ENOBUFS**
No buffer space available

errno.**EISCONN**
Transport endpoint is already connected

errno.**ENOTCONN**
Transport endpoint is not connected

errno.**ESHUTDOWN**
Cannot send after transport endpoint shutdown. This error is mapped to the exception *BrokenPipeError*.

errno.**ETOOMANYREFS**
Too many references: cannot splice

errno.**ETIMEDOUT**
Connection timed out. This error is mapped to the exception *TimeoutError*.

errno.**ECONNREFUSED**
Connection refused. This error is mapped to the exception *ConnectionRefusedError*.

errno.**EHOSTDOWN**
Host is down

errno.**EHOSTUNREACH**
No route to host

errno.**EALREADY**
Operation already in progress. This error is mapped to the exception *BlockingIOError*.

errno.**EINPROGRESS**
Operation now in progress. This error is mapped to the exception *BlockingIOError*.

errno.**ESTALE**
Stale NFS file handle

errno.**EUCLEAN**
Structure needs cleaning

errno.**ENOTNAM**
Not a XENIX named type file

errno.**ENAVAIL**
No XENIX semaphores available

errno.**EISNAM**
Is a named type file

errno.**EREMOTEIO**
Remote I/O error

errno.**EDQUOT**
Quota exceeded

errno.**EQFULL**
Interface output queue is full

Added in version 3.11.

errno.**ENOTCAPABLE**
Capabilities insufficient. This error is mapped to the exception *PermissionError*.

Availability: WASI, FreeBSD

Added in version 3.11.1.

errno.**ECANCELED**
Operation canceled

Added in version 3.2.

errno.**EOWNERDEAD**

Owner died

Added in version 3.2.

errno.**ENOTRECOVERABLE**

State not recoverable

Added in version 3.2.

16.15 ctypes — A foreign function library for Python

Source code: Lib/ctypes

---

ctypes is a foreign function library for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python.

16.15.1 ctypes tutorial

Note: The code samples in this tutorial use *doctest* to make sure that they actually work. Since some code samples behave differently under Linux, Windows, or macOS, they contain doctest directives in comments.

Note: Some code samples reference the ctypes *c_int* type. On platforms where `sizeof(long) == sizeof(int)` it is an alias to *c_long*. So, you should not be confused if *c_long* is printed if you would expect *c_int* — they are actually the same type.

Loading dynamic link libraries

ctypes exports the *cdll*, and on Windows *windll* and *oledll* objects, for loading dynamic link libraries.

You load libraries by accessing them as attributes of these objects. *cdll* loads libraries which export functions using the standard `cdecl` calling convention, while *windll* libraries call functions using the `stdcall` calling convention. *oledll* also uses the `stdcall` calling convention, and assumes the functions return a Windows `HRESULT` error code. The error code is used to automatically raise an *OSError* exception when the function call fails.

Changed in version 3.3: Windows errors used to raise *WindowsError*, which is now an alias of *OSError*.

Here are some examples for Windows. Note that `msvcrt` is the MS standard C library containing most standard C functions, and uses the cdecl calling convention:

```
>>> from ctypes import *
>>> print(windll.kernel32)
<WinDLL 'kernel32', handle ... at ...>
>>> print(cdll.msvcrt)
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

Windows appends the usual `.dll` file suffix automatically.

> **Note**
>
> Accessing the standard C library through `cdll.msvcrt` will use an outdated version of the library that may be incompatible with the one being used by Python. Where possible, use native Python functionality, or else import and use the `msvcrt` module.

On Linux, it is required to specify the filename *including* the extension to load a library, so attribute access can not be used to load libraries. Either the *LoadLibrary()* method of the dll loaders should be used, or you should load the library by creating an instance of CDLL by calling the constructor:

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

Accessing functions from loaded dlls

Functions are accessed as attributes of dll objects:

```
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.GetModuleHandleA)
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.MyOwnFunction)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
>>>
```

Note that win32 system dlls like `kernel32` and `user32` often export ANSI as well as UNICODE versions of a function. The UNICODE version is exported with a `W` appended to the name, while the ANSI version is exported with an `A` appended to the name. The win32 `GetModuleHandle` function, which returns a *module handle* for a given module name, has the following C prototype, and a macro is used to expose one of them as `GetModuleHandle` depending on whether UNICODE is defined or not:

```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

windll does not try to select one of them by magic, you must access the version you need by specifying `GetModuleHandleA` or `GetModuleHandleW` explicitly, and then call it with bytes or string objects respectively.

Sometimes, dlls export functions with names which aren't valid Python identifiers, like `"??2@YAPAXI@Z"`. In this case you have to use *getattr()* to retrieve the function:

```
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z")
<_FuncPtr object at 0x...>
>>>
```

On Windows, some dlls export functions not by name but by ordinal. These functions can be accessed by indexing the dll object with the ordinal number:

```
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>
```

Calling functions

You can call these functions like any other Python callable. This example uses the `rand()` function, which takes no arguments and returns a pseudo-random integer:

```
>>> print(libc.rand())
1804289383
```

On Windows, you can call the `GetModuleHandleA()` function, which returns a win32 module handle (passing `None` as single argument to call it with a `NULL` pointer):

```
>>> print(hex(windll.kernel32.GetModuleHandleA(None)))
0x1d000000
>>>
```

ValueError is raised when you call an `stdcall` function with the `cdecl` calling convention, or vice versa:

```
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>>

>>> windll.msvcrt.printf(b"spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

To find out the correct calling convention you have to look into the C header file or the documentation for the function you want to call.

On Windows, *ctypes* uses win32 structured exception handling to prevent crashes from general protection faults when functions are called with invalid argument values:

```
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: exception: access violation reading 0x00000020
>>>
```

There are, however, enough ways to crash Python with *ctypes*, so you should be careful anyway. The *faulthandler* module can be helpful in debugging crashes (e.g. from segmentation faults produced by erroneous C library calls).

None, integers, bytes objects and (unicode) strings are the only native Python objects that can directly be used as parameters in these function calls. None is passed as a C NULL pointer, bytes objects and strings are passed as pointer to the memory block that contains their data (char* or wchar_t*). Python integers are passed as the platform's default C int type, their value is masked to fit into the C type.

Before we move on calling functions with other parameter types, we have to learn more about *ctypes* data types.

Fundamental data types

ctypes defines a number of primitive C compatible data types:

ctypes type	C type	Python type
c_bool	_Bool	bool (1)
c_char	char	1-character bytes object
c_wchar	wchar_t	1-character string
c_byte	char	int
c_ubyte	unsigned char	int
c_short	short	int
c_ushort	unsigned short	int
c_int	int	int
c_uint	unsigned int	int
c_long	long	int
c_ulong	unsigned long	int
c_longlong	__int64 or long long	int
c_ulonglong	unsigned __int64 or unsigned long long	int
c_size_t	size_t	int
c_ssize_t	ssize_t or Py_ssize_t	int
c_time_t	time_t	int
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char* (NUL terminated)	bytes object or None
c_wchar_p	wchar_t* (NUL terminated)	string or None
c_void_p	void*	int or None

(1) The constructor accepts any object with a truth value.

All these types can be created by calling them with an optional initializer of the correct type and value:

```
>>> c_int()
c_long(0)
>>> c_wchar_p("Hello, World")
c_wchar_p(140018365411392)
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

Since these types are mutable, their value can also be changed afterwards:

```
>>> i = c_int(42)
>>> print(i)
c_long(42)
>>> print(i.value)
42
```

(continues on next page)

(continued from previous page)

```
>>> i.value = -99
>>> print(i.value)
-99
>>>
```

Assigning a new value to instances of the pointer types *c_char_p*, *c_wchar_p*, and *c_void_p* changes the *memory location* they point to, *not the contents* of the memory block (of course not, because Python bytes objects are immutable):

```
>>> s = "Hello, World"
>>> c_s = c_wchar_p(s)
>>> print(c_s)
c_wchar_p(139966785747344)
>>> print(c_s.value)
Hello World
>>> c_s.value = "Hi, there"
>>> print(c_s)              # the memory location has changed
c_wchar_p(139966783348904)
>>> print(c_s.value)
Hi, there
>>> print(s)                # first object is unchanged
Hello, World
>>>
```

You should be careful, however, not to pass them to functions expecting pointers to mutable memory. If you need mutable memory blocks, ctypes has a *create_string_buffer()* function which creates these in various ways. The current memory block contents can be accessed (or changed) with the `raw` property; if you want to access it as NUL terminated string, use the `value` property:

```
>>> from ctypes import *
>>> p = create_string_buffer(3)            # create a 3 byte buffer, initialized to␣
↪NUL bytes
>>> print(sizeof(p), repr(p.raw))
3 b'\x00\x00\x00'
>>> p = create_string_buffer(b"Hello")     # create a buffer containing a NUL␣
↪terminated string
>>> print(sizeof(p), repr(p.raw))
6 b'Hello\x00'
>>> print(repr(p.value))
b'Hello'
>>> p = create_string_buffer(b"Hello", 10) # create a 10 byte buffer
>>> print(sizeof(p), repr(p.raw))
10 b'Hello\x00\x00\x00\x00\x00'
>>> p.value = b"Hi"
>>> print(sizeof(p), repr(p.raw))
10 b'Hi\x00lo\x00\x00\x00\x00\x00'
>>>
```

The *create_string_buffer()* function replaces the old `c_buffer()` function (which is still available as an alias). To create a mutable memory block containing unicode characters of the C type `wchar_t`, use the *create_unicode_buffer()* function.

Calling functions, continued

Note that printf prints to the real standard output channel, *not* to `sys.stdout`, so these examples will only work at the console prompt, not from within *IDLE* or *PythonWin*:

```
>>> printf = libc.printf
>>> printf(b"Hello, %s\n", b"World!")
Hello, World!
14
>>> printf(b"Hello, %S\n", "World!")
Hello, World!
14
>>> printf(b"%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf(b"%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 2: TypeError: Don't know how to convert parameter 2
>>>
```

As has been mentioned before, all Python types except integers, strings, and bytes objects have to be wrapped in their corresponding `ctypes` type, so that they can be converted to the required C data type:

```
>>> printf(b"An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
>>>
```

Calling variadic functions

On a lot of platforms calling variadic functions through ctypes is exactly the same as calling functions with a fixed number of parameters. On some platforms, and in particular ARM64 for Apple Platforms, the calling convention for variadic functions is different than that for regular functions.

On those platforms it is required to specify the `argtypes` attribute for the regular, non-variadic, function arguments:

```
libc.printf.argtypes = [ctypes.c_char_p]
```

Because specifying the attribute does not inhibit portability it is advised to always specify `argtypes` for all variadic functions.

Calling functions with your own custom data types

You can also customize `ctypes` argument conversion to allow instances of your own classes be used as function arguments. `ctypes` looks for an `_as_parameter_` attribute and uses this as the function argument. The attribute must be an integer, string, bytes, a `ctypes` instance, or an object with an `_as_parameter_` attribute:

```
>>> class Bottles:
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf(b"%d bottles of beer\n", bottles)
42 bottles of beer
```

(continues on next page)

(continued from previous page)

```
19
>>>
```

If you don't want to store the instance's data in the `_as_parameter_` instance variable, you could define a *property* which makes the attribute available on request.

Specifying the required argument types (function prototypes)

It is possible to specify the required argument types of functions exported from DLLs by setting the *argtypes* attribute.

argtypes must be a sequence of C data types (the `printf()` function is probably not a good example here, because it takes a variable number and different types of parameters depending on the format string, on the other hand this is quite handy to experiment with this feature):

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf(b"String '%s', Int %d, Double %f\n", b"Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>
```

Specifying a format protects against incompatible argument types (just as a prototype for a C function), and tries to convert the arguments to valid types:

```
>>> printf(b"%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 2: TypeError: 'int' object cannot be interpreted as␣
↪ctypes.c_char_p
>>> printf(b"%s %d %f\n", b"X", 2, 3)
X 2 3.000000
13
>>>
```

If you have defined your own classes which you pass to function calls, you have to implement a *from_param()* class method for them to be able to use them in the *argtypes* sequence. The *from_param()* class method receives the Python object passed to the function call, it should do a typecheck or whatever is needed to make sure this object is acceptable, and then return the object itself, its `_as_parameter_` attribute, or whatever you want to pass as the C function argument in this case. Again, the result should be an integer, string, bytes, a *ctypes* instance, or an object with an `_as_parameter_` attribute.

Return types

By default functions are assumed to return the C `int` type. Other return types can be specified by setting the *restype* attribute of the function object.

The C prototype of `time()` is `time_t time(time_t *)`. Because `time_t` might be of a different type than the default return type `int`, you should specify the `restype` attribute:

```
>>> libc.time.restype = c_time_t
```

The argument types can be specified using *argtypes*:

```
>>> libc.time.argtypes = (POINTER(c_time_t),)
```

To call the function with a `NULL` pointer as first argument, use `None`:

```
>>> print(libc.time(None))
1150640792
```

Here is a more advanced example, it uses the `strchr()` function, which expects a string pointer and a char, and returns a pointer to a string:

```
>>> strchr = libc.strchr
>>> strchr(b"abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p    # c_char_p is a pointer to a string
>>> strchr(b"abcdef", ord("d"))
b'def'
>>> print(strchr(b"abcdef", ord("x")))
None
>>>
```

If you want to avoid the *ord("x")* calls above, you can set the *argtypes* attribute, and the second argument will be converted from a single character Python bytes object into a C char:

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr(b"abcdef", b"d")
b'def'
>>> strchr(b"abcdef", b"def")
Traceback (most recent call last):
ctypes.ArgumentError: argument 2: TypeError: one character bytes, bytearray or␣
↪integer expected
>>> print(strchr(b"abcdef", b"x"))
None
>>> strchr(b"abcdef", b"d")
b'def'
>>>
```

You can also use a callable Python object (a function or a class for example) as the *restype* attribute, if the foreign function returns an integer. The callable will be called with the *integer* the C function returns, and the result of this call will be used as the result of your function call. This is useful to check for error return values and automatically raise an exception:

```
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
...
>>>
>>> GetModuleHandle.restype = ValidHandle
>>> GetModuleHandle(None)
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in ValidHandle
OSError: [Errno 126] The specified module could not be found.
>>>
```

`WinError` is a function which will call Windows `FormatMessage()` api to get the string representation of an error code, and *returns* an exception. `WinError` takes an optional error code parameter, if no one is used, it calls *Get-*

LastError() to retrieve it.

Please note that a much more powerful error checking mechanism is available through the *errcheck* attribute; see the reference manual for details.

Passing pointers (or: passing parameters by reference)

Sometimes a C api function expects a *pointer* to a data type as parameter, probably to write into the corresponding location, or if the data is too large to be passed by value. This is also known as *passing parameters by reference*.

ctypes exports the *byref()* function which is used to pass parameters by reference. The same effect can be achieved with the *pointer()* function, although *pointer()* does a lot more work since it constructs a real pointer object, so it is faster to use *byref()* if you don't need the pointer object in Python itself:

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer(b'\000' * 32)
>>> print(i.value, f.value, repr(s.value))
0 0.0 b''
>>> libc.sscanf(b"1 3.14 Hello", b"%d %f %s",
...             byref(i), byref(f), s)
3
>>> print(i.value, f.value, repr(s.value))
1 3.1400001049 b'Hello'
>>>
```

Structures and unions

Structures and unions must derive from the *Structure* and *Union* base classes which are defined in the *ctypes* module. Each subclass must define a *_fields_* attribute. _fields_ must be a list of *2-tuples*, containing a *field name* and a *field type*.

The field type must be a *ctypes* type like *c_int*, or any other derived *ctypes* type: structure, union, array, pointer.

Here is a simple example of a POINT structure, which contains two integers named *x* and *y*, and also shows how to initialize a structure in the constructor:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
>>> print(point.x, point.y)
10 20
>>> point = POINT(y=5)
>>> print(point.x, point.y)
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: too many initializers
>>>
```

You can, however, build much more complicated structures. A structure can itself contain other structures by using a structure as a field type.

Here is a RECT structure which contains two POINTs named *upperleft* and *lowerright*:

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
...
>>> rc = RECT(point)
>>> print(rc.upperleft.x, rc.upperleft.y)
0 5
>>> print(rc.lowerright.x, rc.lowerright.y)
0 0
>>>
```

Nested structures can also be initialized in the constructor in several ways:

```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

Field *descriptor*s can be retrieved from the *class*, they are useful for debugging because they can provide useful information:

```
>>> print(POINT.x)
<Field type=c_long, ofs=0, size=4>
>>> print(POINT.y)
<Field type=c_long, ofs=4, size=4>
>>>
```

Warning

ctypes does not support passing unions or structures with bit-fields to functions by value. While this may work on 32-bit x86, it's not guaranteed by the library to work in the general case. Unions and structures with bit-fields should always be passed to functions by pointer.

Structure/union alignment and byte order

By default, Structure and Union fields are aligned in the same way the C compiler does it. It is possible to override this behavior by specifying a *_pack_* class attribute in the subclass definition. This must be set to a positive integer and specifies the maximum alignment for the fields. This is what `#pragma pack(n)` also does in MSVC. It is also possible to set a minimum alignment for how the subclass itself is packed in the same way `#pragma align(n)` works in MSVC. This can be achieved by specifying a :*_align_* class attribute in the subclass definition.

ctypes uses the native byte order for Structures and Unions. To build structures with non-native byte order, you can use one of the *BigEndianStructure*, *LittleEndianStructure*, *BigEndianUnion*, and *LittleEndianUnion* base classes. These classes cannot contain pointer fields.

Bit fields in structures and unions

It is possible to create structures and unions containing bit fields. Bit fields are only possible for integer fields, the bit width is specified as the third item in the *_fields_* tuples:

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print(Int.first_16)
<Field type=c_long, ofs=0:0, bits=16>
>>> print(Int.second_16)
<Field type=c_long, ofs=0:16, bits=16>
>>>
```

Arrays

Arrays are sequences, containing a fixed number of instances of the same type.

The recommended way to create array types is by multiplying a data type with a positive integer:

```
TenPointsArrayType = POINT * 10
```

Here is an example of a somewhat artificial data type, a structure containing 4 POINTs among other stuff:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class MyStruct(Structure):
...     _fields_ = [("a", c_int),
...                 ("b", c_float),
...                 ("point_array", POINT * 4)]
>>>
>>> print(len(MyStruct().point_array))
4
>>>
```

Instances are created in the usual way, by calling the class:

```
arr = TenPointsArrayType()
for pt in arr:
    print(pt.x, pt.y)
```

The above code print a series of `0 0` lines, because the array contents is initialized to zeros.

Initializers of the correct type can also be specified:

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print(ii)
<c_long_Array_10 object at 0x...>
>>> for i in ii: print(i, end=" ")
...
1 2 3 4 5 6 7 8 9 10
>>>
```

Pointers

Pointer instances are created by calling the *pointer()* function on a *ctypes* type:

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

Pointer instances have a *contents* attribute which returns the object to which the pointer points, the `i` object above:

```
>>> pi.contents
c_long(42)
>>>
```

Note that *ctypes* does not have OOR (original object return), it constructs a new, equivalent object each time you retrieve an attribute:

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

Assigning another *c_int* instance to the pointer's contents attribute would cause the pointer to point to the memory location where this is stored:

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

Pointer instances can also be indexed with integers:

```
>>> pi[0]
99
>>>
```

Assigning to an integer index changes the pointed to value:

```
>>> print(i)
c_long(99)
>>> pi[0] = 22
>>> print(i)
c_long(22)
>>>
```

It is also possible to use indexes different from 0, but you must know what you're doing, just as in C: You can access or change arbitrary memory locations. Generally you only use this feature if you receive a pointer from a C function, and you *know* that the pointer actually points to an array instead of a single item.

Behind the scenes, the *pointer()* function does more than simply create pointer instances, it has to create pointer *types* first. This is done with the *POINTER()* function, which accepts any *ctypes* type, and returns a new type:

```
>>> PI = POINTER(c_int)
>>> PI
```

(continues on next page)

(continued from previous page)

```
<class 'ctypes.LP_c_long'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
>>>
```

Calling the pointer type without an argument creates a `NULL` pointer. `NULL` pointers have a `False` boolean value:

```
>>> null_ptr = POINTER(c_int)()
>>> print(bool(null_ptr))
False
>>>
```

ctypes checks for `NULL` when dereferencing pointers (but dereferencing invalid non-`NULL` pointers would crash Python):

```
>>> null_ptr[0]
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>

>>> null_ptr[0] = 1234
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>
```

Type conversions

Usually, ctypes does strict type checking. This means, if you have `POINTER(c_int)` in the *argtypes* list of a function or as the type of a member field in a structure definition, only instances of exactly the same type are accepted. There are some exceptions to this rule, where ctypes accepts other objects. For example, you can pass compatible array instances instead of pointer types. So, for `POINTER(c_int)`, ctypes accepts an array of c_int:

```
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values", POINTER(c_int))]
...
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print(bar.values[i])
...
1
2
3
>>>
```

In addition, if a function argument is explicitly declared to be a pointer type (such as `POINTER(c_int)`) in *argtypes*, an object of the pointed type (`c_int` in this case) can be passed to the function. ctypes will apply the required *byref()* conversion in this case automatically.

To set a POINTER type field to NULL, you can assign None:

```
>>> bar.values = None
>>>
```

Sometimes you have instances of incompatible types. In C, you can cast one type into another type. *ctypes* provides a *cast()* function which can be used in the same way. The Bar structure defined above accepts POINTER(c_int) pointers or *c_int* arrays for its values field, but not instances of other types:

```
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: incompatible types, c_byte_Array_4 instance instead of LP_c_long instance
>>>
```

For these cases, the *cast()* function is handy.

The *cast()* function can be used to cast a ctypes instance into a pointer to a different ctypes data type. *cast()* takes two parameters, a ctypes object that is or can be converted to a pointer of some kind, and a ctypes pointer type. It returns an instance of the second argument, which references the same memory block as the first argument:

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

So, *cast()* can be used to assign to the values field of Bar the structure:

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print(bar.values[0])
0
>>>
```

Incomplete Types

Incomplete Types are structures, unions or arrays whose members are not yet specified. In C, they are specified by forward declarations, which are defined later:

```
struct cell; /* forward declaration */

struct cell {
    char *name;
    struct cell *next;
};
```

The straightforward translation into ctypes code would be this, but it does not work:

```
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
>>>
```

because the new `class cell` is not available in the class statement itself. In *ctypes*, we can define the `cell` class and set the *_fields_* attribute later, after the class statement:

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                  ("next", POINTER(cell))]
>>>
```

Let's try it. We create two instances of `cell`, and let them point to each other, and finally follow the pointer chain a few times:

```
>>> c1 = cell()
>>> c1.name = b"foo"
>>> c2 = cell()
>>> c2.name = b"bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print(p.name, end=" ")
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
>>>
```

Callback functions

ctypes allows creating C callable function pointers from Python callables. These are sometimes called *callback functions*.

First, you must create a class for the callback function. The class knows the calling convention, the return type, and the number and types of arguments this function will receive.

The *CFUNCTYPE()* factory function creates types for callback functions using the `cdecl` calling convention. On Windows, the *WINFUNCTYPE()* factory function creates types for callback functions using the `stdcall` calling convention.

Both of these factory functions are called with the result type as first argument, and the callback functions expected argument types as the remaining arguments.

I will present an example here which uses the standard C library's `qsort()` function, that is used to sort items with the help of a callback function. `qsort()` will be used to sort an array of integers:

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

`qsort()` must be called with a pointer to the data to sort, the number of items in the data array, the size of one item, and a pointer to the comparison function, the callback. The callback will then be called with two pointers to items, and it must return a negative integer if the first item is smaller than the second, a zero if they are equal, and a positive integer otherwise.

So our callback function receives pointers to integers, and must return an integer. First we create the `type` for the callback function:

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

To get started, here is a simple callback that shows the values it gets passed:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

The result:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
>>>
```

Now we can actually compare the two items and return a useful result:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>>
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

As we can easily check, our array is sorted now:

```
>>> for i in ia: print(i, end=" ")
...
1 5 7 33 99
>>>
```

The function factories can be used as decorator factories, so we may as well write:

```
>>> @CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
... def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>> qsort(ia, len(ia), sizeof(c_int), py_cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
```

(continues on next page)

(continued from previous page)

```
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

Note

Make sure you keep references to *CFUNCTYPE()* objects as long as they are used from C code. *ctypes* doesn't, and if you don't, they may be garbage collected, crashing your program when a callback is made.

Also, note that if the callback function is called in a thread created outside of Python's control (e.g. by the foreign code that calls the callback), ctypes creates a new dummy Python thread on every invocation. This behavior is correct for most purposes, but it means that values stored with *threading.local* will *not* survive across different callbacks, even when those calls are made from the same C thread.

Accessing values exported from dlls

Some shared libraries not only export functions, they also export variables. An example in the Python library itself is the `Py_Version`, Python runtime version number encoded in a single constant integer.

ctypes can access values like this with the *in_dll()* class methods of the type. *pythonapi* is a predefined symbol giving access to the Python C api:

```
>>> version = ctypes.c_int.in_dll(ctypes.pythonapi, "Py_Version")
>>> print(hex(version.value))
0x30c00a0
```

An extended example which also demonstrates the use of pointers accesses the `PyImport_FrozenModules` pointer exported by Python.

Quoting the docs for that value:

> This pointer is initialized to point to an array of `_frozen` records, terminated by one whose members are all `NULL` or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

So manipulating this pointer could even prove useful. To restrict the example size, we show only how this table can be read with *ctypes*:

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int),
...                 ("get_code", POINTER(c_ubyte)),  # Function pointer
...                ]
...
>>>
```

We have defined the `_frozen` data type, so we can get the pointer to the table:

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "_PyImport_FrozenBootstrap")
>>>
```

Since `table` is a `pointer` to the array of `struct_frozen` records, we can iterate over it, but we just have to make sure that our loop terminates, because pointers have no size. Sooner or later it would probably crash with an access violation or whatever, so it's better to break out of the loop when we hit the `NULL` entry:

```
>>> for item in table:
...     if item.name is None:
...         break
...     print(item.name.decode("ascii"), item.size)
...
_frozen_importlib 31764
_frozen_importlib_external 41499
zipimport 12345
>>>
```

The fact that standard Python has a frozen module and a frozen package (indicated by the negative `size` member) is not well known, it is only used for testing. Try it out with `import __hello__` for example.

Surprises

There are some edges in *ctypes* where you might expect something other than what actually happens.

Consider the following example:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
3 4 3 4
>>>
```

Hm. We certainly expected the last statement to print `3 4 1 2`. What happened? Here are the steps of the `rc.a, rc.b = rc.b, rc.a` line above:

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

Note that `temp0` and `temp1` are objects still using the internal buffer of the `rc` object above. So executing `rc.a = temp0` copies the buffer contents of `temp0` into `rc` 's buffer. This, in turn, changes the contents of `temp1`. So, the last assignment `rc.b = temp1`, doesn't have the expected effect.

Keep in mind that retrieving sub-objects from Structure, Unions, and Arrays doesn't *copy* the sub-object, instead it retrieves a wrapper object accessing the root-object's underlying buffer.

Another example that may behave differently from what one would expect is this:

```
>>> s = c_char_p()
>>> s.value = b"abc def ghi"
>>> s.value
b'abc def ghi'
>>> s.value is s.value
False
>>>
```

Note

Objects instantiated from *c_char_p* can only have their value set to bytes or integers.

Why is it printing `False`? ctypes instances are objects containing a memory block plus some *descriptor*s accessing the contents of the memory. Storing a Python object in the memory block does not store the object itself, instead the `contents` of the object is stored. Accessing the contents again constructs a new Python object each time!

Variable-sized data types

ctypes provides some support for variable-sized arrays and structures.

The *resize()* function can be used to resize the memory buffer of an existing ctypes object. The function takes the object as first argument, and the requested size in bytes as the second argument. The memory block cannot be made smaller than the natural memory block specified by the objects type, a *ValueError* is raised if this is tried:

```
>>> short_array = (c_short * 4)()
>>> print(sizeof(short_array))
8
>>> resize(short_array, 4)
Traceback (most recent call last):
    ...
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>>
```

This is nice and fine, but how would one access the additional elements contained in this array? Since the type still only knows about 4 elements, we get errors accessing other elements:

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
    ...
IndexError: invalid index
>>>
```

Another way to use variable-sized data types with *ctypes* is to use the dynamic nature of Python, and (re-)define the data type after the required size is already known, on a case by case basis.

16.15.2 ctypes reference

Finding shared libraries

When programming in a compiled language, shared libraries are accessed when compiling/linking a program, and when the program is run.

The purpose of the *find_library()* function is to locate a library in a way similar to what the compiler or runtime loader does (on platforms with several versions of a shared library the most recent should be loaded), while the ctypes library loaders act like when a program is run, and call the runtime loader directly.

The `ctypes.util` module provides a function which can help to determine the library to load.

`ctypes.util.`**`find_library`**(*name*)

Try to find a library and return a pathname. *name* is the library name without any prefix like *lib*, suffix like `.so`, `.dylib` or version number (this is the form used for the posix linker option `-l`). If no library can be found, returns `None`.

The exact functionality is system dependent.

On Linux, *find_library()* tries to run external programs (`/sbin/ldconfig`, `gcc`, `objdump` and `ld`) to find the library file. It returns the filename of the library file.

Changed in version 3.6: On Linux, the value of the environment variable `LD_LIBRARY_PATH` is used when searching for libraries, if a library cannot be found by any other means.

Here are some examples:

```
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
>>>
```

On macOS and Android, *find_library()* uses the system's standard naming schemes and paths to locate the library, and returns a full pathname if successful:

```
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
>>>
```

On Windows, *find_library()* searches along the system search path, and returns the full pathname, but since there is no predefined naming scheme a call like `find_library("c")` will fail and return `None`.

If wrapping a shared library with *ctypes*, it *may* be better to determine the shared library name at development time, and hardcode that into the wrapper module instead of using *find_library()* to locate the library at runtime.

Loading shared libraries

There are several ways to load shared libraries into the Python process. One way is to instantiate one of the following classes:

class ctypes.**CDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

Instances of this class represent loaded shared libraries. Functions in these libraries use the standard C calling convention, and are assumed to return `int`.

On Windows creating a *CDLL* instance may fail even if the DLL name exists. When a dependent DLL of the loaded DLL is not found, a *OSError* error is raised with the message *"[WinError 126] The specified module could not be found"*. This error message does not contain the name of the missing DLL because the Windows API does not return this information making this error hard to diagnose. To resolve this error and determine which DLL is not found, you need to find the list of dependent DLLs and determine which one is not found using Windows debugging and tracing tools.

Changed in version 3.12: The *name* parameter can now be a *path-like object*.

See also

Microsoft DUMPBIN tool – A tool to find DLL dependents.

class ctypes.**OleDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return the windows specific *HRESULT* code. *HRESULT* values contain information specifying whether the function call failed or succeeded, together with additional error code. If the return value signals a failure, an *OSError* is automatically raised.

Changed in version 3.3: *WindowsError* used to be raised, which is now an alias of *OSError*.

Changed in version 3.12: The *name* parameter can now be a *path-like object*.

class ctypes.**WinDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return `int` by default.

Changed in version 3.12: The *name* parameter can now be a *path-like object*.

The Python *global interpreter lock* is released before calling any function exported by these libraries, and reacquired afterwards.

class ctypes.**PyDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*)

Instances of this class behave like *CDLL* instances, except that the Python GIL is *not* released during the function call, and after the function execution the Python error flag is checked. If the error flag is set, a Python exception is raised.

Thus, this is only useful to call Python C api functions directly.

Changed in version 3.12: The *name* parameter can now be a *path-like object*.

All these classes can be instantiated by calling them with at least one argument, the pathname of the shared library. If you have an existing handle to an already loaded shared library, it can be passed as the `handle` named parameter, otherwise the underlying platform's `dlopen()` or `LoadLibrary()` function is used to load the library into the process, and to get a handle to it.

The *mode* parameter can be used to specify how the library is loaded. For details, consult the *dlopen(3)* manpage. On Windows, *mode* is ignored. On posix systems, RTLD_NOW is always added, and is not configurable.

The *use_errno* parameter, when set to true, enables a ctypes mechanism that allows accessing the system *errno* error number in a safe way. *ctypes* maintains a thread-local copy of the system's *errno* variable; if you call foreign functions created with `use_errno=True` then the *errno* value before the function call is swapped with the ctypes private copy, the same happens immediately after the function call.

The function *ctypes.get_errno()* returns the value of the ctypes private copy, and the function *ctypes.set_errno()* changes the ctypes private copy to a new value and returns the former value.

The *use_last_error* parameter, when set to true, enables the same mechanism for the Windows error code which is managed by the *GetLastError()* and `SetLastError()` Windows API functions; *ctypes.get_last_error()* and *ctypes.set_last_error()* are used to request and change the ctypes private copy of the windows error code.

The *winmode* parameter is used on Windows to specify how the library is loaded (since *mode* is ignored). It takes any value that is valid for the Win32 API `LoadLibraryEx` flags parameter. When omitted, the default is to use the flags that result in the most secure DLL load, which avoids issues such as DLL hijacking. Passing the full path to the DLL is the safest way to ensure the correct library and dependencies are loaded.

Changed in version 3.8: Added *winmode* parameter.

`ctypes.`**RTLD_GLOBAL**

Flag to use as *mode* parameter. On platforms where this flag is not available, it is defined as the integer zero.

`ctypes.`**RTLD_LOCAL**

Flag to use as *mode* parameter. On platforms where this is not available, it is the same as *RTLD_GLOBAL*.

`ctypes.`**DEFAULT_MODE**

The default mode which is used to load shared libraries. On OSX 10.3, this is *RTLD_GLOBAL*, otherwise it is the same as *RTLD_LOCAL*.

Instances of these classes have no public methods. Functions exported by the shared library can be accessed as attributes or by index. Please note that accessing the function through an attribute caches the result and therefore accessing it repeatedly returns the same object each time. On the other hand, accessing it through an index returns a new object each time:

```
>>> from ctypes import CDLL
>>> libc = CDLL("libc.so.6")  # On Linux
>>> libc.time == libc.time
True
>>> libc['time'] == libc['time']
False
```

The following public attributes are available, their name starts with an underscore to not clash with exported function names:

`PyDLL.`**_handle**

The system handle used to access the library.

`PyDLL.`**_name**

The name of the library passed in the constructor.

Shared libraries can also be loaded by using one of the prefabricated objects, which are instances of the *LibraryLoader* class, either by calling the *LoadLibrary()* method, or by retrieving the library as attribute of the loader instance.

class `ctypes.`**LibraryLoader**(*dlltype*)

Class which loads shared libraries. *dlltype* should be one of the *CDLL*, *PyDLL*, *WinDLL*, or *OleDLL* types.

`__getattr__()` has special behavior: It allows loading a shared library by accessing it as attribute of a library loader instance. The result is cached, so repeated attribute accesses return the same library each time.

LoadLibrary(*name*)

Load a shared library into the process and return it. This method always returns a new instance of the library.

These prefabricated library loaders are available:

ctypes.**cdll**

Creates *CDLL* instances.

ctypes.**windll**

Windows only: Creates *WinDLL* instances.

ctypes.**oledll**

Windows only: Creates *OleDLL* instances.

ctypes.**pydll**

Creates *PyDLL* instances.

For accessing the C Python api directly, a ready-to-use Python shared library object is available:

ctypes.**pythonapi**

An instance of *PyDLL* that exposes Python C API functions as attributes. Note that all these functions are assumed to return C `int`, which is of course not always the truth, so you have to assign the correct `restype` attribute to use these functions.

Loading a library through any of these objects raises an *auditing event* `ctypes.dlopen` with string argument `name`, the name used to load the library.

Accessing a function on a loaded library raises an auditing event `ctypes.dlsym` with arguments `library` (the library object) and `name` (the symbol's name as a string or integer).

In cases when only the library handle is available rather than the object, accessing a function raises an auditing event `ctypes.dlsym/handle` with arguments `handle` (the raw library handle) and `name`.

Foreign functions

As explained in the previous section, foreign functions can be accessed as attributes of loaded shared libraries. The function objects created in this way by default accept any number of arguments, accept any ctypes data instances as arguments, and return the default result type specified by the library loader. They are instances of a private class:

class ctypes.**_FuncPtr**

Base class for C callable foreign functions.

Instances of foreign functions are also C compatible data types; they represent C function pointers.

This behavior can be customized by assigning to special attributes of the foreign function object.

restype

Assign a ctypes type to specify the result type of the foreign function. Use `None` for `void`, a function not returning anything.

It is possible to assign a callable Python object that is not a ctypes type, in this case the function is assumed to return a C `int`, and the callable will be called with this integer, allowing further processing or error

checking. Using this is deprecated, for more flexible post processing or error checking use a ctypes data type as `restype` and assign a callable to the *errcheck* attribute.

argtypes

Assign a tuple of ctypes types to specify the argument types that the function accepts. Functions using the `stdcall` calling convention can only be called with the same number of arguments as the length of this tuple; functions using the C calling convention accept additional, unspecified arguments as well.

When a foreign function is called, each actual argument is passed to the *from_param()* class method of the items in the *argtypes* tuple, this method allows adapting the actual argument to an object that the foreign function accepts. For example, a *c_char_p* item in the *argtypes* tuple will convert a string passed as argument into a bytes object using ctypes conversion rules.

New: It is now possible to put items in argtypes which are not ctypes types, but each item must have a *from_param()* method which returns a value usable as argument (integer, string, ctypes instance). This allows defining adapters that can adapt custom objects as function parameters.

errcheck

Assign a Python function or another callable to this attribute. The callable will be called with three or more arguments:

callable(*result*, *func*, *arguments*)

result is what the foreign function returns, as specified by the `restype` attribute.

func is the foreign function object itself, this allows reusing the same callable object to check or post process the results of several functions.

arguments is a tuple containing the parameters originally passed to the function call, this allows specializing the behavior on the arguments used.

The object that this function returns will be returned from the foreign function call, but it can also check the result value and raise an exception if the foreign function call failed.

exception ctypes.**ArgumentError**

This exception is raised when a foreign function call cannot convert one of the passed arguments.

On Windows, when a foreign function call raises a system exception (for example, due to an access violation), it will be captured and replaced with a suitable Python exception. Further, an auditing event `ctypes.set_exception` with argument `code` will be raised, allowing an audit hook to replace the exception with its own.

Some ways to invoke foreign function calls may raise an auditing event `ctypes.call_function` with arguments `function pointer` and `arguments`.

Function prototypes

Foreign functions can also be created by instantiating function prototypes. Function prototypes are similar to function prototypes in C; they describe a function (return type, argument types, calling convention) without defining an implementation. The factory functions must be called with the desired result type and the argument types of the function, and can be used as decorator factories, and as such, be applied to functions through the `@wrapper` syntax. See *Callback functions* for examples.

ctypes.**CFUNCTYPE**(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)

The returned function prototype creates functions that use the standard C calling convention. The function will release the GIL during the call. If *use_errno* is set to true, the ctypes private copy of the system *errno* variable

is exchanged with the real `errno` value before and after the call; *use_last_error* does the same for the Windows error code.

`ctypes.`**`WINFUNCTYPE`**(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)

Windows only: The returned function prototype creates functions that use the `stdcall` calling convention. The function will release the GIL during the call. *use_errno* and *use_last_error* have the same meaning as above.

`ctypes.`**`PYFUNCTYPE`**(*restype*, **argtypes*)

The returned function prototype creates functions that use the Python calling convention. The function will *not* release the GIL during the call.

Function prototypes created by these factory functions can be instantiated in different ways, depending on the type and number of the parameters in the call:

`prototype`(*address*)

Returns a foreign function at the specified address which must be an integer.

`prototype`(*callable*)

Create a C callable function (a callback function) from a Python *callable*.

`prototype`(*func_spec*[, *paramflags*])

Returns a foreign function exported by a shared library. *func_spec* must be a 2-tuple `(name_or_ordinal, library)`. The first item is the name of the exported function as string, or the ordinal of the exported function as small integer. The second item is the shared library instance.

`prototype`(*vtbl_index*, *name*[, *paramflags*[, *iid*]])

Returns a foreign function that will call a COM method. *vtbl_index* is the index into the virtual function table, a small non-negative integer. *name* is name of the COM method. *iid* is an optional pointer to the interface identifier which is used in extended error reporting.

COM methods use a special calling convention: They require a pointer to the COM interface as first argument, in addition to those parameters that are specified in the `argtypes` tuple.

The optional *paramflags* parameter creates foreign function wrappers with much more functionality than the features described above.

paramflags must be a tuple of the same length as `argtypes`.

Each item in this tuple contains further information about a parameter, it must be a tuple containing one, two, or three items.

The first item is an integer containing a combination of direction flags for the parameter:

1
Specifies an input parameter to the function.

2
Output parameter. The foreign function fills in a value.

4
Input parameter which defaults to the integer zero.

The optional second item is the parameter name as string. If this is specified, the foreign function can be called with named parameters.

The optional third item is the default value for this parameter.

The following example demonstrates how to wrap the Windows `MessageBoxW` function so that it supports default parameters and named arguments. The C declaration from the windows header file is this:

```
WINUSERAPI int WINAPI
MessageBoxW(
    HWND hWnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType);
```

Here is the wrapping with *ctypes*:

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCWSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCWSTR, LPCWSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", "Hello from ctypes
↪"), (1, "flags", 0)
>>> MessageBox = prototype(("MessageBoxW", windll.user32), paramflags)
```

The `MessageBox` foreign function can now be called in these ways:

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
```

A second example demonstrates output parameters. The win32 `GetWindowRect` function retrieves the dimensions of a specified window by copying them into `RECT` structure that the caller has to supply. Here is the C declaration:

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
     HWND hWnd,
     LPRECT lpRect);
```

Here is the wrapping with *ctypes*:

```
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32), paramflags)
>>>
```

Functions with output parameters will automatically return the output parameter value if there is a single one, or a tuple containing the output parameter values when there are more than one, so the GetWindowRect function now returns a RECT instance, when called.

Output parameters can be combined with the *errcheck* protocol to do further output processing and error checking. The win32 `GetWindowRect` api function returns a `BOOL` to signal success or failure, so this function could do the error checking, and raises an exception when the api call failed:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

If the *errcheck* function returns the argument tuple it receives unchanged, *ctypes* continues the normal processing it does on the output parameters. If you want to return a tuple of window coordinates instead of a `RECT` instance, you can retrieve the fields in the function and return them instead, the normal processing will no longer take place:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

Utility functions

ctypes.**addressof**(*obj*)

Returns the address of the memory buffer as integer. *obj* must be an instance of a ctypes type.

Raises an *auditing event* `ctypes.addressof` with argument `obj`.

ctypes.**alignment**(*obj_or_type*)

Returns the alignment requirements of a ctypes type. *obj_or_type* must be a ctypes type or instance.

ctypes.**byref**(*obj*[, *offset*])

Returns a light-weight pointer to *obj*, which must be an instance of a ctypes type. *offset* defaults to zero, and must be an integer that will be added to the internal pointer value.

`byref(obj, offset)` corresponds to this C code:

```
(((char *)&obj) + offset)
```

The returned object can only be used as a foreign function call parameter. It behaves similar to `pointer(obj)`, but the construction is a lot faster.

ctypes.**cast**(*obj*, *type*)

This function is similar to the cast operator in C. It returns a new instance of *type* which points to the same memory block as *obj*. *type* must be a pointer type, and *obj* must be an object that can be interpreted as a pointer.

ctypes.**create_string_buffer**(*init_or_size*, *size=None*)

This function creates a mutable character buffer. The returned object is a ctypes array of *c_char*.

init_or_size must be an integer which specifies the size of the array, or a bytes object which will be used to initialize the array items.

If a bytes object is specified as first argument, the buffer is made one item larger than its length so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows specifying the size of the array if the length of the bytes should not be used.

Raises an *auditing event* `ctypes.create_string_buffer` with arguments `init`, `size`.

ctypes.**create_unicode_buffer**(*init_or_size*, *size=None*)

This function creates a mutable unicode character buffer. The returned object is a ctypes array of *c_wchar*.

init_or_size must be an integer which specifies the size of the array, or a string which will be used to initialize the array items.

If a string is specified as first argument, the buffer is made one item larger than the length of the string so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows specifying the size of the array if the length of the string should not be used.

Raises an *auditing event* `ctypes.create_unicode_buffer` with arguments `init`, `size`.

ctypes.**DllCanUnloadNow**()

Windows only: This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllCanUnloadNow function that the _ctypes extension dll exports.

ctypes.**DllGetClassObject**()

Windows only: This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllGetClassObject function that the `_ctypes` extension dll exports.

ctypes.util.**find_library**(*name*)

Try to find a library and return a pathname. *name* is the library name without any prefix like `lib`, suffix like `.so`, `.dylib` or version number (this is the form used for the posix linker option `-l`). If no library can be found, returns `None`.

The exact functionality is system dependent.

ctypes.util.**find_msvcrt**()

Windows only: return the filename of the VC runtime library used by Python, and by the extension modules. If the name of the library cannot be determined, `None` is returned.

If you need to free memory, for example, allocated by an extension module with a call to the `free(void *)`, it is important that you use the function in the same library that allocated the memory.

ctypes.**FormatError**([*code*])

Windows only: Returns a textual description of the error code *code*. If no error code is specified, the last error code is used by calling the Windows api function GetLastError.

ctypes.**GetLastError**()

Windows only: Returns the last error code set by Windows in the calling thread. This function calls the Windows `GetLastError()` function directly, it does not return the ctypes-private copy of the error code.

ctypes.**get_errno**()

Returns the current value of the ctypes-private copy of the system *errno* variable in the calling thread.

Raises an *auditing event* `ctypes.get_errno` with no arguments.

ctypes.**get_last_error**()

Windows only: returns the current value of the ctypes-private copy of the system `LastError` variable in the calling thread.

Raises an *auditing event* `ctypes.get_last_error` with no arguments.

ctypes.**memmove**(*dst*, *src*, *count*)

Same as the standard C memmove library function: copies *count* bytes from *src* to *dst*. *dst* and *src* must be integers or ctypes instances that can be converted to pointers.

ctypes.**memset**(*dst*, *c*, *count*)

Same as the standard C memset library function: fills the memory block at address *dst* with *count* bytes of value *c*. *dst* must be an integer specifying an address, or a ctypes instance.

ctypes.**POINTER**(*type*, /)

Create and return a new ctypes pointer type. Pointer types are cached and reused internally, so calling this function repeatedly is cheap. *type* must be a ctypes type.

ctypes.**pointer**(*obj*, /)

Create a new pointer instance, pointing to *obj*. The returned object is of the type `POINTER(type(obj))`.

Note: If you just want to pass a pointer to an object to a foreign function call, you should use `byref(obj)` which is much faster.

ctypes.**resize**(*obj*, *size*)

This function resizes the internal memory buffer of *obj*, which must be an instance of a ctypes type. It is not possible to make the buffer smaller than the native size of the objects type, as given by `sizeof(type(obj))`, but it is possible to enlarge the buffer.

ctypes.**set_errno**(*value*)

Set the current value of the ctypes-private copy of the system *errno* variable in the calling thread to *value* and return the previous value.

Raises an *auditing event* `ctypes.set_errno` with argument `errno`.

ctypes.**set_last_error**(*value*)

Windows only: set the current value of the ctypes-private copy of the system `LastError` variable in the calling thread to *value* and return the previous value.

Raises an *auditing event* `ctypes.set_last_error` with argument `error`.

ctypes.**sizeof**(*obj_or_type*)

Returns the size in bytes of a ctypes type or instance memory buffer. Does the same as the C `sizeof` operator.

ctypes.**string_at**(*ptr*, *size=-1*)

Return the byte string at *void *ptr*. If *size* is specified, it is used as size, otherwise the string is assumed to be zero-terminated.

Raises an *auditing event* `ctypes.string_at` with arguments `ptr`, `size`.

ctypes.**WinError**(*code=None*, *descr=None*)

Windows only: this function is probably the worst-named thing in ctypes. It creates an instance of *OSError*. If *code* is not specified, `GetLastError` is called to determine the error code. If *descr* is not specified, *FormatError()* is called to get a textual description of the error.

Changed in version 3.3: An instance of *WindowsError* used to be created, which is now an alias of *OSError*.

ctypes.**wstring_at**(*ptr*, *size=-1*)

Return the wide-character string at *void *ptr*. If *size* is specified, it is used as the number of characters of the string, otherwise the string is assumed to be zero-terminated.

Raises an *auditing event* `ctypes.wstring_at` with arguments `ptr`, `size`.

Data types

class ctypes.**_CData**

This non-public class is the common base class of all ctypes data types. Among other things, all ctypes type instances contain a memory block that hold C compatible data; the address of the memory block is returned by the *addressof()* helper function. Another instance variable is exposed as *_objects*; this contains other Python objects that need to be kept alive in case the memory block contains pointers.

Common methods of ctypes data types, these are all class methods (to be exact, they are methods of the *metaclass*):

from_buffer(*source*[, *offset*])

This method returns a ctypes instance that shares the buffer of the *source* object. The *source* object must support the writeable buffer interface. The optional *offset* parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a *ValueError* is raised.

Raises an *auditing event* `ctypes.cdata/buffer` with arguments `pointer`, `size`, `offset`.

from_buffer_copy(*source*[, *offset*])

This method creates a ctypes instance, copying the buffer from the *source* object buffer which must be readable. The optional *offset* parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a *ValueError* is raised.

Raises an *auditing event* `ctypes.cdata/buffer` with arguments `pointer`, `size`, `offset`.

from_address(*address*)

This method returns a ctypes type instance using the memory specified by *address* which must be an integer.

This method, and others that indirectly call this method, raises an *auditing event* `ctypes.cdata` with argument `address`.

from_param(*obj*)

This method adapts *obj* to a ctypes type. It is called with the actual object used in a foreign function call when the type is present in the foreign function's *argtypes* tuple; it must return an object that can be used as a function call parameter.

All ctypes data types have a default implementation of this classmethod that normally returns *obj* if that is an instance of the type. Some types accept other objects as well.

in_dll(*library*, *name*)

This method returns a ctypes type instance exported by a shared library. *name* is the name of the symbol that exports the data, *library* is the loaded shared library.

Common instance variables of ctypes data types:

_b_base_

Sometimes ctypes data instances do not own the memory block they contain, instead they share part of the memory block of a base object. The *_b_base_* read-only member is the root ctypes object that owns the memory block.

_b_needsfree_

This read-only variable is true when the ctypes data instance has allocated the memory block itself, false otherwise.

_objects

This member is either `None` or a dictionary containing Python objects that need to be kept alive so that the memory block contents is kept valid. This object is only exposed for debugging; never modify the contents of this dictionary.

Fundamental data types

class ctypes.**_SimpleCData**

This non-public class is the base class of all fundamental ctypes data types. It is mentioned here because it contains the common attributes of the fundamental ctypes data types. *_SimpleCData* is a subclass of *_CData*, so it inherits their methods and attributes. ctypes data types that are not and do not contain pointers can now be pickled.

Instances have a single attribute:

value

This attribute contains the actual value of the instance. For integer and pointer types, it is an integer, for character types, it is a single character bytes object or string, for character pointer types it is a Python bytes object or string.

When the `value` attribute is retrieved from a ctypes instance, usually a new object is returned each time. *ctypes* does *not* implement original object return, always a new object is constructed. The same is true for all other ctypes object instances.

Fundamental data types, when returned as foreign function call results, or, for example, by retrieving structure field members or array items, are transparently converted to native Python types. In other words, if a foreign function has a *restype* of *c_char_p*, you will always receive a Python bytes object, *not* a *c_char_p* instance.

Subclasses of fundamental data types do *not* inherit this behavior. So, if a foreign functions `restype` is a subclass of *c_void_p*, you will receive an instance of this subclass from the function call. Of course, you can get the value of the pointer by accessing the `value` attribute.

These are the fundamental ctypes data types:

class ctypes.**c_byte**

Represents the C `signed char` datatype, and interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_char**

Represents the C `char` datatype, and interprets the value as a single character. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.**c_char_p**

Represents the C `char*` datatype when it points to a zero-terminated string. For a general character pointer that may also point to binary data, `POINTER(c_char)` must be used. The constructor accepts an integer address, or a bytes object.

class ctypes.**c_double**

Represents the C `double` datatype. The constructor accepts an optional float initializer.

class ctypes.**c_longdouble**

Represents the C `long double` datatype. The constructor accepts an optional float initializer. On platforms where `sizeof(long double) == sizeof(double)` it is an alias to *c_double*.

class ctypes.**c_float**

Represents the C `float` datatype. The constructor accepts an optional float initializer.

class ctypes.**c_int**

Represents the C `signed int` datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias to *c_long*.

class ctypes.**c_int8**

Represents the C 8-bit `signed int` datatype. Usually an alias for *c_byte*.

class ctypes.**c_int16**

Represents the C 16-bit `signed int` datatype. Usually an alias for *c_short*.

class ctypes.**c_int32**

Represents the C 32-bit `signed int` datatype. Usually an alias for *c_int*.

class ctypes.**c_int64**

Represents the C 64-bit `signed int` datatype. Usually an alias for *c_longlong*.

class ctypes.**c_long**

Represents the C `signed long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_longlong**

Represents the C `signed long long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_short**

Represents the C `signed short` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_size_t**

Represents the C `size_t` datatype.

class ctypes.**c_ssize_t**

Represents the C `ssize_t` datatype.

Added in version 3.2.

class ctypes.**c_time_t**

Represents the C `time_t` datatype.

Added in version 3.12.

class ctypes.**c_ubyte**

Represents the C `unsigned char` datatype, it interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_uint**

Represents the C `unsigned int` datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias for *c_ulong*.

class ctypes.**c_uint8**

Represents the C 8-bit `unsigned int` datatype. Usually an alias for *c_ubyte*.

class ctypes.**c_uint16**

Represents the C 16-bit `unsigned int` datatype. Usually an alias for *c_ushort*.

class ctypes.**c_uint32**

Represents the C 32-bit `unsigned int` datatype. Usually an alias for *c_uint*.

class ctypes.**c_uint64**

Represents the C 64-bit `unsigned int` datatype. Usually an alias for *c_ulonglong*.

class ctypes.**c_ulong**

Represents the C `unsigned long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_ulonglong**

Represents the C `unsigned long long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_ushort**

Represents the C `unsigned short` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_void_p**

Represents the C `void*` type. The value is represented as integer. The constructor accepts an optional integer initializer.

class ctypes.**c_wchar**

Represents the C `wchar_t` datatype, and interprets the value as a single character unicode string. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.**c_wchar_p**

Represents the C `wchar_t*` datatype, which must be a pointer to a zero-terminated wide character string. The constructor accepts an integer address, or a string.

class ctypes.**c_bool**

Represent the C `bool` datatype (more accurately, `_Bool` from C99). Its value can be `True` or `False`, and the constructor accepts any object that has a truth value.

class ctypes.**HRESULT**

Windows only: Represents a `HRESULT` value, which contains success or error information for a function or method call.

class ctypes.**py_object**

Represents the C `PyObject*` datatype. Calling this without an argument creates a `NULL` `PyObject*` pointer.

The `ctypes.wintypes` module provides quite some other Windows specific data types, for example `HWND`, `WPARAM`, or `DWORD`. Some useful structures like `MSG` or `RECT` are also defined.

Structured data types

class ctypes.**Union**(**args*, ***kw*)

Abstract base class for unions in native byte order.

class ctypes.**BigEndianUnion**(**args*, ***kw*)

Abstract base class for unions in *big endian* byte order.

Added in version 3.11.

class ctypes.**LittleEndianUnion**(**args*, ***kw*)

Abstract base class for unions in *little endian* byte order.

Added in version 3.11.

class ctypes.**BigEndianStructure**(**args*, ***kw*)

Abstract base class for structures in *big endian* byte order.

class ctypes.**LittleEndianStructure**(**args*, ***kw*)

Abstract base class for structures in *little endian* byte order.

Structures and unions with non-native byte order cannot contain pointer type fields, or any other data types containing pointer type fields.

class ctypes.**Structure**(**args*, ***kw*)

Abstract base class for structures in *native* byte order.

Concrete structure and union types must be created by subclassing one of these types, and at least define a *_fields_* class variable. *ctypes* will create *descriptor*s which allow reading and writing the fields by direct attribute accesses. These are the

fields

A sequence defining the structure fields. The items must be 2-tuples or 3-tuples. The first item is the name of the field, the second item specifies the type of the field; it can be any ctypes data type.

For integer type fields like *c_int*, a third optional item can be given. It must be a small positive integer defining the bit width of the field.

Field names must be unique within one structure or union. This is not checked, only one field can be accessed when names are repeated.

It is possible to define the *_fields_* class variable *after* the class statement that defines the Structure subclass, this allows creating data types that directly or indirectly reference themselves:

```
class List(Structure):
    pass
List._fields_ = [("pnext", POINTER(List)),
                 ...
                ]
```

The *_fields_* class variable must, however, be defined before the type is first used (an instance is created, *sizeof()* is called on it, and so on). Later assignments to the *_fields_* class variable will raise an AttributeError.

It is possible to define sub-subclasses of structure types, they inherit the fields of the base class plus the *_fields_* defined in the sub-subclass, if any.

pack

An optional small integer that allows overriding the alignment of structure fields in the instance. *_pack_* must already be defined when *_fields_* is assigned, otherwise it will have no effect. Setting this attribute to 0 is the same as not setting it at all.

align

An optional small integer that allows overriding the alignment of the structure when being packed or unpacked to/from memory. Setting this attribute to 0 is the same as not setting it at all.

anonymous

An optional sequence that lists the names of unnamed (anonymous) fields. *_anonymous_* must be already defined when *_fields_* is assigned, otherwise it will have no effect.

The fields listed in this variable must be structure or union type fields. *ctypes* will create descriptors in the structure type that allows accessing the nested fields directly, without the need to create the structure or union field.

Here is an example type (Windows):

```
class _U(Union):
    _fields_ = [("lptdesc", POINTER(TYPEDESC)),
                ("lpadesc", POINTER(ARRAYDESC)),
                ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    _anonymous_ = ("u",)
    _fields_ = [("u", _U),
                ("vt", VARTYPE)]
```

The `TYPEDESC` structure describes a COM data type, the `vt` field specifies which one of the union fields is valid. Since the `u` field is defined as anonymous field, it is now possible to access the members directly off the TYPEDESC instance. `td.lptdesc` and `td.u.lptdesc` are equivalent, but the former is faster since it does not need to create a temporary union instance:

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

It is possible to define sub-subclasses of structures, they inherit the fields of the base class. If the subclass definition has a separate *_fields_* variable, the fields specified in this are appended to the fields of the base class.

Structure and union constructors accept both positional and keyword arguments. Positional arguments are used to initialize member fields in the same order as they are appear in *_fields_*. Keyword arguments in the constructor are interpreted as attribute assignments, so they will initialize *_fields_* with the same name, or create new attributes for names not present in *_fields_*.

Arrays and pointers

class ctypes.**Array**(**args*)

Abstract base class for arrays.

The recommended way to create concrete array types is by multiplying any *ctypes* data type with a non-negative integer. Alternatively, you can subclass this type and define *_length_* and *_type_* class variables. Array elements can be read and written using standard subscript and slice accesses; for slice reads, the resulting object is *not* itself an *Array*.

length

A positive integer specifying the number of elements in the array. Out-of-range subscripts result in an *IndexError*. Will be returned by *len()*.

type

Specifies the type of each element in the array.

Array subclass constructors accept positional arguments, used to initialize the elements in order.

ctypes.**ARRAY**(*type*, *length*)

Create an array. Equivalent to `type * length`, where *type* is a *ctypes* data type and *length* an integer.

This function is *soft deprecated* in favor of multiplication. There are no plans to remove it.

class ctypes.**_Pointer**

Private, abstract base class for pointers.

Concrete pointer types are created by calling *POINTER()* with the type that will be pointed to; this is done automatically by *pointer()*.

If a pointer points to an array, its elements can be read and written using standard subscript and slice accesses. Pointer objects have no size, so *len()* will raise *TypeError*. Negative subscripts will read from the memory *before* the pointer (as in C), and out-of-range subscripts will probably crash with an access violation (if you're lucky).

type

Specifies the type pointed to.

contents

Returns the object to which to pointer points. Assigning to this attribute changes the pointer to point to the assigned object.

CHAPTER

SEVENTEEN

CONCURRENT EXECUTION

The modules described in this chapter provide support for concurrent execution of code. The appropriate choice of tool will depend on the task to be executed (CPU bound vs IO bound) and preferred style of development (event driven cooperative multitasking vs preemptive multitasking). Here's an overview:

17.1 `threading` — Thread-based parallelism

Source code: Lib/threading.py

This module constructs higher-level threading interfaces on top of the lower level *`_thread`* module.

Changed in version 3.7: This module used to be optional, it is now always available.

See also

`concurrent.futures.ThreadPoolExecutor` offers a higher level interface to push tasks to a background thread without blocking execution of the calling thread, while still being able to retrieve their results when needed.

`queue` provides a thread-safe interface for exchanging data between running threads.

`asyncio` offers an alternative approach to achieving task level concurrency without requiring the use of multiple operating system threads.

Note

In the Python 2.x series, this module contained `camelCase` names for some methods and functions. These are deprecated as of Python 3.10, but they are still supported for compatibility with Python 2.5 and lower.

CPython implementation detail: In CPython, due to the *Global Interpreter Lock*, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use *`multiprocessing`* or *`concurrent.futures.ProcessPoolExecutor`*. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

This module defines the following functions:

threading.**active_count**()

Return the number of *Thread* objects currently alive. The returned count is equal to the length of the list returned by *enumerate()*.

The function activeCount is a deprecated alias for this function.

threading.**current_thread**()

Return the current *Thread* object, corresponding to the caller's thread of control. If the caller's thread of control was not created through the *threading* module, a dummy thread object with limited functionality is returned.

The function currentThread is a deprecated alias for this function.

threading.**excepthook**(*args*, /)

Handle uncaught exception raised by *Thread.run()*.

The *args* argument has the following attributes:

- *exc_type*: Exception type.
- *exc_value*: Exception value, can be None.
- *exc_traceback*: Exception traceback, can be None.
- *thread*: Thread which raised the exception, can be None.

If *exc_type* is *SystemExit*, the exception is silently ignored. Otherwise, the exception is printed out on *sys.stderr*.

If this function raises an exception, *sys.excepthook()* is called to handle it.

threading.excepthook() can be overridden to control how uncaught exceptions raised by *Thread.run()* are handled.

Storing *exc_value* using a custom hook can create a reference cycle. It should be cleared explicitly to break the reference cycle when the exception is no longer needed.

Storing *thread* using a custom hook can resurrect it if it is set to an object which is being finalized. Avoid storing *thread* after the custom hook completes to avoid resurrecting objects.

See also

sys.excepthook() handles uncaught exceptions.

Added in version 3.8.

threading.**__excepthook__**

Holds the original value of *threading.excepthook()*. It is saved so that the original value can be restored in case they happen to get replaced with broken or alternative objects.

Added in version 3.10.

threading.**get_ident**()

Return the 'thread identifier' of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

Added in version 3.3.

threading.**get_native_id**()

Return the native integral Thread ID of the current thread assigned by the kernel. This is a non-negative integer. Its value may be used to uniquely identify this particular thread system-wide (until the thread terminates, after which the value may be recycled by the OS).

Availability: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD, GNU/kFreeBSD.

Added in version 3.8.

Changed in version 3.13: Added support for GNU/kFreeBSD.

threading.**enumerate**()

Return a list of all *Thread* objects currently active. The list includes daemonic threads and dummy thread objects created by *current_thread()*. It excludes terminated threads and threads that have not yet been started. However, the main thread is always part of the result, even when terminated.

threading.**main_thread**()

Return the main *Thread* object. In normal conditions, the main thread is the thread from which the Python interpreter was started.

Added in version 3.4.

threading.**settrace**(*func*)

Set a trace function for all threads started from the *threading* module. The *func* will be passed to *sys.settrace()* for each thread, before its *run()* method is called.

threading.**settrace_all_threads**(*func*)

Set a trace function for all threads started from the *threading* module and all Python threads that are currently executing.

The *func* will be passed to *sys.settrace()* for each thread, before its *run()* method is called.

Added in version 3.12.

threading.**gettrace**()

Get the trace function as set by *settrace()*.

Added in version 3.10.

threading.**setprofile**(*func*)

Set a profile function for all threads started from the *threading* module. The *func* will be passed to *sys.setprofile()* for each thread, before its *run()* method is called.

threading.**setprofile_all_threads**(*func*)

Set a profile function for all threads started from the *threading* module and all Python threads that are currently executing.

The *func* will be passed to *sys.setprofile()* for each thread, before its *run()* method is called.

Added in version 3.12.

threading.**getprofile**()

Get the profiler function as set by *setprofile()*.

Added in version 3.10.

threading.**stack_size**([*size*])

Return the thread stack size used when creating new threads. The optional *size* argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32 KiB). If *size* is not specified, 0 is used. If changing the thread stack size is unsupported, a *RuntimeError* is raised. If the specified stack size is invalid, a *ValueError* is raised and the stack size

is unmodified. 32 KiB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32 KiB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4 KiB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information).

Availability: Windows, pthreads.

Unix platforms with POSIX threads support.

This module also defines the following constant:

`threading.`**`TIMEOUT_MAX`**

The maximum value allowed for the *timeout* parameter of blocking functions (*Lock.acquire()*, *RLock.acquire()*, *Condition.wait()*, etc.). Specifying a timeout greater than this value will raise an *OverflowError*.

Added in version 3.2.

This module defines a number of classes, which are detailed in the sections below.

The design of this module is loosely based on Java's threading model. However, where Java makes locks and condition variables basic behavior of every object, they are separate objects in Python. Python's *Thread* class supports a subset of the behavior of Java's Thread class; currently, there are no priorities, no thread groups, and threads cannot be destroyed, stopped, suspended, resumed, or interrupted. The static methods of Java's Thread class, when implemented, are mapped to module-level functions.

All of the methods described below are executed atomically.

17.1.1 Thread-Local Data

Thread-local data is data whose values are thread specific. To manage thread-local data, just create an instance of *local* (or a subclass) and store attributes on it:

```
mydata = threading.local()
mydata.x = 1
```

The instance's values will be different for separate threads.

class `threading.`**`local`**

A class that represents thread-local data.

For more details and extensive examples, see the documentation string of the `_threading_local` module: Lib/_threading_local.py.

17.1.2 Thread Objects

The *Thread* class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the *run()* method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, *only* override the `__init__()` and *run()* methods of this class.

Once a thread object is created, its activity must be started by calling the thread's *start()* method. This invokes the *run()* method in a separate thread of control.

Once the thread's activity is started, the thread is considered 'alive'. It stops being alive when its *run()* method terminates – either normally, or by raising an unhandled exception. The *is_alive()* method tests whether the thread is alive.

Other threads can call a thread's *join()* method. This blocks the calling thread until the thread whose *join()* method is called is terminated.

A thread has a name. The name can be passed to the constructor, and read or changed through the *name* attribute.

If the *run()* method raises an exception, *threading.excepthook()* is called to handle it. By default, *threading.excepthook()* ignores silently *SystemExit*.

A thread can be flagged as a "daemon thread". The significance of this flag is that the entire Python program exits when only daemon threads are left. The initial value is inherited from the creating thread. The flag can be set through the *daemon* property or the *daemon* constructor argument.

Note

Daemon threads are abruptly stopped at shutdown. Their resources (such as open files, database transactions, etc.) may not be released properly. If you want your threads to stop gracefully, make them non-daemonic and use a suitable signalling mechanism such as an *Event*.

There is a "main thread" object; this corresponds to the initial thread of control in the Python program. It is not a daemon thread.

There is the possibility that "dummy thread objects" are created. These are thread objects corresponding to "alien threads", which are threads of control started outside the threading module, such as directly from C code. Dummy thread objects have limited functionality; they are always considered alive and daemonic, and cannot be *joined*. They are never deleted, since it is impossible to detect the termination of alien threads.

class threading.**Thread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, *, *daemon=None*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the *run()* method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-*N*" where *N* is a small decimal number, or "Thread-*N* (target)" where "target" is target.__name__ if the *target* argument is specified.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If not None, *daemon* explicitly sets whether the thread is daemonic. If None (the default), the daemonic property is inherited from the current thread.

If the subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

Changed in version 3.3: Added the *daemon* parameter.

Changed in version 3.10: Use the *target* name if *name* argument is omitted.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's *run()* method to be invoked in a separate thread of control.

This method will raise a *RuntimeError* if called more than once on the same thread object.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard *run()* method invokes the callable object passed to the object's constructor as the *target* argument, if any, with positional and keyword arguments taken from the *args* and *kwargs* arguments, respectively.

Using list or tuple as the *args* argument which passed to the *Thread* could achieve the same effect.

Example:

```
>>> from threading import Thread
>>> t = Thread(target=print, args=[1])
>>> t.run()
1
>>> t = Thread(target=print, args=(1,))
>>> t.run()
1
```

join(*timeout=None*)

Wait until the thread terminates. This blocks the calling thread until the thread whose *join()* method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs.

When the *timeout* argument is present and not `None`, it should be a floating-point number specifying a timeout for the operation in seconds (or fractions thereof). As *join()* always returns `None`, you must call *is_alive()* after *join()* to decide whether a timeout happened – if the thread is still alive, the *join()* call timed out.

When the *timeout* argument is not present or `None`, the operation will block until the thread terminates.

A thread can be joined many times.

join() raises a *RuntimeError* if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to *join()* a thread before it has been started and attempts to do so raise the same exception.

name

A string used for identification purposes only. It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

getName()

setName()

Deprecated getter/setter API for *name*; use it directly as a property instead.

Deprecated since version 3.10.

ident

The 'thread identifier' of this thread or `None` if the thread has not been started. This is a nonzero integer. See the *get_ident()* function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

native_id

The Thread ID (`TID`) of this thread, as assigned by the OS (kernel). This is a non-negative integer, or `None` if the thread has not been started. See the *get_native_id()* function. This value may be used to uniquely identify this particular thread system-wide (until the thread terminates, after which the value may be recycled by the OS).

Note

Similar to Process IDs, Thread IDs are only valid (guaranteed unique system-wide) from the time the thread is created until the thread has been terminated.

Availability: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD.

Added in version 3.8.

is_alive()

Return whether the thread is alive.

This method returns `True` just before the *run()* method starts until just after the *run()* method terminates. The module function *enumerate()* returns a list of all alive threads.

daemon

A boolean value indicating whether this thread is a daemon thread (`True`) or not (`False`). This must be set before *start()* is called, otherwise *RuntimeError* is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to *daemon* = `False`.

The entire Python program exits when no alive non-daemon threads are left.

isDaemon()

setDaemon()

Deprecated getter/setter API for *daemon*; use it directly as a property instead.

Deprecated since version 3.10.

17.1.3 Lock Objects

A primitive lock is a synchronization primitive that is not owned by a particular thread when locked. In Python, it is currently the lowest level synchronization primitive available, implemented directly by the *_thread* extension module.

A primitive lock is in one of two states, "locked" or "unlocked". It is created in the unlocked state. It has two basic methods, *acquire()* and *release()*. When the state is unlocked, *acquire()* changes the state to locked and returns immediately. When the state is locked, *acquire()* blocks until a call to *release()* in another thread changes it to unlocked, then the *acquire()* call resets it to locked and returns. The *release()* method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a *RuntimeError* will be raised.

Locks also support the *context management protocol*.

When more than one thread is blocked in *acquire()* waiting for the state to turn to unlocked, only one thread proceeds when a *release()* call resets the state to unlocked; which one of the waiting threads proceeds is not defined, and may vary across implementations.

All methods are executed atomically.

class threading.**Lock**

The class implementing primitive lock objects. Once a thread has acquired a lock, subsequent attempts to acquire it block, until it is released; any thread may release it.

Changed in version 3.13: `Lock` is now a class. In earlier Pythons, `Lock` was a factory function which returned an instance of the underlying private lock type.

acquire(*blocking=True*, *timeout=-1*)

Acquire a lock, blocking or non-blocking.

When invoked with the *blocking* argument set to `True` (the default), block until the lock is unlocked, then set it to locked and return `True`.

When invoked with the *blocking* argument set to `False`, do not block. If a call with *blocking* set to `True` would block, return `False` immediately; otherwise, set the lock to locked and return `True`.

When invoked with the floating-point *timeout* argument set to a positive value, block for at most the number of seconds specified by *timeout* and as long as the lock cannot be acquired. A *timeout* argument of `-1` specifies an unbounded wait. It is forbidden to specify a *timeout* when *blocking* is `False`.

The return value is `True` if the lock is acquired successfully, `False` if not (for example if the *timeout* expired).

Changed in version 3.2: The *timeout* parameter is new.

Changed in version 3.2: Lock acquisition can now be interrupted by signals on POSIX if the underlying threading implementation supports it.

release()

Release a lock. This can be called from any thread, not only the thread which has acquired the lock.

When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

When invoked on an unlocked lock, a *RuntimeError* is raised.

There is no return value.

locked()

Return `True` if the lock is acquired.

17.1.4 RLock Objects

A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. Internally, it uses the concepts of "owning thread" and "recursion level" in addition to the locked/unlocked state used by primitive locks. In the locked state, some thread owns the lock; in the unlocked state, no thread owns it.

Threads call a lock's *acquire()* method to lock it, and its *release()* method to unlock it.

> **Note**
>
> Reentrant locks support the *context management protocol*, so it is recommended to use `with` instead of manually calling *acquire()* and *release()* to handle acquiring and releasing the lock for a block of code.

RLock's *acquire()*/*release()* call pairs may be nested, unlike Lock's *acquire()*/*release()*. Only the final *release()* (the *release()* of the outermost pair) resets the lock to an unlocked state and allows another thread blocked in *acquire()* to proceed.

acquire()/*release()* must be used in pairs: each acquire must have a release in the thread that has acquired the lock. Failing to call release as many times the lock has been acquired can lead to deadlock.

class threading.**RLock**

This class implements reentrant lock objects. A reentrant lock must be released by the thread that acquired it. Once a thread has acquired a reentrant lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

Note that `RLock` is actually a factory function which returns an instance of the most efficient version of the concrete RLock class that is supported by the platform.

acquire(*blocking=True*, *timeout=-1*)

Acquire a lock, blocking or non-blocking.

> **See also**
>
> ***Using RLock as a context manager***
> Recommended over manual `acquire()` and *`release()`* calls whenever practical.

When invoked with the *blocking* argument set to `True` (the default):

- If no thread owns the lock, acquire the lock and return immediately.
- If another thread owns the lock, block until we are able to acquire lock, or *timeout*, if set to a positive float value.
- If the same thread owns the lock, acquire the lock again, and return immediately. This is the difference between *`Lock`* and `RLock`; *`Lock`* handles this case the same as the previous, blocking until the lock can be acquired.

When invoked with the *blocking* argument set to `False`:

- If no thread owns the lock, acquire the lock and return immediately.
- If another thread owns the lock, return immediately.
- If the same thread owns the lock, acquire the lock again and return immediately.

In all cases, if the thread was able to acquire the lock, return `True`. If the thread was unable to acquire the lock (i.e. if not blocking or the timeout was reached) return `False`.

If called multiple times, failing to call *`release()`* as many times may lead to deadlock. Consider using `RLock` as a context manager rather than calling acquire/release directly.

Changed in version 3.2: The *timeout* parameter is new.

`release``()`

Release a lock, decrementing the recursion level. If after the decrement it is zero, reset the lock to unlocked (not owned by any thread), and if any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling thread.

Only call this method when the calling thread owns the lock. A *`RuntimeError`* is raised if this method is called when the lock is not acquired.

There is no return value.

17.1.5 Condition Objects

A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default. Passing one in is useful when several condition variables must share the same lock. The lock is part of the condition object: you don't have to track it separately.

A condition variable obeys the *context management protocol*: using the `with` statement acquires the associated lock for the duration of the enclosed block. The *`acquire()`* and *`release()`* methods also call the corresponding methods of the associated lock.

Other methods must be called with the associated lock held. The *`wait()`* method releases the lock, and then blocks until another thread awakens it by calling *`notify()`* or *`notify_all()`*. Once awakened, *`wait()`* re-acquires the lock and returns. It is also possible to specify a timeout.

The *`notify()`* method wakes up one of the threads waiting for the condition variable, if any are waiting. The *`notify_all()`* method wakes up all threads waiting for the condition variable.

Note: the *notify()* and *notify_all()* methods don't release the lock; this means that the thread or threads awakened will not return from their *wait()* call immediately, but only when the thread that called *notify()* or *notify_all()* finally relinquishes ownership of the lock.

The typical programming style using condition variables uses the lock to synchronize access to some shared state; threads that are interested in a particular change of state call *wait()* repeatedly until they see the desired state, while threads that modify the state call *notify()* or *notify_all()* when they change the state in such a way that it could possibly be a desired state for one of the waiters. For example, the following code is a generic producer-consumer situation with unlimited buffer capacity:

```
# Consume one item
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# Produce one item
with cv:
    make_an_item_available()
    cv.notify()
```

The `while` loop checking for the application's condition is necessary because *wait()* can return after an arbitrary long time, and the condition which prompted the *notify()* call may no longer hold true. This is inherent to multi-threaded programming. The *wait_for()* method can be used to automate the condition checking, and eases the computation of timeouts:

```
# Consume an item
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

To choose between *notify()* and *notify_all()*, consider whether one state change can be interesting for only one or several waiting threads. E.g. in a typical producer-consumer situation, adding one item to the buffer only needs to wake up one consumer thread.

class threading.**Condition**(*lock=None*)

This class implements condition variable objects. A condition variable allows one or more threads to wait until they are notified by another thread.

If the *lock* argument is given and not `None`, it must be a *Lock* or *RLock* object, and it is used as the underlying lock. Otherwise, a new *RLock* object is created and used as the underlying lock.

Changed in version 3.3: changed from a factory function to a class.

acquire(**args*)

Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.

release()

Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.

wait(*timeout=None*)

Wait until notified or until a timeout occurs. If the calling thread has not acquired the lock when this method is called, a *RuntimeError* is raised.

This method releases the underlying lock, and then blocks until it is awakened by a *notify()* or *notify_all()* call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.

When the *timeout* argument is present and not `None`, it should be a floating-point number specifying a timeout for the operation in seconds (or fractions thereof).

When the underlying lock is an *RLock*, it is not released using its *release()* method, since this may not actually unlock the lock when it was acquired multiple times recursively. Instead, an internal interface of the *RLock* class is used, which really unlocks it even when it has been recursively acquired several times. Another internal interface is then used to restore the recursion level when the lock is reacquired.

The return value is `True` unless a given *timeout* expired, in which case it is `False`.

Changed in version 3.2: Previously, the method always returned `None`.

wait_for(*predicate*, *timeout=None*)

Wait until a condition evaluates to true. *predicate* should be a callable which result will be interpreted as a boolean value. A *timeout* may be provided giving the maximum time to wait.

This utility method may call *wait()* repeatedly until the predicate is satisfied, or until a timeout occurs. The return value is the last return value of the predicate and will evaluate to `False` if the method timed out.

Ignoring the timeout feature, calling this method is roughly equivalent to writing:

```
while not predicate():
    cv.wait()
```

Therefore, the same rules apply as with *wait()*: The lock must be held when called and is re-acquired on return. The predicate is evaluated with the lock held.

Added in version 3.2.

notify(*n=1*)

By default, wake up one thread waiting on this condition, if any. If the calling thread has not acquired the lock when this method is called, a *RuntimeError* is raised.

This method wakes up at most *n* of the threads waiting for the condition variable; it is a no-op if no threads are waiting.

The current implementation wakes up exactly *n* threads, if at least *n* threads are waiting. However, it's not safe to rely on this behavior. A future, optimized implementation may occasionally wake up more than *n* threads.

Note: an awakened thread does not actually return from its *wait()* call until it can reacquire the lock. Since *notify()* does not release the lock, its caller should.

notify_all()

Wake up all threads waiting on this condition. This method acts like *notify()*, but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a *RuntimeError* is raised.

The method `notifyAll` is a deprecated alias for this method.

17.1.6 Semaphore Objects

This is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra (he used the names `P()` and `V()` instead of *acquire()* and *release()*).

A semaphore manages an internal counter which is decremented by each *acquire()* call and incremented by each *release()* call. The counter can never go below zero; when *acquire()* finds that it is zero, it blocks, waiting until some other thread calls *release()*.

Semaphores also support the *context management protocol*.

class threading.**Semaphore**(*value=1*)

This class implements semaphore objects. A semaphore manages an atomic counter representing the number of *release()* calls minus the number of *acquire()* calls, plus an initial value. The *acquire()* method blocks if necessary until it can return without making the counter negative. If not given, *value* defaults to 1.

The optional argument gives the initial *value* for the internal counter; it defaults to `1`. If the *value* given is less than 0, *ValueError* is raised.

Changed in version 3.3: changed from a factory function to a class.

acquire(*blocking=True*, *timeout=None*)

Acquire a semaphore.

When invoked without arguments:

- If the internal counter is larger than zero on entry, decrement it by one and return `True` immediately.
- If the internal counter is zero on entry, block until awoken by a call to *release()*. Once awoken (and the counter is greater than 0), decrement the counter by 1 and return `True`. Exactly one thread will be awoken by each call to *release()*. The order in which threads are awoken should not be relied on.

When invoked with *blocking* set to `False`, do not block. If a call without an argument would block, return `False` immediately; otherwise, do the same thing as when called without arguments, and return `True`.

When invoked with a *timeout* other than `None`, it will block for at most *timeout* seconds. If acquire does not complete successfully in that interval, return `False`. Return `True` otherwise.

Changed in version 3.2: The *timeout* parameter is new.

release(*n=1*)

Release a semaphore, incrementing the internal counter by *n*. When it was zero on entry and other threads are waiting for it to become larger than zero again, wake up *n* of those threads.

Changed in version 3.9: Added the *n* parameter to release multiple waiting threads at once.

class threading.**BoundedSemaphore**(*value=1*)

Class implementing bounded semaphore objects. A bounded semaphore checks to make sure its current value doesn't exceed its initial value. If it does, *ValueError* is raised. In most situations semaphores are used to guard resources with limited capacity. If the semaphore is released too many times it's a sign of a bug. If not given, *value* defaults to 1.

Changed in version 3.3: changed from a factory function to a class.

`Semaphore` Example

Semaphores are often used to guard resources with limited capacity, for example, a database server. In any situation where the size of the resource is fixed, you should use a bounded semaphore. Before spawning any worker threads, your main thread would initialize the semaphore:

```
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

Once spawned, worker threads call the semaphore's acquire and release methods when they need to connect to the server:

```
with pool_sema:
    conn = connectdb()
    try:
        # ... use connection ...
```

(continues on next page)

(continued from previous page)

```
    finally:
        conn.close()
```

The use of a bounded semaphore reduces the chance that a programming error which causes the semaphore to be released more than it's acquired will go undetected.

17.1.7 Event Objects

This is one of the simplest mechanisms for communication between threads: one thread signals an event and other threads wait for it.

An event object manages an internal flag that can be set to true with the *set()* method and reset to false with the *clear()* method. The *wait()* method blocks until the flag is true.

class threading.**Event**

Class implementing event objects. An event manages a flag that can be set to true with the *set()* method and reset to false with the *clear()* method. The *wait()* method blocks until the flag is true. The flag is initially false.

Changed in version 3.3: changed from a factory function to a class.

is_set()

Return `True` if and only if the internal flag is true.

The method `isSet` is a deprecated alias for this method.

set()

Set the internal flag to true. All threads waiting for it to become true are awakened. Threads that call *wait()* once the flag is true will not block at all.

clear()

Reset the internal flag to false. Subsequently, threads calling *wait()* will block until *set()* is called to set the internal flag to true again.

wait(*timeout=None*)

Block as long as the internal flag is false and the timeout, if given, has not expired. The return value represents the reason that this blocking method returned; `True` if returning because the internal flag is set to true, or `False` if a timeout is given and the the internal flag did not become true within the given wait time.

When the timeout argument is present and not `None`, it should be a floating-point number specifying a timeout for the operation in seconds, or fractions thereof.

Changed in version 3.1: Previously, the method always returned `None`.

17.1.8 Timer Objects

This class represents an action that should be run only after a certain amount of time has passed — a timer. *Timer* is a subclass of *Thread* and as such also functions as an example of creating custom threads.

Timers are started, as with threads, by calling their *Timer.start* method. The timer can be stopped (before its action has begun) by calling the *cancel()* method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

For example:

```
def hello():
    print("hello, world")

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

class threading.**Timer**(*interval*, *function*, *args=None*, *kwargs=None*)

Create a timer that will run *function* with arguments *args* and keyword arguments *kwargs*, after *interval* seconds have passed. If *args* is `None` (the default) then an empty list will be used. If *kwargs* is `None` (the default) then an empty dict will be used.

Changed in version 3.3: changed from a factory function to a class.

cancel()

Stop the timer, and cancel the execution of the timer's action. This will only work if the timer is still in its waiting stage.

17.1.9 Barrier Objects

Added in version 3.2.

This class provides a simple synchronization primitive for use by a fixed number of threads that need to wait for each other. Each of the threads tries to pass the barrier by calling the *wait()* method and will block until all of the threads have made their *wait()* calls. At this point, the threads are released simultaneously.

The barrier can be reused any number of times for the same number of threads.

As an example, here is a simple way to synchronize a client and server thread:

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)

def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

class threading.**Barrier**(*parties*, *action=None*, *timeout=None*)

Create a barrier object for *parties* number of threads. An *action*, when provided, is a callable to be called by one of the threads when they are released. *timeout* is the default timeout value if none is specified for the *wait()* method.

wait(*timeout=None*)

Pass the barrier. When all the threads party to the barrier have called this function, they are all released simultaneously. If a *timeout* is provided, it is used in preference to any that was supplied to the class constructor.

The return value is an integer in the range 0 to *parties* – 1, different for each thread. This can be used to select a thread to do some special housekeeping, e.g.:

```
i = barrier.wait()
if i == 0:
    # Only one thread needs to print this
    print("passed the barrier")
```

If an *action* was provided to the constructor, one of the threads will have called it prior to being released. Should this call raise an error, the barrier is put into the broken state.

If the call times out, the barrier is put into the broken state.

This method may raise a *BrokenBarrierError* exception if the barrier is broken or reset while a thread is waiting.

reset()

Return the barrier to the default, empty state. Any threads waiting on it will receive the *BrokenBarrierError* exception.

Note that using this function may require some external synchronization if there are other threads whose state is unknown. If a barrier is broken it may be better to just leave it and create a new one.

abort()

Put the barrier into a broken state. This causes any active or future calls to *wait()* to fail with the *BrokenBarrierError*. Use this for example if one of the threads needs to abort, to avoid deadlocking the application.

It may be preferable to simply create the barrier with a sensible *timeout* value to automatically guard against one of the threads going awry.

parties

The number of threads required to pass the barrier.

n_waiting

The number of threads currently waiting in the barrier.

broken

A boolean that is `True` if the barrier is in the broken state.

exception threading.**BrokenBarrierError**

This exception, a subclass of *RuntimeError*, is raised when the *Barrier* object is reset or broken.

17.1.10 Using locks, conditions, and semaphores in the `with` statement

All of the objects provided by this module that have `acquire` and `release` methods can be used as context managers for a `with` statement. The `acquire` method will be called when the block is entered, and `release` will be called when the block is exited. Hence, the following snippet:

```
with some_lock:
    # do something...
```

is equivalent to:

```
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

Currently, *Lock*, *RLock*, *Condition*, *Semaphore*, and *BoundedSemaphore* objects may be used as `with` statement context managers.

17.2 `multiprocessing` — Process-based parallelism

Source code: Lib/multiprocessing/

Availability: not WASI, not iOS.

This module does not work or is not available on WebAssembly platforms, or on iOS. See *WebAssembly platforms* for more information on WASM availability; see *iOS* for more information on iOS availability.

17.2.1 Introduction

multiprocessing is a package that supports spawning processes using an API similar to the *threading* module. The *multiprocessing* package offers both local and remote concurrency, effectively side-stepping the *Global Interpreter Lock* by using subprocesses instead of threads. Due to this, the *multiprocessing* module allows the programmer to fully leverage multiple processors on a given machine. It runs on both POSIX and Windows.

The *multiprocessing* module also introduces APIs which do not have analogs in the *threading* module. A prime example of this is the *Pool* object which offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data parallelism). The following example demonstrates the common practice of defining such functions in a module so that child processes can successfully import that module. This basic example of data parallelism using *Pool*,

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

will print to standard output

```
[1, 4, 9]
```

See also

concurrent.futures.ProcessPoolExecutor offers a higher level interface to push tasks to a background process without blocking execution of the calling process. Compared to using the *Pool* interface directly, the *concurrent.futures* API more readily allows the submission of work to the underlying process pool to be separated from waiting for the results.

The `Process` class

In *multiprocessing*, processes are spawned by creating a *Process* object and then calling its *start()* method. *Process* follows the API of *threading.Thread*. A trivial example of a multiprocess program is

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

To show the individual process IDs involved, here is an expanded example:

```
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

For an explanation of why the `if __name__ == '__main__'` part is necessary, see *Programming guidelines*.

Contexts and start methods

Depending on the platform, *multiprocessing* supports three ways to start a process. These *start methods* are

spawn
: The parent process starts a fresh Python interpreter process. The child process will only inherit those resources necessary to run the process object's *run()* method. In particular, unnecessary file descriptors and handles from the parent process will not be inherited. Starting a process using this method is rather slow compared to using *fork* or *forkserver*.

  Available on POSIX and Windows platforms. The default on Windows and macOS.

fork
: The parent process uses *os.fork()* to fork the Python interpreter. The child process, when it begins, is effectively identical to the parent process. All resources of the parent are inherited by the child process. Note that safely forking a multithreaded process is problematic.

  Available on POSIX systems. Currently the default on POSIX except macOS.

> **Note**
>
> The default start method will change away from *fork* in Python 3.14. Code that requires *fork* should explicitly specify that via `get_context()` or `set_start_method()`.

Changed in version 3.12: If Python is able to detect that your process has multiple threads, the `os.fork()` function that this start method calls internally will raise a `DeprecationWarning`. Use a different start method. See the `os.fork()` documentation for further explanation.

forkserver

When the program starts and selects the *forkserver* start method, a server process is spawned. From then on, whenever a new process is needed, the parent process connects to the server and requests that it fork a new process. The fork server process is single threaded unless system libraries or preloaded imports spawn threads as a side-effect so it is generally safe for it to use `os.fork()`. No unnecessary resources are inherited.

Available on POSIX platforms which support passing file descriptors over Unix pipes such as Linux.

Changed in version 3.4: *spawn* added on all POSIX platforms, and *forkserver* added for some POSIX platforms. Child processes no longer inherit all of the parents inheritable handles on Windows.

Changed in version 3.8: On macOS, the *spawn* start method is now the default. The *fork* start method should be considered unsafe as it can lead to crashes of the subprocess as macOS system libraries may start threads. See bpo-33725.

On POSIX using the *spawn* or *forkserver* start methods will also start a *resource tracker* process which tracks the unlinked named system resources (such as named semaphores or `SharedMemory` objects) created by processes of the program. When all processes have exited the resource tracker unlinks any remaining tracked object. Usually there should be none, but if a process was killed by a signal there may be some "leaked" resources. (Neither leaked semaphores nor shared memory segments will be automatically unlinked until the next reboot. This is problematic for both objects because the system allows only a limited number of named semaphores, and shared memory segments occupy some space in the main memory.)

To select a start method you use the `set_start_method()` in the `if __name__ == '__main__'` clause of the main module. For example:

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    mp.set_start_method('spawn')
    q = mp.Queue()
    p = mp.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

`set_start_method()` should not be used more than once in the program.

Alternatively, you can use `get_context()` to obtain a context object. Context objects have the same API as the multiprocessing module, and allow one to use multiple start methods in the same program.

```
import multiprocessing as mp

def foo(q):
    q.put('hello')
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    ctx = mp.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

Note that objects related to one context may not be compatible with processes for a different context. In particular, locks created using the *fork* context cannot be passed to processes started using the *spawn* or *forkserver* start methods.

A library which wants to use a particular start method should probably use *get_context()* to avoid interfering with the choice of the library user.

Warning

The `'spawn'` and `'forkserver'` start methods generally cannot be used with "frozen" executables (i.e., binaries produced by packages like **PyInstaller** and **cx_Freeze**) on POSIX systems. The `'fork'` start method may work if code does not use threads.

Exchanging objects between processes

multiprocessing supports two types of communication channel between processes:

Queues

The *Queue* class is a near clone of *queue.Queue*. For example:

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

Queues are thread and process safe. Any object put into a *multiprocessing* queue will be serialized.

Pipes

The *Pipe()* function returns a pair of connection objects connected by a pipe which by default is duplex (two-way). For example:

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # prints "[42, None, 'hello']"
    p.join()
```

The two connection objects returned by *Pipe()* represent the two ends of the pipe. Each connection object has `send()` and `recv()` methods (among others). Note that data in a pipe may become corrupted if two processes (or threads) try to read from or write to the *same* end of the pipe at the same time. Of course there is no risk of corruption from processes using different ends of the pipe at the same time.

The `send()` method serializes the the object and `recv()` re-creates the object.

Synchronization between processes

multiprocessing contains equivalents of all the synchronization primitives from *threading*. For instance one can use a lock to ensure that only one process prints to standard output at a time:

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
        l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

Without using the lock output from the different processes is liable to get all mixed up.

Sharing state between processes

As mentioned above, when doing concurrent programming it is usually best to avoid using shared state as far as possible. This is particularly true when using multiple processes.

However, if you really do need to use some shared data then *multiprocessing* provides a couple of ways of doing so.

Shared memory

Data can be stored in a shared memory map using *Value* or *Array*. For example, the following code

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

will print

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

The `'d'` and `'i'` arguments used when creating `num` and `arr` are typecodes of the kind used by the *array* module: `'d'` indicates a double precision float and `'i'` indicates a signed integer. These shared objects will be process and thread-safe.

For more flexibility in using shared memory one can use the *multiprocessing.sharedctypes* module which supports the creation of arbitrary ctypes objects allocated from shared memory.

Server process

A manager object returned by *Manager()* controls a server process which holds Python objects and allows other processes to manipulate them using proxies.

A manager returned by *Manager()* will support types *list*, *dict*, *Namespace*, *Lock*, *RLock*, *Semaphore*, *BoundedSemaphore*, *Condition*, *Event*, *Barrier*, *Queue*, *Value* and *Array*. For example,

```
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))

        p = Process(target=f, args=(d, l))
        p.start()
        p.join()

        print(d)
        print(l)
```

will print

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Server process managers are more flexible than using shared memory objects because they can be made to support arbitrary object types. Also, a single manager can be shared by processes on different computers

over a network. They are, however, slower than using shared memory.

Using a pool of workers

The *Pool* class represents a pool of worker processes. It has methods which allows tasks to be offloaded to the worker processes in a few different ways.

For example:

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    # start 4 worker processes
    with Pool(processes=4) as pool:

        # print "[0, 1, 4,..., 81]"
        print(pool.map(f, range(10)))

        # print same numbers in arbitrary order
        for i in pool.imap_unordered(f, range(10)):
            print(i)

        # evaluate "f(20)" asynchronously
        res = pool.apply_async(f, (20,))      # runs in *only* one process
        print(res.get(timeout=1))             # prints "400"

        # evaluate "os.getpid()" asynchronously
        res = pool.apply_async(os.getpid, ()) # runs in *only* one process
        print(res.get(timeout=1))             # prints the PID of that process

        # launching multiple evaluations asynchronously *may* use more processes
        multiple_results = [pool.apply_async(os.getpid, ()) for i in range(4)]
        print([res.get(timeout=1) for res in multiple_results])

        # make a single worker sleep for 10 seconds
        res = pool.apply_async(time.sleep, (10,))
        try:
            print(res.get(timeout=1))
        except TimeoutError:
            print("We lacked patience and got a multiprocessing.TimeoutError")

        print("For the moment, the pool remains available for more work")

    # exiting the 'with'-block has stopped the pool
    print("Now the pool is closed and no longer available")
```

Note that the methods of a pool should only ever be used by the process which created it.

Note

Functionality within this package requires that the `__main__` module be importable by the children. This is cov-

ered in *Programming guidelines* however it is worth pointing out here. This means that some examples, such as the `multiprocessing.pool.Pool` examples will not work in the interactive interpreter. For example:

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> with p:
...     p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
Traceback (most recent call last):
Traceback (most recent call last):
AttributeError: Can't get attribute 'f' on <module '__main__' (<class '_frozen_
↪importlib.BuiltinImporter'>)>
AttributeError: Can't get attribute 'f' on <module '__main__' (<class '_frozen_
↪importlib.BuiltinImporter'>)>
AttributeError: Can't get attribute 'f' on <module '__main__' (<class '_frozen_
↪importlib.BuiltinImporter'>)>
```

(If you try this it will actually output three full tracebacks interleaved in a semi-random fashion, and then you may have to stop the parent process somehow.)

17.2.2 Reference

The `multiprocessing` package mostly replicates the API of the `threading` module.

`Process` and exceptions

class multiprocessing.**Process**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, ***, *daemon=None*)

Process objects represent activity that is run in a separate process. The `Process` class has equivalents of all the methods of `threading.Thread`.

The constructor should always be called with keyword arguments. *group* should always be `None`; it exists solely for compatibility with `threading.Thread`. *target* is the callable object to be invoked by the `run()` method. It defaults to `None`, meaning nothing is called. *name* is the process name (see `name` for more details). *args* is the argument tuple for the target invocation. *kwargs* is a dictionary of keyword arguments for the target invocation. If provided, the keyword-only *daemon* argument sets the process `daemon` flag to `True` or `False`. If `None` (the default), this flag will be inherited from the creating process.

By default, no arguments are passed to *target*. The *args* argument, which defaults to `()`, can be used to specify a list or tuple of the arguments to pass to *target*.

If a subclass overrides the constructor, it must make sure it invokes the base class constructor (`Process.__init__()`) before doing anything else to the process.

Changed in version 3.3: Added the *daemon* parameter.

run()

Method representing the process's activity.

You may override this method in a subclass. The standard *run()* method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the *args* and *kwargs* arguments, respectively.

Using a list or tuple as the *args* argument passed to *Process* achieves the same effect.

Example:

```
>>> from multiprocessing import Process
>>> p = Process(target=print, args=[1])
>>> p.run()
1
>>> p = Process(target=print, args=(1,))
>>> p.run()
1
```

start()

Start the process's activity.

This must be called at most once per process object. It arranges for the object's *run()* method to be invoked in a separate process.

join([*timeout*])

If the optional argument *timeout* is None (the default), the method blocks until the process whose *join()* method is called terminates. If *timeout* is a positive number, it blocks at most *timeout* seconds. Note that the method returns None if its process terminates or if the method times out. Check the process's *exitcode* to determine if it terminated.

A process can be joined many times.

A process cannot join itself because this would cause a deadlock. It is an error to attempt to join a process before it has been started.

name

The process's name. The name is a string used for identification purposes only. It has no semantics. Multiple processes may be given the same name.

The initial name is set by the constructor. If no explicit name is provided to the constructor, a name of the form 'Process-N_1:N_2:…:N_k' is constructed, where each N_k is the N-th child of its parent.

is_alive()

Return whether the process is alive.

Roughly, a process object is alive from the moment the *start()* method returns until the child process terminates.

daemon

The process's daemon flag, a Boolean value. This must be set before *start()* is called.

The initial value is inherited from the creating process.

When a process exits, it attempts to terminate all of its daemonic child processes.

Note that a daemonic process is not allowed to create child processes. Otherwise a daemonic process would leave its children orphaned if it gets terminated when its parent process exits. Additionally, these are **not** Unix daemons or services, they are normal processes that will be terminated (and not joined) if non-daemonic processes have exited.

In addition to the *threading.Thread* API, *Process* objects also support the following attributes and methods:

pid

Return the process ID. Before the process is spawned, this will be `None`.

exitcode

The child's exit code. This will be `None` if the process has not yet terminated.

If the child's *run()* method returned normally, the exit code will be 0. If it terminated via *sys.exit()* with an integer argument *N*, the exit code will be *N*.

If the child terminated due to an exception not caught within *run()*, the exit code will be 1. If it was terminated by signal *N*, the exit code will be the negative value *-N*.

authkey

The process's authentication key (a byte string).

When *multiprocessing* is initialized the main process is assigned a random string using *os.urandom()*.

When a *Process* object is created, it will inherit the authentication key of its parent process, although this may be changed by setting *authkey* to another byte string.

See *Authentication keys*.

sentinel

A numeric handle of a system object which will become "ready" when the process ends.

You can use this value if you want to wait on several events at once using *multiprocessing.connection.wait()*. Otherwise calling *join()* is simpler.

On Windows, this is an OS handle usable with the `WaitForSingleObject` and `WaitForMultipleObjects` family of API calls. On POSIX, this is a file descriptor usable with primitives from the *select* module.

Added in version 3.3.

terminate()

Terminate the process. On POSIX this is done using the *SIGTERM* signal; on Windows `TerminateProcess()` is used. Note that exit handlers and finally clauses, etc., will not be executed.

Note that descendant processes of the process will *not* be terminated – they will simply become orphaned.

> **Warning**
>
> If this method is used when the associated process is using a pipe or queue then the pipe or queue is liable to become corrupted and may become unusable by other process. Similarly, if the process has acquired a lock or semaphore etc. then terminating it is liable to cause other processes to deadlock.

kill()

Same as *terminate()* but using the `SIGKILL` signal on POSIX.

Added in version 3.7.

close()

Close the *Process* object, releasing all resources associated with it. *ValueError* is raised if the underlying process is still running. Once *close()* returns successfully, most other methods and attributes of the *Process* object will raise *ValueError*.

Added in version 3.7.

Note that the *start()*, *join()*, *is_alive()*, *terminate()* and *exitcode* methods should only be called by the process that created the process object.

Example usage of some of the methods of *Process*:

```
>>> import multiprocessing, time, signal
>>> mp_context = multiprocessing.get_context('spawn')
>>> p = mp_context.Process(target=time.sleep, args=(1000,))
>>> print(p, p.is_alive())
<...Process ... initial> False
>>> p.start()
>>> print(p, p.is_alive())
<...Process ... started> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print(p, p.is_alive())
<...Process ... stopped exitcode=-SIGTERM> False
>>> p.exitcode == -signal.SIGTERM
True
```

exception multiprocessing.**ProcessError**

The base class of all *multiprocessing* exceptions.

exception multiprocessing.**BufferTooShort**

Exception raised by `Connection.recv_bytes_into()` when the supplied buffer object is too small for the message read.

If `e` is an instance of *BufferTooShort* then `e.args[0]` will give the message as a byte string.

exception multiprocessing.**AuthenticationError**

Raised when there is an authentication error.

exception multiprocessing.**TimeoutError**

Raised by methods with a timeout when the timeout expires.

Pipes and Queues

When using multiple processes, one generally uses message passing for communication between processes and avoids having to use any synchronization primitives like locks.

For passing messages one can use *Pipe()* (for a connection between two processes) or a queue (which allows multiple producers and consumers).

The *Queue*, *SimpleQueue* and *JoinableQueue* types are multi-producer, multi-consumer FIFO queues modelled on the *queue.Queue* class in the standard library. They differ in that *Queue* lacks the *task_done()* and *join()* methods introduced into Python 2.5's *queue.Queue* class.

If you use *JoinableQueue* then you **must** call *JoinableQueue.task_done()* for each task removed from the queue or else the semaphore used to count the number of unfinished tasks may eventually overflow, raising an exception.

One difference from other Python queue implementations, is that *multiprocessing* queues serializes all objects that are put into them using *pickle*. The object return by the get method is a re-created object that does not share memory with the original object.

Note that one can also create a shared queue by using a manager object – see *Managers*.

Note

multiprocessing uses the usual *queue.Empty* and *queue.Full* exceptions to signal a timeout. They are not available in the *multiprocessing* namespace so you need to import them from *queue*.

Note

When an object is put on a queue, the object is pickled and a background thread later flushes the pickled data to an underlying pipe. This has some consequences which are a little surprising, but should not cause any practical difficulties – if they really bother you then you can instead use a queue created with a *manager*.

(1) After putting an object on an empty queue there may be an infinitesimal delay before the queue's *empty()* method returns *False* and *get_nowait()* can return without raising *queue.Empty*.

(2) If multiple processes are enqueuing objects, it is possible for the objects to be received at the other end out-of-order. However, objects enqueued by the same process will always be in the expected order with respect to each other.

Warning

If a process is killed using *Process.terminate()* or *os.kill()* while it is trying to use a *Queue*, then the data in the queue is likely to become corrupted. This may cause any other process to get an exception when it tries to use the queue later on.

Warning

As mentioned above, if a child process has put items on a queue (and it has not used *JoinableQueue.cancel_join_thread*), then that process will not terminate until all buffered items have been flushed to the pipe.

This means that if you try joining that process you may get a deadlock unless you are sure that all items which have been put on the queue have been consumed. Similarly, if the child process is non-daemonic then the parent process may hang on exit when it tries to join all its non-daemonic children.

Note that a queue created using a manager does not have this issue. See *Programming guidelines*.

For an example of the usage of queues for interprocess communication see *Examples*.

`multiprocessing.`**`Pipe`**`(`[*duplex*]`)`

Returns a pair `(conn1, conn2)` of *Connection* objects representing the ends of a pipe.

If *duplex* is `True` (the default) then the pipe is bidirectional. If *duplex* is `False` then the pipe is unidirectional: `conn1` can only be used for receiving messages and `conn2` can only be used for sending messages.

The `send()` method serializes the the object using *pickle* and the `recv()` re-creates the object.

`class` `multiprocessing.`**`Queue`**`(`[*maxsize*]`)`

Returns a process shared queue implemented using a pipe and a few locks/semaphores. When a process first puts an item on the queue a feeder thread is started which transfers objects from a buffer into the pipe.

The usual *queue.Empty* and *queue.Full* exceptions from the standard library's *queue* module are raised to signal timeouts.

Queue implements all the methods of *queue.Queue* except for *task_done()* and *join()*.

qsize()

Return the approximate size of the queue. Because of multithreading/multiprocessing semantics, this number is not reliable.

Note that this may raise *NotImplementedError* on platforms like macOS where `sem_getvalue()` is not implemented.

empty()

Return `True` if the queue is empty, `False` otherwise. Because of multithreading/multiprocessing semantics, this is not reliable.

May raise an *OSError* on closed queues. (not guaranteed)

full()

Return `True` if the queue is full, `False` otherwise. Because of multithreading/multiprocessing semantics, this is not reliable.

put(*obj*[, *block*[, *timeout*]])

Put obj into the queue. If the optional argument *block* is `True` (the default) and *timeout* is `None` (the default), block if necessary until a free slot is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *queue.Full* exception if no free slot was available within that time. Otherwise (*block* is `False`), put an item on the queue if a free slot is immediately available, else raise the *queue.Full* exception (*timeout* is ignored in that case).

Changed in version 3.8: If the queue is closed, *ValueError* is raised instead of *AssertionError*.

put_nowait(*obj*)

Equivalent to `put(obj, False)`.

get([*block*[, *timeout*]])

Remove and return an item from the queue. If optional args *block* is `True` (the default) and *timeout* is `None` (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *queue.Empty* exception if no item was available within that time. Otherwise (block is `False`), return an item if one is immediately available, else raise the *queue.Empty* exception (*timeout* is ignored in that case).

Changed in version 3.8: If the queue is closed, *ValueError* is raised instead of *OSError*.

get_nowait()

Equivalent to `get(False)`.

multiprocessing.Queue has a few additional methods not found in *queue.Queue*. These methods are usually unnecessary for most code:

close()

Indicate that no more data will be put on this queue by the current process. The background thread will quit once it has flushed all buffered data to the pipe. This is called automatically when the queue is garbage collected.

join_thread()

Join the background thread. This can only be used after *close()* has been called. It blocks until the background thread exits, ensuring that all data in the buffer has been flushed to the pipe.

By default if a process is not the creator of the queue then on exit it will attempt to join the queue's background thread. The process can call *cancel_join_thread()* to make *join_thread()* do nothing.

cancel_join_thread()

Prevent *join_thread()* from blocking. In particular, this prevents the background thread from being joined automatically when the process exits – see *join_thread()*.

A better name for this method might be `allow_exit_without_flush()`. It is likely to cause enqueued data to be lost, and you almost certainly will not need to use it. It is really only there if you need the current process to exit immediately without waiting to flush enqueued data to the underlying pipe, and you don't care about lost data.

Note

This class's functionality requires a functioning shared semaphore implementation on the host operating system. Without one, the functionality in this class will be disabled, and attempts to instantiate a *Queue* will result in an *ImportError*. See bpo-3770 for additional information. The same holds true for any of the specialized queue types listed below.

class multiprocessing.**SimpleQueue**

It is a simplified *Queue* type, very close to a locked *Pipe*.

close()

Close the queue: release internal resources.

A queue must not be used anymore after it is closed. For example, *get()*, *put()* and *empty()* methods must no longer be called.

Added in version 3.9.

empty()

Return `True` if the queue is empty, `False` otherwise.

Always raises an *OSError* if the SimpleQueue is closed.

get()

Remove and return an item from the queue.

put(*item*)

Put *item* into the queue.

class multiprocessing.**JoinableQueue**([*maxsize*])

JoinableQueue, a *Queue* subclass, is a queue which additionally has *task_done()* and *join()* methods.

task_done()

Indicate that a formerly enqueued task is complete. Used by queue consumers. For each *get()* used to fetch a task, a subsequent call to *task_done()* tells the queue that the processing on the task is complete.

If a *join()* is currently blocking, it will resume when all items have been processed (meaning that a *task_done()* call was received for every item that had been *put()* into the queue).

Raises a *ValueError* if called more times than there were items placed in the queue.

join()

Block until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer calls *task_done()* to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, *join()* unblocks.

Miscellaneous

multiprocessing.**active_children**()

Return list of all live children of the current process.

Calling this has the side effect of "joining" any processes which have already finished.

multiprocessing.**cpu_count**()

Return the number of CPUs in the system.

This number is not equivalent to the number of CPUs the current process can use. The number of usable CPUs can be obtained with *os.process_cpu_count()* (or `len(os.sched_getaffinity(0))`).

When the number of CPUs cannot be determined a *NotImplementedError* is raised.

See also

os.cpu_count() *os.process_cpu_count()*

Changed in version 3.13: The return value can also be overridden using the `-X cpu_count` flag or `PYTHON_CPU_COUNT` as this is merely a wrapper around the *os* cpu count APIs.

multiprocessing.**current_process**()

Return the *Process* object corresponding to the current process.

An analogue of *threading.current_thread()*.

multiprocessing.**parent_process**()

Return the *Process* object corresponding to the parent process of the *current_process()*. For the main process, `parent_process` will be `None`.

Added in version 3.8.

multiprocessing.**freeze_support**()

Add support for when a program which uses *multiprocessing* has been frozen to produce a Windows executable. (Has been tested with **py2exe**, **PyInstaller** and **cx_Freeze**.)

One needs to call this function straight after the `if __name__ == '__main__'` line of the main module. For example:

```
from multiprocessing import Process, freeze_support

def f():
    print('hello world!')

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

If the `freeze_support()` line is omitted then trying to run the frozen executable will raise *RuntimeError*.

Calling `freeze_support()` has no effect when invoked on any operating system other than Windows. In addition, if the module is being run normally by the Python interpreter on Windows (the program has not been frozen), then `freeze_support()` has no effect.

multiprocessing.**get_all_start_methods**()

Returns a list of the supported start methods, the first of which is the default. The possible start methods are `'fork'`, `'spawn'` and `'forkserver'`. Not all platforms support all methods. See *Contexts and start methods*.

Added in version 3.4.

multiprocessing.**get_context**(*method=None*)

Return a context object which has the same attributes as the *multiprocessing* module.

If *method* is None then the default context is returned. Otherwise *method* should be 'fork', 'spawn', 'forkserver'. *ValueError* is raised if the specified start method is not available. See *Contexts and start methods*.

Added in version 3.4.

multiprocessing.**get_start_method**(*allow_none=False*)

Return the name of start method used for starting processes.

If the start method has not been fixed and *allow_none* is false, then the start method is fixed to the default and the name is returned. If the start method has not been fixed and *allow_none* is true then None is returned.

The return value can be 'fork', 'spawn', 'forkserver' or None. See *Contexts and start methods*.

Added in version 3.4.

Changed in version 3.8: On macOS, the *spawn* start method is now the default. The *fork* start method should be considered unsafe as it can lead to crashes of the subprocess. See bpo-33725.

multiprocessing.**set_executable**(*executable*)

Set the path of the Python interpreter to use when starting a child process. (By default *sys.executable* is used). Embedders will probably need to do some thing like

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

before they can create child processes.

Changed in version 3.4: Now supported on POSIX when the 'spawn' start method is used.

Changed in version 3.11: Accepts a *path-like object*.

multiprocessing.**set_forkserver_preload**(*module_names*)

Set a list of module names for the forkserver main process to attempt to import so that their already imported state is inherited by forked processes. Any *ImportError* when doing so is silently ignored. This can be used as a performance enhancement to avoid repeated work in every process.

For this to work, it must be called before the forkserver process has been launched (before creating a Pool or starting a *Process*).

Only meaningful when using the 'forkserver' start method. See *Contexts and start methods*.

Added in version 3.4.

multiprocessing.**set_start_method**(*method*, *force=False*)

Set the method which should be used to start child processes. The *method* argument can be 'fork', 'spawn' or 'forkserver'. Raises *RuntimeError* if the start method has already been set and *force* is not True. If *method* is None and *force* is True then the start method is set to None. If *method* is None and *force* is False then the context is set to the default context.

Note that this should be called at most once, and it should be protected inside the if __name__ == '__main__' clause of the main module.

See *Contexts and start methods*.

Added in version 3.4.

Note

multiprocessing contains no analogues of *threading.active_count()*, *threading.enumerate()*, *threading.settrace()*, *threading.setprofile()*, *threading.Timer*, or *threading.local*.

Connection Objects

Connection objects allow the sending and receiving of picklable objects or strings. They can be thought of as message oriented connected sockets.

Connection objects are usually created using *Pipe* – see also *Listeners and Clients*.

class multiprocessing.connection.**Connection**

send(*obj*)

Send an object to the other end of the connection which should be read using *recv()*.

The object must be picklable. Very large pickles (approximately 32 MiB+, though it depends on the OS) may raise a *ValueError* exception.

recv()

Return an object sent from the other end of the connection using *send()*. Blocks until there is something to receive. Raises *EOFError* if there is nothing left to receive and the other end was closed.

fileno()

Return the file descriptor or handle used by the connection.

close()

Close the connection.

This is called automatically when the connection is garbage collected.

poll([*timeout*])

Return whether there is any data available to be read.

If *timeout* is not specified then it will return immediately. If *timeout* is a number then this specifies the maximum time in seconds to block. If *timeout* is None then an infinite timeout is used.

Note that multiple connection objects may be polled at once by using *multiprocessing.connection.wait()*.

send_bytes(*buffer*[, *offset*[, *size*]])

Send byte data from a *bytes-like object* as a complete message.

If *offset* is given then data is read from that position in *buffer*. If *size* is given then that many bytes will be read from buffer. Very large buffers (approximately 32 MiB+, though it depends on the OS) may raise a *ValueError* exception

recv_bytes([*maxlength*])

Return a complete message of byte data sent from the other end of the connection as a string. Blocks until there is something to receive. Raises *EOFError* if there is nothing left to receive and the other end has closed.

If *maxlength* is specified and the message is longer than *maxlength* then *OSError* is raised and the connection will no longer be readable.

Changed in version 3.3: This function used to raise *IOError*, which is now an alias of *OSError*.

recv_bytes_into(*buffer*[, *offset*])

Read into *buffer* a complete message of byte data sent from the other end of the connection and return the number of bytes in the message. Blocks until there is something to receive. Raises *EOFError* if there is nothing left to receive and the other end was closed.

buffer must be a writable *bytes-like object*. If *offset* is given then the message will be written into the buffer from that position. Offset must be a non-negative integer less than the length of *buffer* (in bytes).

If the buffer is too short then a `BufferTooShort` exception is raised and the complete message is available as `e.args[0]` where `e` is the exception instance.

Changed in version 3.3: Connection objects themselves can now be transferred between processes using *Connection.send()* and *Connection.recv()*.

Connection objects also now support the context management protocol – see *Context Manager Types*. *__enter__()* returns the connection object, and *__exit__()* calls *close()*.

For example:

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes(b'thank you')
>>> a.recv_bytes()
b'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

Warning

The *Connection.recv()* method automatically unpickles the data it receives, which can be a security risk unless you can trust the process which sent the message.

Therefore, unless the connection object was produced using `Pipe()` you should only use the *recv()* and *send()* methods after performing some sort of authentication. See *Authentication keys*.

Warning

If a process is killed while it is trying to read or write to a pipe then the data in the pipe is likely to become corrupted, because it may become impossible to be sure where the message boundaries lie.

Synchronization primitives

Generally synchronization primitives are not as necessary in a multiprocess program as they are in a multithreaded program. See the documentation for *threading* module.

Note that one can also create synchronization primitives by using a manager object – see *Managers*.

class multiprocessing.**Barrier**(*parties*[, *action*[, *timeout*]])

A barrier object: a clone of *threading.Barrier*.

Added in version 3.3.

class multiprocessing.**BoundedSemaphore**([*value*])

A bounded semaphore object: a close analog of *threading.BoundedSemaphore*.

A solitary difference from its close analog exists: its `acquire` method's first argument is named *block*, as is consistent with *Lock.acquire()*.

> **Note**
>
> On macOS, this is indistinguishable from *Semaphore* because `sem_getvalue()` is not implemented on that platform.

class multiprocessing.**Condition**([*lock*])

A condition variable: an alias for *threading.Condition*.

If *lock* is specified then it should be a *Lock* or *RLock* object from *multiprocessing*.

Changed in version 3.3: The *wait_for()* method was added.

class multiprocessing.**Event**

A clone of *threading.Event*.

class multiprocessing.**Lock**

A non-recursive lock object: a close analog of *threading.Lock*. Once a process or thread has acquired a lock, subsequent attempts to acquire it from any process or thread will block until it is released; any process or thread may release it. The concepts and behaviors of *threading.Lock* as it applies to threads are replicated here in *multiprocessing.Lock* as it applies to either processes or threads, except as noted.

Note that *Lock* is actually a factory function which returns an instance of `multiprocessing.synchronize.Lock` initialized with a default context.

Lock supports the *context manager* protocol and thus may be used in `with` statements.

acquire(*block=True*, *timeout=None*)

Acquire a lock, blocking or non-blocking.

With the *block* argument set to `True` (the default), the method call will block until the lock is in an unlocked state, then set it to locked and return `True`. Note that the name of this first argument differs from that in *threading.Lock.acquire()*.

With the *block* argument set to `False`, the method call does not block. If the lock is currently in a locked state, return `False`; otherwise set the lock to a locked state and return `True`.

When invoked with a positive, floating-point value for *timeout*, block for at most the number of seconds specified by *timeout* as long as the lock can not be acquired. Invocations with a negative value for *timeout* are equivalent to a *timeout* of zero. Invocations with a *timeout* value of `None` (the default) set the timeout period to infinite. Note that the treatment of negative or `None` values for *timeout* differs from the implemented behavior in *threading.Lock.acquire()*. The *timeout* argument has no practical implications if the

block argument is set to `False` and is thus ignored. Returns `True` if the lock has been acquired or `False` if the timeout period has elapsed.

release()

Release a lock. This can be called from any process or thread, not only the process or thread which originally acquired the lock.

Behavior is the same as in *threading.Lock.release()* except that when invoked on an unlocked lock, a *ValueError* is raised.

class multiprocessing.**RLock**

A recursive lock object: a close analog of *threading.RLock*. A recursive lock must be released by the process or thread that acquired it. Once a process or thread has acquired a recursive lock, the same process or thread may acquire it again without blocking; that process or thread must release it once for each time it has been acquired.

Note that *RLock* is actually a factory function which returns an instance of `multiprocessing.synchronize.RLock` initialized with a default context.

RLock supports the *context manager* protocol and thus may be used in `with` statements.

acquire(*block=True*, *timeout=None*)

Acquire a lock, blocking or non-blocking.

When invoked with the *block* argument set to `True`, block until the lock is in an unlocked state (not owned by any process or thread) unless the lock is already owned by the current process or thread. The current process or thread then takes ownership of the lock (if it does not already have ownership) and the recursion level inside the lock increments by one, resulting in a return value of `True`. Note that there are several differences in this first argument's behavior compared to the implementation of *threading.RLock.acquire()*, starting with the name of the argument itself.

When invoked with the *block* argument set to `False`, do not block. If the lock has already been acquired (and thus is owned) by another process or thread, the current process or thread does not take ownership and the recursion level within the lock is not changed, resulting in a return value of `False`. If the lock is in an unlocked state, the current process or thread takes ownership and the recursion level is incremented, resulting in a return value of `True`.

Use and behaviors of the *timeout* argument are the same as in *Lock.acquire()*. Note that some of these behaviors of *timeout* differ from the implemented behaviors in *threading.RLock.acquire()*.

release()

Release a lock, decrementing the recursion level. If after the decrement the recursion level is zero, reset the lock to unlocked (not owned by any process or thread) and if any other processes or threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling process or thread.

Only call this method when the calling process or thread owns the lock. An *AssertionError* is raised if this method is called by a process or thread other than the owner or if the lock is in an unlocked (unowned) state. Note that the type of exception raised in this situation differs from the implemented behavior in *threading.RLock.release()*.

class multiprocessing.**Semaphore**([*value*])

A semaphore object: a close analog of *threading.Semaphore*.

A solitary difference from its close analog exists: its `acquire` method's first argument is named *block*, as is consistent with *Lock.acquire()*.

Note

On macOS, `sem_timedwait` is unsupported, so calling `acquire()` with a timeout will emulate that function's behavior using a sleeping loop.

Note

Some of this package's functionality requires a functioning shared semaphore implementation on the host operating system. Without one, the `multiprocessing.synchronize` module will be disabled, and attempts to import it will result in an *ImportError*. See bpo-3770 for additional information.

Shared `ctypes` Objects

It is possible to create shared objects using shared memory which can be inherited by child processes.

`multiprocessing.`**`Value`**(*typecode_or_type*, **args*, *lock=True*)

Return a *ctypes* object allocated from shared memory. By default the return value is actually a synchronized wrapper for the object. The object itself can be accessed via the *value* attribute of a *Value*.

typecode_or_type determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the *array* module. **args* is passed on to the constructor for the type.

If *lock* is `True` (the default) then a new recursive lock object is created to synchronize access to the value. If *lock* is a *Lock* or *RLock* object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be "process-safe".

Operations like += which involve a read and write are not atomic. So if, for instance, you want to atomically increment a shared value it is insufficient to just do

```
counter.value += 1
```

Assuming the associated lock is recursive (which it is by default) you can instead do

```
with counter.get_lock():
    counter.value += 1
```

Note that *lock* is a keyword-only argument.

`multiprocessing.`**`Array`**(*typecode_or_type*, *size_or_initializer*, ***, *lock=True*)

Return a ctypes array allocated from shared memory. By default the return value is actually a synchronized wrapper for the array.

typecode_or_type determines the type of the elements of the returned array: it is either a ctypes type or a one character typecode of the kind used by the *array* module. If *size_or_initializer* is an integer, then it determines the length of the array, and the array will be initially zeroed. Otherwise, *size_or_initializer* is a sequence which is used to initialize the array and whose length determines the length of the array.

If *lock* is `True` (the default) then a new lock object is created to synchronize access to the value. If *lock* is a *Lock* or *RLock* object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be "process-safe".

Note that *lock* is a keyword only argument.

Note that an array of *ctypes.c_char* has *value* and *raw* attributes which allow one to use it to store and retrieve strings.

The `multiprocessing.sharedctypes` module

The *multiprocessing.sharedctypes* module provides functions for allocating *ctypes* objects from shared memory which can be inherited by child processes.

Note

Although it is possible to store a pointer in shared memory remember that this will refer to a location in the address space of a specific process. However, the pointer is quite likely to be invalid in the context of a second process and trying to dereference the pointer from the second process may cause a crash.

multiprocessing.sharedctypes.**RawArray**(*typecode_or_type*, *size_or_initializer*)

Return a ctypes array allocated from shared memory.

typecode_or_type determines the type of the elements of the returned array: it is either a ctypes type or a one character typecode of the kind used by the *array* module. If *size_or_initializer* is an integer then it determines the length of the array, and the array will be initially zeroed. Otherwise *size_or_initializer* is a sequence which is used to initialize the array and whose length determines the length of the array.

Note that setting and getting an element is potentially non-atomic – use *Array()* instead to make sure that access is automatically synchronized using a lock.

multiprocessing.sharedctypes.**RawValue**(*typecode_or_type*, **args*)

Return a ctypes object allocated from shared memory.

typecode_or_type determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the *array* module. **args* is passed on to the constructor for the type.

Note that setting and getting the value is potentially non-atomic – use *Value()* instead to make sure that access is automatically synchronized using a lock.

Note that an array of *ctypes.c_char* has `value` and `raw` attributes which allow one to use it to store and retrieve strings – see documentation for *ctypes*.

multiprocessing.sharedctypes.**Array**(*typecode_or_type*, *size_or_initializer*, *, *lock=True*)

The same as *RawArray()* except that depending on the value of *lock* a process-safe synchronization wrapper may be returned instead of a raw ctypes array.

If *lock* is `True` (the default) then a new lock object is created to synchronize access to the value. If *lock* is a *Lock* or *RLock* object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be "process-safe".

Note that *lock* is a keyword-only argument.

multiprocessing.sharedctypes.**Value**(*typecode_or_type*, **args*, *lock=True*)

The same as *RawValue()* except that depending on the value of *lock* a process-safe synchronization wrapper may be returned instead of a raw ctypes object.

If *lock* is `True` (the default) then a new lock object is created to synchronize access to the value. If *lock* is a *Lock* or *RLock* object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be "process-safe".

Note that *lock* is a keyword-only argument.

multiprocessing.sharedctypes.**copy**(*obj*)

Return a ctypes object allocated from shared memory which is a copy of the ctypes object *obj*.

multiprocessing.sharedctypes.**synchronized**(*obj*[, *lock*])

Return a process-safe wrapper object for a ctypes object which uses *lock* to synchronize access. If *lock* is None (the default) then a *multiprocessing.RLock* object is created automatically.

A synchronized wrapper will have two methods in addition to those of the object it wraps: get_obj() returns the wrapped object and get_lock() returns the lock object used for synchronization.

Note that accessing the ctypes object through the wrapper can be a lot slower than accessing the raw ctypes object.

Changed in version 3.5: Synchronized objects support the *context manager* protocol.

The table below compares the syntax for creating shared ctypes objects from shared memory with the normal ctypes syntax. (In the table MyStruct is some subclass of *ctypes.Structure*.)

ctypes	sharedctypes using type	sharedctypes using typecode
c_double(2.4)	RawValue(c_double, 2.4)	RawValue('d', 2.4)
MyStruct(4, 6)	RawValue(MyStruct, 4, 6)	
(c_short * 7)()	RawArray(c_short, 7)	RawArray('h', 7)
(c_int * 3)(9, 2, 8)	RawArray(c_int, (9, 2, 8))	RawArray('i', (9, 2, 8))

Below is an example where a number of ctypes objects are modified by a child process:

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', b'hello world', lock=lock)
    A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)], lock=lock)

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()

    print(n.value)
    print(x.value)
    print(s.value)
    print([(a.x, a.y) for a in A])
```

The results printed are

```
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

Managers

Managers provide a way to create data which can be shared between different processes, including sharing over a network between processes running on different machines. A manager object controls a server process which manages *shared objects*. Other processes can access the shared objects by using proxies.

`multiprocessing.`**`Manager`**`()`

Returns a started *SyncManager* object which can be used for sharing objects between processes. The returned manager object corresponds to a spawned child process and has methods which will create shared objects and return corresponding proxies.

Manager processes will be shutdown as soon as they are garbage collected or their parent process exits. The manager classes are defined in the *multiprocessing.managers* module:

class `multiprocessing.managers.`**`BaseManager`**(*address=None*, *authkey=None*, *serializer='pickle'*, *ctx=None*, ***, *shutdown_timeout=1.0*)

Create a BaseManager object.

Once created one should call *start()* or `get_server().serve_forever()` to ensure that the manager object refers to a started manager process.

address is the address on which the manager process listens for new connections. If *address* is `None` then an arbitrary one is chosen.

authkey is the authentication key which will be used to check the validity of incoming connections to the server process. If *authkey* is `None` then `current_process().authkey` is used. Otherwise *authkey* is used and it must be a byte string.

serializer must be `'pickle'` (use *pickle* serialization) or `'xmlrpclib'` (use *xmlrpc.client* serialization).

ctx is a context object, or `None` (use the current context). See the `get_context()` function.

shutdown_timeout is a timeout in seconds used to wait until the process used by the manager completes in the *shutdown()* method. If the shutdown times out, the process is terminated. If terminating the process also times out, the process is killed.

Changed in version 3.11: Added the *shutdown_timeout* parameter.

`start`(*[initializer[, initargs]]*)

Start a subprocess to start the manager. If *initializer* is not `None` then the subprocess will call `initializer(*initargs)` when it starts.

`get_server`()

Returns a `Server` object which represents the actual server under the control of the Manager. The `Server` object supports the `serve_forever()` method:

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('', 50000), authkey=b'abc')
>>> server = manager.get_server()
>>> server.serve_forever()
```

`Server` additionally has an *address* attribute.

connect()

Connect a local manager object to a remote manager process:

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 50000), authkey=b'abc')
>>> m.connect()
```

shutdown()

Stop the process used by the manager. This is only available if *start()* has been used to start the server process.

This can be called multiple times.

register(*typeid*[, *callable*[, *proxytype*[, *exposed*[, *method_to_typeid*[, *create_method*]]]]])

A classmethod which can be used for registering a type or callable with the manager class.

typeid is a "type identifier" which is used to identify a particular type of shared object. This must be a string.

callable is a callable used for creating objects for this type identifier. If a manager instance will be connected to the server using the *connect()* method, or if the *create_method* argument is `False` then this can be left as `None`.

proxytype is a subclass of *BaseProxy* which is used to create proxies for shared objects with this *typeid*. If `None` then a proxy class is created automatically.

exposed is used to specify a sequence of method names which proxies for this typeid should be allowed to access using *BaseProxy._callmethod()*. (If *exposed* is `None` then `proxytype._exposed_` is used instead if it exists.) In the case where no exposed list is specified, all "public methods" of the shared object will be accessible. (Here a "public method" means any attribute which has a `__call__()` method and whose name does not begin with `'_'`.)

method_to_typeid is a mapping used to specify the return type of those exposed methods which should return a proxy. It maps method names to typeid strings. (If *method_to_typeid* is `None` then `proxytype._method_to_typeid_` is used instead if it exists.) If a method's name is not a key of this mapping or if the mapping is `None` then the object returned by the method will be copied by value.

create_method determines whether a method should be created with name *typeid* which can be used to tell the server process to create a new shared object and return a proxy for it. By default it is `True`.

BaseManager instances also have one read-only property:

address

The address used by the manager.

Changed in version 3.3: Manager objects support the context management protocol – see *Context Manager Types*. *__enter__()* starts the server process (if it has not already started) and then returns the manager object. *__exit__()* calls *shutdown()*.

In previous versions *__enter__()* did not start the manager's server process if it was not already started.

class multiprocessing.managers.**SyncManager**

A subclass of *BaseManager* which can be used for the synchronization of processes. Objects of this type are returned by *multiprocessing.Manager()*.

Its methods create and return *Proxy Objects* for a number of commonly used data types to be synchronized across processes. This notably includes shared lists and dictionaries.

Barrier(*parties*[, *action*[, *timeout*]])

Create a shared *threading.Barrier* object and return a proxy for it.

Added in version 3.3.

BoundedSemaphore([*value*])

Create a shared *threading.BoundedSemaphore* object and return a proxy for it.

Condition([*lock*])

Create a shared *threading.Condition* object and return a proxy for it.

If *lock* is supplied then it should be a proxy for a *threading.Lock* or *threading.RLock* object.

Changed in version 3.3: The *wait_for()* method was added.

Event()

Create a shared *threading.Event* object and return a proxy for it.

Lock()

Create a shared *threading.Lock* object and return a proxy for it.

Namespace()

Create a shared *Namespace* object and return a proxy for it.

Queue([*maxsize*])

Create a shared *queue.Queue* object and return a proxy for it.

RLock()

Create a shared *threading.RLock* object and return a proxy for it.

Semaphore([*value*])

Create a shared *threading.Semaphore* object and return a proxy for it.

Array(*typecode*, *sequence*)

Create an array and return a proxy for it.

Value(*typecode*, *value*)

Create an object with a writable `value` attribute and return a proxy for it.

dict()

dict(*mapping*)

dict(*sequence*)

Create a shared *dict* object and return a proxy for it.

list()

list(*sequence*)

Create a shared *list* object and return a proxy for it.

Changed in version 3.6: Shared objects are capable of being nested. For example, a shared container object such as a shared list can contain other shared objects which will all be managed and synchronized by the *SyncManager*.

class multiprocessing.managers.**Namespace**

A type that can register with *SyncManager*.

A namespace object has no public methods, but does have writable attributes. Its representation shows the values of its attributes.

However, when using a proxy for a namespace object, an attribute beginning with `'_'` will be an attribute of the proxy and not an attribute of the referent:

```
>>> mp_context = multiprocessing.get_context('spawn')
>>> manager = mp_context.Manager()
>>> Global = manager.Namespace()
```

(continues on next page)

(continued from previous page)

```
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3    # this is an attribute of the proxy
>>> print(Global)
Namespace(x=10, y='hello')
```

Customized managers

To create one's own manager, one creates a subclass of *BaseManager* and uses the *register()* classmethod to register new types or callables with the manager class. For example:

```
from multiprocessing.managers import BaseManager

class MathsClass:
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    with MyManager() as manager:
        maths = manager.Maths()
        print(maths.add(4, 3))         # prints 7
        print(maths.mul(7, 8))         # prints 56
```

Using a remote manager

It is possible to run a manager server on one machine and have clients use it from other machines (assuming that the firewalls involved allow it).

Running the following commands creates a server for a single shared queue which remote clients can access:

```
>>> from multiprocessing.managers import BaseManager
>>> from queue import Queue
>>> queue = Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda:queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

One client can access the server as follows:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
```

(continues on next page)

(continued from previous page)

```
>>> queue = m.get_queue()
>>> queue.put('hello')
```

Another client can also use it:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

Local processes can also access that queue, using the code from above on the client to access it remotely:

```
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super().__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
...
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

Proxy Objects

A proxy is an object which *refers* to a shared object which lives (presumably) in a different process. The shared object is said to be the *referent* of the proxy. Multiple proxy objects may have the same referent.

A proxy object has methods which invoke corresponding methods of its referent (although not every method of the referent will necessarily be available through the proxy). In this way, a proxy can be used just like its referent can:

```
>>> mp_context = multiprocessing.get_context('spawn')
>>> manager = mp_context.Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print(l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print(repr(l))
<ListProxy object, typeid 'list' at 0x...>
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

Notice that applying *str()* to a proxy will return the representation of the referent, whereas applying *repr()* will return the representation of the proxy.

An important feature of proxy objects is that they are picklable so they can be passed between processes. As such, a referent can contain *Proxy Objects*. This permits nesting of these managed lists, dicts, and other *Proxy Objects*:

```
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)         # referent of a now contains referent of b
>>> print(a, b)
[<ListProxy object, typeid 'list' at ...>] []
>>> b.append('hello')
>>> print(a[0], b)
['hello'] ['hello']
```

Similarly, dict and list proxies may be nested inside one another:

```
>>> l_outer = manager.list([ manager.dict() for i in range(2) ])
>>> d_first_inner = l_outer[0]
>>> d_first_inner['a'] = 1
>>> d_first_inner['b'] = 2
>>> l_outer[1]['c'] = 3
>>> l_outer[1]['z'] = 26
>>> print(l_outer[0])
{'a': 1, 'b': 2}
>>> print(l_outer[1])
{'c': 3, 'z': 26}
```

If standard (non-proxy) *list* or *dict* objects are contained in a referent, modifications to those mutable values will not be propagated through the manager because the proxy has no way of knowing when the values contained within are modified. However, storing a value in a container proxy (which triggers a `__setitem__` on the proxy object) does propagate through the manager and so to effectively modify such an item, one could re-assign the modified value to the container proxy:

```
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# at this point, the changes to d are not yet synced, but by
# updating the dictionary, the proxy is notified of the change
lproxy[0] = d
```

This approach is perhaps less convenient than employing nested *Proxy Objects* for most use cases but also demonstrates a level of control over the synchronization.

Note

The proxy types in *multiprocessing* do nothing to support comparisons by value. So, for instance, we have:

```
>>> manager.list([1,2,3]) == [1,2,3]
False
```

One should just use a copy of the referent instead when making comparisons.

class multiprocessing.managers.**BaseProxy**

Proxy objects are instances of subclasses of *BaseProxy*.

_callmethod(*methodname*[, *args*[, *kwds*]])

Call and return the result of a method of the proxy's referent.

If `proxy` is a proxy whose referent is `obj` then the expression

```
proxy._callmethod(methodname, args, kwds)
```

will evaluate the expression

```
getattr(obj, methodname)(*args, **kwds)
```

in the manager's process.

The returned value will be a copy of the result of the call or a proxy to a new shared object – see documentation for the *method_to_typeid* argument of *BaseManager.register()*.

If an exception is raised by the call, then is re-raised by *_callmethod()*. If some other exception is raised in the manager's process then this is converted into a `RemoteError` exception and is raised by *_callmethod()*.

Note in particular that an exception will be raised if *methodname* has not been *exposed*.

An example of the usage of *_callmethod()*:

```
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getitem__', (slice(2, 7),)) # equivalent to l[2:7]
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))          # equivalent to l[20]
Traceback (most recent call last):
...
IndexError: list index out of range
```

_getvalue()

Return a copy of the referent.

If the referent is unpicklable then this will raise an exception.

__repr__()

Return a representation of the proxy object.

__str__()

Return the representation of the referent.

Cleanup

A proxy object uses a weakref callback so that when it gets garbage collected it deregisters itself from the manager which owns its referent.

A shared object gets deleted from the manager process when there are no longer any proxies referring to it.

Process Pools

One can create a pool of processes which will carry out tasks submitted to it with the `Pool` class.

class multiprocessing.pool.**Pool**([*processes*[, *initializer*[, *initargs*[, *maxtasksperchild*[, *context*]]]]])

A process pool object which controls a pool of worker processes to which jobs can be submitted. It supports asynchronous results with timeouts and callbacks and has a parallel map implementation.

processes is the number of worker processes to use. If *processes* is `None` then the number returned by `os.process_cpu_count()` is used.

If *initializer* is not `None` then each worker process will call `initializer(*initargs)` when it starts.

maxtasksperchild is the number of tasks a worker process can complete before it will exit and be replaced with a fresh worker process, to enable unused resources to be freed. The default *maxtasksperchild* is `None`, which means worker processes will live as long as the pool.

context can be used to specify the context used for starting the worker processes. Usually a pool is created using the function `multiprocessing.Pool()` or the `Pool()` method of a context object. In both cases *context* is set appropriately.

Note that the methods of the pool object should only be called by the process which created the pool.

> **Warning**
>
> `multiprocessing.pool` objects have internal resources that need to be properly managed (like any other resource) by using the pool as a context manager or by calling `close()` and `terminate()` manually. Failure to do this can lead to the process hanging on finalization.
>
> Note that it is **not correct** to rely on the garbage collector to destroy the pool as CPython does not assure that the finalizer of the pool will be called (see `object.__del__()` for more information).

Changed in version 3.2: Added the *maxtasksperchild* parameter.

Changed in version 3.4: Added the *context* parameter.

Changed in version 3.13: *processes* uses `os.process_cpu_count()` by default, instead of `os.cpu_count()`.

> **Note**
>
> Worker processes within a `Pool` typically live for the complete duration of the Pool's work queue. A frequent pattern found in other systems (such as Apache, mod_wsgi, etc) to free resources held by workers is to allow a worker within a pool to complete only a set amount of work before being exiting, being cleaned up and a new process spawned to replace the old one. The *maxtasksperchild* argument to the `Pool` exposes this ability to the end user.

apply(*func*[, *args*[, *kwds*]])

Call *func* with arguments *args* and keyword arguments *kwds*. It blocks until the result is ready. Given this blocks, `apply_async()` is better suited for performing work in parallel. Additionally, *func* is only executed in one of the workers of the pool.

apply_async(*func*[, *args*[, *kwds*[, *callback*[, *error_callback*]]]])

A variant of the `apply()` method which returns a `AsyncResult` object.

If *callback* is specified then it should be a callable which accepts a single argument. When the result becomes ready *callback* is applied to it, that is unless the call failed, in which case the *error_callback* is applied instead.

If *error_callback* is specified then it should be a callable which accepts a single argument. If the target function fails, then the *error_callback* is called with the exception instance.

Callbacks should complete immediately since otherwise the thread which handles the results will get blocked.

map(*func*, *iterable*[, *chunksize*])

A parallel equivalent of the `map()` built-in function (it supports only one *iterable* argument though, for multiple iterables see `starmap()`). It blocks until the result is ready.

This method chops the iterable into a number of chunks which it submits to the process pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer.

Note that it may cause high memory usage for very long iterables. Consider using `imap()` or `imap_unordered()` with explicit *chunksize* option for better efficiency.

map_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

A variant of the `map()` method which returns a `AsyncResult` object.

If *callback* is specified then it should be a callable which accepts a single argument. When the result becomes ready *callback* is applied to it, that is unless the call failed, in which case the *error_callback* is applied instead.

If *error_callback* is specified then it should be a callable which accepts a single argument. If the target function fails, then the *error_callback* is called with the exception instance.

Callbacks should complete immediately since otherwise the thread which handles the results will get blocked.

imap(*func*, *iterable*[, *chunksize*])

A lazier version of `map()`.

The *chunksize* argument is the same as the one used by the `map()` method. For very long iterables using a large value for *chunksize* can make the job complete **much** faster than using the default value of `1`.

Also if *chunksize* is `1` then the `next()` method of the iterator returned by the `imap()` method has an optional *timeout* parameter: `next(timeout)` will raise `multiprocessing.TimeoutError` if the result cannot be returned within *timeout* seconds.

imap_unordered(*func*, *iterable*[, *chunksize*])

The same as `imap()` except that the ordering of the results from the returned iterator should be considered arbitrary. (Only when there is only one worker process is the order guaranteed to be "correct".)

starmap(*func*, *iterable*[, *chunksize*])

Like `map()` except that the elements of the *iterable* are expected to be iterables that are unpacked as arguments.

Hence an *iterable* of `[(1,2), (3, 4)]` results in `[func(1,2), func(3,4)]`.

Added in version 3.3.

starmap_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

A combination of `starmap()` and `map_async()` that iterates over *iterable* of iterables and calls *func* with the iterables unpacked. Returns a result object.

Added in version 3.3.

close()

Prevents any more tasks from being submitted to the pool. Once all the tasks have been completed the worker processes will exit.

terminate()

Stops the worker processes immediately without completing outstanding work. When the pool object is garbage collected `terminate()` will be called immediately.

join()

Wait for the worker processes to exit. One must call *close()* or *terminate()* before using *join()*.

Changed in version 3.3: Pool objects now support the context management protocol – see *Context Manager Types*. *__enter__()* returns the pool object, and *__exit__()* calls *terminate()*.

class multiprocessing.pool.**AsyncResult**

The class of the result returned by *Pool.apply_async()* and *Pool.map_async()*.

get([*timeout*])

Return the result when it arrives. If *timeout* is not None and the result does not arrive within *timeout* seconds then *multiprocessing.TimeoutError* is raised. If the remote call raised an exception then that exception will be reraised by *get()*.

wait([*timeout*])

Wait until the result is available or until *timeout* seconds pass.

ready()

Return whether the call has completed.

successful()

Return whether the call completed without raising an exception. Will raise *ValueError* if the result is not ready.

Changed in version 3.7: If the result is not ready, *ValueError* is raised instead of *AssertionError*.

The following example demonstrates the use of a pool:

```
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:          # start 4 worker processes
        result = pool.apply_async(f, (10,)) # evaluate "f(10)" asynchronously in a␣
↪single process
        print(result.get(timeout=1))        # prints "100" unless your computer is␣
↪*very* slow

        print(pool.map(f, range(10)))       # prints "[0, 1, 4,..., 81]"

        it = pool.imap(f, range(10))
        print(next(it))                     # prints "0"
        print(next(it))                     # prints "1"
        print(it.next(timeout=1))           # prints "4" unless your computer is␣
↪*very* slow

        result = pool.apply_async(time.sleep, (10,))
        print(result.get(timeout=1))        # raises multiprocessing.TimeoutError
```

Listeners and Clients

Usually message passing between processes is done using queues or by using *Connection* objects returned by *Pipe()*.

However, the *multiprocessing.connection* module allows some extra flexibility. It basically gives a high level message oriented API for dealing with sockets or Windows named pipes. It also has support for *digest authentication* using the *hmac* module, and for polling multiple connections at the same time.

multiprocessing.connection.**deliver_challenge**(*connection*, *authkey*)

Send a randomly generated message to the other end of the connection and wait for a reply.

If the reply matches the digest of the message using *authkey* as the key then a welcome message is sent to the other end of the connection. Otherwise *AuthenticationError* is raised.

multiprocessing.connection.**answer_challenge**(*connection*, *authkey*)

Receive a message, calculate the digest of the message using *authkey* as the key, and then send the digest back.

If a welcome message is not received, then *AuthenticationError* is raised.

multiprocessing.connection.**Client**(*address*[, *family*[, *authkey*]])

Attempt to set up a connection to the listener which is using address *address*, returning a *Connection*.

The type of the connection is determined by *family* argument, but this can generally be omitted since it can usually be inferred from the format of *address*. (See *Address Formats*)

If *authkey* is given and not None, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if *authkey* is None. *AuthenticationError* is raised if authentication fails. See *Authentication keys*.

class multiprocessing.connection.**Listener**([*address*[, *family*[, *backlog*[, *authkey*]]]])

A wrapper for a bound socket or Windows named pipe which is 'listening' for connections.

address is the address to be used by the bound socket or named pipe of the listener object.

> **Note**
>
> If an address of '0.0.0.0' is used, the address will not be a connectable end point on Windows. If you require a connectable end-point, you should use '127.0.0.1'.

family is the type of socket (or named pipe) to use. This can be one of the strings 'AF_INET' (for a TCP socket), 'AF_UNIX' (for a Unix domain socket) or 'AF_PIPE' (for a Windows named pipe). Of these only the first is guaranteed to be available. If *family* is None then the family is inferred from the format of *address*. If *address* is also None then a default is chosen. This default is the family which is assumed to be the fastest available. See *Address Formats*. Note that if *family* is 'AF_UNIX' and address is None then the socket will be created in a private temporary directory created using *tempfile.mkstemp()*.

If the listener object uses a socket then *backlog* (1 by default) is passed to the *listen()* method of the socket once it has been bound.

If *authkey* is given and not None, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if *authkey* is None. *AuthenticationError* is raised if authentication fails. See *Authentication keys*.

accept()

Accept a connection on the bound socket or named pipe of the listener object and return a *Connection* object. If authentication is attempted and fails, then *AuthenticationError* is raised.

close()

Close the bound socket or named pipe of the listener object. This is called automatically when the listener is garbage collected. However it is advisable to call it explicitly.

Listener objects have the following read-only properties:

address

The address which is being used by the Listener object.

last_accepted

The address from which the last accepted connection came. If this is unavailable then it is `None`.

Changed in version 3.3: Listener objects now support the context management protocol – see *Context Manager Types*. *__enter__()* returns the listener object, and *__exit__()* calls *close()*.

`multiprocessing.connection.`**wait**(*object_list*, *timeout=None*)

Wait till an object in *object_list* is ready. Returns the list of those objects in *object_list* which are ready. If *timeout* is a float then the call blocks for at most that many seconds. If *timeout* is `None` then it will block for an unlimited period. A negative timeout is equivalent to a zero timeout.

For both POSIX and Windows, an object can appear in *object_list* if it is

- a readable *Connection* object;
- a connected and readable *socket.socket* object; or
- the *sentinel* attribute of a *Process* object.

A connection or socket object is ready when there is data available to be read from it, or the other end has been closed.

POSIX: `wait(object_list, timeout)` almost equivalent `select.select(object_list, [], [], timeout)`. The difference is that, if *select.select()* is interrupted by a signal, it can raise *OSError* with an error number of `EINTR`, whereas *wait()* will not.

Windows: An item in *object_list* must either be an integer handle which is waitable (according to the definition used by the documentation of the Win32 function `WaitForMultipleObjects()`) or it can be an object with a *fileno()* method which returns a socket handle or pipe handle. (Note that pipe handles and socket handles are **not** waitable handles.)

Added in version 3.3.

Examples

The following server code creates a listener which uses `'secret password'` as an authentication key. It then waits for a connection and sends some data to the client:

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)     # family is deduced to be 'AF_INET'

with Listener(address, authkey=b'secret password') as listener:
    with listener.accept() as conn:
        print('connection accepted from', listener.last_accepted)

        conn.send([2.25, None, 'junk', float])

        conn.send_bytes(b'hello')

        conn.send_bytes(array('i', [42, 1729]))
```

The following code connects to the server and receives some data from the server:

```
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)

with Client(address, authkey=b'secret password') as conn:
    print(conn.recv())                  # => [2.25, None, 'junk', float]

    print(conn.recv_bytes())            # => 'hello'

    arr = array('i', [0, 0, 0, 0, 0])
    print(conn.recv_bytes_into(arr))    # => 8
    print(arr)                          # => array('i', [42, 1729, 0, 0, 0])
```

The following code uses *wait()* to wait for messages from multiple processes at once:

```
from multiprocessing import Process, Pipe, current_process
from multiprocessing.connection import wait

def foo(w):
    for i in range(10):
        w.send((i, current_process().name))
    w.close()

if __name__ == '__main__':
    readers = []

    for i in range(4):
        r, w = Pipe(duplex=False)
        readers.append(r)
        p = Process(target=foo, args=(w,))
        p.start()
        # We close the writable end of the pipe now to be sure that
        # p is the only process which owns a handle for it.  This
        # ensures that when p closes its handle for the writable end,
        # wait() will promptly report the readable end as being ready.
        w.close()

    while readers:
        for r in wait(readers):
            try:
                msg = r.recv()
            except EOFError:
                readers.remove(r)
            else:
                print(msg)
```

Address Formats

- An `'AF_INET'` address is a tuple of the form `(hostname, port)` where *hostname* is a string and *port* is an integer.
- An `'AF_UNIX'` address is a string representing a filename on the filesystem.
- An `'AF_PIPE'` address is a string of the form `r'\\.\pipe\PipeName'`. To use `Client()` to connect to a named pipe on a remote computer called *ServerName* one should use an address of the form `r'\\ServerName\pipe\PipeName'` instead.

Note that any string beginning with two backslashes is assumed by default to be an `'AF_PIPE'` address rather than an `'AF_UNIX'` address.

Authentication keys

When one uses `Connection.recv`, the data received is automatically unpickled. Unfortunately unpickling data from an untrusted source is a security risk. Therefore `Listener` and `Client()` use the `hmac` module to provide digest authentication.

An authentication key is a byte string which can be thought of as a password: once a connection is established both ends will demand proof that the other knows the authentication key. (Demonstrating that both ends are using the same key does **not** involve sending the key over the connection.)

If authentication is requested but no authentication key is specified then the return value of `current_process().authkey` is used (see `Process`). This value will be automatically inherited by any `Process` object that the current process creates. This means that (by default) all processes of a multi-process program will share a single authentication key which can be used when setting up connections between themselves.

Suitable authentication keys can also be generated by using `os.urandom()`.

Logging

Some support for logging is available. Note, however, that the `logging` package does not use process shared locks so it is possible (depending on the handler type) for messages from different processes to get mixed up.

`multiprocessing.`**`get_logger`**`()`

Returns the logger used by `multiprocessing`. If necessary, a new one will be created.

When first created the logger has level `logging.NOTSET` and no default handler. Messages sent to this logger will not by default propagate to the root logger.

Note that on Windows child processes will only inherit the level of the parent process's logger – any other customization of the logger will not be inherited.

`multiprocessing.`**`log_to_stderr`**`(`*level=None*`)`

This function performs a call to `get_logger()` but in addition to returning the logger created by get_logger, it adds a handler which sends output to `sys.stderr` using format `'[%(levelname)s/%(processName)s] %(message)s'`. You can modify `levelname` of the logger by passing a `level` argument.

Below is an example session with logging turned on:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
```

(continues on next page)

(continued from previous page)

```
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

For a full table of logging levels, see the *logging* module.

The multiprocessing.dummy module

multiprocessing.dummy replicates the API of *multiprocessing* but is no more than a wrapper around the *threading* module.

In particular, the Pool function provided by *multiprocessing.dummy* returns an instance of *ThreadPool*, which is a subclass of *Pool* that supports all the same method calls but uses a pool of worker threads rather than worker processes.

class multiprocessing.pool.**ThreadPool**([*processes*[, *initializer*[, *initargs*]]])

A thread pool object which controls a pool of worker threads to which jobs can be submitted. *ThreadPool* instances are fully interface compatible with *Pool* instances, and their resources must also be properly managed, either by using the pool as a context manager or by calling *close()* and *terminate()* manually.

processes is the number of worker threads to use. If *processes* is None then the number returned by *os.process_cpu_count()* is used.

If *initializer* is not None then each worker process will call initializer(*initargs) when it starts.

Unlike *Pool*, *maxtasksperchild* and *context* cannot be provided.

> **Note**
>
> A *ThreadPool* shares the same interface as *Pool*, which is designed around a pool of processes and predates the introduction of the *concurrent.futures* module. As such, it inherits some operations that don't make sense for a pool backed by threads, and it has its own type for representing the status of asynchronous jobs, *AsyncResult*, that is not understood by any other libraries.
>
> Users should generally prefer to use *concurrent.futures.ThreadPoolExecutor*, which has a simpler interface that was designed around threads from the start, and which returns *concurrent.futures.Future* instances that are compatible with many other libraries, including *asyncio*.

17.2.3 Programming guidelines

There are certain guidelines and idioms which should be adhered to when using *multiprocessing*.

All start methods

The following applies to all start methods.

Avoid shared state

As far as possible one should try to avoid shifting large amounts of data between processes.

It is probably best to stick to using queues or pipes for communication between processes rather than using the lower level synchronization primitives.

Picklability

Ensure that the arguments to the methods of proxies are picklable.

Thread safety of proxies

Do not use a proxy object from more than one thread unless you protect it with a lock.

(There is never a problem with different processes using the *same* proxy.)

Joining zombie processes

On POSIX when a process finishes but has not been joined it becomes a zombie. There should never be very many because each time a new process starts (or `active_children()` is called) all completed processes which have not yet been joined will be joined. Also calling a finished process's `Process.is_alive` will join the process. Even so it is probably good practice to explicitly join all the processes that you start.

Better to inherit than pickle/unpickle

When using the *spawn* or *forkserver* start methods many types from `multiprocessing` need to be picklable so that child processes can use them. However, one should generally avoid sending shared objects to other processes using pipes or queues. Instead you should arrange the program so that a process which needs access to a shared resource created elsewhere can inherit it from an ancestor process.

Avoid terminating processes

Using the `Process.terminate` method to stop a process is liable to cause any shared resources (such as locks, semaphores, pipes and queues) currently being used by the process to become broken or unavailable to other processes.

Therefore it is probably best to only consider using `Process.terminate` on processes which never use any shared resources.

Joining processes that use queues

Bear in mind that a process that has put items in a queue will wait before terminating until all the buffered items are fed by the "feeder" thread to the underlying pipe. (The child process can call the `Queue.cancel_join_thread` method of the queue to avoid this behaviour.)

This means that whenever you use a queue you need to make sure that all items which have been put on the queue will eventually be removed before the process is joined. Otherwise you cannot be sure that processes which have put items on the queue will terminate. Remember also that non-daemonic processes will be joined automatically.

An example which will deadlock is the following:

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()                    # this deadlocks
    obj = queue.get()
```

A fix here would be to swap the last two lines (or simply remove the `p.join()` line).

Explicitly pass resources to child processes

On POSIX using the *fork* start method, a child process can make use of a shared resource created in a parent process using a global resource. However, it is better to pass the object as an argument to the constructor for the child process.

Apart from making the code (potentially) compatible with Windows and the other start methods this also ensures that as long as the child process is still alive the object will not be garbage collected in the parent process. This might be important if some resource is freed when the object is garbage collected in the parent process.

So for instance

```
from multiprocessing import Process, Lock

def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

should be rewritten as

```
from multiprocessing import Process, Lock

def f(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

Beware of replacing *sys.stdin* with a "file like object"

multiprocessing originally unconditionally called:

```
os.close(sys.stdin.fileno())
```

in the `multiprocessing.Process._bootstrap()` method — this resulted in issues with processes-in-processes. This has been changed to:

```
sys.stdin.close()
sys.stdin = open(os.open(os.devnull, os.O_RDONLY), closefd=False)
```

Which solves the fundamental issue of processes colliding with each other resulting in a bad file descriptor error, but introduces a potential danger to applications which replace *sys.stdin()* with a "file-like object"

with output buffering. This danger is that if multiple processes call *close()* on this file-like object, it could result in the same data being flushed to the object multiple times, resulting in corruption.

If you write a file-like object and implement your own caching, you can make it fork-safe by storing the pid whenever you append to the cache, and discarding the cache when the pid changes. For example:

```
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

For more information, see bpo-5155, bpo-5313 and bpo-5331

The *spawn* and *forkserver* start methods

There are a few extra restrictions which don't apply to the *fork* start method.

More picklability

Ensure that all arguments to `Process.__init__()` are picklable. Also, if you subclass *Process* then make sure that instances will be picklable when the *Process.start* method is called.

Global variables

Bear in mind that if code run in a child process tries to access a global variable, then the value it sees (if any) may not be the same as the value in the parent process at the time that *Process.start* was called.

However, global variables which are just module level constants cause no problems.

Safe importing of main module

Make sure that the main module can be safely imported by a new Python interpreter without causing unintended side effects (such as starting a new process).

For example, using the *spawn* or *forkserver* start method running the following module would fail with a *RuntimeError*:

```
from multiprocessing import Process

def foo():
    print('hello')

p = Process(target=foo)
p.start()
```

Instead one should protect the "entry point" of the program by using `if __name__ == '__main__':` as follows:

```
from multiprocessing import Process, freeze_support, set_start_method

def foo():
    print('hello')

if __name__ == '__main__':
    freeze_support()
    set_start_method('spawn')
```

(continues on next page)

(continued from previous page)

```
    p = Process(target=foo)
    p.start()
```

(The `freeze_support()` line can be omitted if the program will be run normally instead of frozen.)

This allows the newly spawned Python interpreter to safely import the module and then run the module's `foo()` function.

Similar restrictions apply if a pool or manager is created in the main module.

17.2.4 Examples

Demonstration of how to create and use customized managers and proxies:

```
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo:
    def f(self):
        print('you called Foo.f()')
    def g(self):
        print('you called Foo.g()')
    def _h(self):
        print('you called Foo._h()')

# A simple generator function
def baz():
    for i in range(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ['__next__']
    def __iter__(self):
        return self
    def __next__(self):
        return self._callmethod('__next__')

# Function to return the operator module
def get_operator_module():
    return operator

##

class MyManager(BaseManager):
    pass

# register the Foo class; make `f()` and `g()` accessible via proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))
```

(continues on next page)

(continued from previous page)

```
# register the generator function baz; use `GeneratorProxy` to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module)

##

def test():
    manager = MyManager()
    manager.start()

    print('-' * 20)

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print('-' * 20)

    f2 = manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print('-' * 20)

    it = manager.baz()
    for i in it:
        print('<%d>' % i, end=' ')
    print()

    print('-' * 20)

    op = manager.operator()
    print('op.add(23, 45) =', op.add(23, 45))
    print('op.pow(2, 94) =', op.pow(2, 94))
    print('op._exposed_ =', op._exposed_)

##

if __name__ == '__main__':
    freeze_support()
    test()
```

Using *Pool*:

```
import multiprocessing
import time
import random
import sys

#
# Functions used by test code
```

(continues on next page)

(continued from previous page)

```
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
        )

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5 * random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5 * random.random())
    return a + b

def f(x):
    return 1.0 / (x - 5.0)

def pow3(x):
    return x ** 3

def noop(x):
    pass

#
# Test code
#

def test():
    PROCESSES = 4
    print('Creating pool with %d processes\n' % PROCESSES)

    with multiprocessing.Pool(PROCESSES) as pool:
        #
        # Tests
        #

        TASKS = [(mul, (i, 7)) for i in range(10)] + \
                [(plus, (i, 8)) for i in range(10)]

        results = [pool.apply_async(calculate, t) for t in TASKS]
        imap_it = pool.imap(calculatestar, TASKS)
        imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

        print('Ordered results using pool.apply_async():')
        for r in results:
            print('\t', r.get())
        print()

        print('Ordered results using pool.imap():')
        for x in imap_it:
            print('\t', x)
```

(continues on next page)

(continued from previous page)

```
        print()

        print('Unordered results using pool.imap_unordered():')
        for x in imap_unordered_it:
            print('\t', x)
        print()

        print('Ordered results using pool.map() --- will block till complete:')
        for x in pool.map(calculatestar, TASKS):
            print('\t', x)
        print()

        #
        # Test error handling
        #

        print('Testing error handling:')

        try:
            print(pool.apply(f, (5,)))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from pool.apply()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(pool.map(f, list(range(10))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from pool.map()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(list(pool.imap(f, list(range(10)))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from list(pool.imap())')
        else:
            raise AssertionError('expected ZeroDivisionError')

        it = pool.imap(f, list(range(10)))
        for i in range(10):
            try:
                x = next(it)
            except ZeroDivisionError:
                if i == 5:
                    pass
            except StopIteration:
                break
            else:
                if i == 5:
                    raise AssertionError('expected ZeroDivisionError')

        assert i == 9
        print('\tGot ZeroDivisionError as expected from IMapIterator.next()')
        print()

        #
```

(continues on next page)

(continued from previous page)

```
        # Testing timeouts
        #

        print('Testing ApplyResult.get() with timeout:', end=' ')
        res = pool.apply_async(calculate, TASKS[0])
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % res.get(0.02))
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()

        print('Testing IMapIterator.next() with timeout:', end=' ')
        it = pool.imap(calculatestar, TASKS)
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % it.next(0.02))
            except StopIteration:
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()


if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()
```

An example showing how to use queues to feed tasks to a collection of worker processes and collect the results:

```
import time
import random

from multiprocessing import Process, Queue, current_process, freeze_support

#
# Function run by worker processes
#

def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

#
# Function used to calculate result
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % \
        (current_process().name, func.__name__, args, result)
```

(continues on next page)

(continued from previous page)

```
#
# Functions referenced by tasks
#

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

#
#
#

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker, args=(task_queue, done_queue)).start()

    # Get and print results
    print('Unordered results:')
    for i in range(len(TASKS1)):
        print('\t', done_queue.get())

    # Add more tasks using `put()`
    for task in TASKS2:
        task_queue.put(task)

    # Get and print some more results
    for i in range(len(TASKS2)):
        print('\t', done_queue.get())

    # Tell child processes to stop
    for i in range(NUMBER_OF_PROCESSES):
        task_queue.put('STOP')


if __name__ == '__main__':
    freeze_support()
    test()
```

17.3 multiprocessing.shared_memory — Shared memory for direct access across processes

Source code: Lib/multiprocessing/shared_memory.py

Added in version 3.8.

This module provides a class, *SharedMemory*, for the allocation and management of shared memory to be accessed by one or more processes on a multicore or symmetric multiprocessor (SMP) machine. To assist with the life-cycle management of shared memory especially across distinct processes, a *BaseManager* subclass, *SharedMemoryManager*, is also provided in the *multiprocessing.managers* module.

In this module, shared memory refers to "POSIX style" shared memory blocks (though is not necessarily implemented explicitly as such) and does not refer to "distributed shared memory". This style of shared memory permits distinct processes to potentially read and write to a common (or shared) region of volatile memory. Processes are conventionally limited to only have access to their own process memory space but shared memory permits the sharing of data between processes, avoiding the need to instead send messages between processes containing that data. Sharing data directly via memory can provide significant performance benefits compared to sharing data via disk or socket or other communications requiring the serialization/deserialization and copying of data.

class multiprocessing.shared_memory.**SharedMemory**(*name=None*, *create=False*, *size=0*, ***, *track=True*)

Create an instance of the SharedMemory class for either creating a new shared memory block or attaching to an existing shared memory block. Each shared memory block is assigned a unique name. In this way, one process can create a shared memory block with a particular name and a different process can attach to that same shared memory block using that same name.

As a resource for sharing data across processes, shared memory blocks may outlive the original process that created them. When one process no longer needs access to a shared memory block that might still be needed by other processes, the *close()* method should be called. When a shared memory block is no longer needed by any process, the *unlink()* method should be called to ensure proper cleanup.

Parameters

- **name** (str | *None*) – The unique name for the requested shared memory, specified as a string. When creating a new shared memory block, if None (the default) is supplied for the name, a novel name will be generated.
- **create** (bool) – Control whether a new shared memory block is created (True) or an existing shared memory block is attached (False).
- **size** (int) – The requested number of bytes when creating a new shared memory block. Because some platforms choose to allocate chunks of memory based upon that platform's memory page size, the exact size of the shared memory block may be larger or equal to the size requested. When attaching to an existing shared memory block, the *size* parameter is ignored.
- **track** (bool) – When True, register the shared memory block with a resource tracker process on platforms where the OS does not do this automatically. The resource tracker ensures proper cleanup of the shared memory even if all other processes with access to the memory exit without doing so. Python processes created from a common ancestor using *multiprocessing* facilities share a single resource tracker process, and the lifetime of shared memory segments is handled automatically among these processes. Python processes created in any other way will receive their own resource tracker when accessing shared memory with *track* enabled. This will cause the shared memory to be deleted by the resource tracker of the first process that terminates. To avoid this issue, users of *subprocess* or standalone Python

processes should set *track* to `False` when there is already another process in place that does the bookkeeping. *track* is ignored on Windows, which has its own tracking and automatically deletes shared memory when all handles to it have been closed.

Changed in version 3.13: Added the *track* parameter.

close()

Close the file descriptor/handle to the shared memory from this instance. *close()* should be called once access to the shared memory block from this instance is no longer needed. Depending on operating system, the underlying memory may or may not be freed even if all handles to it have been closed. To ensure proper cleanup, use the *unlink()* method.

unlink()

Delete the underlying shared memory block. This should be called only once per shared memory block regardless of the number of handles to it, even in other processes. *unlink()* and *close()* can be called in any order, but trying to access data inside a shared memory block after *unlink()* may result in memory access errors, depending on platform.

This method has no effect on Windows, where the only way to delete a shared memory block is to close all handles.

buf

A memoryview of contents of the shared memory block.

name

Read-only access to the unique name of the shared memory block.

size

Read-only access to size in bytes of the shared memory block.

The following example demonstrates low-level use of *SharedMemory* instances:

```
>>> from multiprocessing import shared_memory
>>> shm_a = shared_memory.SharedMemory(create=True, size=10)
>>> type(shm_a.buf)
<class 'memoryview'>
>>> buffer = shm_a.buf
>>> len(buffer)
10
>>> buffer[:4] = bytearray([22, 33, 44, 55])  # Modify multiple at once
>>> buffer[4] = 100                           # Modify single byte at a time
>>> # Attach to an existing shared memory block
>>> shm_b = shared_memory.SharedMemory(shm_a.name)
>>> import array
>>> array.array('b', shm_b.buf[:5])  # Copy the data into a new array.array
array('b', [22, 33, 44, 55, 100])
>>> shm_b.buf[:5] = b'howdy'  # Modify via shm_b using bytes
>>> bytes(shm_a.buf[:5])      # Access via shm_a
b'howdy'
>>> shm_b.close()   # Close each SharedMemory instance
>>> shm_a.close()
>>> shm_a.unlink()  # Call unlink only once to release the shared memory
```

The following example demonstrates a practical use of the *SharedMemory* class with NumPy arrays, accessing the same `numpy.ndarray` from two distinct Python shells:

```
>>> # In the first Python interactive shell
>>> import numpy as np
```

(continues on next page)

(continued from previous page)

```
>>> a = np.array([1, 1, 2, 3, 5, 8])  # Start with an existing NumPy array
>>> from multiprocessing import shared_memory
>>> shm = shared_memory.SharedMemory(create=True, size=a.nbytes)
>>> # Now create a NumPy array backed by shared memory
>>> b = np.ndarray(a.shape, dtype=a.dtype, buffer=shm.buf)
>>> b[:] = a[:]  # Copy the original data into shared memory
>>> b
array([1, 1, 2, 3, 5, 8])
>>> type(b)
<class 'numpy.ndarray'>
>>> type(a)
<class 'numpy.ndarray'>
>>> shm.name  # We did not specify a name so one was chosen for us
'psm_21467_46075'

>>> # In either the same shell or a new Python shell on the same machine
>>> import numpy as np
>>> from multiprocessing import shared_memory
>>> # Attach to the existing shared memory block
>>> existing_shm = shared_memory.SharedMemory(name='psm_21467_46075')
>>> # Note that a.shape is (6,) and a.dtype is np.int64 in this example
>>> c = np.ndarray((6,), dtype=np.int64, buffer=existing_shm.buf)
>>> c
array([1, 1, 2, 3, 5, 8])
>>> c[-1] = 888
>>> c
array([  1,   1,   2,   3,   5, 888])

>>> # Back in the first Python interactive shell, b reflects this change
>>> b
array([  1,   1,   2,   3,   5, 888])

>>> # Clean up from within the second Python shell
>>> del c  # Unnecessary; merely emphasizing the array is no longer used
>>> existing_shm.close()

>>> # Clean up from within the first Python shell
>>> del b  # Unnecessary; merely emphasizing the array is no longer used
>>> shm.close()
>>> shm.unlink()  # Free and release the shared memory block at the very end
```

class multiprocessing.managers.**SharedMemoryManager**([*address*[, *authkey*]])

A subclass of *multiprocessing.managers.BaseManager* which can be used for the management of shared memory blocks across processes.

A call to *start()* on a SharedMemoryManager instance causes a new process to be started. This new process's sole purpose is to manage the life cycle of all shared memory blocks created through it. To trigger the release of all shared memory blocks managed by that process, call *shutdown()* on the instance. This triggers a *unlink()* call on all of the *SharedMemory* objects managed by that process and then stops the process itself. By creating SharedMemory instances through a SharedMemoryManager, we avoid the need to manually track and trigger the freeing of shared memory resources.

This class provides methods for creating and returning *SharedMemory* instances and for creating a list-like object (*ShareableList*) backed by shared memory.

Refer to *BaseManager* for a description of the inherited *address* and *authkey* optional input arguments and how they may be used to connect to an existing SharedMemoryManager service from other processes.

SharedMemory(*size*)

Create and return a new *SharedMemory* object with the specified *size* in bytes.

ShareableList(*sequence*)

Create and return a new *ShareableList* object, initialized by the values from the input *sequence*.

The following example demonstrates the basic mechanisms of a *SharedMemoryManager*:

```
>>> from multiprocessing.managers import SharedMemoryManager
>>> smm = SharedMemoryManager()
>>> smm.start()  # Start the process that manages the shared memory blocks
>>> sl = smm.ShareableList(range(4))
>>> sl
ShareableList([0, 1, 2, 3], name='psm_6572_7512')
>>> raw_shm = smm.SharedMemory(size=128)
>>> another_sl = smm.ShareableList('alpha')
>>> another_sl
ShareableList(['a', 'l', 'p', 'h', 'a'], name='psm_6572_12221')
>>> smm.shutdown()  # Calls unlink() on sl, raw_shm, and another_sl
```

The following example depicts a potentially more convenient pattern for using *SharedMemoryManager* objects via the `with` statement to ensure that all shared memory blocks are released after they are no longer needed:

```
>>> with SharedMemoryManager() as smm:
...     sl = smm.ShareableList(range(2000))
...     # Divide the work among two processes, storing partial results in sl
...     p1 = Process(target=do_work, args=(sl, 0, 1000))
...     p2 = Process(target=do_work, args=(sl, 1000, 2000))
...     p1.start()
...     p2.start()  # A multiprocessing.Pool might be more efficient
...     p1.join()
...     p2.join()   # Wait for all work to complete in both processes
...     total_result = sum(sl)  # Consolidate the partial results now in sl
```

When using a *SharedMemoryManager* in a `with` statement, the shared memory blocks created using that manager are all released when the `with` statement's code block finishes execution.

class multiprocessing.shared_memory.**ShareableList**(*sequence=None*, *, *name=None*)

Provide a mutable list-like object where all values stored within are stored in a shared memory block. This constrains storable values to the following built-in data types:

- *int* (signed 64-bit)
- *float*
- *bool*
- *str* (less than 10M bytes each when encoded as UTF-8)
- *bytes* (less than 10M bytes each)
- `None`

It also notably differs from the built-in *list* type in that these lists can not change their overall length (i.e. no `append()`, `insert()`, etc.) and do not support the dynamic creation of new `ShareableList` instances via slicing.

sequence is used in populating a new `ShareableList` full of values. Set to `None` to instead attach to an already existing `ShareableList` by its unique shared memory name.

name is the unique name for the requested shared memory, as described in the definition for *SharedMemory*. When attaching to an existing ShareableList, specify its shared memory block's unique name while leaving *sequence* set to None.

Note

A known issue exists for *bytes* and *str* values. If they end with \x00 nul bytes or characters, those may be *silently stripped* when fetching them by index from the ShareableList. This .rstrip(b'\x00') behavior is considered a bug and may go away in the future. See gh-106939.

For applications where rstripping of trailing nulls is a problem, work around it by always unconditionally appending an extra non-0 byte to the end of such values when storing and unconditionally removing it when fetching:

```
>>> from multiprocessing import shared_memory
>>> nul_bug_demo = shared_memory.ShareableList(['?\x00', b'\x03\x02\x01\x00\x00\
↪x00'])
>>> nul_bug_demo[0]
'?'
>>> nul_bug_demo[1]
b'\x03\x02\x01'
>>> nul_bug_demo.shm.unlink()
>>> padded = shared_memory.ShareableList(['?\x00\x07', b'\x03\x02\x01\x00\x00\x00\
↪x07'])
>>> padded[0][:-1]
'?\x00'
>>> padded[1][:-1]
b'\x03\x02\x01\x00\x00\x00'
>>> padded.shm.unlink()
```

count(*value*)

Return the number of occurrences of *value*.

index(*value*)

Return first index position of *value*. Raise *ValueError* if *value* is not present.

format

Read-only attribute containing the *struct* packing format used by all currently stored values.

shm

The *SharedMemory* instance where the values are stored.

The following example demonstrates basic use of a *ShareableList* instance:

```
>>> from multiprocessing import shared_memory
>>> a = shared_memory.ShareableList(['howdy', b'HoWdY', -273.154, 100, None, True,␣
↪42])
>>> [ type(entry) for entry in a ]
[<class 'str'>, <class 'bytes'>, <class 'float'>, <class 'int'>, <class 'NoneType'>,
↪<class 'bool'>, <class 'int'>]
>>> a[2]
-273.154
>>> a[2] = -78.5
>>> a[2]
-78.5
>>> a[2] = 'dry ice'  # Changing data types is supported as well
>>> a[2]
```

(continues on next page)

(continued from previous page)

```
'dry ice'
>>> a[2] = 'larger than previously allocated storage space'
Traceback (most recent call last):
  ...
ValueError: exceeds available storage for existing str
>>> a[2]
'dry ice'
>>> len(a)
7
>>> a.index(42)
6
>>> a.count(b'howdy')
0
>>> a.count(b'HoWdY')
1
>>> a.shm.close()
>>> a.shm.unlink()
>>> del a  # Use of a ShareableList after call to unlink() is unsupported
```

The following example depicts how one, two, or many processes may access the same *ShareableList* by supplying the name of the shared memory block behind it:

```
>>> b = shared_memory.ShareableList(range(5))          # In a first process
>>> c = shared_memory.ShareableList(name=b.shm.name)   # In a second process
>>> c
ShareableList([0, 1, 2, 3, 4], name='...')
>>> c[-1] = -999
>>> b[-1]
-999
>>> b.shm.close()
>>> c.shm.close()
>>> c.shm.unlink()
```

The following examples demonstrates that *ShareableList* (and underlying *SharedMemory*) objects can be pickled and unpickled if needed. Note, that it will still be the same shared object. This happens, because the deserialized object has the same unique name and is just attached to an existing object with the same name (if the object is still alive):

```
>>> import pickle
>>> from multiprocessing import shared_memory
>>> sl = shared_memory.ShareableList(range(10))
>>> list(sl)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> deserialized_sl = pickle.loads(pickle.dumps(sl))
>>> list(deserialized_sl)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> sl[0] = -1
>>> deserialized_sl[1] = -2
>>> list(sl)
[-1, -2, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(deserialized_sl)
[-1, -2, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> sl.shm.close()
>>> sl.shm.unlink()
```

17.4 The concurrent package

Currently, there is only one module in this package:

- *concurrent.futures* – Launching parallel tasks

17.5 concurrent.futures — Launching parallel tasks

Added in version 3.2.

Source code: Lib/concurrent/futures/thread.py and Lib/concurrent/futures/process.py

---

The *concurrent.futures* module provides a high-level interface for asynchronously executing callables.

The asynchronous execution can be performed with threads, using *ThreadPoolExecutor*, or separate processes, using *ProcessPoolExecutor*. Both implement the same interface, which is defined by the abstract *Executor* class.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

17.5.1 Executor Objects

class concurrent.futures.**Executor**

An abstract class that provides methods to execute calls asynchronously. It should not be used directly, but through its concrete subclasses.

submit(*fn*, /, **args*, ***kwargs*)

Schedules the callable, *fn*, to be executed as `fn(*args, **kwargs)` and returns a *Future* object representing the execution of the callable.

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

map(*fn*, **iterables*, *timeout=None*, *chunksize=1*)

Similar to *map(fn, *iterables)* except:

- the *iterables* are collected immediately rather than lazily;
- *fn* is executed asynchronously and several calls to *fn* may be made concurrently.

The returned iterator raises a *TimeoutError* if *__next__()* is called and the result isn't available after *timeout* seconds from the original call to *Executor.map()*. *timeout* can be an int or a float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If a *fn* call raises an exception, then that exception will be raised when its value is retrieved from the iterator.

When using *ProcessPoolExecutor*, this method chops *iterables* into a number of chunks which it submits to the pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer. For very long iterables, using a large value for *chunksize* can significantly improve performance compared to the default size of 1. With *ThreadPoolExecutor*, *chunksize* has no effect.

Changed in version 3.5: Added the *chunksize* argument.

shutdown(*wait=True*, *, *cancel_futures=False*)

Signal the executor that it should free any resources that it is using when the currently pending futures are done executing. Calls to *Executor.submit()* and *Executor.map()* made after shutdown will raise *RuntimeError*.

If *wait* is `True` then this method will not return until all the pending futures are done executing and the resources associated with the executor have been freed. If *wait* is `False` then this method will return immediately and the resources associated with the executor will be freed when all pending futures are done executing. Regardless of the value of *wait*, the entire Python program will not exit until all pending futures are done executing.

If *cancel_futures* is `True`, this method will cancel all pending futures that the executor has not started running. Any futures that are completed or running won't be cancelled, regardless of the value of *cancel_futures*.

If both *cancel_futures* and *wait* are `True`, all futures that the executor has started running will be completed prior to this method returning. The remaining futures are cancelled.

You can avoid having to call this method explicitly if you use the `with` statement, which will shutdown the *Executor* (waiting as if *Executor.shutdown()* were called with *wait* set to `True`):

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

Changed in version 3.9: Added *cancel_futures*.

17.5.2 ThreadPoolExecutor

ThreadPoolExecutor is an *Executor* subclass that uses a pool of threads to execute calls asynchronously.

Deadlocks can occur when the callable associated with a *Future* waits on the results of another *Future*. For example:

```
import time
def wait_on_b():
    time.sleep(5)
    print(b.result())  # b will never complete because it is waiting on a.
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result())  # a will never complete because it is waiting on b.
    return 6


executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
```

And:

```
def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only one worker thread and
    # it is executing this function.
```

(continues on next page)

(continued from previous page)

```
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

class concurrent.futures.**ThreadPoolExecutor**(*max_workers=None*, *thread_name_prefix=''*, *initializer=None*, *initargs=()*)

An *Executor* subclass that uses a pool of at most *max_workers* threads to execute calls asynchronously.

All threads enqueued to `ThreadPoolExecutor` will be joined before the interpreter can exit. Note that the exit handler which does this is executed *before* any exit handlers added using `atexit`. This means exceptions in the main thread must be caught and handled in order to signal threads to exit gracefully. For this reason, it is recommended that `ThreadPoolExecutor` not be used for long-running tasks.

initializer is an optional callable that is called at the start of each worker thread; *initargs* is a tuple of arguments passed to the initializer. Should *initializer* raise an exception, all currently pending jobs will raise a *BrokenThreadPool*, as well as any attempt to submit more jobs to the pool.

Changed in version 3.5: If *max_workers* is `None` or not given, it will default to the number of processors on the machine, multiplied by `5`, assuming that *ThreadPoolExecutor* is often used to overlap I/O instead of CPU work and the number of workers should be higher than the number of workers for *ProcessPoolExecutor*.

Changed in version 3.6: Added the *thread_name_prefix* parameter to allow users to control the *threading.Thread* names for worker threads created by the pool for easier debugging.

Changed in version 3.7: Added the *initializer* and *initargs* arguments.

Changed in version 3.8: Default value of *max_workers* is changed to `min(32, os.cpu_count() + 4)`. This default value preserves at least 5 workers for I/O bound tasks. It utilizes at most 32 CPU cores for CPU bound tasks which release the GIL. And it avoids using very large resources implicitly on many-core machines.

ThreadPoolExecutor now reuses idle worker threads before starting *max_workers* worker threads too.

Changed in version 3.13: Default value of *max_workers* is changed to `min(32, (os.process_cpu_count() or 1) + 4)`.

ThreadPoolExecutor Example

```
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://nonexistant-subdomain.python.org/']

# Retrieve a single page and report the URL and contents
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# We can use a with statement to ensure threads are cleaned up promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # Start the load operations and mark each future with its URL
    future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
```

(continues on next page)

(continued from previous page)

```
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)))
```

17.5.3 ProcessPoolExecutor

The *ProcessPoolExecutor* class is an *Executor* subclass that uses a pool of processes to execute calls asynchronously. *ProcessPoolExecutor* uses the *multiprocessing* module, which allows it to side-step the *Global Interpreter Lock* but also means that only picklable objects can be executed and returned.

The __main__ module must be importable by worker subprocesses. This means that *ProcessPoolExecutor* will not work in the interactive interpreter.

Calling *Executor* or *Future* methods from a callable submitted to a *ProcessPoolExecutor* will result in deadlock.

class concurrent.futures.**ProcessPoolExecutor**(*max_workers=None*, *mp_context=None*, *initializer=None*, *initargs=()*, *max_tasks_per_child=None*)

An *Executor* subclass that executes calls asynchronously using a pool of at most *max_workers* processes. If *max_workers* is `None` or not given, it will default to *os.process_cpu_count()*. If *max_workers* is less than or equal to `0`, then a *ValueError* will be raised. On Windows, *max_workers* must be less than or equal to `61`. If it is not then *ValueError* will be raised. If *max_workers* is `None`, then the default chosen will be at most `61`, even if more processors are available. *mp_context* can be a *multiprocessing* context or `None`. It will be used to launch the workers. If *mp_context* is `None` or not given, the default *multiprocessing* context is used. See *Contexts and start methods*.

initializer is an optional callable that is called at the start of each worker process; *initargs* is a tuple of arguments passed to the initializer. Should *initializer* raise an exception, all currently pending jobs will raise a *BrokenProcessPool*, as well as any attempt to submit more jobs to the pool.

max_tasks_per_child is an optional argument that specifies the maximum number of tasks a single process can execute before it will exit and be replaced with a fresh worker process. By default *max_tasks_per_child* is `None` which means worker processes will live as long as the pool. When a max is specified, the "spawn" multiprocessing start method will be used by default in absence of a *mp_context* parameter. This feature is incompatible with the "fork" start method.

Changed in version 3.3: When one of the worker processes terminates abruptly, a *BrokenProcessPool* error is now raised. Previously, behaviour was undefined but operations on the executor or its futures would often freeze or deadlock.

Changed in version 3.7: The *mp_context* argument was added to allow users to control the start_method for worker processes created by the pool.

Added the *initializer* and *initargs* arguments.

> **Note**
>
> The default *multiprocessing* start method (see *Contexts and start methods*) will change away from *fork* in Python 3.14. Code that requires *fork* be used for their *ProcessPoolExecutor* should explicitly specify

that by passing a `mp_context=multiprocessing.get_context("fork")` parameter.

Changed in version 3.11: The *max_tasks_per_child* argument was added to allow users to control the lifetime of workers in the pool.

Changed in version 3.12: On POSIX systems, if your application has multiple threads and the *multiprocessing* context uses the `"fork"` start method: The *os.fork()* function called internally to spawn workers may raise a *DeprecationWarning*. Pass a *mp_context* configured to use a different start method. See the *os.fork()* documentation for further explanation.

Changed in version 3.13: *max_workers* uses *os.process_cpu_count()* by default, instead of *os.cpu_count()*.

ProcessPoolExecutor Example

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime))

if __name__ == '__main__':
    main()
```

17.5.4 Future Objects

The *Future* class encapsulates the asynchronous execution of a callable. *Future* instances are created by *Executor.submit()*.

class concurrent.futures.**Future**

Encapsulates the asynchronous execution of a callable. *Future* instances are created by *Executor.submit()* and should not be created directly except for testing.

cancel()

Attempt to cancel the call. If the call is currently being executed or finished running and cannot be cancelled then the method will return `False`, otherwise the call will be cancelled and the method will return `True`.

cancelled()

Return `True` if the call was successfully cancelled.

running()

Return `True` if the call is currently being executed and cannot be cancelled.

done()

Return `True` if the call was successfully cancelled or finished running.

result(*timeout=None*)

Return the value returned by the call. If the call hasn't yet completed then this method will wait up to *timeout* seconds. If the call hasn't completed in *timeout* seconds, then a *TimeoutError* will be raised. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If the future is cancelled before completing then *CancelledError* will be raised.

If the call raised an exception, this method will raise the same exception.

exception(*timeout=None*)

Return the exception raised by the call. If the call hasn't yet completed then this method will wait up to *timeout* seconds. If the call hasn't completed in *timeout* seconds, then a *TimeoutError* will be raised. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If the future is cancelled before completing then *CancelledError* will be raised.

If the call completed without raising, `None` is returned.

add_done_callback(*fn*)

Attaches the callable *fn* to the future. *fn* will be called, with the future as its only argument, when the future is cancelled or finishes running.

Added callables are called in the order that they were added and are always called in a thread belonging to the process that added them. If the callable raises an *Exception* subclass, it will be logged and ignored. If the callable raises a *BaseException* subclass, the behavior is undefined.

If the future has already completed or been cancelled, *fn* will be called immediately.

The following *Future* methods are meant for use in unit tests and *Executor* implementations.

set_running_or_notify_cancel()

This method should only be called by *Executor* implementations before executing the work associated with the *Future* and by unit tests.

If the method returns `False` then the *Future* was cancelled, i.e. *Future.cancel()* was called and returned `True`. Any threads waiting on the *Future* completing (i.e. through *as_completed()* or *wait()*) will be woken up.

If the method returns `True` then the *Future* was not cancelled and has been put in the running state, i.e. calls to *Future.running()* will return `True`.

This method can only be called once and cannot be called after *Future.set_result()* or *Future.set_exception()* have been called.

set_result(*result*)

Sets the result of the work associated with the *Future* to *result*.

This method should only be used by *Executor* implementations and unit tests.

Changed in version 3.8: This method raises *concurrent.futures.InvalidStateError* if the *Future* is already done.

set_exception(*exception*)

Sets the result of the work associated with the *Future* to the *Exception* *exception*.

This method should only be used by *Executor* implementations and unit tests.

Changed in version 3.8: This method raises *concurrent.futures.InvalidStateError* if the *Future* is already done.

17.5.5 Module Functions

concurrent.futures.**wait**(*fs*, *timeout=None*, *return_when=ALL_COMPLETED*)

Wait for the *Future* instances (possibly created by different *Executor* instances) given by *fs* to complete. Duplicate futures given to *fs* are removed and will be returned only once. Returns a named 2-tuple of sets. The first set, named `done`, contains the futures that completed (finished or cancelled futures) before the wait completed. The second set, named `not_done`, contains the futures that did not complete (pending or running futures).

timeout can be used to control the maximum number of seconds to wait before returning. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

return_when indicates when this function should return. It must be one of the following constants:

Constant	Description
concurrent.futures.**FIRST_COMPLETED**	The function will return when any future finishes or is cancelled.
concurrent.futures.**FIRST_EXCEPTION**	The function will return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to *ALL_COMPLETED*.
concurrent.futures.**ALL_COMPLETED**	The function will return when all futures finish or are cancelled.

concurrent.futures.**as_completed**(*fs*, *timeout=None*)

Returns an iterator over the *Future* instances (possibly created by different *Executor* instances) given by *fs* that yields futures as they complete (finished or cancelled futures). Any futures given by *fs* that are duplicated will be returned once. Any futures that completed before *as_completed()* is called will be yielded first. The returned iterator raises a *TimeoutError* if *__next__()* is called and the result isn't available after *timeout* seconds from the original call to *as_completed()*. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

See also

PEP 3148 – futures - execute computations asynchronously
The proposal which described this feature for inclusion in the Python standard library.

17.5.6 Exception classes

exception concurrent.futures.**CancelledError**

Raised when a future is cancelled.

exception concurrent.futures.**TimeoutError**

A deprecated alias of *TimeoutError*, raised when a future operation exceeds the given timeout.

Changed in version 3.11: This class was made an alias of *TimeoutError*.

exception concurrent.futures.**BrokenExecutor**

Derived from *RuntimeError*, this exception class is raised when an executor is broken for some reason, and cannot be used to submit or execute new tasks.

Added in version 3.7.

exception concurrent.futures.**InvalidStateError**

Raised when an operation is performed on a future that is not allowed in the current state.

Added in version 3.8.

exception concurrent.futures.thread.**BrokenThreadPool**

Derived from *BrokenExecutor*, this exception class is raised when one of the workers of a *ThreadPoolExecutor* has failed initializing.

Added in version 3.7.

exception concurrent.futures.process.**BrokenProcessPool**

Derived from *BrokenExecutor* (formerly *RuntimeError*), this exception class is raised when one of the workers of a *ProcessPoolExecutor* has terminated in a non-clean fashion (for example, if it was killed from the outside).

Added in version 3.3.

17.6 subprocess — Subprocess management

Source code: Lib/subprocess.py

The *subprocess* module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module intends to replace several older modules and functions:

```
os.system
os.spawn*
```

Information about how the *subprocess* module can be used to replace these modules and functions can be found in the following sections.

See also

PEP 324 – PEP proposing the subprocess module

Availability: not WASI, not iOS.

This module does not work or is not available on WebAssembly platforms, or on iOS. See *WebAssembly platforms* for more information on WASM availability; see *iOS* for more information on iOS availability.

17.6.1 Using the `subprocess` Module

The recommended approach to invoking subprocesses is to use the `run()` function for all use cases it can handle. For more advanced use cases, the underlying `Popen` interface can be used directly.

subprocess.**run**(*args*, *, *stdin=None*, *input=None*, *stdout=None*, *stderr=None*, *capture_output=False*, *shell=False*, *cwd=None*, *timeout=None*, *check=False*, *encoding=None*, *errors=None*, *text=None*, *env=None*, *universal_newlines=None*, **other_popen_kwargs*)

Run the command described by *args*. Wait for command to complete, then return a `CompletedProcess` instance.

The arguments shown above are merely the most common ones, described below in *Frequently Used Arguments* (hence the use of keyword-only notation in the abbreviated signature). The full function signature is largely the same as that of the `Popen` constructor - most of the arguments to this function are passed through to that interface. (*timeout*, *input*, *check*, and *capture_output* are not.)

If *capture_output* is true, stdout and stderr will be captured. When used, the internal `Popen` object is automatically created with *stdout* and *stderr* both set to `PIPE`. The *stdout* and *stderr* arguments may not be supplied at the same time as *capture_output*. If you wish to capture and combine both streams into one, set *stdout* to `PIPE` and *stderr* to `STDOUT`, instead of using *capture_output*.

A *timeout* may be specified in seconds, it is internally passed on to `Popen.communicate()`. If the timeout expires, the child process will be killed and waited for. The `TimeoutExpired` exception will be re-raised after the child process has terminated. The initial process creation itself cannot be interrupted on many platform APIs so you are not guaranteed to see a timeout exception until at least after however long process creation takes.

The *input* argument is passed to `Popen.communicate()` and thus to the subprocess's stdin. If used it must be a byte sequence, or a string if *encoding* or *errors* is specified or *text* is true. When used, the internal `Popen` object is automatically created with *stdin* set to `PIPE`, and the *stdin* argument may not be used as well.

If *check* is true, and the process exits with a non-zero exit code, a `CalledProcessError` exception will be raised. Attributes of that exception hold the arguments, the exit code, and stdout and stderr if they were captured.

If *encoding* or *errors* are specified, or *text* is true, file objects for stdin, stdout and stderr are opened in text mode using the specified *encoding* and *errors* or the `io.TextIOWrapper` default. The *universal_newlines* argument is equivalent to *text* and is provided for backwards compatibility. By default, file objects are opened in binary mode.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process' environment. It is passed directly to `Popen`. This mapping can be str to str on any platform or bytes to bytes on POSIX platforms much like `os.environ` or `os.environb`.

Examples:

```
>>> subprocess.run(["ls", "-l"])  # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)
```

(continues on next page)

(continued from previous page)

```
>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"], capture_output=True)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n', stderr=b'')
```

Added in version 3.5.

Changed in version 3.6: Added *encoding* and *errors* parameters

Changed in version 3.7: Added the *text* parameter, as a more understandable alias of *universal_newlines*. Added the *capture_output* parameter.

Changed in version 3.12: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

class subprocess.**CompletedProcess**

The return value from *run()*, representing a process that has finished.

args

The arguments used to launch the process. This may be a list or a string.

returncode

Exit status of the child process. Typically, an exit status of 0 indicates that it ran successfully.

A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).

stdout

Captured stdout from the child process. A bytes sequence, or a string if *run()* was called with an encoding, errors, or text=True. `None` if stdout was not captured.

If you ran the process with `stderr=subprocess.STDOUT`, stdout and stderr will be combined in this attribute, and *stderr* will be `None`.

stderr

Captured stderr from the child process. A bytes sequence, or a string if *run()* was called with an encoding, errors, or text=True. `None` if stderr was not captured.

check_returncode()

If *returncode* is non-zero, raise a *CalledProcessError*.

Added in version 3.5.

subprocess.**DEVNULL**

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to *Popen* and indicates that the special file *os.devnull* will be used.

Added in version 3.3.

subprocess.**PIPE**

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to *Popen* and indicates that a pipe to the standard stream should be opened. Most useful with *Popen.communicate()*.

subprocess.**STDOUT**

Special value that can be used as the *stderr* argument to *Popen* and indicates that standard error should go into the same handle as standard output.

exception subprocess.**SubprocessError**

Base class for all other exceptions from this module.

Added in version 3.3.

exception subprocess.**TimeoutExpired**

Subclass of *SubprocessError*, raised when a timeout expires while waiting for a child process.

cmd

Command that was used to spawn the child process.

timeout

Timeout in seconds.

output

Output of the child process if it was captured by *run()* or *check_output()*. Otherwise, None. This is always *bytes* when any output was captured regardless of the text=True setting. It may remain None instead of b'' when no output was observed.

stdout

Alias for output, for symmetry with *stderr*.

stderr

Stderr output of the child process if it was captured by *run()*. Otherwise, None. This is always *bytes* when stderr output was captured regardless of the text=True setting. It may remain None instead of b'' when no stderr output was observed.

Added in version 3.3.

Changed in version 3.5: *stdout* and *stderr* attributes added

exception subprocess.**CalledProcessError**

Subclass of *SubprocessError*, raised when a process run by *check_call()*, *check_output()*, or *run()* (with check=True) returns a non-zero exit status.

returncode

Exit status of the child process. If the process exited due to a signal, this will be the negative signal number.

cmd

Command that was used to spawn the child process.

output

Output of the child process if it was captured by *run()* or *check_output()*. Otherwise, None.

stdout

Alias for output, for symmetry with *stderr*.

stderr

Stderr output of the child process if it was captured by *run()*. Otherwise, None.

Changed in version 3.5: *stdout* and *stderr* attributes added

Frequently Used Arguments

To support a wide variety of use cases, the *Popen* constructor (and the convenience functions) accept a large number of optional arguments. For most typical use cases, many of these arguments can be safely left at their default values. The arguments that are most commonly needed are:

args is required for all calls and should be a string, or a sequence of program arguments. Providing a sequence of arguments is generally preferred, as it allows the module to take care of any required escaping and quoting of arguments (e.g. to permit spaces in file names). If passing a single string, either *shell* must be *True* (see below) or else the string must simply name the program to be executed without specifying any arguments.

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are None, *PIPE*, *DEVNULL*, an existing file descriptor (a positive integer), and an existing *file object* with a valid file descriptor. With the default settings of None, no redirection will occur. *PIPE* indicates that a new pipe to the child should be created. *DEVNULL* indicates that the special file *os.devnull* will be used. Additionally, *stderr* can be *STDOUT*, which indicates that the stderr data from the child process should be captured into the same file handle as for *stdout*.

If *encoding* or *errors* are specified, or *text* (also known as *universal_newlines*) is true, the file objects *stdin*, *stdout* and *stderr* will be opened in text mode using the *encoding* and *errors* specified in the call or the defaults for *io.TextIOWrapper*.

For *stdin*, line ending characters '\n' in the input will be converted to the default line separator *os.linesep*. For *stdout* and *stderr*, all line endings in the output will be converted to '\n'. For more information see the documentation of the *io.TextIOWrapper* class when the *newline* argument to its constructor is None.

If text mode is not used, *stdin*, *stdout* and *stderr* will be opened as binary streams. No encoding or line ending conversion is performed.

Changed in version 3.6: Added the *encoding* and *errors* parameters.

Changed in version 3.7: Added the *text* parameter as an alias for *universal_newlines*.

Note

The newlines attribute of the file objects *Popen.stdin*, *Popen.stdout* and *Popen.stderr* are not updated by the *Popen.communicate()* method.

If *shell* is True, the specified command will be executed through the shell. This can be useful if you are using Python primarily for the enhanced control flow it offers over most system shells and still want convenient access to other shell features such as shell pipes, filename wildcards, environment variable expansion, and expansion of ~ to a user's home directory. However, note that Python itself offers implementations of many shell-like features (in particular, *glob*, *fnmatch*, *os.walk()*, *os.path.expandvars()*, *os.path.expanduser()*, and *shutil*).

Changed in version 3.3: When *universal_newlines* is True, the class uses the encoding *locale.getpreferredencoding(False)* instead of locale.getpreferredencoding(). See the *io.TextIOWrapper* class for more information on this change.

Note

Read the *Security Considerations* section before using shell=True.

These options, along with all of the other options, are described in more detail in the *Popen* constructor documentation.

Popen Constructor

The underlying process creation and management in this module is handled by the *Popen* class. It offers a lot of flexibility so that developers are able to handle the less common cases not covered by the convenience functions.

class subprocess.**Popen**(*args*, *bufsize=-1*, *executable=None*, *stdin=None*, *stdout=None*, *stderr=None*, *preexec_fn=None*, *close_fds=True*, *shell=False*, *cwd=None*, *env=None*, *universal_newlines=None*, *startupinfo=None*, *creationflags=0*, *restore_signals=True*, *start_new_session=False*, *pass_fds=()*, ***, *group=None*, *extra_groups=None*, *user=None*, *umask=-1*, *encoding=None*, *errors=None*, *text=None*, *pipesize=-1*, *process_group=None*)

Execute a child program in a new process. On POSIX, the class uses *os.execvpe()*-like behavior to execute the child program. On Windows, the class uses the Windows `CreateProcess()` function. The arguments to *Popen* are as follows.

args should be a sequence of program arguments or else a single string or *path-like object*. By default, the program to execute is the first item in *args* if *args* is a sequence. If *args* is a string, the interpretation is platform-dependent and described below. See the *shell* and *executable* arguments for additional differences from the default behavior. Unless otherwise stated, it is recommended to pass *args* as a sequence.

Warning

For maximum reliability, use a fully qualified path for the executable. To search for an unqualified name on `PATH`, use *shutil.which()*. On all platforms, passing *sys.executable* is the recommended way to launch the current Python interpreter again, and use the `-m` command-line format to launch an installed module.

Resolving the path of *executable* (or the first item of *args*) is platform dependent. For POSIX, see *os.execvpe()*, and note that when resolving or searching for the executable path, *cwd* overrides the current working directory and *env* can override the `PATH` environment variable. For Windows, see the documentation of the `lpApplicationName` and `lpCommandLine` parameters of WinAPI `CreateProcess`, and note that when resolving or searching for the executable path with `shell=False`, *cwd* does not override the current working directory and *env* cannot override the `PATH` environment variable. Using a full path avoids all of these variations.

An example of passing some arguments to an external program as a sequence is:

```
Popen(["/usr/bin/git", "commit", "-m", "Fixes a bug."])
```

On POSIX, if *args* is a string, the string is interpreted as the name or path of the program to execute. However, this can only be done if not passing arguments to the program.

Note

It may not be obvious how to break a shell command into a sequence of arguments, especially in complex cases. *shlex.split()* can illustrate how to determine the correct tokenization for *args*:

```
>>> import shlex, subprocess
>>> command_line = input()
/bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
>>> args = shlex.split(command_line)
>>> print(args)
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd',
↪"echo '$MONEY'"]
>>> p = subprocess.Popen(args) # Success!
```

Note in particular that options (such as *-input*) and arguments (such as *eggs.txt*) that are separated by whitespace in the shell go in separate list elements, while arguments that need quoting or backslash escaping when used in the shell (such as filenames containing spaces or the *echo* command shown above) are single list elements.

On Windows, if *args* is a sequence, it will be converted to a string in a manner described in *Converting an argument sequence to a string on Windows*. This is because the underlying `CreateProcess()` operates on strings.

Changed in version 3.6: *args* parameter accepts a *path-like object* if *shell* is `False` and a sequence containing path-like objects on POSIX.

Changed in version 3.8: *args* parameter accepts a *path-like object* if *shell* is `False` and a sequence containing bytes and path-like objects on Windows.

The *shell* argument (which defaults to `False`) specifies whether to use the shell as the program to execute. If *shell* is `True`, it is recommended to pass *args* as a string rather than as a sequence.

On POSIX with `shell=True`, the shell defaults to `/bin/sh`. If *args* is a string, the string specifies the command to execute through the shell. This means that the string must be formatted exactly as it would be when typed at the shell prompt. This includes, for example, quoting or backslash escaping filenames with spaces in them. If *args* is a sequence, the first item specifies the command string, and any additional items will be treated as additional arguments to the shell itself. That is to say, *Popen* does the equivalent of:

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

On Windows with `shell=True`, the `COMSPEC` environment variable specifies the default shell. The only time you need to specify `shell=True` on Windows is when the command you wish to execute is built into the shell (e.g. **`dir`** or **`copy`**). You do not need `shell=True` to run a batch file or console-based executable.

Note

Read the *Security Considerations* section before using `shell=True`.

bufsize will be supplied as the corresponding argument to the *open()* function when creating the stdin/stdout/stderr pipe file objects:

- `0` means unbuffered (read and write are one system call and can return short)
- `1` means line buffered (only usable if `text=True` or `universal_newlines=True`)
- any other positive value means use a buffer of approximately that size
- negative bufsize (the default) means the system default of io.DEFAULT_BUFFER_SIZE will be used.

Changed in version 3.3.1: *bufsize* now defaults to -1 to enable buffering by default to match the behavior that most code expects. In versions prior to Python 3.2.4 and 3.3.1 it incorrectly defaulted to `0` which was unbuffered and allowed short reads. This was unintentional and did not match the behavior of Python 2 as most code expected.

The *executable* argument specifies a replacement program to execute. It is very seldom needed. When `shell=False`, *executable* replaces the program to execute specified by *args*. However, the original *args* is still passed to the program. Most programs treat the program specified by *args* as the command name, which can then be different from the program actually executed. On POSIX, the *args* name becomes the display name for the executable in utilities such as **`ps`**. If `shell=True`, on POSIX the *executable* argument specifies a replacement shell for the default `/bin/sh`.

Changed in version 3.6: *executable* parameter accepts a *path-like object* on POSIX.

Changed in version 3.8: *executable* parameter accepts a bytes and *path-like object* on Windows.

Changed in version 3.12: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are `None`, *PIPE*, *DEVNULL*, an existing file descriptor (a positive integer), and an existing *file object* with a valid file descriptor. With the default settings of `None`, no redirection will occur. *PIPE* indicates that a new pipe to the child should be created. *DEVNULL* indicates that the special file *os.devnull* will be used. Additionally, *stderr* can be *STDOUT*, which indicates that the stderr data from the applications should be captured into the same file handle as for *stdout*.

If *preexec_fn* is set to a callable object, this object will be called in the child process just before the child is executed. (POSIX only)

Warning

The *preexec_fn* parameter is NOT SAFE to use in the presence of threads in your application. The child process could deadlock before exec is called.

Note

If you need to modify the environment for the child use the *env* parameter rather than doing it in a *preexec_fn*. The *start_new_session* and *process_group* parameters should take the place of code using *preexec_fn* to call *os.setsid()* or *os.setpgid()* in the child.

Changed in version 3.8: The *preexec_fn* parameter is no longer supported in subinterpreters. The use of the parameter in a subinterpreter raises *RuntimeError*. The new restriction may affect applications that are deployed in mod_wsgi, uWSGI, and other embedded environments.

If *close_fds* is true, all file descriptors except `0`, `1` and `2` will be closed before the child process is executed. Otherwise when *close_fds* is false, file descriptors obey their inheritable flag as described in *Inheritance of File Descriptors*.

On Windows, if *close_fds* is true then no handles will be inherited by the child process unless explicitly passed in the `handle_list` element of *STARTUPINFO.lpAttributeList*, or by standard handle redirection.

Changed in version 3.2: The default for *close_fds* was changed from *False* to what is described above.

Changed in version 3.7: On Windows the default for *close_fds* was changed from *False* to *True* when redirecting the standard handles. It's now possible to set *close_fds* to *True* when redirecting the standard handles.

pass_fds is an optional sequence of file descriptors to keep open between the parent and child. Providing any *pass_fds* forces *close_fds* to be *True*. (POSIX only)

Changed in version 3.2: The *pass_fds* parameter was added.

If *cwd* is not `None`, the function changes the working directory to *cwd* before executing the child. *cwd* can be a string, bytes or *path-like* object. On POSIX, the function looks for *executable* (or for the first item in *args*) relative to *cwd* if the executable path is a relative path.

Changed in version 3.6: *cwd* parameter accepts a *path-like object* on POSIX.

Changed in version 3.7: *cwd* parameter accepts a *path-like object* on Windows.

Changed in version 3.8: *cwd* parameter accepts a bytes object on Windows.

If *restore_signals* is true (the default) all signals that Python has set to SIG_IGN are restored to SIG_DFL in the child process before the exec. Currently this includes the SIGPIPE, SIGXFZ and SIGXFSZ signals. (POSIX only)

Changed in version 3.2: *restore_signals* was added.

If *start_new_session* is true the `setsid()` system call will be made in the child process prior to the execution of the subprocess.

Availability: POSIX

Changed in version 3.2: *start_new_session* was added.

If *process_group* is a non-negative integer, the `setpgid(0, value)` system call will be made in the child process prior to the execution of the subprocess.

Availability: POSIX

Changed in version 3.11: *process_group* was added.

If *group* is not `None`, the setregid() system call will be made in the child process prior to the execution of the subprocess. If the provided value is a string, it will be looked up via *grp.getgrnam()* and the value in `gr_gid` will be used. If the value is an integer, it will be passed verbatim. (POSIX only)

Availability: POSIX

Added in version 3.9.

If *extra_groups* is not `None`, the setgroups() system call will be made in the child process prior to the execution of the subprocess. Strings provided in *extra_groups* will be looked up via *grp.getgrnam()* and the values in `gr_gid` will be used. Integer values will be passed verbatim. (POSIX only)

Availability: POSIX

Added in version 3.9.

If *user* is not `None`, the setreuid() system call will be made in the child process prior to the execution of the subprocess. If the provided value is a string, it will be looked up via *pwd.getpwnam()* and the value in `pw_uid` will be used. If the value is an integer, it will be passed verbatim. (POSIX only)

Availability: POSIX

Added in version 3.9.

If *umask* is not negative, the umask() system call will be made in the child process prior to the execution of the subprocess.

Availability: POSIX

Added in version 3.9.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process' environment. This mapping can be str to str on any platform or bytes to bytes on POSIX platforms much like *os.environ* or *os.environb*.

Note

If specified, *env* must provide any variables required for the program to execute. On Windows, in order to run a side-by-side assembly the specified *env* **must** include a valid `SystemRoot`.

If *encoding* or *errors* are specified, or *text* is true, the file objects *stdin*, *stdout* and *stderr* are opened in text mode with the specified *encoding* and *errors*, as described above in *Frequently Used Arguments*. The *universal_newlines* argument is equivalent to *text* and is provided for backwards compatibility. By default, file objects are opened in binary mode.

Added in version 3.6: *encoding* and *errors* were added.

Added in version 3.7: *text* was added as a more readable alias for *universal_newlines*.

If given, *startupinfo* will be a *STARTUPINFO* object, which is passed to the underlying `CreateProcess` function.

If given, *creationflags*, can be one or more of the following flags:

- *CREATE_NEW_CONSOLE*
- *CREATE_NEW_PROCESS_GROUP*
- *ABOVE_NORMAL_PRIORITY_CLASS*
- *BELOW_NORMAL_PRIORITY_CLASS*
- *HIGH_PRIORITY_CLASS*
- *IDLE_PRIORITY_CLASS*
- *NORMAL_PRIORITY_CLASS*
- *REALTIME_PRIORITY_CLASS*
- *CREATE_NO_WINDOW*
- *DETACHED_PROCESS*
- *CREATE_DEFAULT_ERROR_MODE*
- *CREATE_BREAKAWAY_FROM_JOB*

pipesize can be used to change the size of the pipe when *PIPE* is used for *stdin*, *stdout* or *stderr*. The size of the pipe is only changed on platforms that support this (only Linux at this time of writing). Other platforms will ignore this parameter.

Changed in version 3.10: Added the *pipesize* parameter.

Popen objects are supported as context managers via the `with` statement: on exit, standard file descriptors are closed, and the process is waited for.

```
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

Popen and the other functions in this module that use it raise an *auditing event* `subprocess.Popen` with arguments `executable`, `args`, `cwd`, and `env`. The value for `args` may be a single string or a list of strings, depending on platform.

Changed in version 3.2: Added context manager support.

Changed in version 3.6: Popen destructor now emits a *ResourceWarning* warning if the child process is still running.

Changed in version 3.8: Popen can use *os.posix_spawn()* in some cases for better performance. On Windows Subsystem for Linux and QEMU User Emulation, Popen constructor using *os.posix_spawn()* no longer raise an exception on errors like missing program, but the child process fails with a non-zero *returncode*.

Exceptions

Exceptions raised in the child process, before the new program has started to execute, will be re-raised in the parent.

The most common exception raised is *OSError*. This occurs, for example, when trying to execute a non-existent file. Applications should prepare for *OSError* exceptions. Note that, when `shell=True`, *OSError* will be raised by the child only if the selected shell itself was not found. To determine if the shell failed to find the requested application, it is necessary to check the return code or output from the subprocess.

A *ValueError* will be raised if *Popen* is called with invalid arguments.

check_call() and *check_output()* will raise *CalledProcessError* if the called process returns a non-zero return code.

All of the functions and methods that accept a *timeout* parameter, such as *run()* and *Popen.communicate()* will raise *TimeoutExpired* if the timeout expires before the process exits.

Exceptions defined in this module all inherit from *SubprocessError*.

Added in version 3.3: The *SubprocessError* base class was added.

17.6.2 Security Considerations

Unlike some other popen functions, this library will not implicitly choose to call a system shell. This means that all characters, including shell metacharacters, can safely be passed to child processes. If the shell is invoked explicitly, via `shell=True`, it is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately to avoid shell injection vulnerabilities. On *some platforms*, it is possible to use *shlex.quote()* for this escaping.

On Windows, batch files (`*.bat` or `*.cmd`) may be launched by the operating system in a system shell regardless of the arguments passed to this library. This could result in arguments being parsed according to shell rules, but without any escaping added by Python. If you are intentionally launching a batch file with arguments from untrusted sources, consider passing `shell=True` to allow Python to escape special characters. See gh-114539 for additional discussion.

17.6.3 Popen Objects

Instances of the *Popen* class have the following methods:

`Popen.`**`poll`**`()`

Check if child process has terminated. Set and return *returncode* attribute. Otherwise, returns `None`.

`Popen.`**`wait`**`(`*timeout=None*`)`

Wait for child process to terminate. Set and return *returncode* attribute.

If the process does not terminate after *timeout* seconds, raise a *TimeoutExpired* exception. It is safe to catch this exception and retry the wait.

> **Note**
>
> This will deadlock when using `stdout=PIPE` or `stderr=PIPE` and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use *Popen.communicate()* when using pipes to avoid that.

> **Note**
>
> When the `timeout` parameter is not `None`, then (on POSIX) the function is implemented using a busy loop (non-blocking call and short sleeps). Use the *asyncio* module for an asynchronous wait: see *asyncio.create_subprocess_exec*.

Changed in version 3.3: *timeout* was added.

`Popen.`**`communicate`**(*input=None*, *timeout=None*)

Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate and set the *returncode* attribute. The optional *input* argument should be data to be sent to the child process, or `None`, if no data should be sent to the child. If streams were opened in text mode, *input* must be a string. Otherwise, it must be bytes.

communicate() returns a tuple `(stdout_data, stderr_data)`. The data will be strings if streams were opened in text mode; otherwise, bytes.

Note that if you want to send data to the process's stdin, you need to create the Popen object with `stdin=PIPE`. Similarly, to get anything other than `None` in the result tuple, you need to give `stdout=PIPE` and/or `stderr=PIPE` too.

If the process does not terminate after *timeout* seconds, a *TimeoutExpired* exception will be raised. Catching this exception and retrying communication will not lose any output.

The child process is not killed if the timeout expires, so in order to cleanup properly a well-behaved application should kill the child process and finish communication:

```
proc = subprocess.Popen(...)
try:
    outs, errs = proc.communicate(timeout=15)
except TimeoutExpired:
    proc.kill()
    outs, errs = proc.communicate()
```

> **Note**
>
> The data read is buffered in memory, so do not use this method if the data size is large or unlimited.

Changed in version 3.3: *timeout* was added.

`Popen.`**`send_signal`**(*signal*)

Sends the signal *signal* to the child.

Do nothing if the process completed.

> **Note**
>
> On Windows, SIGTERM is an alias for *terminate()*. CTRL_C_EVENT and CTRL_BREAK_EVENT can be sent to processes started with a *creationflags* parameter which includes `CREATE_NEW_PROCESS_GROUP`.

`Popen.`**`terminate`**()

Stop the child. On POSIX OSs the method sends *SIGTERM* to the child. On Windows the Win32 API function `TerminateProcess()` is called to stop the child.

Popen.**kill**()

Kills the child. On POSIX OSs the function sends SIGKILL to the child. On Windows *kill()* is an alias for *terminate()*.

The following attributes are also set by the class for you to access. Reassigning them to new values is unsupported:

Popen.**args**

The *args* argument as it was passed to *Popen* – a sequence of program arguments or else a single string.

Added in version 3.3.

Popen.**stdin**

If the *stdin* argument was *PIPE*, this attribute is a writeable stream object as returned by *open()*. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdin* argument was not *PIPE*, this attribute is `None`.

Popen.**stdout**

If the *stdout* argument was *PIPE*, this attribute is a readable stream object as returned by *open()*. Reading from the stream provides output from the child process. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdout* argument was not *PIPE*, this attribute is `None`.

Popen.**stderr**

If the *stderr* argument was *PIPE*, this attribute is a readable stream object as returned by *open()*. Reading from the stream provides error output from the child process. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stderr* argument was not *PIPE*, this attribute is `None`.

Warning

Use *communicate()* rather than *.stdin.write*, *.stdout.read* or *.stderr.read* to avoid deadlocks due to any of the other OS pipe buffers filling up and blocking the child process.

Popen.**pid**

The process ID of the child process.

Note that if you set the *shell* argument to `True`, this is the process ID of the spawned shell.

Popen.**returncode**

The child return code. Initially `None`, *returncode* is set by a call to the *poll()*, *wait()*, or *communicate()* methods if they detect that the process has terminated.

A `None` value indicates that the process hadn't yet terminated at the time of the last method call.

A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).

17.6.4 Windows Popen Helpers

The *STARTUPINFO* class and following constants are only available on Windows.

class subprocess.**STARTUPINFO**(**, dwFlags=0, hStdInput=None, hStdOutput=None, hStdError=None, wShowWindow=0, lpAttributeList=None*)

Partial support of the Windows STARTUPINFO structure is used for *Popen* creation. The following attributes can be set by passing them as keyword-only arguments.

Changed in version 3.7: Keyword-only argument support was added.

dwFlags

A bit field that determines whether certain *STARTUPINFO* attributes are used when the process creates a window.

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_USESHOWWINDOW
```

hStdInput

If *dwFlags* specifies *STARTF_USESTDHANDLES*, this attribute is the standard input handle for the process. If *STARTF_USESTDHANDLES* is not specified, the default for standard input is the keyboard buffer.

hStdOutput

If *dwFlags* specifies *STARTF_USESTDHANDLES*, this attribute is the standard output handle for the process. Otherwise, this attribute is ignored and the default for standard output is the console window's buffer.

hStdError

If *dwFlags* specifies *STARTF_USESTDHANDLES*, this attribute is the standard error handle for the process. Otherwise, this attribute is ignored and the default for standard error is the console window's buffer.

wShowWindow

If *dwFlags* specifies *STARTF_USESHOWWINDOW*, this attribute can be any of the values that can be specified in the `nCmdShow` parameter for the ShowWindow function, except for `SW_SHOWDEFAULT`. Otherwise, this attribute is ignored.

SW_HIDE is provided for this attribute. It is used when *Popen* is called with `shell=True`.

lpAttributeList

A dictionary of additional attributes for process creation as given in `STARTUPINFOEX`, see UpdateProcThreadAttribute.

Supported attributes:

handle_list

Sequence of handles that will be inherited. *close_fds* must be true if non-empty.

The handles must be temporarily made inheritable by *os.set_handle_inheritable()* when passed to the *Popen* constructor, else *OSError* will be raised with Windows error `ERROR_INVALID_PARAMETER` (87).

> **Warning**
>
> In a multithreaded process, use caution to avoid leaking handles that are marked inheritable when combining this feature with concurrent calls to other process creation functions that inherit all handles such as *os.system()*. This also applies to standard handle redirection, which temporarily creates inheritable handles.

Added in version 3.7.

Windows Constants

The *subprocess* module exposes the following constants.

subprocess.**STD_INPUT_HANDLE**
The standard input device. Initially, this is the console input buffer, CONIN$.

subprocess.**STD_OUTPUT_HANDLE**
The standard output device. Initially, this is the active console screen buffer, CONOUT$.

subprocess.**STD_ERROR_HANDLE**
The standard error device. Initially, this is the active console screen buffer, CONOUT$.

subprocess.**SW_HIDE**
Hides the window. Another window will be activated.

subprocess.**STARTF_USESTDHANDLES**
Specifies that the *STARTUPINFO.hStdInput*, *STARTUPINFO.hStdOutput*, and *STARTUPINFO.hStdError* attributes contain additional information.

subprocess.**STARTF_USESHOWWINDOW**
Specifies that the *STARTUPINFO.wShowWindow* attribute contains additional information.

subprocess.**STARTF_FORCEONFEEDBACK**
A *STARTUPINFO.dwFlags* parameter to specify that the *Working in Background* mouse cursor will be displayed while a process is launching. This is the default behavior for GUI processes.

Added in version 3.13.

subprocess.**STARTF_FORCEOFFFEEDBACK**
A *STARTUPINFO.dwFlags* parameter to specify that the mouse cursor will not be changed when launching a process.

Added in version 3.13.

subprocess.**CREATE_NEW_CONSOLE**
The new process has a new console, instead of inheriting its parent's console (the default).

subprocess.**CREATE_NEW_PROCESS_GROUP**
A *Popen* `creationflags` parameter to specify that a new process group will be created. This flag is necessary for using *os.kill()* on the subprocess.

This flag is ignored if *CREATE_NEW_CONSOLE* is specified.

subprocess.**ABOVE_NORMAL_PRIORITY_CLASS**
A *Popen* `creationflags` parameter to specify that a new process will have an above average priority.

Added in version 3.7.

subprocess.**BELOW_NORMAL_PRIORITY_CLASS**
A *Popen* `creationflags` parameter to specify that a new process will have a below average priority.

Added in version 3.7.

subprocess.**HIGH_PRIORITY_CLASS**
A *Popen* `creationflags` parameter to specify that a new process will have a high priority.

Added in version 3.7.

subprocess.**IDLE_PRIORITY_CLASS**

A *Popen* `creationflags` parameter to specify that a new process will have an idle (lowest) priority.

Added in version 3.7.

subprocess.**NORMAL_PRIORITY_CLASS**

A *Popen* `creationflags` parameter to specify that a new process will have a normal priority. (default)

Added in version 3.7.

subprocess.**REALTIME_PRIORITY_CLASS**

A *Popen* `creationflags` parameter to specify that a new process will have realtime priority. You should almost never use REALTIME_PRIORITY_CLASS, because this interrupts system threads that manage mouse input, keyboard input, and background disk flushing. This class can be appropriate for applications that "talk" directly to hardware or that perform brief tasks that should have limited interruptions.

Added in version 3.7.

subprocess.**CREATE_NO_WINDOW**

A *Popen* `creationflags` parameter to specify that a new process will not create a window.

Added in version 3.7.

subprocess.**DETACHED_PROCESS**

A *Popen* `creationflags` parameter to specify that a new process will not inherit its parent's console. This value cannot be used with CREATE_NEW_CONSOLE.

Added in version 3.7.

subprocess.**CREATE_DEFAULT_ERROR_MODE**

A *Popen* `creationflags` parameter to specify that a new process does not inherit the error mode of the calling process. Instead, the new process gets the default error mode. This feature is particularly useful for multithreaded shell applications that run with hard errors disabled.

Added in version 3.7.

subprocess.**CREATE_BREAKAWAY_FROM_JOB**

A *Popen* `creationflags` parameter to specify that a new process is not associated with the job.

Added in version 3.7.

17.6.5 Older high-level API

Prior to Python 3.5, these three functions comprised the high level API to subprocess. You can now use *run()* in many cases, but lots of existing code calls these functions.

subprocess.**call**(*args*, ***, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, ***other_popen_kwargs*)

Run the command described by *args*. Wait for command to complete, then return the *returncode* attribute.

Code needing to capture stdout or stderr should use *run()* instead:

```
run(...).returncode
```

To suppress stdout or stderr, supply a value of *DEVNULL*.

The arguments shown above are merely some common ones. The full function signature is the same as that of the *Popen* constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

> **Note**
>
> Do not use `stdout=PIPE` or `stderr=PIPE` with this function. The child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

Changed in version 3.3: *timeout* was added.

Changed in version 3.12: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

`subprocess.`**`check_call`**(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, ***other_popen_kwargs*)

Run command with arguments. Wait for command to complete. If the return code was zero then return, otherwise raise *CalledProcessError*. The *CalledProcessError* object will have the return code in the *returncode* attribute. If *check_call()* was unable to start the process it will propagate the exception that was raised.

Code needing to capture stdout or stderr should use *run()* instead:

```
run(..., check=True)
```

To suppress stdout or stderr, supply a value of *DEVNULL*.

The arguments shown above are merely some common ones. The full function signature is the same as that of the *Popen* constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

> **Note**
>
> Do not use `stdout=PIPE` or `stderr=PIPE` with this function. The child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

Changed in version 3.3: *timeout* was added.

Changed in version 3.12: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

`subprocess.`**`check_output`**(*args*, *, *stdin=None*, *stderr=None*, *shell=False*, *cwd=None*, *encoding=None*, *errors=None*, *universal_newlines=None*, *timeout=None*, *text=None*, ***other_popen_kwargs*)

Run command with arguments and return its output.

If the return code was non-zero it raises a *CalledProcessError*. The *CalledProcessError* object will have the return code in the *returncode* attribute and any output in the *output* attribute.

This is equivalent to:

```
run(..., check=True, stdout=PIPE).stdout
```

The arguments shown above are merely some common ones. The full function signature is largely the same as that of *run()* - most arguments are passed directly through to that interface. One API deviation from *run()* behavior exists: passing `input=None` will behave the same as `input=b''` (or `input=''`, depending on other arguments) rather than using the parent's standard input file handle.

By default, this function will return the data as encoded bytes. The actual encoding of the output data may depend on the command being invoked, so the decoding to text will often need to be handled at the application level.

This behaviour may be overridden by setting *text*, *encoding*, *errors*, or *universal_newlines* to `True` as described in *Frequently Used Arguments* and `run()`.

To also capture standard error in the result, use `stderr=subprocess.STDOUT`:

```
>>> subprocess.check_output(
...     "ls non_existent_file; exit 0",
...     stderr=subprocess.STDOUT,
...     shell=True)
'ls: non_existent_file: No such file or directory\n'
```

Added in version 3.1.

Changed in version 3.3: *timeout* was added.

Changed in version 3.4: Support for the *input* keyword argument was added.

Changed in version 3.6: *encoding* and *errors* were added. See `run()` for details.

Added in version 3.7: *text* was added as a more readable alias for *universal_newlines*.

Changed in version 3.12: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

17.6.6 Replacing Older Functions with the `subprocess` Module

In this section, "a becomes b" means that b can be used as a replacement for a.

Note

All "a" functions in this section fail (more or less) silently if the executed program cannot be found; the "b" replacements raise `OSError` instead.

In addition, the replacements using `check_output()` will fail with a `CalledProcessError` if the requested operation produces a non-zero return code. The output is still available as the `output` attribute of the raised exception.

In the following examples, we assume that the relevant functions have already been imported from the `subprocess` module.

Replacing `/bin/sh` shell command substitution

```
output=$(mycmd myarg)
```

becomes:

```
output = check_output(["mycmd", "myarg"])
```

Replacing shell pipeline

```
output=$(dmesg | grep hda)
```

becomes:

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close()  # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

The `p1.stdout.close()` call after starting the p2 is important in order for p1 to receive a SIGPIPE if p2 exits before p1.

Alternatively, for trusted input, the shell's own pipeline support may still be used directly:

```
output=$(dmesg | grep hda)
```

becomes:

```
output = check_output("dmesg | grep hda", shell=True)
```

Replacing `os.system()`

```
sts = os.system("mycmd" + " myarg")
# becomes
retcode = call("mycmd" + " myarg", shell=True)
```

Notes:

- Calling the program through the shell is usually not required.
- The *`call()`* return value is encoded differently to that of *`os.system()`*.
- The *`os.system()`* function ignores SIGINT and SIGQUIT signals while the command is running, but the caller must do this separately when using the *`subprocess`* module.

A more realistic example would look like this:

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode, file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

Replacing the `os.spawn` family

P_NOWAIT example:

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

P_WAIT example:

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

Vector example:

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

Environment example:

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

Replacing `os.popen()`, `os.popen2()`, `os.popen3()`

```
(child_stdin, child_stdout) = os.popen2(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

```
(child_stdin,
 child_stdout,
 child_stderr) = os.popen3(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin,
 child_stdout,
 child_stderr) = (p.stdin, p.stdout, p.stderr)
```

```
(child_stdin, child_stdout_and_stderr) = os.popen4(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

Return code handling translates as follows:

```
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
```

(continues on next page)

(continued from previous page)

```
if rc is not None and rc >> 8:
    print("There were some errors")
==>
process = Popen(cmd, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")
```

Replacing functions from the popen2 module

> **Note**
>
> If the cmd argument to popen2 functions is a string, the command is executed through /bin/sh. If it is a list, the command is directly executed.

```
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize, mode)
==>
p = Popen("somestring", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

```
(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufsize, mode)
==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

`popen2.Popen3` and `popen2.Popen4` basically work as *`subprocess.Popen`*, except that:

- *`Popen`* raises an exception if the execution fails.
- The *capturestderr* argument is replaced with the *stderr* argument.
- `stdin=PIPE` and `stdout=PIPE` must be specified.
- popen2 closes all file descriptors by default, but you have to specify `close_fds=True` with *`Popen`* to guarantee this behavior on all platforms or past Python versions.

17.6.7 Legacy Shell Invocation Functions

This module also provides the following legacy functions from the 2.x `commands` module. These operations implicitly invoke the system shell and none of the guarantees described above regarding security and exception handling consistency are valid for these functions.

`subprocess.`**`getstatusoutput`**(*cmd*, *, *encoding=None*, *errors=None*)

Return `(exitcode, output)` of executing *cmd* in a shell.

Execute the string *cmd* in a shell with `Popen.check_output()` and return a 2-tuple `(exitcode, output)`. *encoding* and *errors* are used to decode output; see the notes on *Frequently Used Arguments* for more details.

A trailing newline is stripped from the output. The exit code for the command can be interpreted as the return code of subprocess. Example:

```
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(1, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(127, 'sh: /bin/junk: not found')
>>> subprocess.getstatusoutput('/bin/kill $$')
(-15, '')
```

Availability: Unix, Windows.

Changed in version 3.3.4: Windows support was added.

The function now returns (exitcode, output) instead of (status, output) as it did in Python 3.3.3 and earlier. exitcode has the same value as *returncode*.

Changed in version 3.11: Added the *encoding* and *errors* parameters.

subprocess.**getoutput**(*cmd*, *, *encoding=None*, *errors=None*)

Return output (stdout and stderr) of executing *cmd* in a shell.

Like *getstatusoutput()*, except the exit code is ignored and the return value is a string containing the command's output. Example:

```
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

Availability: Unix, Windows.

Changed in version 3.3.4: Windows support added

Changed in version 3.11: Added the *encoding* and *errors* parameters.

17.6.8 Notes

Converting an argument sequence to a string on Windows

On Windows, an *args* sequence is converted to a string that can be parsed using the following rules (which correspond to the rules used by the MS C runtime):

1. Arguments are delimited by white space, which is either a space or a tab.
2. A string surrounded by double quotation marks is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.
3. A double quotation mark preceded by a backslash is interpreted as a literal double quotation mark.
4. Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
5. If backslashes immediately precede a double quotation mark, every pair of backslashes is interpreted as a literal backslash. If the number of backslashes is odd, the last backslash escapes the next double quotation mark as described in rule 3.

See also

shlex
Module which provides function to parse and escape command lines.

Disabling use of `vfork()` or `posix_spawn()`

On Linux, *subprocess* defaults to using the `vfork()` system call internally when it is safe to do so rather than `fork()`. This greatly improves performance.

If you ever encounter a presumed highly unusual situation where you need to prevent `vfork()` from being used by Python, you can set the `subprocess._USE_VFORK` attribute to a false value.

```
subprocess._USE_VFORK = False  # See CPython issue gh-NNNNNN.
```

Setting this has no impact on use of `posix_spawn()` which could use `vfork()` internally within its libc implementation. There is a similar `subprocess._USE_POSIX_SPAWN` attribute if you need to prevent use of that.

```
subprocess._USE_POSIX_SPAWN = False  # See CPython issue gh-NNNNNN.
```

It is safe to set these to false on any Python version. They will have no effect on older versions when unsupported. Do not assume the attributes are available to read. Despite their names, a true value does not indicate that the corresponding function will be used, only that it may be.

Please file issues any time you have to use these private knobs with a way to reproduce the issue you were seeing. Link to that issue from a comment in your code.

Added in version 3.8: `_USE_POSIX_SPAWN`

Added in version 3.11: `_USE_VFORK`

17.7 `sched` — Event scheduler

Source code: Lib/sched.py

The *sched* module defines a class which implements a general purpose event scheduler:

class `sched.`**scheduler**(*timefunc=time.monotonic*, *delayfunc=time.sleep*)

The *scheduler* class defines a generic interface to scheduling events. It needs two functions to actually deal with the "outside world" — *timefunc* should be callable without arguments, and return a number (the "time", in any units whatsoever). The *delayfunc* function should be callable with one argument, compatible with the output of *timefunc*, and should delay that many time units. *delayfunc* will also be called with the argument `0` after each event is run to allow other threads an opportunity to run in multi-threaded applications.

Changed in version 3.3: *timefunc* and *delayfunc* parameters are optional.

Changed in version 3.3: *scheduler* class can be safely used in multi-threaded environments.

Example:

```
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(a='default'):
...     print("From print_time", time.time(), a)
...
>>> def print_some_times():
...     print(time.time())
...     s.enter(10, 1, print_time)
...     s.enter(5, 2, print_time, argument=('positional',))
...     # despite having higher priority, 'keyword' runs after 'positional' as␣
↪enter() is relative
```

(continues on next page)

(continued from previous page)

```
...     s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
...     s.enterabs(1_650_000_000, 10, print_time, argument=("first enterabs",))
...     s.enterabs(1_650_000_000, 5, print_time, argument=("second enterabs",))
...     s.run()
...     print(time.time())
...
>>> print_some_times()
1652342830.3640375
From print_time 1652342830.3642538 second enterabs
From print_time 1652342830.3643398 first enterabs
From print_time 1652342835.3694863 positional
From print_time 1652342835.3696074 keyword
From print_time 1652342840.369612 default
1652342840.3697174
```

17.7.1 Scheduler Objects

scheduler instances have the following methods and attributes:

scheduler.**enterabs**(*time*, *priority*, *action*, *argument=()*, *kwargs={}*)

Schedule a new event. The *time* argument should be a numeric type compatible with the return value of the *timefunc* function passed to the constructor. Events scheduled for the same *time* will be executed in the order of their *priority*. A lower number represents a higher priority.

Executing the event means executing `action(*argument, **kwargs)`. *argument* is a sequence holding the positional arguments for *action*. *kwargs* is a dictionary holding the keyword arguments for *action*.

Return value is an event which may be used for later cancellation of the event (see *cancel()*).

Changed in version 3.3: *argument* parameter is optional.

Changed in version 3.3: *kwargs* parameter was added.

scheduler.**enter**(*delay*, *priority*, *action*, *argument=()*, *kwargs={}*)

Schedule an event for *delay* more time units. Other than the relative time, the other arguments, the effect and the return value are the same as those for *enterabs()*.

Changed in version 3.3: *argument* parameter is optional.

Changed in version 3.3: *kwargs* parameter was added.

scheduler.**cancel**(*event*)

Remove the event from the queue. If *event* is not an event currently in the queue, this method will raise a *ValueError*.

scheduler.**empty**()

Return `True` if the event queue is empty.

scheduler.**run**(*blocking=True*)

Run all scheduled events. This method will wait (using the *delayfunc* function passed to the constructor) for the next event, then execute it and so on until there are no more scheduled events.

If *blocking* is false executes the scheduled events due to expire soonest (if any) and then return the deadline of the next scheduled call in the scheduler (if any).

Either *action* or *delayfunc* can raise an exception. In either case, the scheduler will maintain a consistent state and propagate the exception. If an exception is raised by *action*, the event will not be attempted in future calls to *run()*.

If a sequence of events takes longer to run than the time available before the next event, the scheduler will simply fall behind. No events will be dropped; the calling code is responsible for canceling events which are no longer pertinent.

Changed in version 3.3: *blocking* parameter was added.

`scheduler.`**`queue`**

Read-only attribute returning a list of upcoming events in the order they will be run. Each event is shown as a *named tuple* with the following fields: time, priority, action, argument, kwargs.

17.8 `queue` — A synchronized queue class

Source code: Lib/queue.py

The `queue` module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The `Queue` class in this module implements all the required locking semantics.

The module implements three types of queue, which differ only in the order in which the entries are retrieved. In a FIFO queue, the first tasks added are the first retrieved. In a LIFO queue, the most recently added entry is the first retrieved (operating like a stack). With a priority queue, the entries are kept sorted (using the `heapq` module) and the lowest valued entry is retrieved first.

Internally, those three types of queues use locks to temporarily block competing threads; however, they are not designed to handle reentrancy within a thread.

In addition, the module implements a "simple" FIFO queue type, `SimpleQueue`, whose specific implementation provides additional guarantees in exchange for the smaller functionality.

The `queue` module defines the following classes and exceptions:

`class` `queue.`**`Queue`**(*maxsize=0*)

Constructor for a FIFO queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

`class` `queue.`**`LifoQueue`**(*maxsize=0*)

Constructor for a LIFO queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

`class` `queue.`**`PriorityQueue`**(*maxsize=0*)

Constructor for a priority queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

The lowest valued entries are retrieved first (the lowest valued entry is the one that would be returned by `min(entries)`). A typical pattern for entries is a tuple in the form: `(priority_number, data)`.

If the *data* elements are not comparable, the data can be wrapped in a class that ignores the data item and only compares the priority number:

```
from dataclasses import dataclass, field
from typing import Any
```

(continues on next page)

(continued from previous page)

```
@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

class queue.**SimpleQueue**

Constructor for an unbounded FIFO queue. Simple queues lack advanced functionality such as task tracking.

Added in version 3.7.

exception queue.**Empty**

Exception raised when non-blocking *get()* (or *get_nowait()*) is called on a *Queue* object which is empty.

exception queue.**Full**

Exception raised when non-blocking *put()* (or *put_nowait()*) is called on a *Queue* object which is full.

exception queue.**ShutDown**

Exception raised when *put()* or *get()* is called on a *Queue* object which has been shut down.

Added in version 3.13.

17.8.1 Queue Objects

Queue objects (*Queue*, *LifoQueue*, or *PriorityQueue*) provide the public methods described below.

Queue.**qsize**()

Return the approximate size of the queue. Note, qsize() > 0 doesn't guarantee that a subsequent get() will not block, nor will qsize() < maxsize guarantee that put() will not block.

Queue.**empty**()

Return `True` if the queue is empty, `False` otherwise. If empty() returns `True` it doesn't guarantee that a subsequent call to put() will not block. Similarly, if empty() returns `False` it doesn't guarantee that a subsequent call to get() will not block.

Queue.**full**()

Return `True` if the queue is full, `False` otherwise. If full() returns `True` it doesn't guarantee that a subsequent call to get() will not block. Similarly, if full() returns `False` it doesn't guarantee that a subsequent call to put() will not block.

Queue.**put**(*item*, *block=True*, *timeout=None*)

Put *item* into the queue. If optional args *block* is true and *timeout* is `None` (the default), block if necessary until a free slot is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *Full* exception if no free slot was available within that time. Otherwise (*block* is false), put an item on the queue if a free slot is immediately available, else raise the *Full* exception (*timeout* is ignored in that case).

Raises *ShutDown* if the queue has been shut down.

Queue.**put_nowait**(*item*)

Equivalent to `put(item, block=False)`.

Queue.**get**(*block=True*, *timeout=None*)

Remove and return an item from the queue. If optional args *block* is true and *timeout* is `None` (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *Empty* exception if no item was available within that time. Otherwise (*block* is false), return an item if one is immediately available, else raise the *Empty* exception (*timeout* is ignored in that case).

Prior to 3.0 on POSIX systems, and for all versions on Windows, if *block* is true and *timeout* is `None`, this operation goes into an uninterruptible wait on an underlying lock. This means that no exceptions can occur, and in particular a SIGINT will not trigger a *KeyboardInterrupt*.

Raises *ShutDown* if the queue has been shut down and is empty, or if the queue has been shut down immediately.

`Queue.`**`get_nowait`**`()`

Equivalent to `get(False)`.

Two methods are offered to support tracking whether enqueued tasks have been fully processed by daemon consumer threads.

`Queue.`**`task_done`**`()`

Indicate that a formerly enqueued task is complete. Used by queue consumer threads. For each *get()* used to fetch a task, a subsequent call to *task_done()* tells the queue that the processing on the task is complete.

If a *join()* is currently blocking, it will resume when all items have been processed (meaning that a *task_done()* call was received for every item that had been *put()* into the queue).

`shutdown(immediate=True)` calls *task_done()* for each remaining item in the queue.

Raises a *ValueError* if called more times than there were items placed in the queue.

`Queue.`**`join`**`()`

Blocks until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls *task_done()* to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, *join()* unblocks.

Example of how to wait for enqueued tasks to be completed:

```
import threading
import queue

q = queue.Queue()

def worker():
    while True:
        item = q.get()
        print(f'Working on {item}')
        print(f'Finished {item}')
        q.task_done()

# Turn-on the worker thread.
threading.Thread(target=worker, daemon=True).start()

# Send thirty task requests to the worker.
for item in range(30):
    q.put(item)

# Block until all tasks are done.
q.join()
print('All work completed')
```

Terminating queues

Queue objects can be made to prevent further interaction by shutting them down.

Queue.**shutdown**(*immediate=False*)

Shut down the queue, making *get()* and *put()* raise *ShutDown*.

By default, *get()* on a shut down queue will only raise once the queue is empty. Set *immediate* to true to make *get()* raise immediately instead.

All blocked callers of *put()* and *get()* will be unblocked. If *immediate* is true, a task will be marked as done for each remaining item in the queue, which may unblock callers of *join()*.

Added in version 3.13.

17.8.2 SimpleQueue Objects

SimpleQueue objects provide the public methods described below.

SimpleQueue.**qsize**()

Return the approximate size of the queue. Note, qsize() > 0 doesn't guarantee that a subsequent get() will not block.

SimpleQueue.**empty**()

Return `True` if the queue is empty, `False` otherwise. If empty() returns `False` it doesn't guarantee that a subsequent call to get() will not block.

SimpleQueue.**put**(*item*, *block=True*, *timeout=None*)

Put *item* into the queue. The method never blocks and always succeeds (except for potential low-level errors such as failure to allocate memory). The optional args *block* and *timeout* are ignored and only provided for compatibility with *Queue.put()*.

CPython implementation detail: This method has a C implementation which is reentrant. That is, a `put()` or `get()` call can be interrupted by another `put()` call in the same thread without deadlocking or corrupting internal state inside the queue. This makes it appropriate for use in destructors such as `__del__` methods or *weakref* callbacks.

SimpleQueue.**put_nowait**(*item*)

Equivalent to `put(item, block=False)`, provided for compatibility with *Queue.put_nowait()*.

SimpleQueue.**get**(*block=True*, *timeout=None*)

Remove and return an item from the queue. If optional args *block* is true and *timeout* is `None` (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *Empty* exception if no item was available within that time. Otherwise (*block* is false), return an item if one is immediately available, else raise the *Empty* exception (*timeout* is ignored in that case).

SimpleQueue.**get_nowait**()

Equivalent to `get(False)`.

See also

Class *multiprocessing.Queue*
A queue class for use in a multi-processing (rather than multi-threading) context.

collections.deque is an alternative implementation of unbounded queues with fast atomic *append()* and *popleft()* operations that do not require locking and also support indexing.

17.9 contextvars — Context Variables

This module provides APIs to manage, store, and access context-local state. The `ContextVar` class is used to declare and work with *Context Variables*. The `copy_context()` function and the `Context` class should be used to manage the current context in asynchronous frameworks.

Context managers that have state should use Context Variables instead of `threading.local()` to prevent their state from bleeding to other code unexpectedly, when used in concurrent code.

See also **PEP 567** for additional details.

Added in version 3.7.

17.9.1 Context Variables

class contextvars.**ContextVar**(*name*[, *, *default*])

This class is used to declare a new Context Variable, e.g.:

```
var: ContextVar[int] = ContextVar('var', default=42)
```

The required *name* parameter is used for introspection and debug purposes.

The optional keyword-only *default* parameter is returned by `ContextVar.get()` when no value for the variable is found in the current context.

Important: Context Variables should be created at the top module level and never in closures. `Context` objects hold strong references to context variables which prevents context variables from being properly garbage collected.

name

The name of the variable. This is a read-only property.

Added in version 3.7.1.

get([*default*])

Return a value for the context variable for the current context.

If there is no value for the variable in the current context, the method will:

- return the value of the *default* argument of the method, if provided; or
- return the default value for the context variable, if it was created with one; or
- raise a `LookupError`.

set(*value*)

Call to set a new value for the context variable in the current context.

The required *value* argument is the new value for the context variable.

Returns a `Token` object that can be used to restore the variable to its previous value via the `ContextVar.reset()` method.

reset(*token*)

Reset the context variable to the value it had before the `ContextVar.set()` that created the *token* was used.

For example:

```
var = ContextVar('var')

token = var.set('new value')
# code that uses 'var'; var.get() returns 'new value'.
var.reset(token)

# After the reset call the var has no value again, so
# var.get() would raise a LookupError.
```

class contextvars.**Token**

Token objects are returned by the *ContextVar.set()* method. They can be passed to the *ContextVar.reset()* method to revert the value of the variable to what it was before the corresponding *set*.

var

A read-only property. Points to the *ContextVar* object that created the token.

old_value

A read-only property. Set to the value the variable had before the *ContextVar.set()* method call that created the token. It points to *Token.MISSING* if the variable was not set before the call.

MISSING

A marker object used by *Token.old_value*.

17.9.2 Manual Context Management

contextvars.**copy_context**()

Returns a copy of the current *Context* object.

The following snippet gets a copy of the current context and prints all variables and their values that are set in it:

```
ctx: Context = copy_context()
print(list(ctx.items()))
```

The function has an *O*(1) complexity, i.e. works equally fast for contexts with a few context variables and for contexts that have a lot of them.

class contextvars.**Context**

A mapping of *ContextVars* to their values.

`Context()` creates an empty context with no values in it. To get a copy of the current context use the *copy_context()* function.

Every thread will have a different top-level *Context* object. This means that a *ContextVar* object behaves in a similar fashion to *threading.local()* when values are assigned in different threads.

Context implements the *collections.abc.Mapping* interface.

run(*callable*, **args*, ***kwargs*)

Execute `callable(*args, **kwargs)` code in the context object the *run* method is called on. Return the result of the execution or propagate an exception if one occurred.

Any changes to any context variables that *callable* makes will be contained in the context object:

```
var = ContextVar('var')
var.set('spam')
```

(continues on next page)

(continued from previous page)

```
def main():
    # 'var' was set to 'spam' before
    # calling 'copy_context()' and 'ctx.run(main)', so:
    # var.get() == ctx[var] == 'spam'

    var.set('ham')

    # Now, after setting 'var' to 'ham':
    # var.get() == ctx[var] == 'ham'

ctx = copy_context()

# Any changes that the 'main' function makes to 'var'
# will be contained in 'ctx'.
ctx.run(main)

# The 'main()' function was run in the 'ctx' context,
# so changes to 'var' are contained in it:
# ctx[var] == 'ham'

# However, outside of 'ctx', 'var' is still set to 'spam':
# var.get() == 'spam'
```

The method raises a *RuntimeError* when called on the same context object from more than one OS thread, or when called recursively.

copy()

Return a shallow copy of the context object.

var in context

Return `True` if the *context* has a value for *var* set; return `False` otherwise.

context[var]

Return the value of the *var* *ContextVar* variable. If the variable is not set in the context object, a *KeyError* is raised.

get(*var*[, *default*])

Return the value for *var* if *var* has the value in the context object. Return *default* otherwise. If *default* is not given, return `None`.

iter(context)

Return an iterator over the variables stored in the context object.

len(proxy)

Return the number of variables set in the context object.

keys()

Return a list of all variables in the context object.

values()

Return a list of all variables' values in the context object.

items()

Return a list of 2-tuples containing all variables and their values in the context object.

17.9.3 asyncio support

Context variables are natively supported in *asyncio* and are ready to be used without any extra configuration. For example, here is a simple echo server, that uses a context variable to make the address of a remote client available in the Task that handles that client:

```
import asyncio
import contextvars

client_addr_var = contextvars.ContextVar('client_addr')

def render_goodbye():
    # The address of the currently handled client can be accessed
    # without passing it explicitly to this function.

    client_addr = client_addr_var.get()
    return f'Good bye, client @ {client_addr}\n'.encode()

async def handle_request(reader, writer):
    addr = writer.transport.get_extra_info('socket').getpeername()
    client_addr_var.set(addr)

    # In any code that we call is now possible to get
    # client's address by calling 'client_addr_var.get()'.

    while True:
        line = await reader.readline()
        print(line)
        if not line.strip():
            break
        writer.write(line)

    writer.write(render_goodbye())
    writer.close()

async def main():
    srv = await asyncio.start_server(
        handle_request, '127.0.0.1', 8081)

    async with srv:
        await srv.serve_forever()

asyncio.run(main())

# To test it you can use telnet:
#     telnet 127.0.0.1 8081
```

The following are support modules for some of the above services:

17.10 _thread — Low-level threading API

This module provides low-level primitives for working with multiple threads (also called *light-weight processes* or *tasks*) — multiple threads of control sharing their global data space. For synchronization, simple locks (also called *mutexes* or *binary semaphores*) are provided. The `threading` module provides an easier to use and higher-level threading API built on top of this module.

Changed in version 3.7: This module used to be optional, it is now always available.

This module defines the following constants and functions:

exception _thread.**error**

Raised on thread-specific errors.

Changed in version 3.3: This is now a synonym of the built-in `RuntimeError`.

_thread.**LockType**

This is the type of lock objects.

_thread.**start_new_thread**(*function*, *args*[, *kwargs*])

Start a new thread and return its identifier. The thread executes the function *function* with the argument list *args* (which must be a tuple). The optional *kwargs* argument specifies a dictionary of keyword arguments.

When the function returns, the thread silently exits.

When the function terminates with an unhandled exception, `sys.unraisablehook()` is called to handle the exception. The *object* attribute of the hook argument is *function*. By default, a stack trace is printed and then the thread exits (but other threads continue to run).

When the function raises a `SystemExit` exception, it is silently ignored.

Raises an *auditing event* `_thread.start_new_thread` with arguments `function`, `args`, `kwargs`.

Changed in version 3.8: `sys.unraisablehook()` is now used to handle unhandled exceptions.

_thread.**interrupt_main**(*signum=signal.SIGINT*, */*)

Simulate the effect of a signal arriving in the main thread. A thread can use this function to interrupt the main thread, though there is no guarantee that the interruption will happen immediately.

If given, *signum* is the number of the signal to simulate. If *signum* is not given, `signal.SIGINT` is simulated.

If the given signal isn't handled by Python (it was set to `signal.SIG_DFL` or `signal.SIG_IGN`), this function does nothing.

Changed in version 3.10: The *signum* argument is added to customize the signal number.

> **Note**
>
> This does not emit the corresponding signal but schedules a call to the associated handler (if it exists). If you want to truly emit the signal, use `signal.raise_signal()`.

_thread.**exit**()

Raise the `SystemExit` exception. When not caught, this will cause the thread to exit silently.

_thread.**allocate_lock**()

Return a new lock object. Methods of locks are described below. The lock is initially unlocked.

_thread.**get_ident**()

Return the 'thread identifier' of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

_thread.**get_native_id**()

Return the native integral Thread ID of the current thread assigned by the kernel. This is a non-negative integer. Its value may be used to uniquely identify this particular thread system-wide (until the thread terminates, after which the value may be recycled by the OS).

Availability: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD, GNU/kFreeBSD.

Added in version 3.8.

Changed in version 3.13: Added support for GNU/kFreeBSD.

_thread.**stack_size**([*size*])

Return the thread stack size used when creating new threads. The optional *size* argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32 KiB). If *size* is not specified, 0 is used. If changing the thread stack size is unsupported, a `RuntimeError` is raised. If the specified stack size is invalid, a `ValueError` is raised and the stack size is unmodified. 32 KiB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32 KiB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4 KiB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information).

Availability: Windows, pthreads.

Unix platforms with POSIX threads support.

_thread.**TIMEOUT_MAX**

The maximum value allowed for the *timeout* parameter of `Lock.acquire`. Specifying a timeout greater than this value will raise an `OverflowError`.

Added in version 3.2.

Lock objects have the following methods:

lock.**acquire**(*blocking=True*, *timeout=-1*)

Without any optional argument, this method acquires the lock unconditionally, if necessary waiting until it is released by another thread (only one thread at a time can acquire a lock — that's their reason for existence).

If the *blocking* argument is present, the action depends on its value: if it is false, the lock is only acquired if it can be acquired immediately without waiting, while if it is true, the lock is acquired unconditionally as above.

If the floating-point *timeout* argument is present and positive, it specifies the maximum wait time in seconds before returning. A negative *timeout* argument specifies an unbounded wait. You cannot specify a *timeout* if *blocking* is false.

The return value is `True` if the lock is acquired successfully, `False` if not.

Changed in version 3.2: The *timeout* parameter is new.

Changed in version 3.2: Lock acquires can now be interrupted by signals on POSIX.

lock.**release**()

Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

lock.**locked**()

Return the status of the lock: `True` if it has been acquired by some thread, `False` if not.

In addition to these methods, lock objects can also be used via the `with` statement, e.g.:

```
import _thread

a_lock = _thread.allocate_lock()

with a_lock:
    print("a_lock is locked while this executes")
```

Caveats:

- Threads interact strangely with interrupts: the *KeyboardInterrupt* exception will be received by an arbitrary thread. (When the *signal* module is available, interrupts always go to the main thread.)
- Calling *sys.exit()* or raising the *SystemExit* exception is equivalent to calling *_thread.exit()*.
- It is not possible to interrupt the *acquire()* method on a lock — the *KeyboardInterrupt* exception will happen after the lock has been acquired.
- When the main thread exits, it is system defined whether the other threads survive. On most systems, they are killed without executing `try` … `finally` clauses or executing object destructors.
- When the main thread exits, it does not do any of its usual cleanup (except that `try` … `finally` clauses are honored), and the standard I/O files are not flushed.

CHAPTER

EIGHTEEN

NETWORKING AND INTERPROCESS COMMUNICATION

The modules described in this chapter provide mechanisms for networking and inter-processes communication.

Some modules only work for two processes that are on the same machine, e.g. `signal` and `mmap`. Other modules support networking protocols that two or more processes can use to communicate across machines.

The list of modules described in this chapter is:

18.1 `asyncio` — Asynchronous I/O

Hello World!

```
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

asyncio.run(main())
```

asyncio is a library to write **concurrent** code using the **async/await** syntax.

asyncio is used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and web-servers, database connection libraries, distributed task queues, etc.

asyncio is often a perfect fit for IO-bound and high-level **structured** network code.

asyncio provides a set of **high-level** APIs to:

- *run Python coroutines* concurrently and have full control over their execution;
- perform *network IO and IPC*;
- control *subprocesses*;
- distribute tasks via *queues*;
- *synchronize* concurrent code;

Additionally, there are **low-level** APIs for *library and framework developers* to:

- create and manage *event loops*, which provide asynchronous APIs for *networking*, running *subprocesses*, handling *OS signals*, etc;
- implement efficient protocols using *transports*;
- *bridge* callback-based libraries and code with async/await syntax.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

asyncio REPL

You can experiment with an `asyncio` concurrent context in the *REPL*:

```
$ python -m asyncio
asyncio REPL ...
Use "await" directly instead of "asyncio.run()".
Type "help", "copyright", "credits" or "license" for more information.
>>> import asyncio
>>> await asyncio.sleep(10, result='hello')
'hello'
```

Raises an *auditing event* `cpython.run_stdin` with no arguments.

Changed in version 3.12.5: (also 3.11.10, 3.10.15, 3.9.20, and 3.8.20) Emits audit events.

Changed in version 3.13: Uses PyREPL if possible, in which case `PYTHONSTARTUP` is also executed. Emits audit events.

Reference

18.1.1 Runners

Source code: Lib/asyncio/runners.py

This section outlines high-level asyncio primitives to run asyncio code.

They are built on top of an *event loop* with the aim to simplify async code usage for common wide-spread scenarios.

- *Running an asyncio Program*
- *Runner context manager*
- *Handling Keyboard Interruption*

Running an asyncio Program

asyncio.**run**(*coro*, *, *debug=None*, *loop_factory=None*)

Execute the *coroutine* *coro* and return the result.

This function runs the passed coroutine, taking care of managing the asyncio event loop, *finalizing asynchronous generators*, and closing the executor.

This function cannot be called when another asyncio event loop is running in the same thread.

If *debug* is `True`, the event loop will be run in debug mode. `False` disables debug mode explicitly. `None` is used to respect the global *Debug Mode* settings.

If *loop_factory* is not `None`, it is used to create a new event loop; otherwise *asyncio.new_event_loop()* is used. The loop is closed at the end. This function should be used as a main entry point for asyncio programs, and should ideally only be called once. It is recommended to use *loop_factory* to configure the event loop instead of policies. Passing *asyncio.EventLoop* allows running asyncio without the policy system.

The executor is given a timeout duration of 5 minutes to shutdown. If the executor hasn't finished within that duration, a warning is emitted and the executor is closed.

Example:

```
async def main():
    await asyncio.sleep(1)
    print('hello')

asyncio.run(main())
```

Added in version 3.7.

Changed in version 3.9: Updated to use *loop.shutdown_default_executor()*.

Changed in version 3.10: *debug* is `None` by default to respect the global debug mode settings.

Changed in version 3.12: Added *loop_factory* parameter.

Runner context manager

class asyncio.**Runner**(*, *debug=None*, *loop_factory=None*)

A context manager that simplifies *multiple* async function calls in the same context.

Sometimes several top-level async functions should be called in the same *event loop* and *contextvars.Context*.

If *debug* is `True`, the event loop will be run in debug mode. `False` disables debug mode explicitly. `None` is used to respect the global *Debug Mode* settings.

loop_factory could be used for overriding the loop creation. It is the responsibility of the *loop_factory* to set the created loop as the current one. By default *asyncio.new_event_loop()* is used and set as current event loop with *asyncio.set_event_loop()* if *loop_factory* is `None`.

Basically, *asyncio.run()* example can be rewritten with the runner usage:

```
async def main():
    await asyncio.sleep(1)
    print('hello')

with asyncio.Runner() as runner:
    runner.run(main())
```

Added in version 3.11.

run(*coro*, *, *context=None*)

Run a *coroutine* *coro* in the embedded loop.

Return the coroutine's result or raise its exception.

An optional keyword-only *context* argument allows specifying a custom *contextvars.Context* for the *coro* to run in. The runner's default context is used if `None`.

This function cannot be called when another asyncio event loop is running in the same thread.

close()

Close the runner.

Finalize asynchronous generators, shutdown default executor, close the event loop and release embedded *contextvars.Context*.

get_loop()

Return the event loop associated with the runner instance.

> **Note**
>
> *Runner* uses the lazy initialization strategy, its constructor doesn't initialize underlying low-level structures.
>
> Embedded *loop* and *context* are created at the `with` body entering or the first call of *run()* or *get_loop()*.

Handling Keyboard Interruption

Added in version 3.11.

When *signal.SIGINT* is raised by `Ctrl-C`, *KeyboardInterrupt* exception is raised in the main thread by default. However this doesn't work with *asyncio* because it can interrupt asyncio internals and can hang the program from exiting.

To mitigate this issue, *asyncio* handles *signal.SIGINT* as follows:

1. *asyncio.Runner.run()* installs a custom *signal.SIGINT* handler before any user code is executed and removes it when exiting from the function.
2. The *Runner* creates the main task for the passed coroutine for its execution.
3. When *signal.SIGINT* is raised by `Ctrl-C`, the custom signal handler cancels the main task by calling *asyncio.Task.cancel()* which raises *asyncio.CancelledError* inside the main task. This causes the Python stack to unwind, `try/except` and `try/finally` blocks can be used for resource cleanup. After the main task is cancelled, *asyncio.Runner.run()* raises *KeyboardInterrupt*.
4. A user could write a tight loop which cannot be interrupted by *asyncio.Task.cancel()*, in which case the second following `Ctrl-C` immediately raises the *KeyboardInterrupt* without cancelling the main task.

18.1.2 Coroutines and Tasks

This section outlines high-level asyncio APIs to work with coroutines and Tasks.

- *Coroutines*
- *Awaitables*
- *Creating Tasks*
- *Task Cancellation*
- *Task Groups*
- *Sleeping*
- *Running Tasks Concurrently*
- *Eager Task Factory*
- *Shielding From Cancellation*
- *Timeouts*
- *Waiting Primitives*
- *Running in Threads*
- *Scheduling From Other Threads*
- *Introspection*
- *Task Object*

Coroutines

Source code: Lib/asyncio/coroutines.py

Coroutines declared with the async/await syntax is the preferred way of writing asyncio applications. For example, the following snippet of code prints "hello", waits 1 second, and then prints "world":

```
>>> import asyncio

>>> async def main():
...     print('hello')
...     await asyncio.sleep(1)
...     print('world')

>>> asyncio.run(main())
hello
world
```

Note that simply calling a coroutine will not schedule it to be executed:

```
>>> main()
<coroutine object main at 0x1053bb7c8>
```

To actually run a coroutine, asyncio provides the following mechanisms:

- The *asyncio.run()* function to run the top-level entry point "main()" function (see the above example.)
- Awaiting on a coroutine. The following snippet of code will print "hello" after waiting for 1 second, and then print "world" after waiting for *another* 2 seconds:

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")

    await say_after(1, 'hello')
    await say_after(2, 'world')

    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

Expected output:

```
started at 17:13:52
hello
world
finished at 17:13:55
```

- The *asyncio.create_task()* function to run coroutines concurrently as asyncio *Tasks*.

  Let's modify the above example and run two `say_after` coroutines *concurrently*:

```
async def main():
    task1 = asyncio.create_task(
        say_after(1, 'hello'))

    task2 = asyncio.create_task(
        say_after(2, 'world'))

    print(f"started at {time.strftime('%X')}")

    # Wait until both tasks are completed (should take
    # around 2 seconds.)
    await task1
    await task2

    print(f"finished at {time.strftime('%X')}")
```

  Note that expected output now shows that the snippet runs 1 second faster than before:

```
started at 17:14:32
hello
world
finished at 17:14:34
```

- The *asyncio.TaskGroup* class provides a more modern alternative to *create_task()*. Using this API, the last example becomes:

```
async def main():
    async with asyncio.TaskGroup() as tg:
        task1 = tg.create_task(
            say_after(1, 'hello'))

        task2 = tg.create_task(
            say_after(2, 'world'))

        print(f"started at {time.strftime('%X')}")

    # The await is implicit when the context manager exits.

    print(f"finished at {time.strftime('%X')}")
```

The timing and output should be the same as for the previous version.

Added in version 3.11: *asyncio.TaskGroup*.

Awaitables

We say that an object is an **awaitable** object if it can be used in an `await` expression. Many asyncio APIs are designed to accept awaitables.

There are three main types of *awaitable* objects: **coroutines**, **Tasks**, and **Futures**.

Coroutines

Python coroutines are *awaitables* and therefore can be awaited from other coroutines:

```
import asyncio

async def nested():
    return 42

async def main():
    # Nothing happens if we just call "nested()".
    # A coroutine object is created but not awaited,
    # so it *won't run at all*.
    nested()

    # Let's do it differently now and await it:
    print(await nested())  # will print "42".

asyncio.run(main())
```

Important

In this documentation the term "coroutine" can be used for two closely related concepts:

- a *coroutine function*: an `async def` function;
- a *coroutine object*: an object returned by calling a *coroutine function*.

Tasks

Tasks are used to schedule coroutines *concurrently*.

When a coroutine is wrapped into a *Task* with functions like `asyncio.create_task()` the coroutine is automatically scheduled to run soon:

```
import asyncio

async def nested():
    return 42

async def main():
    # Schedule nested() to run soon concurrently
    # with "main()".
    task = asyncio.create_task(nested())

    # "task" can now be used to cancel "nested()", or
    # can simply be awaited to wait until it is complete:
    await task

asyncio.run(main())
```

Futures

A `Future` is a special **low-level** awaitable object that represents an **eventual result** of an asynchronous operation.

When a Future object is *awaited* it means that the coroutine will wait until the Future is resolved in some other place.

Future objects in asyncio are needed to allow callback-based code to be used with async/await.

Normally **there is no need** to create Future objects at the application level code.

Future objects, sometimes exposed by libraries and some asyncio APIs, can be awaited:

```
async def main():
    await function_that_returns_a_future_object()

    # this is also valid:
    await asyncio.gather(
        function_that_returns_a_future_object(),
        some_python_coroutine()
    )
```

A good example of a low-level function that returns a Future object is `loop.run_in_executor()`.

Creating Tasks

Source code: Lib/asyncio/tasks.py

`asyncio.`**`create_task`**(*coro*, ***, *name=None*, *context=None*)

Wrap the *coro* *coroutine* into a `Task` and schedule its execution. Return the Task object.

If *name* is not `None`, it is set as the name of the task using `Task.set_name()`.

An optional keyword-only *context* argument allows specifying a custom *contextvars.Context* for the *coro* to run in. The current context copy is created when no *context* is provided.

The task is executed in the loop returned by *get_running_loop()*, *RuntimeError* is raised if there is no running loop in current thread.

Note

asyncio.TaskGroup.create_task() is a new alternative leveraging structural concurrency; it allows for waiting for a group of related tasks with strong safety guarantees.

Important

Save a reference to the result of this function, to avoid a task disappearing mid-execution. The event loop only keeps weak references to tasks. A task that isn't referenced elsewhere may get garbage collected at any time, even before it's done. For reliable "fire-and-forget" background tasks, gather them in a collection:

```
background_tasks = set()

for i in range(10):
    task = asyncio.create_task(some_coro(param=i))

    # Add task to the set. This creates a strong reference.
    background_tasks.add(task)

    # To prevent keeping references to finished tasks forever,
    # make each task remove its own reference from the set after
    # completion:
    task.add_done_callback(background_tasks.discard)
```

Added in version 3.7.

Changed in version 3.8: Added the *name* parameter.

Changed in version 3.11: Added the *context* parameter.

Task Cancellation

Tasks can easily and safely be cancelled. When a task is cancelled, *asyncio.CancelledError* will be raised in the task at the next opportunity.

It is recommended that coroutines use `try/finally` blocks to robustly perform clean-up logic. In case *asyncio.CancelledError* is explicitly caught, it should generally be propagated when clean-up is complete. *asyncio.CancelledError* directly subclasses *BaseException* so most code will not need to be aware of it.

The asyncio components that enable structured concurrency, like *asyncio.TaskGroup* and *asyncio.timeout()*, are implemented using cancellation internally and might misbehave if a coroutine swallows *asyncio.CancelledError*. Similarly, user code should not generally call *uncancel*. However, in cases when suppressing *asyncio.CancelledError* is truly desired, it is necessary to also call `uncancel()` to completely remove the cancellation state.

Task Groups

Task groups combine a task creation API with a convenient and reliable way to wait for all tasks in the group to finish.

class asyncio.**TaskGroup**

An asynchronous context manager holding a group of tasks. Tasks can be added to the group using *create_task()*. All tasks are awaited when the context manager exits.

Added in version 3.11.

create_task(*coro*, ***, *name=None*, *context=None*)

Create a task in this task group. The signature matches that of *asyncio.create_task()*. If the task group is inactive (e.g. not yet entered, already finished, or in the process of shutting down), we will close the given `coro`.

Changed in version 3.13: Close the given coroutine if the task group is not active.

Example:

```
async def main():
    async with asyncio.TaskGroup() as tg:
        task1 = tg.create_task(some_coro(...))
        task2 = tg.create_task(another_coro(...))
    print(f"Both tasks have completed now: {task1.result()}, {task2.result()}")
```

The `async with` statement will wait for all tasks in the group to finish. While waiting, new tasks may still be added to the group (for example, by passing `tg` into one of the coroutines and calling `tg.create_task()` in that coroutine). Once the last task has finished and the `async with` block is exited, no new tasks may be added to the group.

The first time any of the tasks belonging to the group fails with an exception other than *asyncio.CancelledError*, the remaining tasks in the group are cancelled. No further tasks can then be added to the group. At this point, if the body of the `async with` statement is still active (i.e., `__aexit__()` hasn't been called yet), the task directly containing the `async with` statement is also cancelled. The resulting *asyncio.CancelledError* will interrupt an `await`, but it will not bubble out of the containing `async with` statement.

Once all tasks have finished, if any tasks have failed with an exception other than *asyncio.CancelledError*, those exceptions are combined in an *ExceptionGroup* or *BaseExceptionGroup* (as appropriate; see their documentation) which is then raised.

Two base exceptions are treated specially: If any task fails with *KeyboardInterrupt* or *SystemExit*, the task group still cancels the remaining tasks and waits for them, but then the initial *KeyboardInterrupt* or *SystemExit* is re-raised instead of *ExceptionGroup* or *BaseExceptionGroup*.

If the body of the `async with` statement exits with an exception (so `__aexit__()` is called with an exception set), this is treated the same as if one of the tasks failed: the remaining tasks are cancelled and then waited for, and non-cancellation exceptions are grouped into an exception group and raised. The exception passed into `__aexit__()`, unless it is *asyncio.CancelledError*, is also included in the exception group. The same special case is made for *KeyboardInterrupt* and *SystemExit* as in the previous paragraph.

Task groups are careful not to mix up the internal cancellation used to "wake up" their `__aexit__()` with cancellation requests for the task in which they are running made by other parties. In particular, when one task group is syntactically nested in another, and both experience an exception in one of their child tasks simultaneously, the inner task group will process its exceptions, and then the outer task group will receive another cancellation and process its own exceptions.

In the case where a task group is cancelled externally and also must raise an *ExceptionGroup*, it will call the parent task's *cancel()* method. This ensures that a *asyncio.CancelledError* will be raised at the next `await`, so the cancellation is not lost.

Task groups preserve the cancellation count reported by *asyncio.Task.cancelling()*.

Changed in version 3.13: Improved handling of simultaneous internal and external cancellations and correct preservation of cancellation counts.

Sleeping

coroutine asyncio.**sleep**(*delay*, *result=None*)

Block for *delay* seconds.

If *result* is provided, it is returned to the caller when the coroutine completes.

`sleep()` always suspends the current task, allowing other tasks to run.

Setting the delay to 0 provides an optimized path to allow other tasks to run. This can be used by long-running functions to avoid blocking the event loop for the full duration of the function call.

Example of coroutine displaying the current date every second for 5 seconds:

```
import asyncio
import datetime

async def display_date():
    loop = asyncio.get_running_loop()
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

asyncio.run(display_date())
```

Changed in version 3.10: Removed the *loop* parameter.

Changed in version 3.13: Raises *ValueError* if *delay* is *nan*.

Running Tasks Concurrently

awaitable asyncio.**gather**(**aws*, *return_exceptions=False*)

Run *awaitable objects* in the *aws* sequence *concurrently*.

If any awaitable in *aws* is a coroutine, it is automatically scheduled as a Task.

If all awaitables are completed successfully, the result is an aggregate list of returned values. The order of result values corresponds to the order of awaitables in *aws*.

If *return_exceptions* is `False` (default), the first raised exception is immediately propagated to the task that awaits on `gather()`. Other awaitables in the *aws* sequence **won't be cancelled** and will continue to run.

If *return_exceptions* is `True`, exceptions are treated the same as successful results, and aggregated in the result list.

If `gather()` is *cancelled*, all submitted awaitables (that have not completed yet) are also *cancelled*.

If any Task or Future from the *aws* sequence is *cancelled*, it is treated as if it raised *CancelledError* – the `gather()` call is **not** cancelled in this case. This is to prevent the cancellation of one submitted Task/Future to cause other Tasks/Futures to be cancelled.

Note

A new alternative to create and run tasks concurrently and wait for their completion is `asyncio.TaskGroup`. *TaskGroup* provides stronger safety guarantees than *gather* for scheduling a nesting of subtasks: if a task (or a subtask, a task scheduled by a task) raises an exception, *TaskGroup* will, while *gather* will not, cancel the remaining scheduled tasks).

Example:

```
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({number}), currently i={i}...")
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}")
    return f

async def main():
    # Schedule three calls *concurrently*:
    L = await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )
    print(L)

asyncio.run(main())

# Expected output:
#
#     Task A: Compute factorial(2), currently i=2...
#     Task B: Compute factorial(3), currently i=2...
#     Task C: Compute factorial(4), currently i=2...
#     Task A: factorial(2) = 2
#     Task B: Compute factorial(3), currently i=3...
#     Task C: Compute factorial(4), currently i=3...
#     Task B: factorial(3) = 6
#     Task C: Compute factorial(4), currently i=4...
#     Task C: factorial(4) = 24
#     [2, 6, 24]
```

Note

If *return_exceptions* is false, cancelling gather() after it has been marked done won't cancel any submitted awaitables. For instance, gather can be marked done after propagating an exception to the caller, therefore, calling `gather.cancel()` after catching an exception (raised by one of the awaitables) from gather won't cancel any other awaitables.

Changed in version 3.7: If the *gather* itself is cancelled, the cancellation is propagated regardless of *return_exceptions*.

Changed in version 3.10: Removed the *loop* parameter.

Deprecated since version 3.10: Deprecation warning is emitted if no positional arguments are provided or not all positional arguments are Future-like objects and there is no running event loop.

Eager Task Factory

asyncio.**eager_task_factory**(*loop*, *coro*, *, *name=None*, *context=None*)

A task factory for eager task execution.

When using this factory (via `loop.set_task_factory(asyncio.eager_task_factory)`), coroutines begin execution synchronously during `Task` construction. Tasks are only scheduled on the event loop if they block. This can be a performance improvement as the overhead of loop scheduling is avoided for coroutines that complete synchronously.

A common example where this is beneficial is coroutines which employ caching or memoization to avoid actual I/O when possible.

Note

Immediate execution of the coroutine is a semantic change. If the coroutine returns or raises, the task is never scheduled to the event loop. If the coroutine execution blocks, the task is scheduled to the event loop. This change may introduce behavior changes to existing applications. For example, the application's task execution order is likely to change.

Added in version 3.12.

asyncio.**create_eager_task_factory**(*custom_task_constructor*)

Create an eager task factory, similar to `eager_task_factory()`, using the provided *custom_task_constructor* when creating a new task instead of the default `Task`.

custom_task_constructor must be a *callable* with the signature matching the signature of `Task.__init__`. The callable must return a `asyncio.Task`-compatible object.

This function returns a *callable* intended to be used as a task factory of an event loop via `loop.set_task_factory(factory)`).

Added in version 3.12.

Shielding From Cancellation

awaitable asyncio.**shield**(*aw*)

Protect an *awaitable object* from being `cancelled`.

If *aw* is a coroutine it is automatically scheduled as a Task.

The statement:

```
task = asyncio.create_task(something())
res = await shield(task)
```

is equivalent to:

```
res = await something()
```

except that if the coroutine containing it is cancelled, the Task running in `something()` is not cancelled. From the point of view of `something()`, the cancellation did not happen. Although its caller is still cancelled, so the "await" expression still raises a *CancelledError*.

If `something()` is cancelled by other means (i.e. from within itself) that would also cancel `shield()`.

If it is desired to completely ignore cancellation (not recommended) the `shield()` function should be combined with a try/except clause, as follows:

```
task = asyncio.create_task(something())
try:
    res = await shield(task)
except CancelledError:
    res = None
```

Important

Save a reference to tasks passed to this function, to avoid a task disappearing mid-execution. The event loop only keeps weak references to tasks. A task that isn't referenced elsewhere may get garbage collected at any time, even before it's done.

Changed in version 3.10: Removed the *loop* parameter.

Deprecated since version 3.10: Deprecation warning is emitted if *aw* is not Future-like object and there is no running event loop.

Timeouts

`asyncio.timeout`(*delay*)

Return an asynchronous context manager that can be used to limit the amount of time spent waiting on something.

delay can either be `None`, or a float/int number of seconds to wait. If *delay* is `None`, no time limit will be applied; this can be useful if the delay is unknown when the context manager is created.

In either case, the context manager can be rescheduled after creation using *Timeout.reschedule()*.

Example:

```
async def main():
    async with asyncio.timeout(10):
        await long_running_task()
```

If `long_running_task` takes more than 10 seconds to complete, the context manager will cancel the current task and handle the resulting *asyncio.CancelledError* internally, transforming it into a *TimeoutError* which can be caught and handled.

Note

The *asyncio.timeout()* context manager is what transforms the *asyncio.CancelledError* into a *TimeoutError*, which means the *TimeoutError* can only be caught *outside* of the context manager.

Example of catching *TimeoutError*:

```
async def main():
    try:
        async with asyncio.timeout(10):
            await long_running_task()
    except TimeoutError:
        print("The long operation timed out, but we've handled it.")

    print("This statement will run regardless.")
```

The context manager produced by *asyncio.timeout()* can be rescheduled to a different deadline and inspected.

class asyncio.**Timeout**(*when*)

An asynchronous context manager for cancelling overdue coroutines.

`when` should be an absolute time at which the context should time out, as measured by the event loop's clock:

- If `when` is `None`, the timeout will never trigger.
- If `when < loop.time()`, the timeout will trigger on the next iteration of the event loop.

when() → *float* | *None*

Return the current deadline, or `None` if the current deadline is not set.

reschedule(*when:* float | None)

Reschedule the timeout.

expired() → *bool*

Return whether the context manager has exceeded its deadline (expired).

Example:

```
async def main():
    try:
        # We do not know the timeout when starting, so we pass ``None``.
        async with asyncio.timeout(None) as cm:
            # We know the timeout now, so we reschedule it.
            new_deadline = get_running_loop().time() + 10
            cm.reschedule(new_deadline)

            await long_running_task()
    except TimeoutError:
        pass

    if cm.expired():
        print("Looks like we haven't finished on time.")
```

Timeout context managers can be safely nested.

Added in version 3.11.

asyncio.**timeout_at**(*when*)

Similar to *asyncio.timeout()*, except *when* is the absolute time to stop waiting, or `None`.

Example:

```
async def main():
    loop = get_running_loop()
```

(continues on next page)

(continued from previous page)

```
    deadline = loop.time() + 20
    try:
        async with asyncio.timeout_at(deadline):
            await long_running_task()
    except TimeoutError:
        print("The long operation timed out, but we've handled it.")

    print("This statement will run regardless.")
```

Added in version 3.11.

coroutine asyncio.**wait_for**(*aw*, *timeout*)

Wait for the *aw* *awaitable* to complete with a timeout.

If *aw* is a coroutine it is automatically scheduled as a Task.

timeout can either be `None` or a float or int number of seconds to wait for. If *timeout* is `None`, block until the future completes.

If a timeout occurs, it cancels the task and raises *TimeoutError*.

To avoid the task *cancellation*, wrap it in *shield()*.

The function will wait until the future is actually cancelled, so the total wait time may exceed the *timeout*. If an exception happens during cancellation, it is propagated.

If the wait is cancelled, the future *aw* is also cancelled.

Example:

```
async def eternity():
    # Sleep for one hour
    await asyncio.sleep(3600)
    print('yay!')

async def main():
    # Wait for at most 1 second
    try:
        await asyncio.wait_for(eternity(), timeout=1.0)
    except TimeoutError:
        print('timeout!')

asyncio.run(main())

# Expected output:
#
#     timeout!
```

Changed in version 3.7: When *aw* is cancelled due to a timeout, `wait_for` waits for *aw* to be cancelled. Previously, it raised *TimeoutError* immediately.

Changed in version 3.10: Removed the *loop* parameter.

Changed in version 3.11: Raises *TimeoutError* instead of *asyncio.TimeoutError*.

Waiting Primitives

coroutine asyncio.**wait**(*aws*, *, *timeout=None*, *return_when=ALL_COMPLETED*)

Run *Future* and *Task* instances in the *aws* iterable concurrently and block until the condition specified by *return_when*.

The *aws* iterable must not be empty.

Returns two sets of Tasks/Futures: `(done, pending)`.

Usage:

```
done, pending = await asyncio.wait(aws)
```

timeout (a float or int), if specified, can be used to control the maximum number of seconds to wait before returning.

Note that this function does not raise *TimeoutError*. Futures or Tasks that aren't done when the timeout occurs are simply returned in the second set.

return_when indicates when this function should return. It must be one of the following constants:

Constant	Description
asyncio.**FIRST_COMPLETED**	The function will return when any future finishes or is cancelled.
asyncio.**FIRST_EXCEPTION**	The function will return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to *ALL_COMPLETED*.
asyncio.**ALL_COMPLETED**	The function will return when all futures finish or are cancelled.

Unlike *wait_for()*, `wait()` does not cancel the futures when a timeout occurs.

Changed in version 3.10: Removed the *loop* parameter.

Changed in version 3.11: Passing coroutine objects to `wait()` directly is forbidden.

Changed in version 3.12: Added support for generators yielding tasks.

asyncio.**as_completed**(*aws*, *, *timeout=None*)

Run *awaitable objects* in the *aws* iterable concurrently. The returned object can be iterated to obtain the results of the awaitables as they finish.

The object returned by `as_completed()` can be iterated as an *asynchronous iterator* or a plain *iterator*. When asynchronous iteration is used, the originally-supplied awaitables are yielded if they are tasks or futures. This makes it easy to correlate previously-scheduled tasks with their results. Example:

```
ipv4_connect = create_task(open_connection("127.0.0.1", 80))
ipv6_connect = create_task(open_connection("::1", 80))
tasks = [ipv4_connect, ipv6_connect]

async for earliest_connect in as_completed(tasks):
    # earliest_connect is done. The result can be obtained by
    # awaiting it or calling earliest_connect.result()
    reader, writer = await earliest_connect

    if earliest_connect is ipv6_connect:
```

(continues on next page)

(continued from previous page)

```
        print("IPv6 connection established.")
    else:
        print("IPv4 connection established.")
```

During asynchronous iteration, implicitly-created tasks will be yielded for supplied awaitables that aren't tasks or futures.

When used as a plain iterator, each iteration yields a new coroutine that returns the result or raises the exception of the next completed awaitable. This pattern is compatible with Python versions older than 3.13:

```
ipv4_connect = create_task(open_connection("127.0.0.1", 80))
ipv6_connect = create_task(open_connection("::1", 80))
tasks = [ipv4_connect, ipv6_connect]

for next_connect in as_completed(tasks):
    # next_connect is not one of the original task objects. It must be
    # awaited to obtain the result value or raise the exception of the
    # awaitable that finishes next.
    reader, writer = await next_connect
```

A *TimeoutError* is raised if the timeout occurs before all awaitables are done. This is raised by the `async for` loop during asynchronous iteration or by the coroutines yielded during plain iteration.

Changed in version 3.10: Removed the *loop* parameter.

Deprecated since version 3.10: Deprecation warning is emitted if not all awaitable objects in the *aws* iterable are Future-like objects and there is no running event loop.

Changed in version 3.12: Added support for generators yielding tasks.

Changed in version 3.13: The result can now be used as either an *asynchronous iterator* or as a plain *iterator* (previously it was only a plain iterator).

Running in Threads

coroutine asyncio.**to_thread**(*func*, */*, **args*, ***kwargs*)

Asynchronously run function *func* in a separate thread.

Any *args and **kwargs supplied for this function are directly passed to *func*. Also, the current *contextvars.Context* is propagated, allowing context variables from the event loop thread to be accessed in the separate thread.

Return a coroutine that can be awaited to get the eventual result of *func*.

This coroutine function is primarily intended to be used for executing IO-bound functions/methods that would otherwise block the event loop if they were run in the main thread. For example:

```
def blocking_io():
    print(f"start blocking_io at {time.strftime('%X')}")
    # Note that time.sleep() can be replaced with any blocking
    # IO-bound operation, such as file operations.
    time.sleep(1)
    print(f"blocking_io complete at {time.strftime('%X')}")

async def main():
    print(f"started main at {time.strftime('%X')}")
```

(continues on next page)

(continued from previous page)

```
    await asyncio.gather(
        asyncio.to_thread(blocking_io),
        asyncio.sleep(1))

    print(f"finished main at {time.strftime('%X')}")


asyncio.run(main())

# Expected output:
#
# started main at 19:50:53
# start blocking_io at 19:50:53
# blocking_io complete at 19:50:54
# finished main at 19:50:54
```

Directly calling `blocking_io()` in any coroutine would block the event loop for its duration, resulting in an additional 1 second of run time. Instead, by using `asyncio.to_thread()`, we can run it in a separate thread without blocking the event loop.

Note

Due to the *GIL*, `asyncio.to_thread()` can typically only be used to make IO-bound functions non-blocking. However, for extension modules that release the GIL or alternative Python implementations that don't have one, `asyncio.to_thread()` can also be used for CPU-bound functions.

Added in version 3.9.

Scheduling From Other Threads

`asyncio.`**`run_coroutine_threadsafe`**(*coro*, *loop*)

Submit a coroutine to the given event loop. Thread-safe.

Return a *concurrent.futures.Future* to wait for the result from another OS thread.

This function is meant to be called from a different OS thread than the one where the event loop is running. Example:

```
# Create a coroutine
coro = asyncio.sleep(1, result=3)

# Submit the coroutine to a given loop
future = asyncio.run_coroutine_threadsafe(coro, loop)

# Wait for the result with an optional timeout argument
assert future.result(timeout) == 3
```

If an exception is raised in the coroutine, the returned Future will be notified. It can also be used to cancel the task in the event loop:

```
try:
    result = future.result(timeout)
except TimeoutError:
    print('The coroutine took too long, cancelling the task...')
    future.cancel()
```

(continues on next page)

(continued from previous page)

```
except Exception as exc:
    print(f'The coroutine raised an exception: {exc!r}')
else:
    print(f'The coroutine returned: {result!r}')
```

See the *concurrency and multithreading* section of the documentation.

Unlike other asyncio functions this function requires the *loop* argument to be passed explicitly.

Added in version 3.5.1.

Introspection

`asyncio.`**`current_task`**(*loop=None*)

Return the currently running *Task* instance, or `None` if no task is running.

If *loop* is `None` *get_running_loop()* is used to get the current loop.

Added in version 3.7.

`asyncio.`**`all_tasks`**(*loop=None*)

Return a set of not yet finished *Task* objects run by the loop.

If *loop* is `None`, *get_running_loop()* is used for getting current loop.

Added in version 3.7.

`asyncio.`**`iscoroutine`**(*obj*)

Return `True` if *obj* is a coroutine object.

Added in version 3.4.

Task Object

class `asyncio.`**`Task`**(*coro*, ***, *loop=None*, *name=None*, *context=None*, *eager_start=False*)

A *Future-like* object that runs a Python *coroutine*. Not thread-safe.

Tasks are used to run coroutines in event loops. If a coroutine awaits on a Future, the Task suspends the execution of the coroutine and waits for the completion of the Future. When the Future is *done*, the execution of the wrapped coroutine resumes.

Event loops use cooperative scheduling: an event loop runs one Task at a time. While a Task awaits for the completion of a Future, the event loop runs other Tasks, callbacks, or performs IO operations.

Use the high-level *asyncio.create_task()* function to create Tasks, or the low-level *loop.create_task()* or *ensure_future()* functions. Manual instantiation of Tasks is discouraged.

To cancel a running Task use the *cancel()* method. Calling it will cause the Task to throw a *CancelledError* exception into the wrapped coroutine. If a coroutine is awaiting on a Future object during cancellation, the Future object will be cancelled.

cancelled() can be used to check if the Task was cancelled. The method returns `True` if the wrapped coroutine did not suppress the *CancelledError* exception and was actually cancelled.

asyncio.Task inherits from *Future* all of its APIs except *Future.set_result()* and *Future.set_exception()*.

An optional keyword-only *context* argument allows specifying a custom *contextvars.Context* for the *coro* to run in. If no *context* is provided, the Task copies the current context and later runs its coroutine in the copied context.

An optional keyword-only *eager_start* argument allows eagerly starting the execution of the *asyncio.Task* at task creation time. If set to `True` and the event loop is running, the task will start executing the coroutine immediately, until the first time the coroutine blocks. If the coroutine returns or raises without blocking, the task will be finished eagerly and will skip scheduling to the event loop.

Changed in version 3.7: Added support for the *contextvars* module.

Changed in version 3.8: Added the *name* parameter.

Deprecated since version 3.10: Deprecation warning is emitted if *loop* is not specified and there is no running event loop.

Changed in version 3.11: Added the *context* parameter.

Changed in version 3.12: Added the *eager_start* parameter.

done()

Return `True` if the Task is *done*.

A Task is *done* when the wrapped coroutine either returned a value, raised an exception, or the Task was cancelled.

result()

Return the result of the Task.

If the Task is *done*, the result of the wrapped coroutine is returned (or if the coroutine raised an exception, that exception is re-raised.)

If the Task has been *cancelled*, this method raises a *CancelledError* exception.

If the Task's result isn't yet available, this method raises an *InvalidStateError* exception.

exception()

Return the exception of the Task.

If the wrapped coroutine raised an exception that exception is returned. If the wrapped coroutine returned normally this method returns `None`.

If the Task has been *cancelled*, this method raises a *CancelledError* exception.

If the Task isn't *done* yet, this method raises an *InvalidStateError* exception.

add_done_callback(*callback*, *, *context=None*)

Add a callback to be run when the Task is *done*.

This method should only be used in low-level callback-based code.

See the documentation of *Future.add_done_callback()* for more details.

remove_done_callback(*callback*)

Remove *callback* from the callbacks list.

This method should only be used in low-level callback-based code.

See the documentation of *Future.remove_done_callback()* for more details.

get_stack(*, *limit=None*)

Return the list of stack frames for this Task.

If the wrapped coroutine is not done, this returns the stack where it is suspended. If the coroutine has completed successfully or was cancelled, this returns an empty list. If the coroutine was terminated by an exception, this returns the list of traceback frames.

The frames are always ordered from oldest to newest.

Only one stack frame is returned for a suspended coroutine.

The optional *limit* argument sets the maximum number of frames to return; by default all available frames are returned. The ordering of the returned list differs depending on whether a stack or a traceback is returned: the newest frames of a stack are returned, but the oldest frames of a traceback are returned. (This matches the behavior of the traceback module.)

print_stack(*, *limit=None*, *file=None*)

Print the stack or traceback for this Task.

This produces output similar to that of the traceback module for the frames retrieved by *get_stack()*.

The *limit* argument is passed to *get_stack()* directly.

The *file* argument is an I/O stream to which the output is written; by default output is written to *sys.stdout*.

get_coro()

Return the coroutine object wrapped by the *Task*.

> **Note**
>
> This will return `None` for Tasks which have already completed eagerly. See the *Eager Task Factory*.

Added in version 3.8.

Changed in version 3.12: Newly added eager task execution means result may be `None`.

get_context()

Return the *contextvars.Context* object associated with the task.

Added in version 3.12.

get_name()

Return the name of the Task.

If no name has been explicitly assigned to the Task, the default asyncio Task implementation generates a default name during instantiation.

Added in version 3.8.

set_name(*value*)

Set the name of the Task.

The *value* argument can be any object, which is then converted to a string.

In the default Task implementation, the name will be visible in the *repr()* output of a task object.

Added in version 3.8.

cancel(*msg=None*)

Request the Task to be cancelled.

This arranges for a *CancelledError* exception to be thrown into the wrapped coroutine on the next cycle of the event loop.

The coroutine then has a chance to clean up or even deny the request by suppressing the exception with a `try` … … `except CancelledError` … `finally` block. Therefore, unlike *Future.cancel()*, *Task.cancel()* does not guarantee that the Task will be cancelled, although suppressing cancellation completely is not common and is actively discouraged. Should the coroutine nevertheless decide to suppress the cancellation, it needs to call *Task.uncancel()* in addition to catching the exception.

Changed in version 3.9: Added the *msg* parameter.

Changed in version 3.11: The `msg` parameter is propagated from cancelled task to its awaiter. The following example illustrates how coroutines can intercept the cancellation request:

```
async def cancel_me():
    print('cancel_me(): before sleep')

    try:
        # Wait for 1 hour
        await asyncio.sleep(3600)
    except asyncio.CancelledError:
        print('cancel_me(): cancel sleep')
        raise
    finally:
        print('cancel_me(): after sleep')

async def main():
    # Create a "cancel_me" Task
    task = asyncio.create_task(cancel_me())

    # Wait for 1 second
    await asyncio.sleep(1)

    task.cancel()
    try:
        await task
    except asyncio.CancelledError:
        print("main(): cancel_me is cancelled now")

asyncio.run(main())

# Expected output:
#
#     cancel_me(): before sleep
#     cancel_me(): cancel sleep
#     cancel_me(): after sleep
#     main(): cancel_me is cancelled now
```

cancelled()

Return `True` if the Task is *cancelled*.

The Task is *cancelled* when the cancellation was requested with *cancel()* and the wrapped coroutine propagated the *CancelledError* exception thrown into it.

uncancel()

Decrement the count of cancellation requests to this Task.

Returns the remaining number of cancellation requests.

Note that once execution of a cancelled task completed, further calls to *uncancel()* are ineffective.

Added in version 3.11.

This method is used by asyncio's internals and isn't expected to be used by end-user code. In particular, if a Task gets successfully uncancelled, this allows for elements of structured concurrency like *Task Groups* and *asyncio.timeout()* to continue running, isolating cancellation to the respective structured block. For example:

```
async def make_request_with_timeout():
    try:
        async with asyncio.timeout(1):
            # Structured block affected by the timeout:
            await make_request()
            await make_another_request()
    except TimeoutError:
        log("There was a timeout")
    # Outer code not affected by the timeout:
    await unrelated_code()
```

While the block with `make_request()` and `make_another_request()` might get cancelled due to the timeout, `unrelated_code()` should continue running even in case of the timeout. This is implemented with *uncancel()*. *TaskGroup* context managers use *uncancel()* in a similar fashion.

If end-user code is, for some reason, suppressing cancellation by catching *CancelledError*, it needs to call this method to remove the cancellation state.

When this method decrements the cancellation count to zero, the method checks if a previous *cancel()* call had arranged for *CancelledError* to be thrown into the task. If it hasn't been thrown yet, that arrangement will be rescinded (by resetting the internal `_must_cancel` flag).

Changed in version 3.13: Changed to rescind pending cancellation requests upon reaching zero.

cancelling()

Return the number of pending cancellation requests to this Task, i.e., the number of calls to *cancel()* less the number of *uncancel()* calls.

Note that if this number is greater than zero but the Task is still executing, *cancelled()* will still return `False`. This is because this number can be lowered by calling *uncancel()*, which can lead to the task not being cancelled after all if the cancellation requests go down to zero.

This method is used by asyncio's internals and isn't expected to be used by end-user code. See *uncancel()* for more details.

Added in version 3.11.

18.1.3 Streams

Source code: Lib/asyncio/streams.py

Streams are high-level async/await-ready primitives to work with network connections. Streams allow sending and receiving data without using callbacks or low-level protocols and transports.

Here is an example of a TCP echo client written using asyncio streams:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)
```

(continues on next page)

(continued from previous page)

```
    print(f'Send: {message!r}')
    writer.write(message.encode())
    await writer.drain()

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

See also the *Examples* section below.

Stream Functions

The following top-level asyncio functions can be used to create and work with streams:

coroutine asyncio.**open_connection**(*host=None*, *port=None*, ***, *limit=None*, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *happy_eyeballs_delay=None*, *interleave=None*)

Establish a network connection and return a pair of `(reader, writer)` objects.

The returned *reader* and *writer* objects are instances of *StreamReader* and *StreamWriter* classes.

limit determines the buffer size limit used by the returned *StreamReader* instance. By default the *limit* is set to 64 KiB.

The rest of the arguments are passed directly to *loop.create_connection()*.

> **Note**
>
> The *sock* argument transfers ownership of the socket to the *StreamWriter* created. To close the socket, call its *close()* method.

Changed in version 3.7: Added the *ssl_handshake_timeout* parameter.

Changed in version 3.8: Added the *happy_eyeballs_delay* and *interleave* parameters.

Changed in version 3.10: Removed the *loop* parameter.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

coroutine asyncio.**start_server**(*client_connected_cb*, *host=None*, *port=None*, ***, *limit=None*, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*)

Start a socket server.

The *client_connected_cb* callback is called whenever a new client connection is established. It receives a `(reader, writer)` pair as two arguments, instances of the *StreamReader* and *StreamWriter* classes.

client_connected_cb can be a plain callable or a *coroutine function*; if it is a coroutine function, it will be automatically scheduled as a `Task`.

limit determines the buffer size limit used by the returned `StreamReader` instance. By default the *limit* is set to 64 KiB.

The rest of the arguments are passed directly to `loop.create_server()`.

> **Note**
>
> The *sock* argument transfers ownership of the socket to the server created. To close the socket, call the server's `close()` method.

Changed in version 3.7: Added the *ssl_handshake_timeout* and *start_serving* parameters.

Changed in version 3.10: Removed the *loop* parameter.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

Unix Sockets

coroutine asyncio.**open_unix_connection**(*path=None*, ***, *limit=None*, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

Establish a Unix socket connection and return a pair of `(reader, writer)`.

Similar to `open_connection()` but operates on Unix sockets.

See also the documentation of `loop.create_unix_connection()`.

> **Note**
>
> The *sock* argument transfers ownership of the socket to the `StreamWriter` created. To close the socket, call its `close()` method.

Availability: Unix.

Changed in version 3.7: Added the *ssl_handshake_timeout* parameter. The *path* parameter can now be a *path-like object*

Changed in version 3.10: Removed the *loop* parameter.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

coroutine asyncio.**start_unix_server**(*client_connected_cb*, *path=None*, ***, *limit=None*, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*)

Start a Unix socket server.

Similar to `start_server()` but works with Unix sockets.

See also the documentation of `loop.create_unix_server()`.

> **Note**
>
> The *sock* argument transfers ownership of the socket to the server created. To close the socket, call the server's `close()` method.

Availability: Unix.

Changed in version 3.7: Added the *ssl_handshake_timeout* and *start_serving* parameters. The *path* parameter can now be a *path-like object*.

Changed in version 3.10: Removed the *loop* parameter.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

StreamReader

class asyncio.**StreamReader**

Represents a reader object that provides APIs to read data from the IO stream. As an *asynchronous iterable*, the object supports the `async for` statement.

It is not recommended to instantiate *StreamReader* objects directly; use `open_connection()` and `start_server()` instead.

feed_eof()

Acknowledge the EOF.

coroutine read(*n=-1*)

Read up to *n* bytes from the stream.

If *n* is not provided or set to `-1`, read until EOF, then return all read `bytes`. If EOF was received and the internal buffer is empty, return an empty `bytes` object.

If *n* is `0`, return an empty `bytes` object immediately.

If *n* is positive, return at most *n* available `bytes` as soon as at least 1 byte is available in the internal buffer. If EOF is received before any byte is read, return an empty `bytes` object.

coroutine readline()

Read one line, where "line" is a sequence of bytes ending with `\n`.

If EOF is received and `\n` was not found, the method returns partially read data.

If EOF is received and the internal buffer is empty, return an empty `bytes` object.

coroutine readexactly(*n*)

Read exactly *n* bytes.

Raise an `IncompleteReadError` if EOF is reached before *n* can be read. Use the `IncompleteReadError.partial` attribute to get the partially read data.

coroutine readuntil(*separator=b'\n'*)

Read data from the stream until *separator* is found.

On success, the data and separator will be removed from the internal buffer (consumed). Returned data will include the separator at the end.

If the amount of data read exceeds the configured stream limit, a `LimitOverrunError` exception is raised, and the data is left in the internal buffer and can be read again.

If EOF is reached before the complete separator is found, an *IncompleteReadError* exception is raised, and the internal buffer is reset. The *IncompleteReadError.partial* attribute may contain a portion of the separator.

The *separator* may also be a tuple of separators. In this case the return value will be the shortest possible that has any separator as the suffix. For the purposes of *LimitOverrunError*, the shortest possible separator is considered to be the one that matched.

Added in version 3.5.2.

Changed in version 3.13: The *separator* parameter may now be a *tuple* of separators.

at_eof()

Return `True` if the buffer is empty and *feed_eof()* was called.

StreamWriter

class asyncio.**StreamWriter**

Represents a writer object that provides APIs to write data to the IO stream.

It is not recommended to instantiate *StreamWriter* objects directly; use *open_connection()* and *start_server()* instead.

write(*data*)

The method attempts to write the *data* to the underlying socket immediately. If that fails, the data is queued in an internal write buffer until it can be sent.

The method should be used along with the `drain()` method:

```
stream.write(data)
await stream.drain()
```

writelines(*data*)

The method writes a list (or any iterable) of bytes to the underlying socket immediately. If that fails, the data is queued in an internal write buffer until it can be sent.

The method should be used along with the `drain()` method:

```
stream.writelines(lines)
await stream.drain()
```

close()

The method closes the stream and the underlying socket.

The method should be used, though not mandatory, along with the `wait_closed()` method:

```
stream.close()
await stream.wait_closed()
```

can_write_eof()

Return `True` if the underlying transport supports the *write_eof()* method, `False` otherwise.

write_eof()

Close the write end of the stream after the buffered write data is flushed.

transport

Return the underlying asyncio transport.

get_extra_info(*name*, *default=None*)

Access optional transport information; see *BaseTransport.get_extra_info()* for details.

coroutine drain()

Wait until it is appropriate to resume writing to the stream. Example:

```
writer.write(data)
await writer.drain()
```

This is a flow control method that interacts with the underlying IO write buffer. When the size of the buffer reaches the high watermark, *drain()* blocks until the size of the buffer is drained down to the low watermark and writing can be resumed. When there is nothing to wait for, the *drain()* returns immediately.

coroutine start_tls(*sslcontext*, *, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

Upgrade an existing stream-based connection to TLS.

Parameters:

- *sslcontext*: a configured instance of *SSLContext*.
- *server_hostname*: sets or overrides the host name that the target server's certificate will be matched against.
- *ssl_handshake_timeout* is the time in seconds to wait for the TLS handshake to complete before aborting the connection. `60.0` seconds if `None` (default).
- *ssl_shutdown_timeout* is the time in seconds to wait for the SSL shutdown to complete before aborting the connection. `30.0` seconds if `None` (default).

Added in version 3.11.

Changed in version 3.12: Added the *ssl_shutdown_timeout* parameter.

is_closing()

Return `True` if the stream is closed or in the process of being closed.

Added in version 3.7.

coroutine wait_closed()

Wait until the stream is closed.

Should be called after *close()* to wait until the underlying connection is closed, ensuring that all data has been flushed before e.g. exiting the program.

Added in version 3.7.

Examples

TCP echo client using streams

TCP echo client using the *asyncio.open_connection()* function:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)
```

(continues on next page)

(continued from previous page)

```
    print(f'Send: {message!r}')
    writer.write(message.encode())
    await writer.drain()

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

See also

The *TCP echo client protocol* example uses the low-level `loop.create_connection()` method.

TCP echo server using streams

TCP echo server using the `asyncio.start_server()` function:

```
import asyncio

async def handle_echo(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')

    print(f"Received {message!r} from {addr!r}")

    print(f"Send: {message!r}")
    writer.write(data)
    await writer.drain()

    print("Close the connection")
    writer.close()
    await writer.wait_closed()

async def main():
    server = await asyncio.start_server(
        handle_echo, '127.0.0.1', 8888)

    addrs = ', '.join(str(sock.getsockname()) for sock in server.sockets)
    print(f'Serving on {addrs}')

    async with server:
        await server.serve_forever()

asyncio.run(main())
```

See also

The *TCP echo server protocol* example uses the `loop.create_server()` method.

Get HTTP headers

Simple example querying HTTP headers of the URL passed on the command line:

```
import asyncio
import urllib.parse
import sys

async def print_http_headers(url):
    url = urllib.parse.urlsplit(url)
    if url.scheme == 'https':
        reader, writer = await asyncio.open_connection(
            url.hostname, 443, ssl=True)
    else:
        reader, writer = await asyncio.open_connection(
            url.hostname, 80)

    query = (
        f"HEAD {url.path or '/'} HTTP/1.0\r\n"
        f"Host: {url.hostname}\r\n"
        f"\r\n"
    )

    writer.write(query.encode('latin-1'))
    while True:
        line = await reader.readline()
        if not line:
            break

        line = line.decode('latin1').rstrip()
        if line:
            print(f'HTTP header> {line}')

    # Ignore the body, close the socket
    writer.close()
    await writer.wait_closed()

url = sys.argv[1]
asyncio.run(print_http_headers(url))
```

Usage:

```
python example.py http://example.com/path/page.html
```

or with HTTPS:

```
python example.py https://example.com/path/page.html
```

Register an open socket to wait for data using streams

Coroutine waiting until a socket receives data using the *open_connection()* function:

```
import asyncio
import socket

async def wait_for_data():
    # Get a reference to the current event loop because
    # we want to access low-level APIs.
    loop = asyncio.get_running_loop()

    # Create a pair of connected sockets.
    rsock, wsock = socket.socketpair()

    # Register the open socket to wait for data.
    reader, writer = await asyncio.open_connection(sock=rsock)

    # Simulate the reception of data from the network
    loop.call_soon(wsock.send, 'abc'.encode())

    # Wait for data
    data = await reader.read(100)

    # Got data, we are done: close the socket
    print("Received:", data.decode())
    writer.close()
    await writer.wait_closed()

    # Close the second socket
    wsock.close()

asyncio.run(wait_for_data())
```

See also

The *register an open socket to wait for data using a protocol* example uses a low-level protocol and the *loop.create_connection()* method.

The *watch a file descriptor for read events* example uses the low-level *loop.add_reader()* method to watch a file descriptor.

18.1.4 Synchronization Primitives

Source code: Lib/asyncio/locks.py

---

asyncio synchronization primitives are designed to be similar to those of the *threading* module with two important caveats:

- asyncio primitives are not thread-safe, therefore they should not be used for OS thread synchronization (use *threading* for that);
- methods of these synchronization primitives do not accept the *timeout* argument; use the *asyncio.wait_for()* function to perform operations with timeouts.

asyncio has the following basic synchronization primitives:

- *Lock*
- *Event*
- *Condition*
- *Semaphore*
- *BoundedSemaphore*
- *Barrier*

Lock

class asyncio.**Lock**

Implements a mutex lock for asyncio tasks. Not thread-safe.

An asyncio lock can be used to guarantee exclusive access to a shared resource.

The preferred way to use a Lock is an `async with` statement:

```
lock = asyncio.Lock()

# ... later
async with lock:
    # access shared state
```

which is equivalent to:

```
lock = asyncio.Lock()

# ... later
await lock.acquire()
try:
    # access shared state
finally:
    lock.release()
```

Changed in version 3.10: Removed the *loop* parameter.

coroutine acquire()

Acquire the lock.

This method waits until the lock is *unlocked*, sets it to *locked* and returns `True`.

When more than one coroutine is blocked in *acquire()* waiting for the lock to be unlocked, only one coroutine eventually proceeds.

Acquiring a lock is *fair*: the coroutine that proceeds will be the first coroutine that started waiting on the lock.

release()

Release the lock.

When the lock is *locked*, reset it to *unlocked* and return.

If the lock is *unlocked*, a *RuntimeError* is raised.

locked()

Return `True` if the lock is *locked*.

Event

class asyncio.**Event**

An event object. Not thread-safe.

An asyncio event can be used to notify multiple asyncio tasks that some event has happened.

An Event object manages an internal flag that can be set to *true* with the *set()* method and reset to *false* with the *clear()* method. The *wait()* method blocks until the flag is set to *true*. The flag is set to *false* initially.

Changed in version 3.10: Removed the *loop* parameter. Example:

```
async def waiter(event):
    print('waiting for it ...')
    await event.wait()
    print('... got it!')

async def main():
    # Create an Event object.
    event = asyncio.Event()

    # Spawn a Task to wait until 'event' is set.
    waiter_task = asyncio.create_task(waiter(event))

    # Sleep for 1 second and set the event.
    await asyncio.sleep(1)
    event.set()

    # Wait until the waiter task is finished.
    await waiter_task

asyncio.run(main())
```

coroutine wait()

Wait until the event is set.

If the event is set, return `True` immediately. Otherwise block until another task calls *set()*.

set()

Set the event.

All tasks waiting for event to be set will be immediately awakened.

clear()

Clear (unset) the event.

Tasks awaiting on *wait()* will now block until the *set()* method is called again.

is_set()

Return `True` if the event is set.

Condition

class asyncio.**Condition**(*lock=None*)

A Condition object. Not thread-safe.

An asyncio condition primitive can be used by a task to wait for some event to happen and then get exclusive access to a shared resource.

In essence, a Condition object combines the functionality of an *Event* and a *Lock*. It is possible to have multiple Condition objects share one Lock, which allows coordinating exclusive access to a shared resource between different tasks interested in particular states of that shared resource.

The optional *lock* argument must be a *Lock* object or `None`. In the latter case a new Lock object is created automatically.

Changed in version 3.10: Removed the *loop* parameter.

The preferred way to use a Condition is an `async with` statement:

```
cond = asyncio.Condition()

# ... later
async with cond:
    await cond.wait()
```

which is equivalent to:

```
cond = asyncio.Condition()

# ... later
await cond.acquire()
try:
    await cond.wait()
finally:
    cond.release()
```

coroutine acquire()

Acquire the underlying lock.

This method waits until the underlying lock is *unlocked*, sets it to *locked* and returns `True`.

notify(*n=1*)

Wake up *n* tasks (1 by default) waiting on this condition. If fewer than *n* tasks are waiting they are all awakened.

The lock must be acquired before this method is called and released shortly after. If called with an *unlocked* lock a *RuntimeError* error is raised.

locked()

Return `True` if the underlying lock is acquired.

notify_all()

Wake up all tasks waiting on this condition.

This method acts like *notify()*, but wakes up all waiting tasks.

The lock must be acquired before this method is called and released shortly after. If called with an *unlocked* lock a *RuntimeError* error is raised.

release()

Release the underlying lock.

When invoked on an unlocked lock, a *RuntimeError* is raised.

coroutine wait()

Wait until notified.

If the calling task has not acquired the lock when this method is called, a *RuntimeError* is raised.

This method releases the underlying lock, and then blocks until it is awakened by a *notify()* or *notify_all()* call. Once awakened, the Condition re-acquires its lock and this method returns `True`.

Note that a task *may* return from this call spuriously, which is why the caller should always re-check the state and be prepared to *wait()* again. For this reason, you may prefer to use *wait_for()* instead.

coroutine wait_for(*predicate*)

Wait until a predicate becomes *true*.

The predicate must be a callable which result will be interpreted as a boolean value. The method will repeatedly *wait()* until the predicate evaluates to *true*. The final value is the return value.

Semaphore

class asyncio.**Semaphore**(*value=1*)

A Semaphore object. Not thread-safe.

A semaphore manages an internal counter which is decremented by each *acquire()* call and incremented by each *release()* call. The counter can never go below zero; when *acquire()* finds that it is zero, it blocks, waiting until some task calls *release()*.

The optional *value* argument gives the initial value for the internal counter (`1` by default). If the given value is less than `0` a *ValueError* is raised.

Changed in version 3.10: Removed the *loop* parameter.

The preferred way to use a Semaphore is an `async with` statement:

```
sem = asyncio.Semaphore(10)

# ... later
async with sem:
    # work with shared resource
```

which is equivalent to:

```
sem = asyncio.Semaphore(10)

# ... later
await sem.acquire()
try:
    # work with shared resource
finally:
    sem.release()
```

coroutine acquire()

Acquire a semaphore.

If the internal counter is greater than zero, decrement it by one and return `True` immediately. If it is zero, wait until a *release()* is called and return `True`.

locked()

Returns `True` if semaphore can not be acquired immediately.

release()

Release a semaphore, incrementing the internal counter by one. Can wake up a task waiting to acquire the semaphore.

Unlike *BoundedSemaphore*, *Semaphore* allows making more `release()` calls than `acquire()` calls.

BoundedSemaphore

class asyncio.**BoundedSemaphore**(*value=1*)

A bounded semaphore object. Not thread-safe.

Bounded Semaphore is a version of *Semaphore* that raises a *ValueError* in *release()* if it increases the internal counter above the initial *value*.

Changed in version 3.10: Removed the *loop* parameter.

Barrier

class asyncio.**Barrier**(*parties*)

A barrier object. Not thread-safe.

A barrier is a simple synchronization primitive that allows to block until *parties* number of tasks are waiting on it. Tasks can wait on the *wait()* method and would be blocked until the specified number of tasks end up waiting on *wait()*. At that point all of the waiting tasks would unblock simultaneously.

`async with` can be used as an alternative to awaiting on *wait()*.

The barrier can be reused any number of times.

Example:

```
async def example_barrier():
   # barrier with 3 parties
   b = asyncio.Barrier(3)

   # create 2 new waiting tasks
   asyncio.create_task(b.wait())
```

(continues on next page)

(continued from previous page)

```
    asyncio.create_task(b.wait())

    await asyncio.sleep(0)
    print(b)

    # The third .wait() call passes the barrier
    await b.wait()
    print(b)
    print("barrier passed")

    await asyncio.sleep(0)
    print(b)

asyncio.run(example_barrier())
```

Result of this example is:

```
<asyncio.locks.Barrier object at 0x... [filling, waiters:2/3]>
<asyncio.locks.Barrier object at 0x... [draining, waiters:0/3]>
barrier passed
<asyncio.locks.Barrier object at 0x... [filling, waiters:0/3]>
```

Added in version 3.11.

coroutine wait()

Pass the barrier. When all the tasks party to the barrier have called this function, they are all unblocked simultaneously.

When a waiting or blocked task in the barrier is cancelled, this task exits the barrier which stays in the same state. If the state of the barrier is "filling", the number of waiting task decreases by 1.

The return value is an integer in the range of 0 to `parties-1`, different for each task. This can be used to select a task to do some special housekeeping, e.g.:

```
...
async with barrier as position:
    if position == 0:
        # Only one task prints this
        print('End of *draining phase*')
```

This method may raise a *BrokenBarrierError* exception if the barrier is broken or reset while a task is waiting. It could raise a *CancelledError* if a task is cancelled.

coroutine reset()

Return the barrier to the default, empty state. Any tasks waiting on it will receive the *BrokenBarrierError* exception.

If a barrier is broken it may be better to just leave it and create a new one.

coroutine abort()

Put the barrier into a broken state. This causes any active or future calls to *wait()* to fail with the *BrokenBarrierError*. Use this for example if one of the tasks needs to abort, to avoid infinite waiting tasks.

parties

The number of tasks required to pass the barrier.

n_waiting

The number of tasks currently waiting in the barrier while filling.

broken

A boolean that is `True` if the barrier is in the broken state.

exception asyncio.**BrokenBarrierError**

This exception, a subclass of *RuntimeError*, is raised when the *Barrier* object is reset or broken.

Changed in version 3.9: Acquiring a lock using `await lock` or `yield from lock` and/or `with` statement (`with await lock`, `with (yield from lock)`) was removed. Use `async with lock` instead.

18.1.5 Subprocesses

Source code: Lib/asyncio/subprocess.py, Lib/asyncio/base_subprocess.py

---

This section describes high-level async/await asyncio APIs to create and manage subprocesses.

Here's an example of how asyncio can run a shell command and obtain its result:

```
import asyncio

async def run(cmd):
    proc = await asyncio.create_subprocess_shell(
        cmd,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE)

    stdout, stderr = await proc.communicate()

    print(f'[{cmd!r} exited with {proc.returncode}]')
    if stdout:
        print(f'[stdout]\n{stdout.decode()}')
    if stderr:
        print(f'[stderr]\n{stderr.decode()}')

asyncio.run(run('ls /zzz'))
```

will print:

```
['ls /zzz' exited with 1]
[stderr]
ls: /zzz: No such file or directory
```

Because all asyncio subprocess functions are asynchronous and asyncio provides many tools to work with such functions, it is easy to execute and monitor multiple subprocesses in parallel. It is indeed trivial to modify the above example to run several commands simultaneously:

```
async def main():
    await asyncio.gather(
        run('ls /zzz'),
        run('sleep 1; echo "hello"'))

asyncio.run(main())
```

See also the *Examples* subsection.

Creating Subprocesses

coroutine asyncio.**create_subprocess_exec**(*program*, **args*, *stdin=None*, *stdout=None*, *stderr=None*, *limit=None*, ***kwds*)

Create a subprocess.

The *limit* argument sets the buffer limit for `StreamReader` wrappers for `Process.stdout` and `Process.stderr` (if `subprocess.PIPE` is passed to *stdout* and *stderr* arguments).

Return a `Process` instance.

See the documentation of `loop.subprocess_exec()` for other parameters.

Changed in version 3.10: Removed the *loop* parameter.

coroutine asyncio.**create_subprocess_shell**(*cmd*, *stdin=None*, *stdout=None*, *stderr=None*, *limit=None*, ***kwds*)

Run the *cmd* shell command.

The *limit* argument sets the buffer limit for `StreamReader` wrappers for `Process.stdout` and `Process.stderr` (if `subprocess.PIPE` is passed to *stdout* and *stderr* arguments).

Return a `Process` instance.

See the documentation of `loop.subprocess_shell()` for other parameters.

Important

It is the application's responsibility to ensure that all whitespace and special characters are quoted appropriately to avoid shell injection vulnerabilities. The `shlex.quote()` function can be used to properly escape whitespace and special shell characters in strings that are going to be used to construct shell commands.

Changed in version 3.10: Removed the *loop* parameter.

Note

Subprocesses are available for Windows if a `ProactorEventLoop` is used. See *Subprocess Support on Windows* for details.

See also

asyncio also has the following *low-level* APIs to work with subprocesses: `loop.subprocess_exec()`, `loop.subprocess_shell()`, `loop.connect_read_pipe()`, `loop.connect_write_pipe()`, as well as the *Subprocess Transports* and *Subprocess Protocols*.

Constants

asyncio.subprocess.**PIPE**

Can be passed to the *stdin*, *stdout* or *stderr* parameters.

If *PIPE* is passed to *stdin* argument, the *Process.stdin* attribute will point to a `StreamWriter` instance.

If *PIPE* is passed to *stdout* or *stderr* arguments, the *Process.stdout* and *Process.stderr* attributes will point to `StreamReader` instances.

asyncio.subprocess.**STDOUT**

Special value that can be used as the *stderr* argument and indicates that standard error should be redirected into standard output.

asyncio.subprocess.**DEVNULL**

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to process creation functions. It indicates that the special file *os.devnull* will be used for the corresponding subprocess stream.

Interacting with Subprocesses

Both *create_subprocess_exec()* and *create_subprocess_shell()* functions return instances of the *Process* class. *Process* is a high-level wrapper that allows communicating with subprocesses and watching for their completion.

class asyncio.subprocess.**Process**

An object that wraps OS processes created by the `create_subprocess_exec()` and `create_subprocess_shell()` functions.

This class is designed to have a similar API to the *subprocess.Popen* class, but there are some notable differences:

- unlike Popen, Process instances do not have an equivalent to the *poll()* method;
- the *communicate()* and *wait()* methods don't have a *timeout* parameter: use the *wait_for()* function;
- the *Process.wait()* method is asynchronous, whereas *subprocess.Popen.wait()* method is implemented as a blocking busy loop;
- the *universal_newlines* parameter is not supported.

This class is *not thread safe*.

See also the *Subprocess and Threads* section.

coroutine wait()

Wait for the child process to terminate.

Set and return the *returncode* attribute.

> **Note**
>
> This method can deadlock when using `stdout=PIPE` or `stderr=PIPE` and the child process generates so much output that it blocks waiting for the OS pipe buffer to accept more data. Use the *communicate()* method when using pipes to avoid this condition.

coroutine communicate(*input=None*)

Interact with process:

1. send data to *stdin* (if *input* is not `None`);
2. closes *stdin*;
3. read data from *stdout* and *stderr*, until EOF is reached;
4. wait for process to terminate.

The optional *input* argument is the data (*bytes* object) that will be sent to the child process.

Return a tuple `(stdout_data, stderr_data)`.

If either *BrokenPipeError* or *ConnectionResetError* exception is raised when writing *input* into *stdin*, the exception is ignored. This condition occurs when the process exits before all data are written into *stdin*.

If it is desired to send data to the process' *stdin*, the process needs to be created with `stdin=PIPE`. Similarly, to get anything other than `None` in the result tuple, the process has to be created with `stdout=PIPE` and/or `stderr=PIPE` arguments.

Note, that the data read is buffered in memory, so do not use this method if the data size is large or unlimited.

Changed in version 3.12: *stdin* gets closed when *input=None* too.

send_signal(*signal*)

Sends the signal *signal* to the child process.

> **Note**
>
> On Windows, *SIGTERM* is an alias for *terminate()*. `CTRL_C_EVENT` and `CTRL_BREAK_EVENT` can be sent to processes started with a *creationflags* parameter which includes `CREATE_NEW_PROCESS_GROUP`.

terminate()

Stop the child process.

On POSIX systems this method sends *SIGTERM* to the child process.

On Windows the Win32 API function `TerminateProcess()` is called to stop the child process.

kill()

Kill the child process.

On POSIX systems this method sends `SIGKILL` to the child process.

On Windows this method is an alias for *terminate()*.

stdin

Standard input stream (`StreamWriter`) or `None` if the process was created with `stdin=None`.

stdout

Standard output stream (`StreamReader`) or `None` if the process was created with `stdout=None`.

stderr

Standard error stream (`StreamReader`) or `None` if the process was created with `stderr=None`.

> **Warning**
>
> Use the *communicate()* method rather than *process.stdin.write()*, *await process.stdout.read()* or *await process.stderr.read()*. This avoids deadlocks due to streams pausing reading or writing and blocking the child process.

pid

Process identification number (PID).

Note that for processes created by the `create_subprocess_shell()` function, this attribute is the PID of the spawned shell.

returncode

Return code of the process when it exits.

A `None` value indicates that the process has not terminated yet.

A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).

Subprocess and Threads

Standard asyncio event loop supports running subprocesses from different threads by default.

On Windows subprocesses are provided by *ProactorEventLoop* only (default), *SelectorEventLoop* has no subprocess support.

On UNIX *child watchers* are used for subprocess finish waiting, see *Process Watchers* for more info.

Changed in version 3.8: UNIX switched to use *ThreadedChildWatcher* for spawning subprocesses from different threads without any limitation.

Spawning a subprocess with *inactive* current child watcher raises *RuntimeError*.

Note that alternative event loop implementations might have own limitations; please refer to their documentation.

> **See also**
>
> The *Concurrency and multithreading in asyncio* section.

Examples

An example using the *Process* class to control a subprocess and the *StreamReader* class to read from its standard output.

The subprocess is created by the *create_subprocess_exec()* function:

```
import asyncio
import sys

async def get_date():
    code = 'import datetime; print(datetime.datetime.now())'

    # Create the subprocess; redirect the standard output
    # into a pipe.
```

(continues on next page)

(continued from previous page)

```
    proc = await asyncio.create_subprocess_exec(
        sys.executable, '-c', code,
        stdout=asyncio.subprocess.PIPE)

    # Read one line of output.
    data = await proc.stdout.readline()
    line = data.decode('ascii').rstrip()

    # Wait for the subprocess exit.
    await proc.wait()
    return line

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

See also the *same example* written using low-level APIs.

18.1.6 Queues

Source code: Lib/asyncio/queues.py

---

asyncio queues are designed to be similar to classes of the `queue` module. Although asyncio queues are not thread-safe, they are designed to be used specifically in async/await code.

Note that methods of asyncio queues don't have a *timeout* parameter; use `asyncio.wait_for()` function to do queue operations with a timeout.

See also the *Examples* section below.

Queue

class asyncio.**Queue**(*maxsize=0*)

A first in, first out (FIFO) queue.

If *maxsize* is less than or equal to zero, the queue size is infinite. If it is an integer greater than `0`, then `await put()` blocks when the queue reaches *maxsize* until an item is removed by `get()`.

Unlike the standard library threading `queue`, the size of the queue is always known and can be returned by calling the `qsize()` method.

Changed in version 3.10: Removed the *loop* parameter.

This class is *not thread safe*.

maxsize

Number of items allowed in the queue.

empty()

Return `True` if the queue is empty, `False` otherwise.

full()

Return `True` if there are `maxsize` items in the queue.

If the queue was initialized with `maxsize=0` (the default), then `full()` never returns `True`.

coroutine get()

Remove and return an item from the queue. If queue is empty, wait until an item is available.

Raises *QueueShutDown* if the queue has been shut down and is empty, or if the queue has been shut down immediately.

get_nowait()

Return an item if one is immediately available, else raise *QueueEmpty*.

coroutine join()

Block until all items in the queue have been received and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer coroutine calls *task_done()* to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, *join()* unblocks.

coroutine put(*item*)

Put an item into the queue. If the queue is full, wait until a free slot is available before adding the item.

Raises *QueueShutDown* if the queue has been shut down.

put_nowait(*item*)

Put an item into the queue without blocking.

If no free slot is immediately available, raise *QueueFull*.

qsize()

Return the number of items in the queue.

shutdown(*immediate=False*)

Shut down the queue, making *get()* and *put()* raise *QueueShutDown*.

By default, *get()* on a shut down queue will only raise once the queue is empty. Set *immediate* to true to make *get()* raise immediately instead.

All blocked callers of *put()* and *get()* will be unblocked. If *immediate* is true, a task will be marked as done for each remaining item in the queue, which may unblock callers of *join()*.

Added in version 3.13.

task_done()

Indicate that a formerly enqueued task is complete.

Used by queue consumers. For each *get()* used to fetch a task, a subsequent call to *task_done()* tells the queue that the processing on the task is complete.

If a *join()* is currently blocking, it will resume when all items have been processed (meaning that a *task_done()* call was received for every item that had been *put()* into the queue).

`shutdown(immediate=True)` calls *task_done()* for each remaining item in the queue.

Raises *ValueError* if called more times than there were items placed in the queue.

Priority Queue

class asyncio.**PriorityQueue**

A variant of *Queue*; retrieves entries in priority order (lowest first).

Entries are typically tuples of the form `(priority_number, data)`.

LIFO Queue

class asyncio.**LifoQueue**

A variant of *Queue* that retrieves most recently added entries first (last in, first out).

Exceptions

exception asyncio.**QueueEmpty**

This exception is raised when the *get_nowait()* method is called on an empty queue.

exception asyncio.**QueueFull**

Exception raised when the *put_nowait()* method is called on a queue that has reached its *maxsize*.

exception asyncio.**QueueShutDown**

Exception raised when *put()* or *get()* is called on a queue which has been shut down.

Added in version 3.13.

Examples

Queues can be used to distribute workload between several concurrent tasks:

```
import asyncio
import random
import time


async def worker(name, queue):
    while True:
        # Get a "work item" out of the queue.
        sleep_for = await queue.get()

        # Sleep for the "sleep_for" seconds.
        await asyncio.sleep(sleep_for)

        # Notify the queue that the "work item" has been processed.
        queue.task_done()

        print(f'{name} has slept for {sleep_for:.2f} seconds')


async def main():
    # Create a queue that we will use to store our "workload".
    queue = asyncio.Queue()

    # Generate random timings and put them into the queue.
    total_sleep_time = 0
```

(continues on next page)

(continued from previous page)

```
    for _ in range(20):
        sleep_for = random.uniform(0.05, 1.0)
        total_sleep_time += sleep_for
        queue.put_nowait(sleep_for)

    # Create three worker tasks to process the queue concurrently.
    tasks = []
    for i in range(3):
        task = asyncio.create_task(worker(f'worker-{i}', queue))
        tasks.append(task)

    # Wait until the queue is fully processed.
    started_at = time.monotonic()
    await queue.join()
    total_slept_for = time.monotonic() - started_at

    # Cancel our worker tasks.
    for task in tasks:
        task.cancel()
    # Wait until all worker tasks are cancelled.
    await asyncio.gather(*tasks, return_exceptions=True)

    print('====')
    print(f'3 workers slept in parallel for {total_slept_for:.2f} seconds')
    print(f'total expected sleep time: {total_sleep_time:.2f} seconds')


asyncio.run(main())
```

18.1.7 Exceptions

Source code: Lib/asyncio/exceptions.py

---

exception asyncio.**TimeoutError**

A deprecated alias of *TimeoutError*, raised when the operation has exceeded the given deadline.

Changed in version 3.11: This class was made an alias of *TimeoutError*.

exception asyncio.**CancelledError**

The operation has been cancelled.

This exception can be caught to perform custom operations when asyncio Tasks are cancelled. In almost all situations the exception must be re-raised.

Changed in version 3.8: *CancelledError* is now a subclass of *BaseException* rather than *Exception*.

exception asyncio.**InvalidStateError**

Invalid internal state of *Task* or *Future*.

Can be raised in situations like setting a result value for a *Future* object that already has a result value set.

exception asyncio.**SendfileNotAvailableError**

The "sendfile" syscall is not available for the given socket or file type.

A subclass of *RuntimeError*.

exception asyncio.**IncompleteReadError**

The requested read operation did not complete fully.

Raised by the *asyncio stream APIs*.

This exception is a subclass of *EOFError*.

expected

The total number (*int*) of expected bytes.

partial

A string of *bytes* read before the end of stream was reached.

exception asyncio.**LimitOverrunError**

Reached the buffer size limit while looking for a separator.

Raised by the *asyncio stream APIs*.

consumed

The total number of to be consumed bytes.

18.1.8 Event Loop

Source code: Lib/asyncio/events.py, Lib/asyncio/base_events.py

---

Preface

The event loop is the core of every asyncio application. Event loops run asynchronous tasks and callbacks, perform network IO operations, and run subprocesses.

Application developers should typically use the high-level asyncio functions, such as *asyncio.run()*, and should rarely need to reference the loop object or call its methods. This section is intended mostly for authors of lower-level code, libraries, and frameworks, who need finer control over the event loop behavior.

Obtaining the Event Loop

The following low-level functions can be used to get, set, or create an event loop:

asyncio.**get_running_loop**()

Return the running event loop in the current OS thread.

Raise a *RuntimeError* if there is no running event loop.

This function can only be called from a coroutine or a callback.

Added in version 3.7.

asyncio.**get_event_loop**()

Get the current event loop.

When called from a coroutine or a callback (e.g. scheduled with call_soon or similar API), this function will always return the running event loop.

If there is no running event loop set, the function will return the result of the `get_event_loop_policy().get_event_loop()` call.

Because this function has rather complex behavior (especially when custom event loop policies are in use), using the `get_running_loop()` function is preferred to `get_event_loop()` in coroutines and callbacks.

As noted above, consider using the higher-level `asyncio.run()` function, instead of using these lower level functions to manually create and close an event loop.

Deprecated since version 3.12: Deprecation warning is emitted if there is no current event loop. In some future Python release this will become an error.

`asyncio.`**`set_event_loop`**`(`*loop*`)`

Set *loop* as the current event loop for the current OS thread.

`asyncio.`**`new_event_loop`**`()`

Create and return a new event loop object.

Note that the behaviour of `get_event_loop()`, `set_event_loop()`, and `new_event_loop()` functions can be altered by *setting a custom event loop policy*.

Contents

This documentation page contains the following sections:

- The *Event Loop Methods* section is the reference documentation of the event loop APIs;
- The *Callback Handles* section documents the `Handle` and `TimerHandle` instances which are returned from scheduling methods such as `loop.call_soon()` and `loop.call_later()`;
- The *Server Objects* section documents types returned from event loop methods like `loop.create_server()`;
- The *Event Loop Implementations* section documents the `SelectorEventLoop` and `ProactorEventLoop` classes;
- The *Examples* section showcases how to work with some event loop APIs.

Event Loop Methods

Event loops have **low-level** APIs for the following:

- *Running and stopping the loop*
- *Scheduling callbacks*
- *Scheduling delayed callbacks*
- *Creating Futures and Tasks*
- *Opening network connections*
- *Creating network servers*
- *Transferring files*
- *TLS Upgrade*
- *Watching file descriptors*
- *Working with socket objects directly*
- *DNS*

- *Working with pipes*
- *Unix signals*
- *Executing code in thread or process pools*
- *Error Handling API*
- *Enabling debug mode*
- *Running Subprocesses*

Running and stopping the loop

`loop.`**`run_until_complete`**`(`*future*`)`

Run until the *future* (an instance of `Future`) has completed.

If the argument is a *coroutine object* it is implicitly scheduled to run as a `asyncio.Task`.

Return the Future's result or raise its exception.

`loop.`**`run_forever`**`()`

Run the event loop until `stop()` is called.

If `stop()` is called before `run_forever()` is called, the loop will poll the I/O selector once with a timeout of zero, run all callbacks scheduled in response to I/O events (and those that were already scheduled), and then exit.

If `stop()` is called while `run_forever()` is running, the loop will run the current batch of callbacks and then exit. Note that new callbacks scheduled by callbacks will not run in this case; instead, they will run the next time `run_forever()` or `run_until_complete()` is called.

`loop.`**`stop`**`()`

Stop the event loop.

`loop.`**`is_running`**`()`

Return `True` if the event loop is currently running.

`loop.`**`is_closed`**`()`

Return `True` if the event loop was closed.

`loop.`**`close`**`()`

Close the event loop.

The loop must not be running when this function is called. Any pending callbacks will be discarded.

This method clears all queues and shuts down the executor, but does not wait for the executor to finish.

This method is idempotent and irreversible. No other methods should be called after the event loop is closed.

`coroutine` `loop.`**`shutdown_asyncgens`**`()`

Schedule all currently open *asynchronous generator* objects to close with an `aclose()` call. After calling this method, the event loop will issue a warning if a new asynchronous generator is iterated. This should be used to reliably finalize all scheduled asynchronous generators.

Note that there is no need to call this function when `asyncio.run()` is used.

Example:

```
try:
    loop.run_forever()
finally:
    loop.run_until_complete(loop.shutdown_asyncgens())
    loop.close()
```

Added in version 3.6.

coroutine loop.**shutdown_default_executor**(*timeout=None*)

Schedule the closure of the default executor and wait for it to join all of the threads in the *ThreadPoolExecutor*. Once this method has been called, using the default executor with *loop.run_in_executor()* will raise a *RuntimeError*.

The *timeout* parameter specifies the amount of time (in *float* seconds) the executor will be given to finish joining. With the default, None, the executor is allowed an unlimited amount of time.

If the *timeout* is reached, a *RuntimeWarning* is emitted and the default executor is terminated without waiting for its threads to finish joining.

Note

Do not call this method when using *asyncio.run()*, as the latter handles default executor shutdown automatically.

Added in version 3.9.

Changed in version 3.12: Added the *timeout* parameter.

Scheduling callbacks

loop.**call_soon**(*callback*, **args*, *context=None*)

Schedule the *callback* *callback* to be called with *args* arguments at the next iteration of the event loop.

Return an instance of *asyncio.Handle*, which can be used later to cancel the callback.

Callbacks are called in the order in which they are registered. Each callback will be called exactly once.

The optional keyword-only *context* argument specifies a custom *contextvars.Context* for the *callback* to run in. Callbacks use the current context when no *context* is provided.

Unlike *call_soon_threadsafe()*, this method is not thread-safe.

loop.**call_soon_threadsafe**(*callback*, **args*, *context=None*)

A thread-safe variant of *call_soon()*. When scheduling callbacks from another thread, this function *must* be used, since *call_soon()* is not thread-safe.

Raises *RuntimeError* if called on a loop that's been closed. This can happen on a secondary thread when the main application is shutting down.

See the *concurrency and multithreading* section of the documentation.

Changed in version 3.7: The *context* keyword-only parameter was added. See **PEP 567** for more details.

Note

Most *asyncio* scheduling functions don't allow passing keyword arguments. To do that, use *functools.partial()*:

```
# will schedule "print("Hello", flush=True)"
loop.call_soon(
    functools.partial(print, "Hello", flush=True))
```

Using partial objects is usually more convenient than using lambdas, as asyncio can render partial objects better in debug and error messages.

Scheduling delayed callbacks

Event loop provides mechanisms to schedule callback functions to be called at some point in the future. Event loop uses monotonic clocks to track time.

`loop.`**`call_later`**(*delay*, *callback*, **args*, *context=None*)

Schedule *callback* to be called after the given *delay* number of seconds (can be either an int or a float).

An instance of *asyncio.TimerHandle* is returned which can be used to cancel the callback.

callback will be called exactly once. If two callbacks are scheduled for exactly the same time, the order in which they are called is undefined.

The optional positional *args* will be passed to the callback when it is called. If you want the callback to be called with keyword arguments use *functools.partial()*.

An optional keyword-only *context* argument allows specifying a custom *contextvars.Context* for the *callback* to run in. The current context is used when no *context* is provided.

Changed in version 3.7: The *context* keyword-only parameter was added. See **PEP 567** for more details.

Changed in version 3.8: In Python 3.7 and earlier with the default event loop implementation, the *delay* could not exceed one day. This has been fixed in Python 3.8.

`loop.`**`call_at`**(*when*, *callback*, **args*, *context=None*)

Schedule *callback* to be called at the given absolute timestamp *when* (an int or a float), using the same time reference as *loop.time()*.

This method's behavior is the same as *call_later()*.

An instance of *asyncio.TimerHandle* is returned which can be used to cancel the callback.

Changed in version 3.7: The *context* keyword-only parameter was added. See **PEP 567** for more details.

Changed in version 3.8: In Python 3.7 and earlier with the default event loop implementation, the difference between *when* and the current time could not exceed one day. This has been fixed in Python 3.8.

`loop.`**`time`**()

Return the current time, as a *float* value, according to the event loop's internal monotonic clock.

Note

Changed in version 3.8: In Python 3.7 and earlier timeouts (relative *delay* or absolute *when*) should not exceed one day. This has been fixed in Python 3.8.

See also

The *asyncio.sleep()* function.

Creating Futures and Tasks

`loop.`**`create_future`**`()`

Create an *asyncio.Future* object attached to the event loop.

This is the preferred way to create Futures in asyncio. This lets third-party event loops provide alternative implementations of the Future object (with better performance or instrumentation).

Added in version 3.5.2.

`loop.`**`create_task`**`(`*coro*, *, *name=None*, *context=None*`)`

Schedule the execution of *coroutine* *coro*. Return a *Task* object.

Third-party event loops can use their own subclass of *Task* for interoperability. In this case, the result type is a subclass of *Task*.

If the *name* argument is provided and not `None`, it is set as the name of the task using *Task.set_name()*.

An optional keyword-only *context* argument allows specifying a custom *contextvars.Context* for the *coro* to run in. The current context copy is created when no *context* is provided.

Changed in version 3.8: Added the *name* parameter.

Changed in version 3.11: Added the *context* parameter.

`loop.`**`set_task_factory`**`(`*factory*`)`

Set a task factory that will be used by *loop.create_task()*.

If *factory* is `None` the default task factory will be set. Otherwise, *factory* must be a *callable* with the signature matching `(loop, coro, context=None)`, where *loop* is a reference to the active event loop, and *coro* is a coroutine object. The callable must return a *asyncio.Future*-compatible object.

`loop.`**`get_task_factory`**`()`

Return a task factory or `None` if the default one is in use.

Opening network connections

`coroutine` `loop.`**`create_connection`**`(`*protocol_factory*, *host=None*, *port=None*, *, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *happy_eyeballs_delay=None*, *interleave=None*, *all_errors=False*`)`

Open a streaming transport connection to a given address specified by *host* and *port*.

The socket family can be either *AF_INET* or *AF_INET6* depending on *host* (or the *family* argument, if provided).

The socket type will be *SOCK_STREAM*.

protocol_factory must be a callable returning an *asyncio protocol* implementation.

This method will try to establish the connection in the background. When successful, it returns a `(transport, protocol)` pair.

The chronological synopsis of the underlying operation is as follows:

1. The connection is established and a *transport* is created for it.
2. *protocol_factory* is called without arguments and is expected to return a *protocol* instance.
3. The protocol instance is coupled with the transport by calling its `connection_made()` method.
4. A `(transport, protocol)` tuple is returned on success.

The created transport is an implementation-dependent bidirectional stream.

Other arguments:

- *ssl*: if given and not false, a SSL/TLS transport is created (by default a plain TCP transport is created). If *ssl* is a `ssl.SSLContext` object, this context is used to create the transport; if *ssl* is `True`, a default context returned from `ssl.create_default_context()` is used.

  See also

  SSL/TLS security considerations

- *server_hostname* sets or overrides the hostname that the target server's certificate will be matched against. Should only be passed if *ssl* is not `None`. By default the value of the *host* argument is used. If *host* is empty, there is no default and you must pass a value for *server_hostname*. If *server_hostname* is an empty string, hostname matching is disabled (which is a serious security risk, allowing for potential man-in-the-middle attacks).
- *family*, *proto*, *flags* are the optional address family, protocol and flags to be passed through to getaddrinfo() for *host* resolution. If given, these should all be integers from the corresponding `socket` module constants.
- *happy_eyeballs_delay*, if given, enables Happy Eyeballs for this connection. It should be a floating-point number representing the amount of time in seconds to wait for a connection attempt to complete, before starting the next attempt in parallel. This is the "Connection Attempt Delay" as defined in **RFC 8305**. A sensible default value recommended by the RFC is `0.25` (250 milliseconds).
- *interleave* controls address reordering when a host name resolves to multiple IP addresses. If `0` or unspecified, no reordering is done, and addresses are tried in the order returned by `getaddrinfo()`. If a positive integer is specified, the addresses are interleaved by address family, and the given integer is interpreted as "First Address Family Count" as defined in **RFC 8305**. The default is `0` if *happy_eyeballs_delay* is not specified, and `1` if it is.
- *sock*, if given, should be an existing, already connected `socket.socket` object to be used by the transport. If *sock* is given, none of *host*, *port*, *family*, *proto*, *flags*, *happy_eyeballs_delay*, *interleave* and *local_addr* should be specified.

  Note

  The *sock* argument transfers ownership of the socket to the transport created. To close the socket, call the transport's `close()` method.

- *local_addr*, if given, is a `(local_host, local_port)` tuple used to bind the socket locally. The *local_host* and *local_port* are looked up using `getaddrinfo()`, similarly to *host* and *port*.
- *ssl_handshake_timeout* is (for a TLS connection) the time in seconds to wait for the TLS handshake to complete before aborting the connection. `60.0` seconds if `None` (default).
- *ssl_shutdown_timeout* is the time in seconds to wait for the SSL shutdown to complete before aborting the connection. `30.0` seconds if `None` (default).

- *all_errors* determines what exceptions are raised when a connection cannot be created. By default, only a single `Exception` is raised: the first exception if there is only one or all errors have same message, or a single `OSError` with the error messages combined. When `all_errors` is `True`, an `ExceptionGroup` will be raised containing all exceptions (even if there is only one).

Changed in version 3.5: Added support for SSL/TLS in *ProactorEventLoop*.

Changed in version 3.6: The socket option *socket.TCP_NODELAY* is set by default for all TCP connections.

Changed in version 3.7: Added the *ssl_handshake_timeout* parameter.

Changed in version 3.8: Added the *happy_eyeballs_delay* and *interleave* parameters.

Happy Eyeballs Algorithm: Success with Dual-Stack Hosts. When a server's IPv4 path and protocol are working, but the server's IPv6 path and protocol are not working, a dual-stack client application experiences significant connection delay compared to an IPv4-only client. This is undesirable because it causes the dual-stack client to have a worse user experience. This document specifies requirements for algorithms that reduce this user-visible delay and provides an algorithm.

For more information: https://datatracker.ietf.org/doc/html/rfc6555

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

Changed in version 3.12: *all_errors* was added.

See also

The *open_connection()* function is a high-level alternative API. It returns a pair of (*StreamReader*, *StreamWriter*) that can be used directly in async/await code.

coroutine loop.**create_datagram_endpoint**(*protocol_factory*, *local_addr=None*, *remote_addr=None*, ***, *family=0*, *proto=0*, *flags=0*, *reuse_port=None*, *allow_broadcast=None*, *sock=None*)

Create a datagram connection.

The socket family can be either *AF_INET*, *AF_INET6*, or *AF_UNIX*, depending on *host* (or the *family* argument, if provided).

The socket type will be *SOCK_DGRAM*.

protocol_factory must be a callable returning a *protocol* implementation.

A tuple of `(transport, protocol)` is returned on success.

Other arguments:

- *local_addr*, if given, is a `(local_host, local_port)` tuple used to bind the socket locally. The *local_host* and *local_port* are looked up using *getaddrinfo()*.
- *remote_addr*, if given, is a `(remote_host, remote_port)` tuple used to connect the socket to a remote address. The *remote_host* and *remote_port* are looked up using *getaddrinfo()*.
- *family*, *proto*, *flags* are the optional address family, protocol and flags to be passed through to *getaddrinfo()* for *host* resolution. If given, these should all be integers from the corresponding *socket* module constants.
- *reuse_port* tells the kernel to allow this endpoint to be bound to the same port as other existing endpoints are bound to, so long as they all set this flag when being created. This option is not supported on Windows and some Unixes. If the *socket.SO_REUSEPORT* constant is not defined then this capability is unsupported.
- *allow_broadcast* tells the kernel to allow this endpoint to send messages to the broadcast address.

- *sock* can optionally be specified in order to use a preexisting, already connected, `socket.socket` object to be used by the transport. If specified, *local_addr* and *remote_addr* should be omitted (must be `None`).

> **Note**
>
> The *sock* argument transfers ownership of the socket to the transport created. To close the socket, call the transport's `close()` method.

See *UDP echo client protocol* and *UDP echo server protocol* examples.

Changed in version 3.4.4: The *family*, *proto*, *flags*, *reuse_address*, *reuse_port*, *allow_broadcast*, and *sock* parameters were added.

Changed in version 3.8: Added support for Windows.

Changed in version 3.8.1: The *reuse_address* parameter is no longer supported, as using *socket.SO_REUSEADDR* poses a significant security concern for UDP. Explicitly passing `reuse_address=True` will raise an exception.

When multiple processes with differing UIDs assign sockets to an identical UDP socket address with `SO_REUSEADDR`, incoming packets can become randomly distributed among the sockets.

For supported platforms, *reuse_port* can be used as a replacement for similar functionality. With *reuse_port*, *socket.SO_REUSEPORT* is used instead, which specifically prevents processes with differing UIDs from assigning sockets to the same socket address.

Changed in version 3.11: The *reuse_address* parameter, disabled since Python 3.8.1, 3.7.6 and 3.6.10, has been entirely removed.

coroutine `loop.`**create_unix_connection**(*protocol_factory*, *path=None*, *, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

Create a Unix connection.

The socket family will be `AF_UNIX`; socket type will be `SOCK_STREAM`.

A tuple of `(transport, protocol)` is returned on success.

path is the name of a Unix domain socket and is required, unless a *sock* parameter is specified. Abstract Unix sockets, `str`, `bytes`, and `Path` paths are supported.

See the documentation of the `loop.create_connection()` method for information about arguments to this method.

Availability: Unix.

Changed in version 3.7: Added the *ssl_handshake_timeout* parameter. The *path* parameter can now be a *path-like object*.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

Creating network servers

coroutine loop.**create_server**(*protocol_factory*, *host=None*, *port=None*, *, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *keep_alive=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*)

Create a TCP server (socket type *SOCK_STREAM*) listening on *port* of the *host* address.

Returns a *Server* object.

Arguments:

- *protocol_factory* must be a callable returning a *protocol* implementation.
- The *host* parameter can be set to several types which determine where the server would be listening:
  - If *host* is a string, the TCP server is bound to a single network interface specified by *host*.
  - If *host* is a sequence of strings, the TCP server is bound to all network interfaces specified by the sequence.
  - If *host* is an empty string or `None`, all interfaces are assumed and a list of multiple sockets will be returned (most likely one for IPv4 and another one for IPv6).
- The *port* parameter can be set to specify which port the server should listen on. If `0` or `None` (the default), a random unused port will be selected (note that if *host* resolves to multiple network interfaces, a different random port will be selected for each interface).
- *family* can be set to either *socket.AF_INET* or *AF_INET6* to force the socket to use IPv4 or IPv6. If not set, the *family* will be determined from host name (defaults to *AF_UNSPEC*).
- *flags* is a bitmask for *getaddrinfo()*.
- *sock* can optionally be specified in order to use a preexisting socket object. If specified, *host* and *port* must not be specified.

> **Note**
>
> The *sock* argument transfers ownership of the socket to the server created. To close the socket, call the server's *close()* method.

- *backlog* is the maximum number of queued connections passed to *listen()* (defaults to 100).
- *ssl* can be set to an *SSLContext* instance to enable TLS over the accepted connections.
- *reuse_address* tells the kernel to reuse a local socket in `TIME_WAIT` state, without waiting for its natural timeout to expire. If not specified will automatically be set to `True` on Unix.
- *reuse_port* tells the kernel to allow this endpoint to be bound to the same port as other existing endpoints are bound to, so long as they all set this flag when being created. This option is not supported on Windows.
- *keep_alive* set to `True` keeps connections active by enabling the periodic transmission of messages.

Changed in version 3.13: Added the *keep_alive* parameter.

- *ssl_handshake_timeout* is (for a TLS server) the time in seconds to wait for the TLS handshake to complete before aborting the connection. `60.0` seconds if `None` (default).
- *ssl_shutdown_timeout* is the time in seconds to wait for the SSL shutdown to complete before aborting the connection. `30.0` seconds if `None` (default).

- *start_serving* set to `True` (the default) causes the created server to start accepting connections immediately. When set to `False`, the user should await on `Server.start_serving()` or `Server.serve_forever()` to make the server to start accepting connections.

Changed in version 3.5: Added support for SSL/TLS in `ProactorEventLoop`.

Changed in version 3.5.1: The *host* parameter can be a sequence of strings.

Changed in version 3.6: Added *ssl_handshake_timeout* and *start_serving* parameters. The socket option *socket.TCP_NODELAY* is set by default for all TCP connections.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

See also

The `start_server()` function is a higher-level alternative API that returns a pair of `StreamReader` and `StreamWriter` that can be used in an async/await code.

coroutine loop.**create_unix_server**(*protocol_factory*, *path=None*, ***, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*, *cleanup_socket=True*)

Similar to `loop.create_server()` but works with the `AF_UNIX` socket family.

path is the name of a Unix domain socket, and is required, unless a *sock* argument is provided. Abstract Unix sockets, `str`, `bytes`, and `Path` paths are supported.

If *cleanup_socket* is true then the Unix socket will automatically be removed from the filesystem when the server is closed, unless the socket has been replaced after the server has been created.

See the documentation of the `loop.create_server()` method for information about arguments to this method.

Availability: Unix.

Changed in version 3.7: Added the *ssl_handshake_timeout* and *start_serving* parameters. The *path* parameter can now be a `Path` object.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

Changed in version 3.13: Added the *cleanup_socket* parameter.

coroutine loop.**connect_accepted_socket**(*protocol_factory*, *sock*, ***, *ssl=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

Wrap an already accepted connection into a transport/protocol pair.

This method can be used by servers that accept connections outside of asyncio but that use asyncio to handle them.

Parameters:

- *protocol_factory* must be a callable returning a *protocol* implementation.
- *sock* is a preexisting socket object returned from `socket.accept`.

Note

The *sock* argument transfers ownership of the socket to the transport created. To close the socket, call the transport's `close()` method.

- *ssl* can be set to an *SSLContext* to enable SSL over the accepted connections.
- *ssl_handshake_timeout* is (for an SSL connection) the time in seconds to wait for the SSL handshake to complete before aborting the connection. `60.0` seconds if `None` (default).
- *ssl_shutdown_timeout* is the time in seconds to wait for the SSL shutdown to complete before aborting the connection. `30.0` seconds if `None` (default).

Returns a `(transport, protocol)` pair.

Added in version 3.5.3.

Changed in version 3.7: Added the *ssl_handshake_timeout* parameter.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

Transferring files

coroutine loop.**sendfile**(*transport*, *file*, *offset=0*, *count=None*, *, *fallback=True*)

Send a *file* over a *transport*. Return the total number of bytes sent.

The method uses high-performance *os.sendfile()* if available.

file must be a regular file object opened in binary mode.

offset tells from where to start reading the file. If specified, *count* is the total number of bytes to transmit as opposed to sending the file until EOF is reached. File position is always updated, even when this method raises an error, and *file.tell()* can be used to obtain the actual number of bytes sent.

fallback set to `True` makes asyncio to manually read and send the file when the platform does not support the sendfile system call (e.g. Windows or SSL socket on Unix).

Raise *SendfileNotAvailableError* if the system does not support the *sendfile* syscall and *fallback* is `False`.

Added in version 3.7.

TLS Upgrade

coroutine loop.**start_tls**(*transport*, *protocol*, *sslcontext*, *, *server_side=False*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

Upgrade an existing transport-based connection to TLS.

Create a TLS coder/decoder instance and insert it between the *transport* and the *protocol*. The coder/decoder implements both *transport*-facing protocol and *protocol*-facing transport.

Return the created two-interface instance. After *await*, the *protocol* must stop using the original *transport* and communicate with the returned object only because the coder caches *protocol*-side data and sporadically exchanges extra TLS session packets with *transport*.

In some situations (e.g. when the passed transport is already closing) this may return `None`.

Parameters:

- *transport* and *protocol* instances that methods like *create_server()* and *create_connection()* return.
- *sslcontext*: a configured instance of *SSLContext*.
- *server_side* pass `True` when a server-side connection is being upgraded (like the one created by *create_server()*).

- *server_hostname*: sets or overrides the host name that the target server's certificate will be matched against.
- *ssl_handshake_timeout* is (for a TLS connection) the time in seconds to wait for the TLS handshake to complete before aborting the connection. `60.0` seconds if `None` (default).
- *ssl_shutdown_timeout* is the time in seconds to wait for the SSL shutdown to complete before aborting the connection. `30.0` seconds if `None` (default).

Added in version 3.7.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

Watching file descriptors

`loop.`**`add_reader`**(*fd*, *callback*, **args*)

Start monitoring the *fd* file descriptor for read availability and invoke *callback* with the specified arguments once *fd* is available for reading.

`loop.`**`remove_reader`**(*fd*)

Stop monitoring the *fd* file descriptor for read availability. Returns `True` if *fd* was previously being monitored for reads.

`loop.`**`add_writer`**(*fd*, *callback*, **args*)

Start monitoring the *fd* file descriptor for write availability and invoke *callback* with the specified arguments once *fd* is available for writing.

Use *`functools.partial()` to pass keyword arguments* to *callback*.

`loop.`**`remove_writer`**(*fd*)

Stop monitoring the *fd* file descriptor for write availability. Returns `True` if *fd* was previously being monitored for writes.

See also *Platform Support* section for some limitations of these methods.

Working with socket objects directly

In general, protocol implementations that use transport-based APIs such as *`loop.create_connection()`* and *`loop.create_server()`* are faster than implementations that work with sockets directly. However, there are some use cases when performance is not critical, and working with *`socket`* objects directly is more convenient.

`coroutine` `loop.`**`sock_recv`**(*sock*, *nbytes*)

Receive up to *nbytes* from *sock*. Asynchronous version of *`socket.recv()`*.

Return the received data as a bytes object.

sock must be a non-blocking socket.

Changed in version 3.7: Even though this method was always documented as a coroutine method, releases before Python 3.7 returned a *`Future`*. Since Python 3.7 this is an `async def` method.

`coroutine` `loop.`**`sock_recv_into`**(*sock*, *buf*)

Receive data from *sock* into the *buf* buffer. Modeled after the blocking *`socket.recv_into()`* method.

Return the number of bytes written to the buffer.

sock must be a non-blocking socket.

Added in version 3.7.

coroutine loop.**sock_recvfrom**(*sock*, *bufsize*)

Receive a datagram of up to *bufsize* from *sock*. Asynchronous version of *socket.recvfrom()*.

Return a tuple of (received data, remote address).

sock must be a non-blocking socket.

Added in version 3.11.

coroutine loop.**sock_recvfrom_into**(*sock*, *buf*, *nbytes=0*)

Receive a datagram of up to *nbytes* from *sock* into *buf*. Asynchronous version of *socket.recvfrom_into()*.

Return a tuple of (number of bytes received, remote address).

sock must be a non-blocking socket.

Added in version 3.11.

coroutine loop.**sock_sendall**(*sock*, *data*)

Send *data* to the *sock* socket. Asynchronous version of *socket.sendall()*.

This method continues to send to the socket until either all data in *data* has been sent or an error occurs. None is returned on success. On error, an exception is raised. Additionally, there is no way to determine how much data, if any, was successfully processed by the receiving end of the connection.

sock must be a non-blocking socket.

Changed in version 3.7: Even though the method was always documented as a coroutine method, before Python 3.7 it returned a *Future*. Since Python 3.7, this is an async def method.

coroutine loop.**sock_sendto**(*sock*, *data*, *address*)

Send a datagram from *sock* to *address*. Asynchronous version of *socket.sendto()*.

Return the number of bytes sent.

sock must be a non-blocking socket.

Added in version 3.11.

coroutine loop.**sock_connect**(*sock*, *address*)

Connect *sock* to a remote socket at *address*.

Asynchronous version of *socket.connect()*.

sock must be a non-blocking socket.

Changed in version 3.5.2: address no longer needs to be resolved. sock_connect will try to check if the *address* is already resolved by calling *socket.inet_pton()*. If not, *loop.getaddrinfo()* will be used to resolve the *address*.

> **See also**
>
> *loop.create_connection()* and *asyncio.open_connection()*.

coroutine loop.**sock_accept**(*sock*)

Accept a connection. Modeled after the blocking *socket.accept()* method.

The socket must be bound to an address and listening for connections. The return value is a pair (conn, address) where *conn* is a *new* socket object usable to send and receive data on the connection, and *address* is the address bound to the socket on the other end of the connection.

sock must be a non-blocking socket.

Changed in version 3.7: Even though the method was always documented as a coroutine method, before Python 3.7 it returned a *Future*. Since Python 3.7, this is an `async def` method.

See also

loop.create_server() and *start_server()*.

coroutine loop.**sock_sendfile**(*sock*, *file*, *offset=0*, *count=None*, *, *fallback=True*)

Send a file using high-performance *os.sendfile* if possible. Return the total number of bytes sent.

Asynchronous version of *socket.sendfile()*.

sock must be a non-blocking *socket.SOCK_STREAM* *socket*.

file must be a regular file object open in binary mode.

offset tells from where to start reading the file. If specified, *count* is the total number of bytes to transmit as opposed to sending the file until EOF is reached. File position is always updated, even when this method raises an error, and *file.tell()* can be used to obtain the actual number of bytes sent.

fallback, when set to `True`, makes asyncio manually read and send the file when the platform does not support the sendfile syscall (e.g. Windows or SSL socket on Unix).

Raise *SendfileNotAvailableError* if the system does not support *sendfile* syscall and *fallback* is `False`.

sock must be a non-blocking socket.

Added in version 3.7.

DNS

coroutine loop.**getaddrinfo**(*host*, *port*, *, *family=0*, *type=0*, *proto=0*, *flags=0*)

Asynchronous version of *socket.getaddrinfo()*.

coroutine loop.**getnameinfo**(*sockaddr*, *flags=0*)

Asynchronous version of *socket.getnameinfo()*.

Note

Both *getaddrinfo* and *getnameinfo* internally utilize their synchronous versions through the loop's default thread pool executor. When this executor is saturated, these methods may experience delays, which higher-level networking libraries may report as increased timeouts. To mitigate this, consider using a custom executor for other user tasks, or setting a default executor with a larger number of workers.

Changed in version 3.7: Both *getaddrinfo* and *getnameinfo* methods were always documented to return a coroutine, but prior to Python 3.7 they were, in fact, returning *asyncio.Future* objects. Starting with Python 3.7 both methods are coroutines.

Working with pipes

coroutine loop.**connect_read_pipe**(*protocol_factory*, *pipe*)

Register the read end of *pipe* in the event loop.

protocol_factory must be a callable returning an *asyncio protocol* implementation.

pipe is a *file-like object*.

Return pair `(transport, protocol)`, where *transport* supports the `ReadTransport` interface and *protocol* is an object instantiated by the *protocol_factory*.

With `SelectorEventLoop` event loop, the *pipe* is set to non-blocking mode.

coroutine loop.**connect_write_pipe**(*protocol_factory*, *pipe*)

Register the write end of *pipe* in the event loop.

protocol_factory must be a callable returning an *asyncio protocol* implementation.

pipe is *file-like object*.

Return pair `(transport, protocol)`, where *transport* supports `WriteTransport` interface and *protocol* is an object instantiated by the *protocol_factory*.

With `SelectorEventLoop` event loop, the *pipe* is set to non-blocking mode.

Note

`SelectorEventLoop` does not support the above methods on Windows. Use `ProactorEventLoop` instead for Windows.

See also

The `loop.subprocess_exec()` and `loop.subprocess_shell()` methods.

Unix signals

loop.**add_signal_handler**(*signum*, *callback*, **args*)

Set *callback* as the handler for the *signum* signal.

The callback will be invoked by *loop*, along with other queued callbacks and runnable coroutines of that event loop. Unlike signal handlers registered using `signal.signal()`, a callback registered with this function is allowed to interact with the event loop.

Raise `ValueError` if the signal number is invalid or uncatchable. Raise `RuntimeError` if there is a problem setting up the handler.

Use `functools.partial()` *to pass keyword arguments* to *callback*.

Like `signal.signal()`, this function must be invoked in the main thread.

loop.**remove_signal_handler**(*sig*)

Remove the handler for the *sig* signal.

Return `True` if the signal handler was removed, or `False` if no handler was set for the given signal.

Availability: Unix.

See also

The *signal* module.

Executing code in thread or process pools

awaitable loop.**run_in_executor**(*executor*, *func*, **args*)

Arrange for *func* to be called in the specified executor.

The *executor* argument should be an *concurrent.futures.Executor* instance. The default executor is used if *executor* is None. The default executor can be set by *loop.set_default_executor()*, otherwise, a *concurrent.futures.ThreadPoolExecutor* will be lazy-initialized and used by *run_in_executor()* if needed.

Example:

```
import asyncio
import concurrent.futures

def blocking_io():
    # File operations (such as logging) can block the
    # event loop: run them in a thread pool.
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

def cpu_bound():
    # CPU-bound operations will block the event loop:
    # in general it is preferable to run them in a
    # process pool.
    return sum(i * i for i in range(10 ** 7))

async def main():
    loop = asyncio.get_running_loop()

    ## Options:

    # 1. Run in the default loop's executor:
    result = await loop.run_in_executor(
        None, blocking_io)
    print('default thread pool', result)

    # 2. Run in a custom thread pool:
    with concurrent.futures.ThreadPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, blocking_io)
        print('custom thread pool', result)

    # 3. Run in a custom process pool:
    with concurrent.futures.ProcessPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom process pool', result)

if __name__ == '__main__':
    asyncio.run(main())
```

Note that the entry point guard (`if __name__ == '__main__'`) is required for option 3 due to the peculiarities of *multiprocessing*, which is used by *ProcessPoolExecutor*. See *Safe importing of main module*.

This method returns a *asyncio.Future* object.

Use *functools.partial() to pass keyword arguments* to *func*.

Changed in version 3.5.3: *loop.run_in_executor()* no longer configures the `max_workers` of the thread pool executor it creates, instead leaving it up to the thread pool executor (*ThreadPoolExecutor*) to set the default.

`loop.`**`set_default_executor`**(*executor*)

Set *executor* as the default executor used by *run_in_executor()*. *executor* must be an instance of *ThreadPoolExecutor*.

Changed in version 3.11: *executor* must be an instance of *ThreadPoolExecutor*.

Error Handling API

Allows customizing how exceptions are handled in the event loop.

`loop.`**`set_exception_handler`**(*handler*)

Set *handler* as the new event loop exception handler.

If *handler* is `None`, the default exception handler will be set. Otherwise, *handler* must be a callable with the signature matching `(loop, context)`, where `loop` is a reference to the active event loop, and `context` is a `dict` object containing the details of the exception (see *call_exception_handler()* documentation for details about context).

If the handler is called on behalf of a *Task* or *Handle*, it is run in the *contextvars.Context* of that task or callback handle.

Changed in version 3.12: The handler may be called in the *Context* of the task or handle where the exception originated.

`loop.`**`get_exception_handler`**()

Return the current exception handler, or `None` if no custom exception handler was set.

Added in version 3.5.2.

`loop.`**`default_exception_handler`**(*context*)

Default exception handler.

This is called when an exception occurs and no exception handler is set. This can be called by a custom exception handler that wants to defer to the default handler behavior.

context parameter has the same meaning as in *call_exception_handler()*.

`loop.`**`call_exception_handler`**(*context*)

Call the current event loop exception handler.

context is a `dict` object containing the following keys (new keys may be introduced in future Python versions):

- 'message': Error message;
- 'exception' (optional): Exception object;
- 'future' (optional): *asyncio.Future* instance;
- 'task' (optional): *asyncio.Task* instance;

- 'handle' (optional): *asyncio.Handle* instance;
- 'protocol' (optional): *Protocol* instance;
- 'transport' (optional): *Transport* instance;
- 'socket' (optional): *socket.socket* instance;
- **'asyncgen' (optional): Asynchronous generator that caused** the exception.

> **Note**
>
> This method should not be overloaded in subclassed event loops. For custom exception handling, use the *set_exception_handler()* method.

Enabling debug mode

`loop.`**`get_debug`**`()`

Get the debug mode (*bool*) of the event loop.

The default value is `True` if the environment variable `PYTHONASYNCIODEBUG` is set to a non-empty string, `False` otherwise.

`loop.`**`set_debug`**`(`*enabled:* bool`)`

Set the debug mode of the event loop.

Changed in version 3.7: The new *Python Development Mode* can now also be used to enable the debug mode.

`loop.`**`slow_callback_duration`**

This attribute can be used to set the minimum execution duration in seconds that is considered "slow". When debug mode is enabled, "slow" callbacks are logged.

Default value is 100 milliseconds.

> **See also**
>
> The *debug mode of asyncio*.

Running Subprocesses

Methods described in this subsections are low-level. In regular async/await code consider using the high-level *asyncio.create_subprocess_shell()* and *asyncio.create_subprocess_exec()* convenience functions instead.

> **Note**
>
> On Windows, the default event loop *ProactorEventLoop* supports subprocesses, whereas *SelectorEventLoop* does not. See *Subprocess Support on Windows* for details.

coroutine loop.**subprocess_exec**(*protocol_factory*, **args*, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*)

Create a subprocess from one or more string arguments specified by *args*.

args must be a list of strings represented by:

- `str`;
- or `bytes`, encoded to the *filesystem encoding*.

The first string specifies the program executable, and the remaining strings specify the arguments. Together, string arguments form the `argv` of the program.

This is similar to the standard library `subprocess.Popen` class called with `shell=False` and the list of strings passed as the first argument; however, where `Popen` takes a single argument which is list of strings, *subprocess_exec* takes multiple string arguments.

The *protocol_factory* must be a callable returning a subclass of the `asyncio.SubprocessProtocol` class.

Other parameters:

- *stdin* can be any of these:
  - a file-like object
  - an existing file descriptor (a positive integer), for example those created with `os.pipe()`
  - the `subprocess.PIPE` constant (default) which will create a new pipe and connect it,
  - the value `None` which will make the subprocess inherit the file descriptor from this process
  - the `subprocess.DEVNULL` constant which indicates that the special `os.devnull` file will be used
- *stdout* can be any of these:
  - a file-like object
  - the `subprocess.PIPE` constant (default) which will create a new pipe and connect it,
  - the value `None` which will make the subprocess inherit the file descriptor from this process
  - the `subprocess.DEVNULL` constant which indicates that the special `os.devnull` file will be used
- *stderr* can be any of these:
  - a file-like object
  - the `subprocess.PIPE` constant (default) which will create a new pipe and connect it,
  - the value `None` which will make the subprocess inherit the file descriptor from this process
  - the `subprocess.DEVNULL` constant which indicates that the special `os.devnull` file will be used
  - the `subprocess.STDOUT` constant which will connect the standard error stream to the process' standard output stream
- All other keyword arguments are passed to `subprocess.Popen` without interpretation, except for *bufsize*, *universal_newlines*, *shell*, *text*, *encoding* and *errors*, which should not be specified at all.

  The `asyncio` subprocess API does not support decoding the streams as text. `bytes.decode()` can be used to convert the bytes returned from the stream to text.

If a file-like object passed as *stdin*, *stdout* or *stderr* represents a pipe, then the other side of this pipe should be registered with `connect_write_pipe()` or `connect_read_pipe()` for use with the event loop.

See the constructor of the `subprocess.Popen` class for documentation on other arguments.

Returns a pair of `(transport, protocol)`, where *transport* conforms to the *asyncio.SubprocessTransport* base class and *protocol* is an object instantiated by the *protocol_factory*.

coroutine loop.**subprocess_shell**(*protocol_factory*, *cmd*, *, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*)

Create a subprocess from *cmd*, which can be a *str* or a *bytes* string encoded to the *filesystem encoding*, using the platform's "shell" syntax.

This is similar to the standard library *subprocess.Popen* class called with `shell=True`.

The *protocol_factory* must be a callable returning a subclass of the *SubprocessProtocol* class.

See *subprocess_exec()* for more details about the remaining arguments.

Returns a pair of `(transport, protocol)`, where *transport* conforms to the *SubprocessTransport* base class and *protocol* is an object instantiated by the *protocol_factory*.

Note

It is the application's responsibility to ensure that all whitespace and special characters are quoted appropriately to avoid shell injection vulnerabilities. The *shlex.quote()* function can be used to properly escape whitespace and special characters in strings that are going to be used to construct shell commands.

Callback Handles

class asyncio.**Handle**

A callback wrapper object returned by *loop.call_soon()*, *loop.call_soon_threadsafe()*.

get_context()

Return the *contextvars.Context* object associated with the handle.

Added in version 3.12.

cancel()

Cancel the callback. If the callback has already been canceled or executed, this method has no effect.

cancelled()

Return `True` if the callback was cancelled.

Added in version 3.7.

class asyncio.**TimerHandle**

A callback wrapper object returned by *loop.call_later()*, and *loop.call_at()*.

This class is a subclass of *Handle*.

when()

Return a scheduled callback time as *float* seconds.

The time is an absolute timestamp, using the same time reference as *loop.time()*.

Added in version 3.7.

Server Objects

Server objects are created by *loop.create_server()*, *loop.create_unix_server()*, *start_server()*, and *start_unix_server()* functions.

Do not instantiate the *Server* class directly.

class asyncio.**Server**

Server objects are asynchronous context managers. When used in an `async with` statement, it's guaranteed that the Server object is closed and not accepting new connections when the `async with` statement is completed:

```
srv = await loop.create_server(...)

async with srv:
    # some code

# At this point, srv is closed and no longer accepts new connections.
```

Changed in version 3.7: Server object is an asynchronous context manager since Python 3.7.

Changed in version 3.11: This class was exposed publicly as `asyncio.Server` in Python 3.9.11, 3.10.3 and 3.11.

close()

Stop serving: close listening sockets and set the *sockets* attribute to `None`.

The sockets that represent existing incoming client connections are left open.

The server is closed asynchronously; use the *wait_closed()* coroutine to wait until the server is closed (and no more connections are active).

close_clients()

Close all existing incoming client connections.

Calls *close()* on all associated transports.

close() should be called before *close_clients()* when closing the server to avoid races with new clients connecting.

Added in version 3.13.

abort_clients()

Close all existing incoming client connections immediately, without waiting for pending operations to complete.

Calls *abort()* on all associated transports.

close() should be called before *abort_clients()* when closing the server to avoid races with new clients connecting.

Added in version 3.13.

get_loop()

Return the event loop associated with the server object.

Added in version 3.7.

coroutine start_serving()

Start accepting connections.

This method is idempotent, so it can be called when the server is already serving.

The *start_serving* keyword-only parameter to *loop.create_server()* and *asyncio.start_server()* allows creating a Server object that is not accepting connections initially. In this case `Server.start_serving()`, or *Server.serve_forever()* can be used to make the Server start accepting connections.

Added in version 3.7.

coroutine serve_forever()

Start accepting connections until the coroutine is cancelled. Cancellation of `serve_forever` task causes the server to be closed.

This method can be called if the server is already accepting connections. Only one `serve_forever` task can exist per one *Server* object.

Example:

```
async def client_connected(reader, writer):
    # Communicate with the client with
    # reader/writer streams.  For example:
    await reader.readline()

async def main(host, port):
    srv = await asyncio.start_server(
        client_connected, host, port)
    await srv.serve_forever()

asyncio.run(main('127.0.0.1', 0))
```

Added in version 3.7.

is_serving()

Return `True` if the server is accepting new connections.

Added in version 3.7.

coroutine wait_closed()

Wait until the *close()* method completes and all active connections have finished.

sockets

List of socket-like objects, `asyncio.trsock.TransportSocket`, which the server is listening on.

Changed in version 3.7: Prior to Python 3.7 `Server.sockets` used to return an internal list of server sockets directly. In 3.7 a copy of that list is returned.

Event Loop Implementations

asyncio ships with two different event loop implementations: *SelectorEventLoop* and *ProactorEventLoop*.

By default asyncio is configured to use *EventLoop*.

class asyncio.**SelectorEventLoop**

A subclass of *AbstractEventLoop* based on the *selectors* module.

Uses the most efficient *selector* available for the given platform. It is also possible to manually configure the exact selector implementation to be used:

```
import asyncio
import selectors
```

(continues on next page)

(continued from previous page)

```
class MyPolicy(asyncio.DefaultEventLoopPolicy):
   def new_event_loop(self):
      selector = selectors.SelectSelector()
      return asyncio.SelectorEventLoop(selector)

asyncio.set_event_loop_policy(MyPolicy())
```

Availability: Unix, Windows.

class asyncio.**ProactorEventLoop**

A subclass of *AbstractEventLoop* for Windows that uses "I/O Completion Ports" (IOCP).

Availability: Windows.

> **See also**
>
> MSDN documentation on I/O Completion Ports.

class asyncio.**EventLoop**

An alias to the most efficient available subclass of *AbstractEventLoop* for the given platform.

It is an alias to *SelectorEventLoop* on Unix and *ProactorEventLoop* on Windows.

Added in version 3.13.

class asyncio.**AbstractEventLoop**

Abstract base class for asyncio-compliant event loops.

The *Event Loop Methods* section lists all methods that an alternative implementation of `AbstractEventLoop` should have defined.

Examples

Note that all examples in this section **purposefully** show how to use the low-level event loop APIs, such as *loop.run_forever()* and *loop.call_soon()*. Modern asyncio applications rarely need to be written this way; consider using the high-level functions like *asyncio.run()*.

Hello World with call_soon()

An example using the *loop.call_soon()* method to schedule a callback. The callback displays `"Hello World"` and then stops the event loop:

```
import asyncio

def hello_world(loop):
    """A callback to print 'Hello World' and stop the event loop"""
    print('Hello World')
    loop.stop()

loop = asyncio.new_event_loop()

# Schedule a call to hello_world()
```

(continues on next page)

(continued from previous page)

```
loop.call_soon(hello_world, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

See also

A similar *Hello World* example created with a coroutine and the `run()` function.

Display the current date with call_later()

An example of a callback displaying the current date every second. The callback uses the `loop.call_later()` method to reschedule itself after 5 seconds, and then stops the event loop:

```
import asyncio
import datetime

def display_date(end_time, loop):
    print(datetime.datetime.now())
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, display_date, end_time, loop)
    else:
        loop.stop()

loop = asyncio.new_event_loop()

# Schedule the first call to display_date()
end_time = loop.time() + 5.0
loop.call_soon(display_date, end_time, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

See also

A similar *current date* example created with a coroutine and the `run()` function.

Watch a file descriptor for read events

Wait until a file descriptor received some data using the *loop.add_reader()* method and then close the event loop:

```
import asyncio
from socket import socketpair

# Create a pair of connected file descriptors
rsock, wsock = socketpair()

loop = asyncio.new_event_loop()

def reader():
    data = rsock.recv(100)
    print("Received:", data.decode())

    # We are done: unregister the file descriptor
    loop.remove_reader(rsock)

    # Stop the event loop
    loop.stop()

# Register the file descriptor for read event
loop.add_reader(rsock, reader)

# Simulate the reception of data from the network
loop.call_soon(wsock.send, 'abc'.encode())

try:
    # Run the event loop
    loop.run_forever()
finally:
    # We are done. Close sockets and the event loop.
    rsock.close()
    wsock.close()
    loop.close()
```

See also

- A similar *example* using transports, protocols, and the *loop.create_connection()* method.
- Another similar *example* using the high-level *asyncio.open_connection()* function and streams.

Set signal handlers for SIGINT and SIGTERM

(This `signals` example only works on Unix.)

Register handlers for signals *SIGINT* and *SIGTERM* using the *loop.add_signal_handler()* method:

```
import asyncio
import functools
import os
import signal

def ask_exit(signame, loop):
```

(continues on next page)

(continued from previous page)

```
    print("got signal %s: exit" % signame)
    loop.stop()

async def main():
    loop = asyncio.get_running_loop()

    for signame in {'SIGINT', 'SIGTERM'}:
        loop.add_signal_handler(
            getattr(signal, signame),
            functools.partial(ask_exit, signame, loop))

    await asyncio.sleep(3600)

print("Event loop running for 1 hour, press Ctrl+C to interrupt.")
print(f"pid {os.getpid()}: send SIGINT or SIGTERM to exit.")

asyncio.run(main())
```

18.1.9 Futures

Source code: Lib/asyncio/futures.py, Lib/asyncio/base_futures.py

---

Future objects are used to bridge **low-level callback-based code** with high-level async/await code.

Future Functions

`asyncio.`**`isfuture`**`(`*obj*`)`

Return `True` if *obj* is either of:

- an instance of *`asyncio.Future`*,
- an instance of *`asyncio.Task`*,
- a Future-like object with a `_asyncio_future_blocking` attribute.

Added in version 3.5.

`asyncio.`**`ensure_future`**`(`*obj*, *, *loop=None*`)`

Return:

- *obj* argument as is, if *obj* is a *`Future`*, a *`Task`*, or a Future-like object (*`isfuture()`* is used for the test.)
- a *`Task`* object wrapping *obj*, if *obj* is a coroutine (*`iscoroutine()`* is used for the test); in this case the coroutine will be scheduled by `ensure_future()`.
- a *`Task`* object that would await on *obj*, if *obj* is an awaitable (*`inspect.isawaitable()`* is used for the test.)

If *obj* is neither of the above a *`TypeError`* is raised.

> **Important**
>
> See also the *`create_task()`* function which is the preferred way for creating new Tasks.
>
> Save a reference to the result of this function, to avoid a task disappearing mid-execution.

Changed in version 3.5.1: The function accepts any *awaitable* object.

Deprecated since version 3.10: Deprecation warning is emitted if *obj* is not a Future-like object and *loop* is not specified and there is no running event loop.

asyncio.**wrap_future**(*future*, *, *loop=None*)

Wrap a `concurrent.futures.Future` object in a `asyncio.Future` object.

Deprecated since version 3.10: Deprecation warning is emitted if *future* is not a Future-like object and *loop* is not specified and there is no running event loop.

Future Object

class asyncio.**Future**(*, *loop=None*)

A Future represents an eventual result of an asynchronous operation. Not thread-safe.

Future is an *awaitable* object. Coroutines can await on Future objects until they either have a result or an exception set, or until they are cancelled. A Future can be awaited multiple times and the result is same.

Typically Futures are used to enable low-level callback-based code (e.g. in protocols implemented using asyncio *transports*) to interoperate with high-level async/await code.

The rule of thumb is to never expose Future objects in user-facing APIs, and the recommended way to create a Future object is to call `loop.create_future()`. This way alternative event loop implementations can inject their own optimized implementations of a Future object.

Changed in version 3.7: Added support for the `contextvars` module.

Deprecated since version 3.10: Deprecation warning is emitted if *loop* is not specified and there is no running event loop.

result()

Return the result of the Future.

If the Future is *done* and has a result set by the `set_result()` method, the result value is returned.

If the Future is *done* and has an exception set by the `set_exception()` method, this method raises the exception.

If the Future has been *cancelled*, this method raises a `CancelledError` exception.

If the Future's result isn't yet available, this method raises an `InvalidStateError` exception.

set_result(*result*)

Mark the Future as *done* and set its result.

Raises an `InvalidStateError` error if the Future is already *done*.

set_exception(*exception*)

Mark the Future as *done* and set an exception.

Raises an `InvalidStateError` error if the Future is already *done*.

done()

Return `True` if the Future is *done*.

A Future is *done* if it was *cancelled* or if it has a result or an exception set with `set_result()` or `set_exception()` calls.

cancelled()

Return `True` if the Future was *cancelled*.

The method is usually used to check if a Future is not *cancelled* before setting a result or an exception for it:

```
if not fut.cancelled():
    fut.set_result(42)
```

add_done_callback(*callback*, *, *context=None*)

Add a callback to be run when the Future is *done*.

The *callback* is called with the Future object as its only argument.

If the Future is already *done* when this method is called, the callback is scheduled with *loop.call_soon()*.

An optional keyword-only *context* argument allows specifying a custom *contextvars.Context* for the *callback* to run in. The current context is used when no *context* is provided.

functools.partial() can be used to pass parameters to the callback, e.g.:

```
# Call 'print("Future:", fut)' when "fut" is done.
fut.add_done_callback(
    functools.partial(print, "Future:"))
```

Changed in version 3.7: The *context* keyword-only parameter was added. See **PEP 567** for more details.

remove_done_callback(*callback*)

Remove *callback* from the callbacks list.

Returns the number of callbacks removed, which is typically 1, unless a callback was added more than once.

cancel(*msg=None*)

Cancel the Future and schedule callbacks.

If the Future is already *done* or *cancelled*, return `False`. Otherwise, change the Future's state to *cancelled*, schedule the callbacks, and return `True`.

Changed in version 3.9: Added the *msg* parameter.

exception()

Return the exception that was set on this Future.

The exception (or `None` if no exception was set) is returned only if the Future is *done*.

If the Future has been *cancelled*, this method raises a *CancelledError* exception.

If the Future isn't *done* yet, this method raises an *InvalidStateError* exception.

get_loop()

Return the event loop the Future object is bound to.

Added in version 3.7.

This example creates a Future object, creates and schedules an asynchronous Task to set result for the Future, and waits until the Future has a result:

```
async def set_after(fut, delay, value):
    # Sleep for *delay* seconds.
    await asyncio.sleep(delay)

    # Set *value* as a result of *fut* Future.
```

(continues on next page)

(continued from previous page)

```
    fut.set_result(value)

async def main():
    # Get the current event loop.
    loop = asyncio.get_running_loop()

    # Create a new Future object.
    fut = loop.create_future()

    # Run "set_after()" coroutine in a parallel Task.
    # We are using the low-level "loop.create_task()" API here because
    # we already have a reference to the event loop at hand.
    # Otherwise we could have just used "asyncio.create_task()".
    loop.create_task(
        set_after(fut, 1, '... world'))

    print('hello ...')

    # Wait until *fut* has a result (1 second) and print it.
    print(await fut)

asyncio.run(main())
```

Important

The Future object was designed to mimic *concurrent.futures.Future*. Key differences include:

- unlike asyncio Futures, *concurrent.futures.Future* instances cannot be awaited.
- *asyncio.Future.result()* and *asyncio.Future.exception()* do not accept the *timeout* argument.
- *asyncio.Future.result()* and *asyncio.Future.exception()* raise an *InvalidStateError* exception when the Future is not *done*.
- Callbacks registered with *asyncio.Future.add_done_callback()* are not called immediately. They are scheduled with *loop.call_soon()* instead.
- asyncio Future is not compatible with the *concurrent.futures.wait()* and *concurrent.futures.as_completed()* functions.
- *asyncio.Future.cancel()* accepts an optional `msg` argument, but *concurrent.futures.Future.cancel()* does not.

18.1.10 Transports and Protocols

Preface

Transports and Protocols are used by the **low-level** event loop APIs such as *loop.create_connection()*. They use callback-based programming style and enable high-performance implementations of network or IPC protocols (e.g. HTTP).

Essentially, transports and protocols should only be used in libraries and frameworks and never in high-level asyncio applications.

This documentation page covers both *Transports* and *Protocols*.

Introduction

At the highest level, the transport is concerned with *how* bytes are transmitted, while the protocol determines *which* bytes to transmit (and to some extent when).

A different way of saying the same thing: a transport is an abstraction for a socket (or similar I/O endpoint) while a protocol is an abstraction for an application, from the transport's point of view.

Yet another view is the transport and protocol interfaces together define an abstract interface for using network I/O and interprocess I/O.

There is always a 1:1 relationship between transport and protocol objects: the protocol calls transport methods to send data, while the transport calls protocol methods to pass it data that has been received.

Most of connection oriented event loop methods (such as `loop.create_connection()`) usually accept a *protocol_factory* argument used to create a *Protocol* object for an accepted connection, represented by a *Transport* object. Such methods usually return a tuple of `(transport, protocol)`.

Contents

This documentation page contains the following sections:

- The *Transports* section documents asyncio `BaseTransport`, `ReadTransport`, `WriteTransport`, `Transport`, `DatagramTransport`, and `SubprocessTransport` classes.
- The *Protocols* section documents asyncio `BaseProtocol`, `Protocol`, `BufferedProtocol`, `DatagramProtocol`, and `SubprocessProtocol` classes.
- The *Examples* section showcases how to work with transports, protocols, and low-level event loop APIs.

Transports

Source code: Lib/asyncio/transports.py

Transports are classes provided by `asyncio` in order to abstract various kinds of communication channels.

Transport objects are always instantiated by an *asyncio event loop*.

asyncio implements transports for TCP, UDP, SSL, and subprocess pipes. The methods available on a transport depend on the transport's kind.

The transport classes are *not thread safe*.

Transports Hierarchy

class asyncio.**BaseTransport**

Base class for all transports. Contains methods that all asyncio transports share.

class asyncio.**WriteTransport**(*BaseTransport*)

A base transport for write-only connections.

Instances of the *WriteTransport* class are returned from the `loop.connect_write_pipe()` event loop method and are also used by subprocess-related methods like `loop.subprocess_exec()`.

class asyncio.**ReadTransport**(*BaseTransport*)

A base transport for read-only connections.

Instances of the *ReadTransport* class are returned from the *loop.connect_read_pipe()* event loop method and are also used by subprocess-related methods like *loop.subprocess_exec()*.

class asyncio.**Transport**(*WriteTransport*, *ReadTransport*)

Interface representing a bidirectional transport, such as a TCP connection.

The user does not instantiate a transport directly; they call a utility function, passing it a protocol factory and other information necessary to create the transport and protocol.

Instances of the *Transport* class are returned from or used by event loop methods like *loop.create_connection()*, *loop.create_unix_connection()*, *loop.create_server()*, *loop.sendfile()*, etc.

class asyncio.**DatagramTransport**(*BaseTransport*)

A transport for datagram (UDP) connections.

Instances of the *DatagramTransport* class are returned from the *loop.create_datagram_endpoint()* event loop method.

class asyncio.**SubprocessTransport**(*BaseTransport*)

An abstraction to represent a connection between a parent and its child OS process.

Instances of the *SubprocessTransport* class are returned from event loop methods *loop.subprocess_shell()* and *loop.subprocess_exec()*.

Base Transport

BaseTransport.**close**()

Close the transport.

If the transport has a buffer for outgoing data, buffered data will be flushed asynchronously. No more data will be received. After all buffered data is flushed, the protocol's *protocol.connection_lost()* method will be called with *None* as its argument. The transport should not be used once it is closed.

BaseTransport.**is_closing**()

Return `True` if the transport is closing or is closed.

BaseTransport.**get_extra_info**(*name*, *default=None*)

Return information about the transport or underlying resources it uses.

name is a string representing the piece of transport-specific information to get.

default is the value to return if the information is not available, or if the transport does not support querying it with the given third-party event loop implementation or on the current platform.

For example, the following code attempts to get the underlying socket object of the transport:

```
sock = transport.get_extra_info('socket')
if sock is not None:
    print(sock.getsockopt(...))
```

Categories of information that can be queried on some transports:

- socket:
  - `'peername'`: the remote address to which the socket is connected, result of *socket.socket.getpeername()* (`None` on error)

  - `'socket'`: *socket.socket* instance
  - `'sockname'`: the socket's own address, result of *socket.socket.getsockname()*
- SSL socket:
  - `'compression'`: the compression algorithm being used as a string, or `None` if the connection isn't compressed; result of *ssl.SSLSocket.compression()*
  - `'cipher'`: a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used; result of *ssl.SSLSocket.cipher()*
  - `'peercert'`: peer certificate; result of *ssl.SSLSocket.getpeercert()*
  - `'sslcontext'`: *ssl.SSLContext* instance
  - `'ssl_object'`: *ssl.SSLObject* or *ssl.SSLSocket* instance
- pipe:
  - `'pipe'`: pipe object
- subprocess:
  - `'subprocess'`: *subprocess.Popen* instance

`BaseTransport.`**`set_protocol`**(*protocol*)

Set a new protocol.

Switching protocol should only be done when both protocols are documented to support the switch.

`BaseTransport.`**`get_protocol`**()

Return the current protocol.

Read-only Transports

`ReadTransport.`**`is_reading`**()

Return `True` if the transport is receiving new data.

Added in version 3.7.

`ReadTransport.`**`pause_reading`**()

Pause the receiving end of the transport. No data will be passed to the protocol's *protocol.data_received()* method until *resume_reading()* is called.

Changed in version 3.7: The method is idempotent, i.e. it can be called when the transport is already paused or closed.

`ReadTransport.`**`resume_reading`**()

Resume the receiving end. The protocol's *protocol.data_received()* method will be called once again if some data is available for reading.

Changed in version 3.7: The method is idempotent, i.e. it can be called when the transport is already reading.

Write-only Transports

WriteTransport.**abort**()

Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol's *protocol.connection_lost()* method will eventually be called with *None* as its argument.

WriteTransport.**can_write_eof**()

Return *True* if the transport supports *write_eof()*, *False* if not.

WriteTransport.**get_write_buffer_size**()

Return the current size of the output buffer used by the transport.

WriteTransport.**get_write_buffer_limits**()

Get the *high* and *low* watermarks for write flow control. Return a tuple `(low, high)` where *low* and *high* are positive number of bytes.

Use *set_write_buffer_limits()* to set the limits.

Added in version 3.4.2.

WriteTransport.**set_write_buffer_limits**(*high=None*, *low=None*)

Set the *high* and *low* watermarks for write flow control.

These two values (measured in number of bytes) control when the protocol's *protocol.pause_writing()* and *protocol.resume_writing()* methods are called. If specified, the low watermark must be less than or equal to the high watermark. Neither *high* nor *low* can be negative.

pause_writing() is called when the buffer size becomes greater than or equal to the *high* value. If writing has been paused, *resume_writing()* is called when the buffer size becomes less than or equal to the *low* value.

The defaults are implementation-specific. If only the high watermark is given, the low watermark defaults to an implementation-specific value less than or equal to the high watermark. Setting *high* to zero forces *low* to zero as well, and causes *pause_writing()* to be called whenever the buffer becomes non-empty. Setting *low* to zero causes *resume_writing()* to be called only once the buffer is empty. Use of zero for either limit is generally sub-optimal as it reduces opportunities for doing I/O and computation concurrently.

Use *get_write_buffer_limits()* to get the limits.

WriteTransport.**write**(*data*)

Write some *data* bytes to the transport.

This method does not block; it buffers the data and arranges for it to be sent out asynchronously.

WriteTransport.**writelines**(*list_of_data*)

Write a list (or any iterable) of data bytes to the transport. This is functionally equivalent to calling *write()* on each element yielded by the iterable, but may be implemented more efficiently.

WriteTransport.**write_eof**()

Close the write end of the transport after flushing all buffered data. Data may still be received.

This method can raise *NotImplementedError* if the transport (e.g. SSL) doesn't support half-closed connections.

Datagram Transports

DatagramTransport.**sendto**(*data*, *addr=None*)

Send the *data* bytes to the remote peer given by *addr* (a transport-dependent target address). If *addr* is `None`, the data is sent to the target address given on transport creation.

This method does not block; it buffers the data and arranges for it to be sent out asynchronously.

Changed in version 3.13: This method can be called with an empty bytes object to send a zero-length datagram. The buffer size calculation used for flow control is also updated to account for the datagram header.

DatagramTransport.**abort**()

Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol's *protocol.connection_lost()* method will eventually be called with `None` as its argument.

Subprocess Transports

SubprocessTransport.**get_pid**()

Return the subprocess process id as an integer.

SubprocessTransport.**get_pipe_transport**(*fd*)

Return the transport for the communication pipe corresponding to the integer file descriptor *fd*:

- `0`: readable streaming transport of the standard input (*stdin*), or `None` if the subprocess was not created with `stdin=PIPE`
- `1`: writable streaming transport of the standard output (*stdout*), or `None` if the subprocess was not created with `stdout=PIPE`
- `2`: writable streaming transport of the standard error (*stderr*), or `None` if the subprocess was not created with `stderr=PIPE`
- other *fd*: `None`

SubprocessTransport.**get_returncode**()

Return the subprocess return code as an integer or `None` if it hasn't returned, which is similar to the *subprocess.Popen.returncode* attribute.

SubprocessTransport.**kill**()

Kill the subprocess.

On POSIX systems, the function sends SIGKILL to the subprocess. On Windows, this method is an alias for *terminate()*.

See also *subprocess.Popen.kill()*.

SubprocessTransport.**send_signal**(*signal*)

Send the *signal* number to the subprocess, as in *subprocess.Popen.send_signal()*.

SubprocessTransport.**terminate**()

Stop the subprocess.

On POSIX systems, this method sends *SIGTERM* to the subprocess. On Windows, the Windows API function `TerminateProcess()` is called to stop the subprocess.

See also *subprocess.Popen.terminate()*.

SubprocessTransport.**close**()

Kill the subprocess by calling the *kill()* method.

If the subprocess hasn't returned yet, and close transports of *stdin*, *stdout*, and *stderr* pipes.

Protocols

Source code: Lib/asyncio/protocols.py

---

asyncio provides a set of abstract base classes that should be used to implement network protocols. Those classes are meant to be used together with *transports*.

Subclasses of abstract base protocol classes may implement some or all methods. All these methods are callbacks: they are called by transports on certain events, for example when some data is received. A base protocol method should be called by the corresponding transport.

Base Protocols

class asyncio.**BaseProtocol**

Base protocol with methods that all protocols share.

class asyncio.**Protocol**(*BaseProtocol*)

The base class for implementing streaming protocols (TCP, Unix sockets, etc).

class asyncio.**BufferedProtocol**(*BaseProtocol*)

A base class for implementing streaming protocols with manual control of the receive buffer.

class asyncio.**DatagramProtocol**(*BaseProtocol*)

The base class for implementing datagram (UDP) protocols.

class asyncio.**SubprocessProtocol**(*BaseProtocol*)

The base class for implementing protocols communicating with child processes (unidirectional pipes).

Base Protocol

All asyncio protocols can implement Base Protocol callbacks.

Connection Callbacks

Connection callbacks are called on all protocols, exactly once per a successful connection. All other protocol callbacks can only be called between those two methods.

BaseProtocol.**connection_made**(*transport*)

Called when a connection is made.

The *transport* argument is the transport representing the connection. The protocol is responsible for storing the reference to its transport.

BaseProtocol.**connection_lost**(*exc*)

Called when the connection is lost or closed.

The argument is either an exception object or *None*. The latter means a regular EOF is received, or the connection was aborted or closed by this side of the connection.

Flow Control Callbacks

Flow control callbacks can be called by transports to pause or resume writing performed by the protocol.

See the documentation of the *set_write_buffer_limits()* method for more details.

BaseProtocol.**pause_writing**()

Called when the transport's buffer goes over the high watermark.

BaseProtocol.**resume_writing**()

Called when the transport's buffer drains below the low watermark.

If the buffer size equals the high watermark, *pause_writing()* is not called: the buffer size must go strictly over.

Conversely, *resume_writing()* is called when the buffer size is equal or lower than the low watermark. These end conditions are important to ensure that things go as expected when either mark is zero.

Streaming Protocols

Event methods, such as *loop.create_server()*, *loop.create_unix_server()*, *loop.create_connection()*, *loop.create_unix_connection()*, *loop.connect_accepted_socket()*, *loop.connect_read_pipe()*, and *loop.connect_write_pipe()* accept factories that return streaming protocols.

Protocol.**data_received**(*data*)

Called when some data is received. *data* is a non-empty bytes object containing the incoming data.

Whether the data is buffered, chunked or reassembled depends on the transport. In general, you shouldn't rely on specific semantics and instead make your parsing generic and flexible. However, data is always received in the correct order.

The method can be called an arbitrary number of times while a connection is open.

However, *protocol.eof_received()* is called at most once. Once `eof_received()` is called, `data_received()` is not called anymore.

Protocol.**eof_received**()

Called when the other end signals it won't send any more data (for example by calling *transport.write_eof()*, if the other end also uses asyncio).

This method may return a false value (including `None`), in which case the transport will close itself. Conversely, if this method returns a true value, the protocol used determines whether to close the transport. Since the default implementation returns `None`, it implicitly closes the connection.

Some transports, including SSL, don't support half-closed connections, in which case returning true from this method will result in the connection being closed.

State machine:

```
start -> connection_made
    [-> data_received]*
    [-> eof_received]?
-> connection_lost -> end
```

Buffered Streaming Protocols

Added in version 3.7.

Buffered Protocols can be used with any event loop method that supports *Streaming Protocols*.

`BufferedProtocol` implementations allow explicit manual allocation and control of the receive buffer. Event loops can then use the buffer provided by the protocol to avoid unnecessary data copies. This can result in noticeable performance improvement for protocols that receive big amounts of data. Sophisticated protocol implementations can significantly reduce the number of buffer allocations.

The following callbacks are called on *`BufferedProtocol`* instances:

`BufferedProtocol.`**`get_buffer`**(*sizehint*)

Called to allocate a new receive buffer.

sizehint is the recommended minimum size for the returned buffer. It is acceptable to return smaller or larger buffers than what *sizehint* suggests. When set to -1, the buffer size can be arbitrary. It is an error to return a buffer with a zero size.

`get_buffer()` must return an object implementing the buffer protocol.

`BufferedProtocol.`**`buffer_updated`**(*nbytes*)

Called when the buffer was updated with the received data.

nbytes is the total number of bytes that were written to the buffer.

`BufferedProtocol.`**`eof_received`**()

See the documentation of the *`protocol.eof_received()`* method.

`get_buffer()` can be called an arbitrary number of times during a connection. However, *`protocol.eof_received()`* is called at most once and, if called, *`get_buffer()`* and *`buffer_updated()`* won't be called after it.

State machine:

```
start -> connection_made
    [-> get_buffer
        [-> buffer_updated]?
    ]*
    [-> eof_received]?
-> connection_lost -> end
```

Datagram Protocols

Datagram Protocol instances should be constructed by protocol factories passed to the *`loop.create_datagram_endpoint()`* method.

`DatagramProtocol.`**`datagram_received`**(*data*, *addr*)

Called when a datagram is received. *data* is a bytes object containing the incoming data. *addr* is the address of the peer sending the data; the exact format depends on the transport.

`DatagramProtocol.`**`error_received`**(*exc*)

Called when a previous send or receive operation raises an *`OSError`*. *exc* is the *`OSError`* instance.

This method is called in rare conditions, when the transport (e.g. UDP) detects that a datagram could not be delivered to its recipient. In many conditions though, undeliverable datagrams will be silently dropped.

Note

On BSD systems (macOS, FreeBSD, etc.) flow control is not supported for datagram protocols, because there is no reliable way to detect send failures caused by writing too many packets.

The socket always appears 'ready' and excess packets are dropped. An *OSError* with `errno` set to *errno.ENOBUFS* may or may not be raised; if it is raised, it will be reported to *DatagramProtocol.error_received()* but otherwise ignored.

Subprocess Protocols

Subprocess Protocol instances should be constructed by protocol factories passed to the *loop.subprocess_exec()* and *loop.subprocess_shell()* methods.

`SubprocessProtocol.`**`pipe_data_received`**`(`*fd*, *data*`)`

Called when the child process writes data into its stdout or stderr pipe.

fd is the integer file descriptor of the pipe.

data is a non-empty bytes object containing the received data.

`SubprocessProtocol.`**`pipe_connection_lost`**`(`*fd*, *exc*`)`

Called when one of the pipes communicating with the child process is closed.

fd is the integer file descriptor that was closed.

`SubprocessProtocol.`**`process_exited`**`()`

Called when the child process has exited.

It can be called before *pipe_data_received()* and *pipe_connection_lost()* methods.

Examples

TCP Echo Server

Create a TCP echo server using the *loop.create_server()* method, send back received data, and close the connection:

```
import asyncio


class EchoServerProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info('peername')
        print('Connection from {}'.format(peername))
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
        print('Data received: {!r}'.format(message))

        print('Send: {!r}'.format(message))
        self.transport.write(data)

        print('Close the client socket')
```

(continues on next page)

(continued from previous page)

```
        self.transport.close()


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    server = await loop.create_server(
        EchoServerProtocol,
        '127.0.0.1', 8888)

    async with server:
        await server.serve_forever()


asyncio.run(main())
```

See also

The *TCP echo server using streams* example uses the high-level `asyncio.start_server()` function.

TCP Echo Client

A TCP echo client using the `loop.create_connection()` method, sends data, and waits until the connection is closed:

```
import asyncio


class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('Data sent: {!r}'.format(self.message))

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))

    def connection_lost(self, exc):
        print('The server closed the connection')
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
```

(continues on next page)

(continued from previous page)

```
    message = 'Hello World!'

    transport, protocol = await loop.create_connection(
        lambda: EchoClientProtocol(message, on_con_lost),
        '127.0.0.1', 8888)

    # Wait until the protocol signals that the connection
    # is lost and close the transport.
    try:
        await on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

See also

The *TCP echo client using streams* example uses the high-level `asyncio.open_connection()` function.

UDP Echo Server

A UDP echo server, using the `loop.create_datagram_endpoint()` method, sends back received data:

```
import asyncio


class EchoServerProtocol:
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        message = data.decode()
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)


async def main():
    print("Starting UDP server")

    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    # One protocol instance will be created to serve all
    # client requests.
    transport, protocol = await loop.create_datagram_endpoint(
        EchoServerProtocol,
        local_addr=('127.0.0.1', 9999))

    try:
        await asyncio.sleep(3600)  # Serve for 1 hour.
```

(continues on next page)

(continued from previous page)

```
    finally:
        transport.close()


asyncio.run(main())
```

UDP Echo Client

A UDP echo client, using the *loop.create_datagram_endpoint()* method, sends data and closes the transport when it receives the answer:

```
import asyncio


class EchoClientProtocol:
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost
        self.transport = None

    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())

    def datagram_received(self, data, addr):
        print("Received:", data.decode())

        print("Close the socket")
        self.transport.close()

    def error_received(self, exc):
        print('Error received:', exc)

    def connection_lost(self, exc):
        print("Connection closed")
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = "Hello World!"

    transport, protocol = await loop.create_datagram_endpoint(
        lambda: EchoClientProtocol(message, on_con_lost),
        remote_addr=('127.0.0.1', 9999))

    try:
        await on_con_lost
    finally:
        transport.close()
```

(continues on next page)

(continued from previous page)

```
asyncio.run(main())
```

Connecting Existing Sockets

Wait until a socket receives data using the *loop.create_connection()* method with a protocol:

```
import asyncio
import socket


class MyProtocol(asyncio.Protocol):

    def __init__(self, on_con_lost):
        self.transport = None
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        self.transport = transport

    def data_received(self, data):
        print("Received:", data.decode())

        # We are done: close the transport;
        # connection_lost() will be called automatically.
        self.transport.close()

    def connection_lost(self, exc):
        # The socket has been closed
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()
    on_con_lost = loop.create_future()

    # Create a pair of connected sockets
    rsock, wsock = socket.socketpair()

    # Register the socket to wait for data.
    transport, protocol = await loop.create_connection(
        lambda: MyProtocol(on_con_lost), sock=rsock)

    # Simulate the reception of data from the network.
    loop.call_soon(wsock.send, 'abc'.encode())

    try:
        await protocol.on_con_lost
    finally:
        transport.close()
        wsock.close()
```

(continues on next page)

(continued from previous page)

```
asyncio.run(main())
```

See also

The *watch a file descriptor for read events* example uses the low-level `loop.add_reader()` method to register an FD.

The *register an open socket to wait for data using streams* example uses high-level streams created by the `open_connection()` function in a coroutine.

loop.subprocess_exec() and SubprocessProtocol

An example of a subprocess protocol used to get the output of a subprocess and to wait for the subprocess exit.

The subprocess is created by the `loop.subprocess_exec()` method:

```
import asyncio
import sys

class DateProtocol(asyncio.SubprocessProtocol):
    def __init__(self, exit_future):
        self.exit_future = exit_future
        self.output = bytearray()
        self.pipe_closed = False
        self.exited = False

    def pipe_connection_lost(self, fd, exc):
        self.pipe_closed = True
        self.check_for_exit()

    def pipe_data_received(self, fd, data):
        self.output.extend(data)

    def process_exited(self):
        self.exited = True
        # process_exited() method can be called before
        # pipe_connection_lost() method: wait until both methods are
        # called.
        self.check_for_exit()

    def check_for_exit(self):
        if self.pipe_closed and self.exited:
            self.exit_future.set_result(True)

async def get_date():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    code = 'import datetime; print(datetime.datetime.now())'
    exit_future = asyncio.Future(loop=loop)

    # Create the subprocess controlled by DateProtocol;
    # redirect the standard output into a pipe.
```

(continues on next page)

(continued from previous page)

```
    transport, protocol = await loop.subprocess_exec(
        lambda: DateProtocol(exit_future),
        sys.executable, '-c', code,
        stdin=None, stderr=None)

    # Wait for the subprocess exit using the process_exited()
    # method of the protocol.
    await exit_future

    # Close the stdout pipe.
    transport.close()

    # Read the output which was collected by the
    # pipe_data_received() method of the protocol.
    data = bytes(protocol.output)
    return data.decode('ascii').rstrip()

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

See also the *same example* written using high-level APIs.

18.1.11 Policies

An event loop policy is a global object used to get and set the current *event loop*, as well as create new event loops. The default policy can be *replaced* with *built-in alternatives* to use different event loop implementations, or substituted by a *custom policy* that can override these behaviors.

The *policy object* gets and sets a separate event loop per *context*. This is per-thread by default, though custom policies could define *context* differently.

Custom event loop policies can control the behavior of `get_event_loop()`, `set_event_loop()`, and `new_event_loop()`.

Policy objects should implement the APIs defined in the `AbstractEventLoopPolicy` abstract base class.

Getting and Setting the Policy

The following functions can be used to get and set the policy for the current process:

`asyncio.get_event_loop_policy()`

Return the current process-wide policy.

`asyncio.set_event_loop_policy(policy)`

Set the current process-wide policy to *policy*.

If *policy* is set to `None`, the default policy is restored.

Policy Objects

The abstract event loop policy base class is defined as follows:

class asyncio.**AbstractEventLoopPolicy**

An abstract base class for asyncio policies.

get_event_loop()

Get the event loop for the current context.

Return an event loop object implementing the *AbstractEventLoop* interface.

This method should never return None.

Changed in version 3.6.

set_event_loop(*loop*)

Set the event loop for the current context to *loop*.

new_event_loop()

Create and return a new event loop object.

This method should never return None.

get_child_watcher()

Get a child process watcher object.

Return a watcher object implementing the *AbstractChildWatcher* interface.

This function is Unix specific.

Deprecated since version 3.12.

set_child_watcher(*watcher*)

Set the current child process watcher to *watcher*.

This function is Unix specific.

Deprecated since version 3.12.

asyncio ships with the following built-in policies:

class asyncio.**DefaultEventLoopPolicy**

The default asyncio policy. Uses *SelectorEventLoop* on Unix and *ProactorEventLoop* on Windows.

There is no need to install the default policy manually. asyncio is configured to use the default policy automatically.

Changed in version 3.8: On Windows, *ProactorEventLoop* is now used by default.

Deprecated since version 3.12: The *get_event_loop()* method of the default asyncio policy now emits a *DeprecationWarning* if there is no current event loop set and it decides to create one. In some future Python release this will become an error.

class asyncio.**WindowsSelectorEventLoopPolicy**

An alternative event loop policy that uses the *SelectorEventLoop* event loop implementation.

Availability: Windows.

class asyncio.**WindowsProactorEventLoopPolicy**

An alternative event loop policy that uses the *ProactorEventLoop* event loop implementation.

Availability: Windows.

Process Watchers

A process watcher allows customization of how an event loop monitors child processes on Unix. Specifically, the event loop needs to know when a child process has exited.

In asyncio, child processes are created with *create_subprocess_exec()* and *loop.subprocess_exec()* functions.

asyncio defines the *AbstractChildWatcher* abstract base class, which child watchers should implement, and has four different implementations: *ThreadedChildWatcher* (configured to be used by default), *MultiLoopChildWatcher*, *SafeChildWatcher*, and *FastChildWatcher*.

See also the *Subprocess and Threads* section.

The following two functions can be used to customize the child process watcher implementation used by the asyncio event loop:

asyncio.**get_child_watcher**()

Return the current child watcher for the current policy.

Deprecated since version 3.12.

asyncio.**set_child_watcher**(*watcher*)

Set the current child watcher to *watcher* for the current policy. *watcher* must implement methods defined in the *AbstractChildWatcher* base class.

Deprecated since version 3.12.

Note

Third-party event loops implementations might not support custom child watchers. For such event loops, using *set_child_watcher()* might be prohibited or have no effect.

class asyncio.**AbstractChildWatcher**

add_child_handler(*pid*, *callback*, **args*)

Register a new child handler.

Arrange for `callback(pid, returncode, *args)` to be called when a process with PID equal to *pid* terminates. Specifying another callback for the same process replaces the previous handler.

The *callback* callable must be thread-safe.

remove_child_handler(*pid*)

Removes the handler for process with PID equal to *pid*.

The function returns `True` if the handler was successfully removed, `False` if there was nothing to remove.

attach_loop(*loop*)

Attach the watcher to an event loop.

If the watcher was previously attached to an event loop, then it is first detached before attaching to the new loop.

Note: loop may be `None`.

is_active()

Return `True` if the watcher is ready to use.

Spawning a subprocess with *inactive* current child watcher raises *RuntimeError*.

Added in version 3.8.

close()

Close the watcher.

This method has to be called to ensure that underlying resources are cleaned-up.

Deprecated since version 3.12.

class asyncio.**ThreadedChildWatcher**

This implementation starts a new waiting thread for every subprocess spawn.

It works reliably even when the asyncio event loop is run in a non-main OS thread.

There is no noticeable overhead when handling a big number of children ($O(1)$ each time a child terminates), but starting a thread per process requires extra memory.

This watcher is used by default.

Added in version 3.8.

class asyncio.**MultiLoopChildWatcher**

This implementation registers a SIGCHLD signal handler on instantiation. That can break third-party code that installs a custom handler for SIGCHLD signal.

The watcher avoids disrupting other code spawning processes by polling every process explicitly on a SIGCHLD signal.

There is no limitation for running subprocesses from different threads once the watcher is installed.

The solution is safe but it has a significant overhead when handling a big number of processes ($O(n)$ each time a SIGCHLD is received).

Added in version 3.8.

Deprecated since version 3.12.

class asyncio.**SafeChildWatcher**

This implementation uses active event loop from the main thread to handle SIGCHLD signal. If the main thread has no running event loop another thread cannot spawn a subprocess (*RuntimeError* is raised).

The watcher avoids disrupting other code spawning processes by polling every process explicitly on a SIGCHLD signal.

This solution is as safe as *MultiLoopChildWatcher* and has the same $O(n)$ complexity but requires a running event loop in the main thread to work.

Deprecated since version 3.12.

class asyncio.**FastChildWatcher**

This implementation reaps every terminated processes by calling os.waitpid(-1) directly, possibly breaking other code spawning processes and waiting for their termination.

There is no noticeable overhead when handling a big number of children ($O(1)$ each time a child terminates).

This solution requires a running event loop in the main thread to work, as *SafeChildWatcher*.

Deprecated since version 3.12.

class asyncio.**PidfdChildWatcher**

This implementation polls process file descriptors (pidfds) to await child process termination. In some respects, *PidfdChildWatcher* is a "Goldilocks" child watcher implementation. It doesn't require signals or threads,

doesn't interfere with any processes launched outside the event loop, and scales linearly with the number of subprocesses launched by the event loop. The main disadvantage is that pidfds are specific to Linux, and only work on recent (5.3+) kernels.

Added in version 3.9.

Custom Policies

To implement a new event loop policy, it is recommended to subclass *DefaultEventLoopPolicy* and override the methods for which custom behavior is wanted, e.g.:

```
class MyEventLoopPolicy(asyncio.DefaultEventLoopPolicy):

    def get_event_loop(self):
        """Get the event loop.

        This may be None or an instance of EventLoop.
        """
        loop = super().get_event_loop()
        # Do something with loop ...
        return loop

asyncio.set_event_loop_policy(MyEventLoopPolicy())
```

18.1.12 Platform Support

The *asyncio* module is designed to be portable, but some platforms have subtle differences and limitations due to the platforms' underlying architecture and capabilities.

All Platforms

- *loop.add_reader()* and *loop.add_writer()* cannot be used to monitor file I/O.

Windows

Source code: Lib/asyncio/proactor_events.py, Lib/asyncio/windows_events.py, Lib/asyncio/windows_utils.py

---

Changed in version 3.8: On Windows, *ProactorEventLoop* is now the default event loop.

All event loops on Windows do not support the following methods:

- *loop.create_unix_connection()* and *loop.create_unix_server()* are not supported. The *socket.AF_UNIX* socket family is specific to Unix.
- *loop.add_signal_handler()* and *loop.remove_signal_handler()* are not supported.

SelectorEventLoop has the following limitations:

- *SelectSelector* is used to wait on socket events: it supports sockets and is limited to 512 sockets.
- *loop.add_reader()* and *loop.add_writer()* only accept socket handles (e.g. pipe file descriptors are not supported).

- Pipes are not supported, so the *loop.connect_read_pipe()* and *loop.connect_write_pipe()* methods are not implemented.
- *Subprocesses* are not supported, i.e. *loop.subprocess_exec()* and *loop.subprocess_shell()* methods are not implemented.

ProactorEventLoop has the following limitations:

- The *loop.add_reader()* and *loop.add_writer()* methods are not supported.

The resolution of the monotonic clock on Windows is usually around 15.6 milliseconds. The best resolution is 0.5 milliseconds. The resolution depends on the hardware (availability of HPET) and on the Windows configuration.

Subprocess Support on Windows

On Windows, the default event loop *ProactorEventLoop* supports subprocesses, whereas *SelectorEventLoop* does not.

The *policy.set_child_watcher()* function is also not supported, as *ProactorEventLoop* has a different mechanism to watch child processes.

macOS

Modern macOS versions are fully supported.

macOS <= 10.8

On macOS 10.6, 10.7 and 10.8, the default event loop uses *selectors.KqueueSelector*, which does not support character devices on these versions. The *SelectorEventLoop* can be manually configured to use *SelectSelector* or *PollSelector* to support character devices on these older versions of macOS. Example:

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

18.1.13 Extending

The main direction for *asyncio* extending is writing custom *event loop* classes. Asyncio has helpers that could be used to simplify this task.

Note

Third-parties should reuse existing asyncio code with caution, a new Python version is free to break backward compatibility in *internal* part of API.

Writing a Custom Event Loop

asyncio.AbstractEventLoop declares very many methods. Implementing all them from scratch is a tedious job.

A loop can get many common methods implementation for free by inheriting from `asyncio.BaseEventLoop`.

In turn, the successor should implement a bunch of *private* methods declared but not implemented in `asyncio.BaseEventLoop`.

For example, `loop.create_connection()` checks arguments, resolves DNS addresses, and calls `loop._make_socket_transport()` that should be implemented by inherited class. The `_make_socket_transport()` method is not documented and is considered as an *internal* API.

Future and Task private constructors

asyncio.Future and *asyncio.Task* should be never created directly, please use corresponding *loop.create_future()* and *loop.create_task()*, or *asyncio.create_task()* factories instead.

However, third-party *event loops* may *reuse* built-in future and task implementations for the sake of getting a complex and highly optimized code for free.

For this purpose the following, *private* constructors are listed:

`Future.__init__`(*, *loop=None*)

Create a built-in future instance.

loop is an optional event loop instance.

`Task.__init__`(*coro*, *, *loop=None*, *name=None*, *context=None*)

Create a built-in task instance.

loop is an optional event loop instance. The rest of arguments are described in *loop.create_task()* description.

Changed in version 3.11: *context* argument is added.

Task lifetime support

A third party task implementation should call the following functions to keep a task visible by *asyncio.all_tasks()* and *asyncio.current_task()*:

`asyncio._register_task`(*task*)

Register a new *task* as managed by *asyncio*.

Call the function from a task constructor.

`asyncio._unregister_task`(*task*)

Unregister a *task* from *asyncio* internal structures.

The function should be called when a task is about to finish.

`asyncio._enter_task`(*loop*, *task*)

Switch the current task to the *task* argument.

Call the function just before executing a portion of embedded *coroutine* (`coroutine.send()` or `coroutine.throw()`).

asyncio.**_leave_task**(*loop*, *task*)

Switch the current task back from *task* to None.

Call the function just after coroutine.send() or coroutine.throw() execution.

18.1.14 High-level API Index

This page lists all high-level async/await enabled asyncio APIs.

Tasks

Utilities to run asyncio programs, create Tasks, and await on multiple things with timeouts.

run()	Create event loop, run a coroutine, close the loop.
Runner	A context manager that simplifies multiple async function calls.
Task	Task object.
TaskGroup	A context manager that holds a group of tasks. Provides a convenient and reliable way to wait for all tasks in the group to finish.
create_task()	Start an asyncio Task, then returns it.
current_task()	Return the current Task.
all_tasks()	Return all tasks that are not yet finished for an event loop.
await *sleep()*	Sleep for a number of seconds.
await *gather()*	Schedule and wait for things concurrently.
await *wait_for()*	Run with a timeout.
await *shield()*	Shield from cancellation.
await *wait()*	Monitor for completion.
timeout()	Run with a timeout. Useful in cases when wait_for is not suitable.
to_thread()	Asynchronously run a function in a separate OS thread.
run_coroutine_threadsafe()	Schedule a coroutine from another OS thread.
for in *as_completed()*	Monitor for completion with a for loop.

Examples

- *Using asyncio.gather() to run things in parallel*.
- *Using asyncio.wait_for() to enforce a timeout*.
- *Cancellation*.
- *Using asyncio.sleep()*.
- See also the main *Tasks documentation page*.

Queues

Queues should be used to distribute work amongst multiple asyncio Tasks, implement connection pools, and pub/sub patterns.

Queue	A FIFO queue.
PriorityQueue	A priority queue.
LifoQueue	A LIFO queue.

Examples

- *Using asyncio.Queue to distribute workload between several Tasks*.
- See also the *Queues documentation page*.

Subprocesses

Utilities to spawn subprocesses and run shell commands.

await *create_subprocess_exec()*	Create a subprocess.
await *create_subprocess_shell()*	Run a shell command.

Examples

- *Executing a shell command*.
- See also the *subprocess APIs* documentation.

Streams

High-level APIs to work with network IO.

await *open_connection()*	Establish a TCP connection.
await *open_unix_connection()*	Establish a Unix socket connection.
await *start_server()*	Start a TCP server.
await *start_unix_server()*	Start a Unix socket server.
StreamReader	High-level async/await object to receive network data.
StreamWriter	High-level async/await object to send network data.

Examples

- *Example TCP client*.
- See also the *streams APIs* documentation.

Synchronization

Threading-like synchronization primitives that can be used in Tasks.

`Lock`	A mutex lock.
`Event`	An event object.
`Condition`	A condition object.
`Semaphore`	A semaphore.
`BoundedSemaphore`	A bounded semaphore.
`Barrier`	A barrier object.

Examples

- *Using asyncio.Event*.
- *Using asyncio.Barrier*.
- See also the documentation of asyncio *synchronization primitives*.

Exceptions

`asyncio.CancelledError`	Raised when a Task is cancelled. See also `Task.cancel()`.
`asyncio.BrokenBarrierError`	Raised when a Barrier is broken. See also `Barrier.wait()`.

Examples

- *Handling CancelledError to run code on cancellation request*.
- See also the full list of *asyncio-specific exceptions*.

18.1.15 Low-level API Index

This page lists all low-level asyncio APIs.

Obtaining the Event Loop

asyncio.get_running_loop()	The **preferred** function to get the running event loop.
asyncio.get_event_loop()	Get an event loop instance (running or current via the current policy).
asyncio.set_event_loop()	Set the event loop as current via the current policy.
asyncio.new_event_loop()	Create a new event loop.

Examples

- *Using asyncio.get_running_loop()*.

Event Loop Methods

See also the main documentation section about the *Event Loop Methods*.

Lifecycle

loop.run_until_complete()	Run a Future/Task/awaitable until complete.
loop.run_forever()	Run the event loop forever.
loop.stop()	Stop the event loop.
loop.close()	Close the event loop.
loop.is_running()	Return `True` if the event loop is running.
loop.is_closed()	Return `True` if the event loop is closed.
`await` *loop.shutdown_asyncgens()*	Close asynchronous generators.

Debugging

loop.set_debug()	Enable or disable the debug mode.
loop.get_debug()	Get the current debug mode.

Scheduling Callbacks

loop.call_soon()	Invoke a callback soon.
loop.call_soon_threadsafe()	A thread-safe variant of *loop.call_soon()*.
loop.call_later()	Invoke a callback *after* the given time.
loop.call_at()	Invoke a callback *at* the given time.

Thread/Process Pool

await *loop.run_in_executor()*	Run a CPU-bound or other blocking function in a *concurrent.futures* executor.
loop.set_default_executor()	Set the default executor for *loop.run_in_executor()*.

Tasks and Futures

loop.create_future()	Create a *Future* object.
loop.create_task()	Schedule coroutine as a *Task*.
loop.set_task_factory()	Set a factory used by *loop.create_task()* to create *Tasks*.
loop.get_task_factory()	Get the factory *loop.create_task()* uses to create *Tasks*.

DNS

await *loop.getaddrinfo()*	Asynchronous version of *socket.getaddrinfo()*.
await *loop.getnameinfo()*	Asynchronous version of *socket.getnameinfo()*.

Networking and IPC

await *loop.create_connection()*	Open a TCP connection.
await *loop.create_server()*	Create a TCP server.
await *loop.create_unix_connection()*	Open a Unix socket connection.
await *loop.create_unix_server()*	Create a Unix socket server.
await *loop.connect_accepted_socket()*	Wrap a *socket* into a `(transport, protocol)` pair.
await *loop.create_datagram_endpoint()*	Open a datagram (UDP) connection.
await *loop.sendfile()*	Send a file over a transport.
await *loop.start_tls()*	Upgrade an existing connection to TLS.
await *loop.connect_read_pipe()*	Wrap a read end of a pipe into a `(transport, protocol)` pair.
await *loop.connect_write_pipe()*	Wrap a write end of a pipe into a `(transport, protocol)` pair.

Sockets

`await loop.sock_recv()`	Receive data from the `socket`.
`await loop.sock_recv_into()`	Receive data from the `socket` into a buffer.
`await loop.sock_recvfrom()`	Receive a datagram from the `socket`.
`await loop.sock_recvfrom_into()`	Receive a datagram from the `socket` into a buffer.
`await loop.sock_sendall()`	Send data to the `socket`.
`await loop.sock_sendto()`	Send a datagram via the `socket` to the given address.
`await loop.sock_connect()`	Connect the `socket`.
`await loop.sock_accept()`	Accept a `socket` connection.
`await loop.sock_sendfile()`	Send a file over the `socket`.
`loop.add_reader()`	Start watching a file descriptor for read availability.
`loop.remove_reader()`	Stop watching a file descriptor for read availability.
`loop.add_writer()`	Start watching a file descriptor for write availability.
`loop.remove_writer()`	Stop watching a file descriptor for write availability.

Unix Signals

`loop.add_signal_handler()`	Add a handler for a `signal`.
`loop.remove_signal_handler()`	Remove a handler for a `signal`.

Subprocesses

`loop.subprocess_exec()`	Spawn a subprocess.
`loop.subprocess_shell()`	Spawn a subprocess from a shell command.

Error Handling

`loop.call_exception_handler()`	Call the exception handler.
`loop.set_exception_handler()`	Set a new exception handler.
`loop.get_exception_handler()`	Get the current exception handler.
`loop.default_exception_handler()`	The default exception handler implementation.

Examples

- *Using asyncio.new_event_loop() and loop.run_forever()*.
- *Using loop.call_later()*.
- Using `loop.create_connection()` to implement *an echo-client*.
- Using `loop.create_connection()` to *connect a socket*.
- *Using add_reader() to watch an FD for read events*.
- *Using loop.add_signal_handler()*.
- *Using loop.subprocess_exec()*.

Transports

All transports implement the following methods:

transport.close()	Close the transport.
transport.is_closing()	Return `True` if the transport is closing or is closed.
transport.get_extra_info()	Request for information about the transport.
transport.set_protocol()	Set a new protocol.
transport.get_protocol()	Return the current protocol.

Transports that can receive data (TCP and Unix connections, pipes, etc). Returned from methods like *loop.create_connection()*, *loop.create_unix_connection()*, *loop.connect_read_pipe()*, etc:

Read Transports

transport.is_reading()	Return `True` if the transport is receiving.
transport.pause_reading()	Pause receiving.
transport.resume_reading()	Resume receiving.

Transports that can Send data (TCP and Unix connections, pipes, etc). Returned from methods like *loop.create_connection()*, *loop.create_unix_connection()*, *loop.connect_write_pipe()*, etc:

Write Transports

transport.write()	Write data to the transport.
transport.writelines()	Write buffers to the transport.
transport.can_write_eof()	Return *True* if the transport supports sending EOF.
transport.write_eof()	Close and send EOF after flushing buffered data.
transport.abort()	Close the transport immediately.
transport.get_write_buffer_size()	Return the current size of the output buffer.
transport.get_write_buffer_limits()	Return high and low water marks for write flow control.
transport.set_write_buffer_limits()	Set new high and low water marks for write flow control.

Transports returned by *loop.create_datagram_endpoint()*:

Datagram Transports

transport.sendto()	Send data to the remote peer.
transport.abort()	Close the transport immediately.

Low-level transport abstraction over subprocesses. Returned by *loop.subprocess_exec()* and *loop.subprocess_shell()*:

Subprocess Transports

transport.get_pid()	Return the subprocess process id.
transport.get_pipe_transport()	Return the transport for the requested communication pipe (*stdin*, *stdout*, or *stderr*).
transport.get_returncode()	Return the subprocess return code.
transport.kill()	Kill the subprocess.
transport.send_signal()	Send a signal to the subprocess.
transport.terminate()	Stop the subprocess.
transport.close()	Kill the subprocess and close all pipes.

Protocols

Protocol classes can implement the following **callback methods**:

callback *connection_made()*	Called when a connection is made.
callback *connection_lost()*	Called when the connection is lost or closed.
callback *pause_writing()*	Called when the transport's buffer goes over the high water mark.
callback *resume_writing()*	Called when the transport's buffer drains below the low water mark.

Streaming Protocols (TCP, Unix Sockets, Pipes)

callback *data_received()*	Called when some data is received.
callback *eof_received()*	Called when an EOF is received.

Buffered Streaming Protocols

callback *get_buffer()*	Called to allocate a new receive buffer.
callback *buffer_updated()*	Called when the buffer was updated with the received data.
callback *eof_received()*	Called when an EOF is received.

Datagram Protocols

callback *datagram_received()*	Called when a datagram is received.
callback *error_received()*	Called when a previous send or receive operation raises an *OSError*.

Subprocess Protocols

callback *pipe_data_received()*	Called when the child process writes data into its *stdout* or *stderr* pipe.
callback *pipe_connection_lost()*	Called when one of the pipes communicating with the child process is closed.
callback *process_exited()*	Called when the child process has exited. It can be called before *pipe_data_received()* and *pipe_connection_lost()* methods.

Event Loop Policies

Policies is a low-level mechanism to alter the behavior of functions like *asyncio.get_event_loop()*. See also the main *policies section* for more details.

Accessing Policies

asyncio.get_event_loop_policy()	Return the current process-wide policy.
asyncio.set_event_loop_policy()	Set a new process-wide policy.
AbstractEventLoopPolicy	Base class for policy objects.

18.1.16 Developing with asyncio

Asynchronous programming is different from classic "sequential" programming.

This page lists common mistakes and traps and explains how to avoid them.

Debug Mode

By default asyncio runs in production mode. In order to ease the development asyncio has a *debug mode*.

There are several ways to enable asyncio debug mode:

- Setting the `PYTHONASYNCIODEBUG` environment variable to `1`.
- Using the *Python Development Mode*.
- Passing `debug=True` to *asyncio.run()*.
- Calling *loop.set_debug()*.

In addition to enabling the debug mode, consider also:

- setting the log level of the *asyncio logger* to *logging.DEBUG*, for example the following snippet of code can be run at startup of the application:

```
logging.basicConfig(level=logging.DEBUG)
```

- configuring the *warnings* module to display *ResourceWarning* warnings. One way of doing that is by using the `-W default` command line option.

When the debug mode is enabled:

- asyncio checks for *coroutines that were not awaited* and logs them; this mitigates the "forgotten await" pitfall.
- Many non-threadsafe asyncio APIs (such as `loop.call_soon()` and `loop.call_at()` methods) raise an exception if they are called from a wrong thread.
- The execution time of the I/O selector is logged if it takes too long to perform an I/O operation.
- Callbacks taking longer than 100 milliseconds are logged. The `loop.slow_callback_duration` attribute can be used to set the minimum execution duration in seconds that is considered "slow".

Concurrency and Multithreading

An event loop runs in a thread (typically the main thread) and executes all callbacks and Tasks in its thread. While a Task is running in the event loop, no other Tasks can run in the same thread. When a Task executes an `await` expression, the running Task gets suspended, and the event loop executes the next Task.

To schedule a *callback* from another OS thread, the `loop.call_soon_threadsafe()` method should be used. Example:

```
loop.call_soon_threadsafe(callback, *args)
```

Almost all asyncio objects are not thread safe, which is typically not a problem unless there is code that works with them from outside of a Task or a callback. If there's a need for such code to call a low-level asyncio API, the `loop.call_soon_threadsafe()` method should be used, e.g.:

```
loop.call_soon_threadsafe(fut.cancel)
```

To schedule a coroutine object from a different OS thread, the `run_coroutine_threadsafe()` function should be used. It returns a `concurrent.futures.Future` to access the result:

```
async def coro_func():
    return await asyncio.sleep(1, 42)

# Later in another OS thread:

future = asyncio.run_coroutine_threadsafe(coro_func(), loop)
# Wait for the result:
result = future.result()
```

To handle signals the event loop must be run in the main thread.

The `loop.run_in_executor()` method can be used with a `concurrent.futures.ThreadPoolExecutor` to execute blocking code in a different OS thread without blocking the OS thread that the event loop runs in.

There is currently no way to schedule coroutines or callbacks directly from a different process (such as one started with `multiprocessing`). The *Event Loop Methods* section lists APIs that can read from pipes and watch file descriptors without blocking the event loop. In addition, asyncio's *Subprocess* APIs provide a way to start a process and communicate with it from the event loop. Lastly, the aforementioned `loop.run_in_executor()` method can also be used with a `concurrent.futures.ProcessPoolExecutor` to execute code in a different process.

Running Blocking Code

Blocking (CPU-bound) code should not be called directly. For example, if a function performs a CPU-intensive calculation for 1 second, all concurrent asyncio Tasks and IO operations would be delayed by 1 second.

An executor can be used to run a task in a different thread or even in a different process to avoid blocking the OS thread with the event loop. See the *loop.run_in_executor()* method for more details.

Logging

asyncio uses the *logging* module and all logging is performed via the `"asyncio"` logger.

The default log level is *logging.INFO*, which can be easily adjusted:

```
logging.getLogger("asyncio").setLevel(logging.WARNING)
```

Network logging can block the event loop. It is recommended to use a separate thread for handling logs or use non-blocking IO. For example, see blocking-handlers.

Detect never-awaited coroutines

When a coroutine function is called, but not awaited (e.g. `coro()` instead of `await coro()`) or the coroutine is not scheduled with *asyncio.create_task()*, asyncio will emit a *RuntimeWarning*:

```
import asyncio

async def test():
    print("never scheduled")

async def main():
    test()

asyncio.run(main())
```

Output:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
  test()
```

Output in debug mode:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
Coroutine created at (most recent call last)
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

  < .. >

  File "../t.py", line 7, in main
    test()
  test()
```

The usual fix is to either await the coroutine or call the *asyncio.create_task()* function:

```
async def main():
    await test()
```

Detect never-retrieved exceptions

If a *Future.set_exception()* is called but the Future object is never awaited on, the exception would never be propagated to the user code. In this case, asyncio would emit a log message when the Future object is garbage collected.

Example of an unhandled exception:

```
import asyncio

async def bug():
    raise Exception("not consumed")

async def main():
    asyncio.create_task(bug())

asyncio.run(main())
```

Output:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
  exception=Exception('not consumed')>

Traceback (most recent call last):
  File "test.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

Enable the debug mode to get the traceback where the task was created:

```
asyncio.run(main(), debug=True)
```

Output in debug mode:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
    exception=Exception('not consumed') created at asyncio/tasks.py:321>

source_traceback: Object created at (most recent call last):
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

< .. >

Traceback (most recent call last):
  File "../t.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

Note

The source code for asyncio can be found in Lib/asyncio/.

18.2 socket — Low-level networking interface

Source code: Lib/socket.py

This module provides access to the BSD *socket* interface. It is available on all modern Unix systems, Windows, MacOS, and probably additional platforms.

Note

Some behavior may be platform dependent, since calls are made to the operating system socket APIs.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python's object-oriented style: the `socket()` function returns a *socket object* whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with `read()` and `write()` operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

See also

Module *socketserver*
Classes that simplify writing network servers.

Module *ssl*
A TLS/SSL wrapper for socket objects.

18.2.1 Socket families

Depending on the system and the build options, various socket families are supported by this module.

The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created. Socket addresses are represented as follows:

- The address of an *AF_UNIX* socket bound to a file system node is represented as a string, using the file system encoding and the `'surrogateescape'` error handler (see **PEP 383**). An address in Linux's abstract namespace is returned as a *bytes-like object* with an initial null byte; note that sockets in this namespace can communicate with normal file system sockets, so programs intended to run on Linux may need to deal with both types of address. A string or bytes-like object can be used for either type of address when passing it as an argument.

  Changed in version 3.3: Previously, *AF_UNIX* socket paths were assumed to use UTF-8 encoding.

  Changed in version 3.5: Writable *bytes-like object* is now accepted.

- A pair `(host, port)` is used for the *AF_INET* address family, where *host* is a string representing either a hostname in internet domain notation like `'daring.cwi.nl'` or an IPv4 address like `'100.50.200.5'`, and *port* is an integer.

  - For IPv4 addresses, two special forms are accepted instead of a host address: `''` represents `INADDR_ANY`, which is used to bind to all interfaces, and the string `'<broadcast>'` represents `INADDR_BROADCAST`. This behavior is not compatible with IPv6, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs.

- For *AF_INET6* address family, a four-tuple `(host, port, flowinfo, scope_id)` is used, where *flowinfo* and *scope_id* represent the `sin6_flowinfo` and `sin6_scope_id` members in `struct sockaddr_in6` in C. For *socket* module methods, *flowinfo* and *scope_id* can be omitted just for backward compatibility. Note, however, omission of *scope_id* can cause problems in manipulating scoped IPv6 addresses.

  Changed in version 3.7: For multicast addresses (with *scope_id* meaningful) *address* may not contain `%scope_id` (or `zone id`) part. This information is superfluous and may be safely omitted (recommended).

- `AF_NETLINK` sockets are represented as pairs `(pid, groups)`.

- Linux-only support for TIPC is available using the `AF_TIPC` address family. TIPC is an open, non-IP based networked protocol designed for use in clustered computer environments. Addresses are represented by a tuple, and the fields depend on the address type. The general tuple form is `(addr_type, v1, v2, v3 [, scope])`, where:

  - *addr_type* is one of `TIPC_ADDR_NAMESEQ`, `TIPC_ADDR_NAME`, or `TIPC_ADDR_ID`.
  - *scope* is one of `TIPC_ZONE_SCOPE`, `TIPC_CLUSTER_SCOPE`, and `TIPC_NODE_SCOPE`.
  - If *addr_type* is `TIPC_ADDR_NAME`, then *v1* is the server type, *v2* is the port identifier, and *v3* should be 0.

    If *addr_type* is `TIPC_ADDR_NAMESEQ`, then *v1* is the server type, *v2* is the lower port number, and *v3* is the upper port number.

    If *addr_type* is `TIPC_ADDR_ID`, then *v1* is the node, *v2* is the reference, and *v3* should be set to 0.

- A tuple `(interface, )` is used for the *AF_CAN* address family, where *interface* is a string representing a network interface name like `'can0'`. The network interface name `''` can be used to receive packets from all network interfaces of this family.

  - *CAN_ISOTP* protocol require a tuple `(interface, rx_addr, tx_addr)` where both additional parameters are unsigned long integer that represent a CAN identifier (standard or extended).
  - *CAN_J1939* protocol require a tuple `(interface, name, pgn, addr)` where additional parameters are 64-bit unsigned integer representing the ECU name, a 32-bit unsigned integer representing the Parameter Group Number (PGN), and an 8-bit integer representing the address.

- A string or a tuple `(id, unit)` is used for the `SYSPROTO_CONTROL` protocol of the `PF_SYSTEM` family. The string is the name of a kernel control using a dynamically assigned ID. The tuple can be used if ID and unit number of the kernel control are known or if a registered ID is used.

  Added in version 3.3.

- `AF_BLUETOOTH` supports the following protocols and address formats:

  - `BTPROTO_L2CAP` accepts `(bdaddr, psm)` where `bdaddr` is the Bluetooth address as a string and `psm` is an integer.
  - `BTPROTO_RFCOMM` accepts `(bdaddr, channel)` where `bdaddr` is the Bluetooth address as a string and `channel` is an integer.
  - `BTPROTO_HCI` accepts `(device_id,)` where `device_id` is either an integer or a string with the Bluetooth address of the interface. (This depends on your OS; NetBSD and DragonFlyBSD expect a Bluetooth address while everything else expects an integer.)

    Changed in version 3.2: NetBSD and DragonFlyBSD support added.

  - `BTPROTO_SCO` accepts `bdaddr` where `bdaddr` is a *bytes* object containing the Bluetooth address in a string format. (ex. `b'12:23:34:45:56:67'`) This protocol is not supported under FreeBSD.

- *AF_ALG* is a Linux-only socket based interface to Kernel cryptography. An algorithm socket is configured with a tuple of two to four elements `(type, name [, feat [, mask]])`, where:

  - *type* is the algorithm type as string, e.g. `aead`, `hash`, `skcipher` or `rng`.

- *name* is the algorithm name and operation mode as string, e.g. `sha256`, `hmac(sha256)`, `cbc(aes)` or `drbg_nopr_ctr_aes256`.
- *feat* and *mask* are unsigned 32bit integers.

Availability: Linux >= 2.6.38.

Some algorithm types require more recent Kernels.

Added in version 3.6.

- *AF_VSOCK* allows communication between virtual machines and their hosts. The sockets are represented as a `(CID, port)` tuple where the context ID or CID and port are integers.

  Availability: Linux >= 3.9

  See *vsock(7)*

  Added in version 3.7.

- *AF_PACKET* is a low-level interface directly to network devices. The addresses are represented by the tuple `(ifname, proto[, pkttype[, hatype[, addr]]])` where:
  - *ifname* - String specifying the device name.
  - *proto* - The Ethernet protocol number. May be *ETH_P_ALL* to capture all protocols, one of the *ETHERTYPE_* constants* or any other Ethernet protocol number.
  - *pkttype* - Optional integer specifying the packet type:
    - `PACKET_HOST` (the default) - Packet addressed to the local host.
    - `PACKET_BROADCAST` - Physical-layer broadcast packet.
    - `PACKET_MULTICAST` - Packet sent to a physical-layer multicast address.
    - `PACKET_OTHERHOST` - Packet to some other host that has been caught by a device driver in promiscuous mode.
    - `PACKET_OUTGOING` - Packet originating from the local host that is looped back to a packet socket.
  - *hatype* - Optional integer specifying the ARP hardware address type.
  - *addr* - Optional bytes-like object specifying the hardware physical address, whose interpretation depends on the device.

  Availability: Linux >= 2.2.

- *AF_QIPCRTR* is a Linux-only socket based interface for communicating with services running on co-processors in Qualcomm platforms. The address family is represented as a `(node, port)` tuple where the *node* and *port* are non-negative integers.

  Availability: Linux >= 4.7.

  Added in version 3.8.

- `IPPROTO_UDPLITE` is a variant of UDP which allows you to specify what portion of a packet is covered with the checksum. It adds two socket options that you can change. `self.setsockopt(IPPROTO_UDPLITE, UDPLITE_SEND_CSCOV, length)` will change what portion of outgoing packets are covered by the checksum and `self.setsockopt(IPPROTO_UDPLITE, UDPLITE_RECV_CSCOV, length)` will filter out packets which cover too little of their data. In both cases `length` should be in `range(8, 2**16, 8)`.

  Such a socket should be constructed with `socket(AF_INET, SOCK_DGRAM, IPPROTO_UDPLITE)` for IPv4 or `socket(AF_INET6, SOCK_DGRAM, IPPROTO_UDPLITE)` for IPv6.

  Availability: Linux >= 2.6.20, FreeBSD >= 10.1

Added in version 3.9.

- *AF_HYPERV* is a Windows-only socket based interface for communicating with Hyper-V hosts and guests. The address family is represented as a `(vm_id, service_id)` tuple where the `vm_id` and `service_id` are UUID strings.

  The `vm_id` is the virtual machine identifier or a set of known VMID values if the target is not a specific virtual machine. Known VMID constants defined on `socket` are:

  - `HV_GUID_ZERO`
  - `HV_GUID_BROADCAST`
  - `HV_GUID_WILDCARD` - Used to bind on itself and accept connections from all partitions.
  - `HV_GUID_CHILDREN` - Used to bind on itself and accept connection from child partitions.
  - `HV_GUID_LOOPBACK` - Used as a target to itself.
  - `HV_GUID_PARENT` - When used as a bind accepts connection from the parent partition. When used as an address target it will connect to the parent partition.

  The `service_id` is the service identifier of the registered service.

  Added in version 3.12.

If you use a hostname in the *host* portion of IPv4/v6 socket address, the program may show a nondeterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/v6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in *host* portion.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised. Errors related to socket or address semantics raise *OSError* or one of its subclasses.

Non-blocking mode is supported through *setblocking()*. A generalization of this based on timeouts is supported through *settimeout()*.

18.2.2 Module contents

The module *socket* exports the following elements.

Exceptions

exception socket.**error**

A deprecated alias of *OSError*.

Changed in version 3.3: Following **PEP 3151**, this class was made an alias of *OSError*.

exception socket.**herror**

A subclass of *OSError*, this exception is raised for address-related errors, i.e. for functions that use *h_errno* in the POSIX C API, including *gethostbyname_ex()* and *gethostbyaddr()*. The accompanying value is a pair `(h_errno, string)` representing an error returned by a library call. *h_errno* is a numeric value, while *string* represents the description of *h_errno*, as returned by the `hstrerror()` C function.

Changed in version 3.3: This class was made a subclass of *OSError*.

exception socket.**gaierror**

A subclass of *OSError*, this exception is raised for address-related errors by *getaddrinfo()* and *getnameinfo()*. The accompanying value is a pair `(error, string)` representing an error returned by a library call. *string* represents the description of *error*, as returned by the `gai_strerror()` C function. The numeric *error* value will match one of the `EAI_*` constants defined in this module.

Changed in version 3.3: This class was made a subclass of *OSError*.

exception socket.**timeout**

A deprecated alias of *TimeoutError*.

A subclass of *OSError*, this exception is raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to *settimeout()* (or implicitly through *setdefaulttimeout()*). The accompanying value is a string whose value is currently always "timed out".

Changed in version 3.3: This class was made a subclass of *OSError*.

Changed in version 3.10: This class was made an alias of *TimeoutError*.

Constants

The AF_* and SOCK_* constants are now `AddressFamily` and `SocketKind` *IntEnum* collections.

Added in version 3.4.

socket.**AF_UNIX**

socket.**AF_INET**

socket.**AF_INET6**

These constants represent the address (and protocol) families, used for the first argument to *socket()*. If the *AF_UNIX* constant is not defined then this protocol is unsupported. More constants may be available depending on the system.

socket.**AF_UNSPEC**

AF_UNSPEC means that *getaddrinfo()* should return socket addresses for any address family (either IPv4, IPv6, or any other) that can be used.

socket.**SOCK_STREAM**

socket.**SOCK_DGRAM**

socket.**SOCK_RAW**

socket.**SOCK_RDM**

socket.**SOCK_SEQPACKET**

These constants represent the socket types, used for the second argument to *socket()*. More constants may be available depending on the system. (Only *SOCK_STREAM* and *SOCK_DGRAM* appear to be generally useful.)

socket.**SOCK_CLOEXEC**

socket.**SOCK_NONBLOCK**

These two constants, if defined, can be combined with the socket types and allow you to set some flags atomically (thus avoiding possible race conditions and the need for separate calls).

See also

Secure File Descriptor Handling for a more thorough explanation.

Availability: Linux >= 2.6.27.

Added in version 3.2.

SO_*

socket.**SOMAXCONN**

MSG_*

SOL_*

SCM_*

IPPROTO_*

IPPORT_*

INADDR_*

IP_*

IPV6_*

EAI_*

AI_*

NI_*

TCP_*

Many constants of these forms, documented in the Unix documentation on sockets and/or the IP protocol, are also defined in the socket module. They are generally used in arguments to the *setsockopt()* and *getsockopt()* methods of socket objects. In most cases, only those symbols that are defined in the Unix header files are defined; for a few symbols, default values are provided.

Changed in version 3.6: SO_DOMAIN, SO_PROTOCOL, SO_PEERSEC, SO_PASSSEC, TCP_USER_TIMEOUT, TCP_CONGESTION were added.

Changed in version 3.6.5: On Windows, TCP_FASTOPEN, TCP_KEEPCNT appear if run-time Windows supports.

Changed in version 3.7: TCP_NOTSENT_LOWAT was added.

On Windows, TCP_KEEPIDLE, TCP_KEEPINTVL appear if run-time Windows supports.

Changed in version 3.10: IP_RECVTOS was added. Added TCP_KEEPALIVE. On MacOS this constant can be used in the same way that TCP_KEEPIDLE is used on Linux.

Changed in version 3.11: Added TCP_CONNECTION_INFO. On MacOS this constant can be used in the same way that TCP_INFO is used on Linux and BSD.

Changed in version 3.12: Added SO_RTABLE and SO_USER_COOKIE. On OpenBSD and FreeBSD respectively those constants can be used in the same way that SO_MARK is used on Linux. Also added missing TCP socket options from Linux: TCP_MD5SIG, TCP_THIN_LINEAR_TIMEOUTS, TCP_THIN_DUPACK, TCP_REPAIR, TCP_REPAIR_QUEUE, TCP_QUEUE_SEQ, TCP_REPAIR_OPTIONS, TCP_TIMESTAMP, TCP_CC_INFO, TCP_SAVE_SYN, TCP_SAVED_SYN, TCP_REPAIR_WINDOW, TCP_FASTOPEN_CONNECT, TCP_ULP, TCP_MD5SIG_EXT, TCP_FASTOPEN_KEY, TCP_FASTOPEN_NO_COOKIE, TCP_ZEROCOPY_RECEIVE, TCP_INQ, TCP_TX_DELAY. Added IP_PKTINFO, IP_UNBLOCK_SOURCE, IP_BLOCK_SOURCE, IP_ADD_SOURCE_MEMBERSHIP, IP_DROP_SOURCE_MEMBERSHIP.

Changed in version 3.13: Added SO_BINDTOIFINDEX. On Linux this constant can be used in the same way that SO_BINDTODEVICE is used, but with the index of a network interface instead of its name.

socket.**AF_CAN**

socket.**PF_CAN**

SOL_CAN_*

CAN_*

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux >= 2.6.25, NetBSD >= 8.

Added in version 3.3.

Changed in version 3.11: NetBSD support was added.

socket.**CAN_BCM**

CAN_BCM_*

CAN_BCM, in the CAN protocol family, is the broadcast manager (BCM) protocol. Broadcast manager constants, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux >= 2.6.25.

> **Note**
>
> The CAN_BCM_CAN_FD_FRAME flag is only available on Linux >= 4.8.

Added in version 3.4.

socket.**CAN_RAW_FD_FRAMES**

Enables CAN FD support in a CAN_RAW socket. This is disabled by default. This allows your application to send both CAN and CAN FD frames; however, you must accept both CAN and CAN FD frames when reading from the socket.

This constant is documented in the Linux documentation.

Availability: Linux >= 3.6.

Added in version 3.5.

socket.**CAN_RAW_JOIN_FILTERS**

Joins the applied CAN filters such that only CAN frames that match all given CAN filters are passed to user space.

This constant is documented in the Linux documentation.

Availability: Linux >= 4.1.

Added in version 3.9.

socket.**CAN_ISOTP**

CAN_ISOTP, in the CAN protocol family, is the ISO-TP (ISO 15765-2) protocol. ISO-TP constants, documented in the Linux documentation.

Availability: Linux >= 2.6.25.

Added in version 3.7.

socket.**CAN_J1939**

CAN_J1939, in the CAN protocol family, is the SAE J1939 protocol. J1939 constants, documented in the Linux documentation.

Availability: Linux >= 5.4.

Added in version 3.9.

socket.**AF_DIVERT**

socket.**PF_DIVERT**

These two constants, documented in the FreeBSD divert(4) manual page, are also defined in the socket module.

Availability: FreeBSD >= 14.0.

Added in version 3.12.

socket.**AF_PACKET**

socket.**PF_PACKET**

PACKET_*

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux >= 2.2.

socket.**ETH_P_ALL**

ETH_P_ALL can be used in the *socket* constructor as *proto* for the *AF_PACKET* family in order to capture every packet, regardless of protocol.

For more information, see the *packet(7)* manpage.

Availability: Linux.

Added in version 3.12.

socket.**AF_RDS**

socket.**PF_RDS**

socket.**SOL_RDS**

RDS_*

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux >= 2.6.30.

Added in version 3.3.

socket.**SIO_RCVALL**

socket.**SIO_KEEPALIVE_VALS**

socket.**SIO_LOOPBACK_FAST_PATH**

RCVALL_*

Constants for Windows' WSAIoctl(). The constants are used as arguments to the *ioctl()* method of socket objects.

Changed in version 3.6: SIO_LOOPBACK_FAST_PATH was added.

TIPC_*

TIPC related constants, matching the ones exported by the C socket API. See the TIPC documentation for more information.

socket.**AF_ALG**

socket.**SOL_ALG**

ALG_*

Constants for Linux Kernel cryptography.

Availability: Linux >= 2.6.38.

Added in version 3.6.

socket.**AF_VSOCK**

socket.**IOCTL_VM_SOCKETS_GET_LOCAL_CID**

VMADDR*

SO_VM*

Constants for Linux host/guest communication.

Availability: Linux >= 4.8.

Added in version 3.7.

socket.**AF_LINK**

Availability: BSD, macOS.

Added in version 3.4.

socket.**has_ipv6**

This constant contains a boolean value which indicates if IPv6 is supported on this platform.

socket.**BDADDR_ANY**

socket.**BDADDR_LOCAL**

These are string constants containing Bluetooth addresses with special meanings. For example, *BDADDR_ANY* can be used to indicate any address when specifying the binding socket with BTPROTO_RFCOMM.

socket.**HCI_FILTER**

socket.**HCI_TIME_STAMP**

socket.**HCI_DATA_DIR**

For use with BTPROTO_HCI. *HCI_FILTER* is not available for NetBSD or DragonFlyBSD. *HCI_TIME_STAMP* and *HCI_DATA_DIR* are not available for FreeBSD, NetBSD, or DragonFlyBSD.

socket.**AF_QIPCRTR**

Constant for Qualcomm's IPC router protocol, used to communicate with service providing remote processors.

Availability: Linux >= 4.7.

socket.**SCM_CREDS2**

socket.**LOCAL_CREDS**

socket.**LOCAL_CREDS_PERSISTENT**

LOCAL_CREDS and LOCAL_CREDS_PERSISTENT can be used with SOCK_DGRAM, SOCK_STREAM sockets, equivalent to Linux/DragonFlyBSD SO_PASSCRED, while LOCAL_CREDS sends the credentials at first read, LOCAL_CREDS_PERSISTENT sends for each read, SCM_CREDS2 must be then used for the latter for the message type.

Added in version 3.11.

Availability: FreeBSD.

socket.**SO_INCOMING_CPU**

Constant to optimize CPU locality, to be used in conjunction with SO_REUSEPORT.

Added in version 3.11.

Availability: Linux >= 3.9

socket.**AF_HYPERV**

socket.**HV_PROTOCOL_RAW**

socket.**HVSOCKET_CONNECT_TIMEOUT**

socket.**HVSOCKET_CONNECT_TIMEOUT_MAX**

socket.**HVSOCKET_CONNECTED_SUSPEND**

socket.**HVSOCKET_ADDRESS_FLAG_PASSTHRU**

socket.**HV_GUID_ZERO**

socket.**HV_GUID_WILDCARD**

socket.**HV_GUID_BROADCAST**

socket.**HV_GUID_CHILDREN**

socket.**HV_GUID_LOOPBACK**

socket.**HV_GUID_PARENT**

Constants for Windows Hyper-V sockets for host/guest communications.

Availability: Windows.

Added in version 3.12.

socket.**ETHERTYPE_ARP**

socket.**ETHERTYPE_IP**

socket.**ETHERTYPE_IPV6**

socket.**ETHERTYPE_VLAN**

IEEE 802.3 protocol number. constants.

Availability: Linux, FreeBSD, macOS.

Added in version 3.12.

Functions

Creating sockets

The following functions all create *socket objects*.

class socket.**socket**(*family=AF_INET*, *type=SOCK_STREAM*, *proto=0*, *fileno=None*)

Create a new socket using the given address family, socket type and protocol number. The address family should be *AF_INET* (the default), *AF_INET6*, *AF_UNIX*, *AF_CAN*, *AF_PACKET*, or *AF_RDS*. The socket type should be *SOCK_STREAM* (the default), *SOCK_DGRAM*, *SOCK_RAW* or perhaps one of the other SOCK_ constants. The protocol number is usually zero and may be omitted or in the case where the address family is *AF_CAN* the protocol should be one of CAN_RAW, *CAN_BCM*, *CAN_ISOTP* or *CAN_J1939*.

If *fileno* is specified, the values for *family*, *type*, and *proto* are auto-detected from the specified file descriptor. Auto-detection can be overruled by calling the function with explicit *family*, *type*, or *proto* arguments. This only affects how Python represents e.g. the return value of *socket.getpeername()* but not the actual OS resource. Unlike *socket.fromfd()*, *fileno* will return the same socket and not a duplicate. This may help close a detached socket using *socket.close()*.

The newly created socket is *non-inheritable*.

Raises an *auditing event* socket.__new__ with arguments self, family, type, protocol.

Changed in version 3.3: The AF_CAN family was added. The AF_RDS family was added.

Changed in version 3.4: The CAN_BCM protocol was added.

Changed in version 3.4: The returned socket is now non-inheritable.

Changed in version 3.7: The CAN_ISOTP protocol was added.

Changed in version 3.7: When *SOCK_NONBLOCK* or *SOCK_CLOEXEC* bit flags are applied to *type* they are cleared, and *socket.type* will not reflect them. They are still passed to the underlying system socket() call. Therefore,

```
sock = socket.socket(
    socket.AF_INET,
    socket.SOCK_STREAM | socket.SOCK_NONBLOCK)
```

will still create a non-blocking socket on OSes that support `SOCK_NONBLOCK`, but `sock.type` will be set to `socket.SOCK_STREAM`.

Changed in version 3.9: The CAN_J1939 protocol was added.

Changed in version 3.10: The IPPROTO_MPTCP protocol was added.

`socket.`**`socketpair`**(*[family[, type[, proto]]]*)

Build a pair of connected socket objects using the given address family, socket type, and protocol number. Address family, socket type, and protocol number are as for the *socket()* function above. The default family is *AF_UNIX* if defined on the platform; otherwise, the default is *AF_INET*.

The newly created sockets are *non-inheritable*.

Changed in version 3.2: The returned socket objects now support the whole socket API, rather than a subset.

Changed in version 3.4: The returned sockets are now non-inheritable.

Changed in version 3.5: Windows support added.

`socket.`**`create_connection`**(*address*, *timeout=GLOBAL_DEFAULT*, *source_address=None*, *, *all_errors=False*)

Connect to a TCP service listening on the internet *address* (a 2-tuple `(host, port)`), and return the socket object. This is a higher-level function than *socket.connect()*: if *host* is a non-numeric hostname, it will try to resolve it for both *AF_INET* and *AF_INET6*, and then try to connect to all possible addresses in turn until a connection succeeds. This makes it easy to write clients that are compatible to both IPv4 and IPv6.

Passing the optional *timeout* parameter will set the timeout on the socket instance before attempting to connect. If no *timeout* is supplied, the global default timeout setting returned by *getdefaulttimeout()* is used.

If supplied, *source_address* must be a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting. If host or port are '' or 0 respectively the OS default behavior will be used.

When a connection cannot be created, an exception is raised. By default, it is the exception from the last address in the list. If *all_errors* is `True`, it is an *ExceptionGroup* containing the errors of all attempts.

Changed in version 3.2: *source_address* was added.

Changed in version 3.11: *all_errors* was added.

`socket.`**`create_server`**(*address*, *, *family=AF_INET*, *backlog=None*, *reuse_port=False*, *dualstack_ipv6=False*)

Convenience function which creates a TCP socket bound to *address* (a 2-tuple `(host, port)`) and returns the socket object.

family should be either *AF_INET* or *AF_INET6*. *backlog* is the queue size passed to *socket.listen()*; if not specified , a default reasonable value is chosen. *reuse_port* dictates whether to set the `SO_REUSEPORT` socket option.

If *dualstack_ipv6* is true and the platform supports it the socket will be able to accept both IPv4 and IPv6 connections, else it will raise *ValueError*. Most POSIX platforms and Windows are supposed to support this functionality. When this functionality is enabled the address returned by *socket.getpeername()* when an IPv4 connection occurs will be an IPv6 address represented as an IPv4-mapped IPv6 address. If *dualstack_ipv6* is false it will explicitly disable this functionality on platforms that enable it by default (e.g. Linux). This parameter can be used in conjunction with *has_dualstack_ipv6()*:

```
import socket

addr = ("", 8080)  # all interfaces, port 8080
if socket.has_dualstack_ipv6():
    s = socket.create_server(addr, family=socket.AF_INET6, dualstack_ipv6=True)
else:
    s = socket.create_server(addr)
```

Note

On POSIX platforms the SO_REUSEADDR socket option is set in order to immediately reuse previous sockets which were bound on the same *address* and remained in TIME_WAIT state.

Added in version 3.8.

socket.**has_dualstack_ipv6**()

Return True if the platform supports creating a TCP socket which can handle both IPv4 and IPv6 connections.

Added in version 3.8.

socket.**fromfd**(*fd*, *family*, *type*, *proto=0*)

Duplicate the file descriptor *fd* (an integer as returned by a file object's *fileno()* method) and build a socket object from the result. Address family, socket type and protocol number are as for the *socket()* function above. The file descriptor should refer to a socket, but this is not checked — subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (such as a server started by the Unix inet daemon). The socket is assumed to be in blocking mode.

The newly created socket is *non-inheritable*.

Changed in version 3.4: The returned socket is now non-inheritable.

socket.**fromshare**(*data*)

Instantiate a socket from data obtained from the *socket.share()* method. The socket is assumed to be in blocking mode.

Availability: Windows.

Added in version 3.3.

socket.**SocketType**

This is a Python type object that represents the socket object type. It is the same as type(socket(...)).

Other functions

The *socket* module also offers various network-related services:

socket.**close**(*fd*)

Close a socket file descriptor. This is like *os.close()*, but for sockets. On some platforms (most noticeable Windows) *os.close()* does not work for socket file descriptors.

Added in version 3.7.

socket.**getaddrinfo**(*host*, *port*, *family=0*, *type=0*, *proto=0*, *flags=0*)

Translate the *host*/*port* argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. *host* is a domain name, a string representation of an IPv4/v6 address or None.

port is a string service name such as `'http'`, a numeric port number or `None`. By passing `None` as the value of *host* and *port*, you can pass `NULL` to the underlying C API.

The *family*, *type* and *proto* arguments can be optionally specified in order to narrow the list of addresses returned. Passing zero as a value for each of these arguments selects the full range of results. The *flags* argument can be one or several of the `AI_*` constants, and will influence how results are computed and returned. For example, `AI_NUMERICHOST` will disable domain name resolution and will raise an error if *host* is a domain name.

The function returns a list of 5-tuples with the following structure:

```
(family, type, proto, canonname, sockaddr)
```

In these tuples, *family*, *type*, *proto* are all integers and are meant to be passed to the *socket()* function. *canonname* will be a string representing the canonical name of the *host* if `AI_CANONNAME` is part of the *flags* argument; else *canonname* will be empty. *sockaddr* is a tuple describing a socket address, whose format depends on the returned *family* (a `(address, port)` 2-tuple for *AF_INET*, a `(address, port, flowinfo, scope_id)` 4-tuple for *AF_INET6*), and is meant to be passed to the *socket.connect()* method.

Raises an *auditing event* `socket.getaddrinfo` with arguments `host`, `port`, `family`, `type`, `protocol`.

The following example fetches address information for a hypothetical TCP connection to `example.org` on port 80 (results may differ on your system if IPv6 isn't enabled):

```
>>> socket.getaddrinfo("example.org", 80, proto=socket.IPPROTO_TCP)
[(socket.AF_INET6, socket.SOCK_STREAM,
 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (socket.AF_INET, socket.SOCK_STREAM,
 6, '', ('93.184.216.34', 80))]
```

Changed in version 3.2: parameters can now be passed using keyword arguments.

Changed in version 3.7: for IPv6 multicast addresses, string representing an address will not contain `%scope_id` part.

`socket.`**`getfqdn`**`(`[*name*]`)`

Return a fully qualified domain name for *name*. If *name* is omitted or empty, it is interpreted as the local host. To find the fully qualified name, the hostname returned by *gethostbyaddr()* is checked, followed by aliases for the host, if available. The first name which includes a period is selected. In case no fully qualified domain name is available and *name* was provided, it is returned unchanged. If *name* was empty or equal to `'0.0.0.0'`, the hostname from *gethostname()* is returned.

`socket.`**`gethostbyname`**`(`*hostname*`)`

Translate a host name to IPv4 address format. The IPv4 address is returned as a string, such as `'100.50.200.5'`. If the host name is an IPv4 address itself it is returned unchanged. See *gethostbyname_ex()* for a more complete interface. *gethostbyname()* does not support IPv6 name resolution, and *getaddrinfo()* should be used instead for IPv4/v6 dual stack support.

Raises an *auditing event* `socket.gethostbyname` with argument `hostname`.

Availability: not WASI.

`socket.`**`gethostbyname_ex`**`(`*hostname*`)`

Translate a host name to IPv4 address format, extended interface. Return a 3-tuple `(hostname, aliaslist, ipaddrlist)` where *hostname* is the host's primary host name, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4 addresses for the same interface on the same host (often but not always a single address). *gethostbyname_ex()* does not support IPv6 name resolution, and *getaddrinfo()* should be used instead for IPv4/v6 dual stack support.

Raises an *auditing event* `socket.gethostbyname` with argument `hostname`.

Availability: not WASI.

socket.**gethostname**()

Return a string containing the hostname of the machine where the Python interpreter is currently executing.

Raises an *auditing event* `socket.gethostname` with no arguments.

Note: *gethostname()* doesn't always return the fully qualified domain name; use *getfqdn()* for that.

Availability: not WASI.

socket.**gethostbyaddr**(*ip_address*)

Return a 3-tuple `(hostname, aliaslist, ipaddrlist)` where *hostname* is the primary host name responding to the given *ip_address*, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4/v6 addresses for the same interface on the same host (most likely containing only a single address). To find the fully qualified domain name, use the function *getfqdn()*. *gethostbyaddr()* supports both IPv4 and IPv6.

Raises an *auditing event* `socket.gethostbyaddr` with argument `ip_address`.

Availability: not WASI.

socket.**getnameinfo**(*sockaddr*, *flags*)

Translate a socket address *sockaddr* into a 2-tuple `(host, port)`. Depending on the settings of *flags*, the result can contain a fully qualified domain name or numeric address representation in *host*. Similarly, *port* can contain a string port name or a numeric port number.

For IPv6 addresses, `%scope_id` is appended to the host part if *sockaddr* contains meaningful *scope_id*. Usually this happens for multicast addresses.

For more information about *flags* you can consult *getnameinfo(3)*.

Raises an *auditing event* `socket.getnameinfo` with argument `sockaddr`.

Availability: not WASI.

socket.**getprotobyname**(*protocolname*)

Translate an internet protocol name (for example, `'icmp'`) to a constant suitable for passing as the (optional) third argument to the *socket()* function. This is usually only needed for sockets opened in "raw" mode (*SOCK_RAW*); for the normal socket modes, the correct protocol is chosen automatically if the protocol is omitted or zero.

Availability: not WASI.

socket.**getservbyname**(*servicename*[, *protocolname*])

Translate an internet service name and protocol name to a port number for that service. The optional protocol name, if given, should be `'tcp'` or `'udp'`, otherwise any protocol will match.

Raises an *auditing event* `socket.getservbyname` with arguments `servicename`, `protocolname`.

Availability: not WASI.

socket.**getservbyport**(*port*[, *protocolname*])

Translate an internet port number and protocol name to a service name for that service. The optional protocol name, if given, should be `'tcp'` or `'udp'`, otherwise any protocol will match.

Raises an *auditing event* `socket.getservbyport` with arguments `port`, `protocolname`.

Availability: not WASI.

socket.**ntohl**(*x*)

Convert 32-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.**ntohs**(*x*)

Convert 16-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

Changed in version 3.10: Raises *OverflowError* if *x* does not fit in a 16-bit unsigned integer.

socket.**htonl**(*x*)

Convert 32-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.**htons**(*x*)

Convert 16-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

Changed in version 3.10: Raises *OverflowError* if *x* does not fit in a 16-bit unsigned integer.

socket.**inet_aton**(*ip_string*)

Convert an IPv4 address from dotted-quad string format (for example, '123.45.67.89') to 32-bit packed binary format, as a bytes object four characters in length. This is useful when conversing with a program that uses the standard C library and needs objects of type `in_addr`, which is the C type for the 32-bit packed binary this function returns.

inet_aton() also accepts strings with less than three dots; see the Unix manual page *inet(3)* for details.

If the IPv4 address string passed to this function is invalid, *OSError* will be raised. Note that exactly what is valid depends on the underlying C implementation of `inet_aton()`.

inet_aton() does not support IPv6, and *inet_pton()* should be used instead for IPv4/v6 dual stack support.

socket.**inet_ntoa**(*packed_ip*)

Convert a 32-bit packed IPv4 address (a *bytes-like object* four bytes in length) to its standard dotted-quad string representation (for example, '123.45.67.89'). This is useful when conversing with a program that uses the standard C library and needs objects of type `in_addr`, which is the C type for the 32-bit packed binary data this function takes as an argument.

If the byte sequence passed to this function is not exactly 4 bytes in length, *OSError* will be raised. *inet_ntoa()* does not support IPv6, and *inet_ntop()* should be used instead for IPv4/v6 dual stack support.

Changed in version 3.5: Writable *bytes-like object* is now accepted.

socket.**inet_pton**(*address_family*, *ip_string*)

Convert an IP address from its family-specific string format to a packed, binary format. *inet_pton()* is useful when a library or network protocol calls for an object of type `in_addr` (similar to *inet_aton()*) or `in6_addr`.

Supported values for *address_family* are currently *AF_INET* and *AF_INET6*. If the IP address string *ip_string* is invalid, *OSError* will be raised. Note that exactly what is valid depends on both the value of *address_family* and the underlying implementation of `inet_pton()`.

Availability: Unix, Windows.

Changed in version 3.4: Windows support added

socket.**inet_ntop**(*address_family*, *packed_ip*)

Convert a packed IP address (a *bytes-like object* of some number of bytes) to its standard, family-specific string representation (for example, `'7.10.0.5'` or `'5aef:2b::8'`). *inet_ntop()* is useful when a library or network protocol returns an object of type `in_addr` (similar to *inet_ntoa()*) or `in6_addr`.

Supported values for *address_family* are currently *AF_INET* and *AF_INET6*. If the bytes object *packed_ip* is not the correct length for the specified address family, *ValueError* will be raised. *OSError* is raised for errors from the call to *inet_ntop()*.

Availability: Unix, Windows.

Changed in version 3.4: Windows support added

Changed in version 3.5: Writable *bytes-like object* is now accepted.

socket.**CMSG_LEN**(*length*)

Return the total length, without trailing padding, of an ancillary data item with associated data of the given *length*. This value can often be used as the buffer size for *recvmsg()* to receive a single item of ancillary data, but **RFC 3542** requires portable applications to use *CMSG_SPACE()* and thus include space for padding, even when the item will be the last in the buffer. Raises *OverflowError* if *length* is outside the permissible range of values.

Availability: Unix, not WASI.

Most Unix platforms.

Added in version 3.3.

socket.**CMSG_SPACE**(*length*)

Return the buffer size needed for *recvmsg()* to receive an ancillary data item with associated data of the given *length*, along with any trailing padding. The buffer space needed to receive multiple items is the sum of the *CMSG_SPACE()* values for their associated data lengths. Raises *OverflowError* if *length* is outside the permissible range of values.

Note that some systems might support ancillary data without providing this function. Also note that setting the buffer size using the results of this function may not precisely limit the amount of ancillary data that can be received, since additional data may be able to fit into the padding area.

Availability: Unix, not WASI.

most Unix platforms.

Added in version 3.3.

socket.**getdefaulttimeout**()

Return the default timeout in seconds (float) for new socket objects. A value of `None` indicates that new socket objects have no timeout. When the socket module is first imported, the default is `None`.

socket.**setdefaulttimeout**(*timeout*)

Set the default timeout in seconds (float) for new socket objects. When the socket module is first imported, the default is `None`. See *settimeout()* for possible values and their respective meanings.

socket.**sethostname**(*name*)

Set the machine's hostname to *name*. This will raise an *OSError* if you don't have enough rights.

Raises an *auditing event* `socket.sethostname` with argument `name`.

Availability: Unix.

Added in version 3.3.

socket.**if_nameindex**()

Return a list of network interface information (index int, name string) tuples. *OSError* if the system call fails.

Availability: Unix, Windows, not WASI.

Added in version 3.3.

Changed in version 3.8: Windows support was added.

Note

On Windows network interfaces have different names in different contexts (all names are examples):

- UUID: `{FB605B73-AAC2-49A6-9A2F-25416AEA0573}`
- name: `ethernet_32770`
- friendly name: `vEthernet (nat)`
- description: `Hyper-V Virtual Ethernet Adapter`

This function returns names of the second form from the list, `ethernet_32770` in this example case.

`socket.`**`if_nametoindex`**(*if_name*)

Return a network interface index number corresponding to an interface name. *OSError* if no interface with the given name exists.

Availability: Unix, Windows, not WASI.

Added in version 3.3.

Changed in version 3.8: Windows support was added.

See also

"Interface name" is a name as documented in *if_nameindex()*.

`socket.`**`if_indextoname`**(*if_index*)

Return a network interface name corresponding to an interface index number. *OSError* if no interface with the given index exists.

Availability: Unix, Windows, not WASI.

Added in version 3.3.

Changed in version 3.8: Windows support was added.

See also

"Interface name" is a name as documented in *if_nameindex()*.

`socket.`**`send_fds`**(*sock*, *buffers*, *fds*[, *flags*[, *address*]])

Send the list of file descriptors *fds* over an *AF_UNIX* socket *sock*. The *fds* parameter is a sequence of file descriptors. Consult *sendmsg()* for the documentation of these parameters.

Availability: Unix, Windows, not WASI.

Unix platforms supporting *sendmsg()* and `SCM_RIGHTS` mechanism.

Added in version 3.9.

`socket.`**`recv_fds`**(*sock*, *bufsize*, *maxfds*[, *flags*])

Receive up to *maxfds* file descriptors from an *AF_UNIX* socket *sock*. Return `(msg, list(fds), flags, addr)`. Consult *recvmsg()* for the documentation of these parameters.

Availability: Unix, Windows, not WASI.

Unix platforms supporting *sendmsg()* and `SCM_RIGHTS` mechanism.

Added in version 3.9.

> **Note**
>
> Any truncated integers at the end of the list of file descriptors.

18.2.3 Socket Objects

Socket objects have the following methods. Except for `makefile()`, these correspond to Unix system calls applicable to sockets.

Changed in version 3.2: Support for the *context manager* protocol was added. Exiting the context manager is equivalent to calling `close()`.

`socket.`**`accept`**`()`

Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair `(conn, address)` where *conn* is a *new* socket object usable to send and receive data on the connection, and *address* is the address bound to the socket on the other end of the connection.

The newly created socket is *non-inheritable*.

Changed in version 3.4: The socket is now non-inheritable.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** for the rationale).

`socket.`**`bind`**`(`*address*`)`

Bind the socket to *address*. The socket must not already be bound. (The format of *address* depends on the address family — see above.)

Raises an *auditing event* `socket.bind` with arguments `self`, `address`.

Availability: not WASI.

`socket.`**`close`**`()`

Mark the socket closed. The underlying system resource (e.g. a file descriptor) is also closed when all file objects from `makefile()` are closed. Once that happens, all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed).

Sockets are automatically closed when they are garbage-collected, but it is recommended to `close()` them explicitly, or to use a `with` statement around them.

Changed in version 3.6: `OSError` is now raised if an error occurs when the underlying `close()` call is made.

> **Note**
>
> `close()` releases the resource associated with a connection but does not necessarily close the connection immediately. If you want to close the connection in a timely fashion, call `shutdown()` before `close()`.

`socket.`**`connect`**`(`*address*`)`

Connect to a remote socket at *address*. (The format of *address* depends on the address family — see above.)

If the connection is interrupted by a signal, the method waits until the connection completes, or raise a `TimeoutError` on timeout, if the signal handler doesn't raise an exception and the socket is blocking or has a timeout.

For non-blocking sockets, the method raises an *InterruptedError* exception if the connection is interrupted by a signal (or the exception raised by the signal handler).

Raises an *auditing event* `socket.connect` with arguments `self`, `address`.

Changed in version 3.5: The method now waits until the connection completes instead of raising an *InterruptedError* exception if the connection is interrupted by a signal, the signal handler doesn't raise an exception and the socket is blocking or has a timeout (see the **PEP 475** for the rationale).

Availability: not WASI.

`socket.`**`connect_ex`**`(`*address*`)`

Like `connect(address)`, but return an error indicator instead of raising an exception for errors returned by the C-level `connect()` call (other problems, such as "host not found," can still raise exceptions). The error indicator is `0` if the operation succeeded, otherwise the value of the `errno` variable. This is useful to support, for example, asynchronous connects.

Raises an *auditing event* `socket.connect` with arguments `self`, `address`.

Availability: not WASI.

`socket.`**`detach`**`()`

Put the socket object into closed state without actually closing the underlying file descriptor. The file descriptor is returned, and can be reused for other purposes.

Added in version 3.2.

`socket.`**`dup`**`()`

Duplicate the socket.

The newly created socket is *non-inheritable*.

Changed in version 3.4: The socket is now non-inheritable.

Availability: not WASI.

`socket.`**`fileno`**`()`

Return the socket's file descriptor (a small integer), or -1 on failure. This is useful with *select.select()*.

Under Windows the small integer returned by this method cannot be used where a file descriptor can be used (such as *os.fdopen()*). Unix does not have this limitation.

`socket.`**`get_inheritable`**`()`

Get the *inheritable flag* of the socket's file descriptor or socket's handle: `True` if the socket can be inherited in child processes, `False` if it cannot.

Added in version 3.4.

`socket.`**`getpeername`**`()`

Return the remote address to which the socket is connected. This is useful to find out the port number of a remote IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.) On some systems this function is not supported.

`socket.`**`getsockname`**`()`

Return the socket's own address. This is useful to find out the port number of an IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.)

`socket.`**`getsockopt`**`(`*level*, *optname*[, *buflen*]`)`

Return the value of the given socket option (see the Unix man page *getsockopt(2)*). The needed symbolic constants (*SO_* etc.*) are defined in this module. If *buflen* is absent, an integer option is assumed and its integer value is returned by the function. If *buflen* is present, it specifies the maximum length of the buffer used to receive

the option in, and this buffer is returned as a bytes object. It is up to the caller to decode the contents of the buffer (see the optional built-in module *struct* for a way to decode C structures encoded as byte strings).

Availability: not WASI.

socket.**getblocking**()

Return `True` if socket is in blocking mode, `False` if in non-blocking.

This is equivalent to checking `socket.gettimeout() != 0`.

Added in version 3.7.

socket.**gettimeout**()

Return the timeout in seconds (float) associated with socket operations, or `None` if no timeout is set. This reflects the last call to *setblocking()* or *settimeout()*.

socket.**ioctl**(*control*, *option*)

Platform
Windows

The *ioctl()* method is a limited interface to the WSAIoctl system interface. Please refer to the Win32 documentation for more information.

On other platforms, the generic *fcntl.fcntl()* and *fcntl.ioctl()* functions may be used; they accept a socket object as their first argument.

Currently only the following control codes are supported: `SIO_RCVALL`, `SIO_KEEPALIVE_VALS`, and `SIO_LOOPBACK_FAST_PATH`.

Changed in version 3.6: `SIO_LOOPBACK_FAST_PATH` was added.

socket.**listen**([*backlog*])

Enable a server to accept connections. If *backlog* is specified, it must be at least 0 (if it is lower, it is set to 0); it specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.

Availability: not WASI.

Changed in version 3.5: The *backlog* parameter is now optional.

socket.**makefile**(*mode='r'*, *buffering=None*, ***, *encoding=None*, *errors=None*, *newline=None*)

Return a *file object* associated with the socket. The exact returned type depends on the arguments given to *makefile()*. These arguments are interpreted the same way as by the built-in *open()* function, except the only supported *mode* values are `'r'` (default), `'w'`, `'b'`, or a combination of those.

The socket must be in blocking mode; it can have a timeout, but the file object's internal buffer may end up in an inconsistent state if a timeout occurs.

Closing the file object returned by *makefile()* won't close the original socket unless all other file objects have been closed and *socket.close()* has been called on the socket object.

Note

On Windows, the file-like object created by *makefile()* cannot be used where a file object with a file descriptor is expected, such as the stream arguments of *subprocess.Popen()*.

socket.**recv**(*bufsize*[, *flags*])

Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by *bufsize*. A returned empty bytes object indicates that the client

has disconnected. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero.

> **Note**
>
> For best match with hardware and network realities, the value of *bufsize* should be a relatively small power of 2, for example, 4096.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

`socket.`**`recvfrom`**(*bufsize*[, *flags*])

Receive data from the socket. The return value is a pair `(bytes, address)` where *bytes* is a bytes object representing the data received and *address* is the address of the socket sending the data. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family — see above.)

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

Changed in version 3.7: For multicast IPv6 address, first item of *address* does not contain `%scope_id` part anymore. In order to get full IPv6 address use *getnameinfo()*.

`socket.`**`recvmsg`**(*bufsize*[, *ancbufsize*[, *flags*]])

Receive normal data (up to *bufsize* bytes) and ancillary data from the socket. The *ancbufsize* argument sets the size in bytes of the internal buffer used to receive the ancillary data; it defaults to 0, meaning that no ancillary data will be received. Appropriate buffer sizes for ancillary data can be calculated using *CMSG_SPACE()* or *CMSG_LEN()*, and items which do not fit into the buffer might be truncated or discarded. The *flags* argument defaults to 0 and has the same meaning as for *recv()*.

The return value is a 4-tuple: `(data, ancdata, msg_flags, address)`. The *data* item is a *bytes* object holding the non-ancillary data received. The *ancdata* item is a list of zero or more tuples `(cmsg_level, cmsg_type, cmsg_data)` representing the ancillary data (control messages) received: *cmsg_level* and *cmsg_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg_data* is a *bytes* object holding the associated data. The *msg_flags* item is the bitwise OR of various flags indicating conditions on the received message; see your system documentation for details. If the receiving socket is unconnected, *address* is the address of the sending socket, if available; otherwise, its value is unspecified.

On some systems, *sendmsg()* and *recvmsg()* can be used to pass file descriptors between processes over an *AF_UNIX* socket. When this facility is used (it is often restricted to *SOCK_STREAM* sockets), *recvmsg()* will return, in its ancillary data, items of the form `(socket.SOL_SOCKET, socket.SCM_RIGHTS, fds)`, where *fds* is a *bytes* object representing the new file descriptors as a binary array of the native C `int` type. If *recvmsg()* raises an exception after the system call returns, it will first attempt to close any file descriptors received via this mechanism.

Some systems do not indicate the truncated length of ancillary data items which have been only partially received. If an item appears to extend beyond the end of the buffer, *recvmsg()* will issue a *RuntimeWarning*, and will return the part of it which is inside the buffer provided it has not been truncated before the start of its associated data.

On systems which support the `SCM_RIGHTS` mechanism, the following function will receive up to *maxfds* file descriptors, returning the message data and a list containing the descriptors (while ignoring unexpected conditions such as unrelated control messages being received). See also *sendmsg()*.

```
import socket, array

def recv_fds(sock, msglen, maxfds):
    fds = array.array("i")   # Array of ints
    msg, ancdata, flags, addr = sock.recvmsg(msglen, socket.CMSG_LEN(maxfds * fds.
↪itemsize))
    for cmsg_level, cmsg_type, cmsg_data in ancdata:
        if cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS:
            # Append data, ignoring any truncated integers at the end.
            fds.frombytes(cmsg_data[:len(cmsg_data) - (len(cmsg_data) % fds.
↪itemsize)])
    return msg, list(fds)
```

Availability: Unix.

Most Unix platforms.

Added in version 3.3.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** for the rationale).

`socket.`**`recvmsg_into`**(*buffers*[, *ancbufsize*[, *flags*]])

Receive normal data and ancillary data from the socket, behaving as `recvmsg()` would, but scatter the non-ancillary data into a series of buffers instead of returning a new bytes object. The *buffers* argument must be an iterable of objects that export writable buffers (e.g. `bytearray` objects); these will be filled with successive chunks of the non-ancillary data until it has all been written or there are no more buffers. The operating system may set a limit (`sysconf()` value `SC_IOV_MAX`) on the number of buffers that can be used. The *ancbufsize* and *flags* arguments have the same meaning as for `recvmsg()`.

The return value is a 4-tuple: `(nbytes, ancdata, msg_flags, address)`, where *nbytes* is the total number of bytes of non-ancillary data written into the buffers, and *ancdata*, *msg_flags* and *address* are the same as for `recvmsg()`.

Example:

```
>>> import socket
>>> s1, s2 = socket.socketpair()
>>> b1 = bytearray(b'----')
>>> b2 = bytearray(b'0123456789')
>>> b3 = bytearray(b'--------------')
>>> s1.send(b'Mary had a little lamb')
22
>>> s2.recvmsg_into([b1, memoryview(b2)[2:9], b3])
(22, [], 0, None)
>>> [b1, b2, b3]
[bytearray(b'Mary'), bytearray(b'01 had a 9'), bytearray(b'little lamb---')]
```

Availability: Unix.

Most Unix platforms.

Added in version 3.3.

`socket.`**`recvfrom_into`**(*buffer*[, *nbytes*[, *flags*]])

Receive data from the socket, writing it into *buffer* instead of creating a new bytestring. The return value is a pair `(nbytes, address)` where *nbytes* is the number of bytes received and *address* is the address of the socket sending the data. See the Unix manual page `recv(2)` for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family — see above.)

socket.**recv_into**(*buffer*[, *nbytes*[, *flags*]])

Receive up to *nbytes* bytes from the socket, storing the data into a buffer rather than creating a new bytestring. If *nbytes* is not specified (or 0), receive up to the size available in the given buffer. Returns the number of bytes received. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero.

socket.**send**(*bytes*[, *flags*])

Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for *recv()* above. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data. For further information on this topic, consult the socket-howto.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.**sendall**(*bytes*[, *flags*])

Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for *recv()* above. Unlike *send()*, this method continues to send data from *bytes* until either all data has been sent or an error occurs. `None` is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

Changed in version 3.5: The socket timeout is no longer reset each time data is sent successfully. The socket timeout is now the maximum total duration to send all data.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.**sendto**(*bytes*, *address*)

socket.**sendto**(*bytes*, *flags*, *address*)

Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*. The optional *flags* argument has the same meaning as for *recv()* above. Return the number of bytes sent. (The format of *address* depends on the address family — see above.)

Raises an *auditing event* `socket.sendto` with arguments `self`, `address`.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.**sendmsg**(*buffers*[, *ancdata*[, *flags*[, *address*]]])

Send normal and ancillary data to the socket, gathering the non-ancillary data from a series of buffers and concatenating it into a single message. The *buffers* argument specifies the non-ancillary data as an iterable of *bytes-like objects* (e.g. *bytes* objects); the operating system may set a limit (*sysconf()* value `SC_IOV_MAX`) on the number of buffers that can be used. The *ancdata* argument specifies the ancillary data (control messages) as an iterable of zero or more tuples `(cmsg_level, cmsg_type, cmsg_data)`, where *cmsg_level* and *cmsg_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg_data* is a bytes-like object holding the associated data. Note that some systems (in particular, systems without *CMSG_SPACE()*) might support sending only one control message per call. The *flags* argument defaults to 0 and has the same meaning as for *send()*. If *address* is supplied and not `None`, it sets a destination address for the message. The return value is the number of bytes of non-ancillary data sent.

The following function sends the list of file descriptors *fds* over an *AF_UNIX* socket, on systems which support the `SCM_RIGHTS` mechanism. See also *recvmsg()*.

```
import socket, array
```

(continues on next page)

(continued from previous page)

```
def send_fds(sock, msg, fds):
    return sock.sendmsg([msg], [(socket.SOL_SOCKET, socket.SCM_RIGHTS, array.
↪array("i", fds))])
```

Availability: Unix, not WASI.

Most Unix platforms.

Raises an *auditing event* `socket.sendmsg` with arguments `self`, `address`.

Added in version 3.3.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** for the rationale).

`socket.`**`sendmsg_afalg`**(*[msg,]*, op[, iv[, assoclen[, flags]]]*)

Specialized version of `sendmsg()` for `AF_ALG` socket. Set mode, IV, AEAD associated data length and flags for `AF_ALG` socket.

Availability: Linux >= 2.6.38.

Added in version 3.6.

`socket.`**`sendfile`**(*file*, *offset=0*, *count=None*)

Send a file until EOF is reached by using high-performance `os.sendfile` and return the total number of bytes which were sent. *file* must be a regular file object opened in binary mode. If `os.sendfile` is not available (e.g. Windows) or *file* is not a regular file `send()` will be used instead. *offset* tells from where to start reading the file. If specified, *count* is the total number of bytes to transmit as opposed to sending the file until EOF is reached. File position is updated on return or also in case of error in which case `file.tell()` can be used to figure out the number of bytes which were sent. The socket must be of `SOCK_STREAM` type. Non-blocking sockets are not supported.

Added in version 3.5.

`socket.`**`set_inheritable`**(*inheritable*)

Set the *inheritable flag* of the socket's file descriptor or socket's handle.

Added in version 3.4.

`socket.`**`setblocking`**(*flag*)

Set blocking or non-blocking mode of the socket: if *flag* is false, the socket is set to non-blocking, else to blocking mode.

This method is a shorthand for certain `settimeout()` calls:

- `sock.setblocking(True)` is equivalent to `sock.settimeout(None)`
- `sock.setblocking(False)` is equivalent to `sock.settimeout(0.0)`

Changed in version 3.7: The method no longer applies `SOCK_NONBLOCK` flag on `socket.type`.

`socket.`**`settimeout`**(*value*)

Set a timeout on blocking socket operations. The *value* argument can be a nonnegative floating-point number expressing seconds, or `None`. If a non-zero value is given, subsequent socket operations will raise a `timeout` exception if the timeout period *value* has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If `None` is given, the socket is put in blocking mode.

For further information, please consult the *notes on socket timeouts*.

Changed in version 3.7: The method no longer toggles `SOCK_NONBLOCK` flag on `socket.type`.

socket.**setsockopt**(*level*, *optname*, *value:* int)

socket.**setsockopt**(*level*, *optname*, *value: buffer*)

socket.**setsockopt**(*level*, *optname*, *None*, *optlen: int*)

Set the value of the given socket option (see the Unix manual page *setsockopt(2)*). The needed symbolic constants are defined in this module (SO_* etc. <socket-unix-constants>). The value can be an integer, None or a *bytes-like object* representing a buffer. In the later case it is up to the caller to ensure that the bytestring contains the proper bits (see the optional built-in module *struct* for a way to encode C structures as bytestrings). When *value* is set to None, *optlen* argument is required. It's equivalent to call setsockopt() C function with optval=NULL and optlen=optlen.

Changed in version 3.5: Writable *bytes-like object* is now accepted.

Changed in version 3.6: setsockopt(level, optname, None, optlen: int) form added.

Availability: not WASI.

socket.**shutdown**(*how*)

Shut down one or both halves of the connection. If *how* is SHUT_RD, further receives are disallowed. If *how* is SHUT_WR, further sends are disallowed. If *how* is SHUT_RDWR, further sends and receives are disallowed.

Availability: not WASI.

socket.**share**(*process_id*)

Duplicate a socket and prepare it for sharing with a target process. The target process must be provided with *process_id*. The resulting bytes object can then be passed to the target process using some form of interprocess communication and the socket can be recreated there using *fromshare()*. Once this method has been called, it is safe to close the socket since the operating system has already duplicated it for the target process.

Availability: Windows.

Added in version 3.3.

Note that there are no methods read() or write(); use *recv()* and *send()* without *flags* argument instead.

Socket objects also have these (read-only) attributes that correspond to the values given to the *socket* constructor.

socket.**family**

The socket family.

socket.**type**

The socket type.

socket.**proto**

The socket protocol.

18.2.4 Notes on socket timeouts

A socket object can be in one of three modes: blocking, non-blocking, or timeout. Sockets are by default always created in blocking mode, but this can be changed by calling *setdefaulttimeout()*.

- In *blocking mode*, operations block until complete or the system returns an error (such as connection timed out).
- In *non-blocking mode*, operations fail (with an error that is unfortunately system-dependent) if they cannot be completed immediately: functions from the *select* module can be used to know when and whether a socket is available for reading or writing.
- In *timeout mode*, operations fail if they cannot be completed within the timeout specified for the socket (they raise a *timeout* exception) or if the system returns an error.

> **Note**
>
> At the operating system level, sockets in *timeout mode* are internally set in non-blocking mode. Also, the blocking and timeout modes are shared between file descriptors and socket objects that refer to the same network endpoint. This implementation detail can have visible consequences if e.g. you decide to use the `fileno()` of a socket.

Timeouts and the `connect` method

The `connect()` operation is also subject to the timeout setting, and in general it is recommended to call `settimeout()` before calling `connect()` or pass a timeout parameter to `create_connection()`. However, the system network stack may also return a connection timeout error of its own regardless of any Python socket timeout setting.

Timeouts and the `accept` method

If `getdefaulttimeout()` is not `None`, sockets returned by the `accept()` method inherit that timeout. Otherwise, the behaviour depends on settings of the listening socket:

- if the listening socket is in *blocking mode* or in *timeout mode*, the socket returned by `accept()` is in *blocking mode*;
- if the listening socket is in *non-blocking mode*, whether the socket returned by `accept()` is in blocking or non-blocking mode is operating system-dependent. If you want to ensure cross-platform behaviour, it is recommended you manually override this setting.

18.2.5 Example

Here are four minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence `socket()`, `bind()`, `listen()`, `accept()` (possibly repeating the `accept()` to service more than one client), while a client only needs the sequence `socket()`, `connect()`. Also note that the server does not `sendall()`/`recv()` on the socket it is listening on but on the new socket returned by `accept()`.

The first two examples support IPv4 only.

```
# Echo server program
import socket

HOST = ''                 # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data: break
            conn.sendall(data)
```

```
# Echo client program
import socket
```

(continues on next page)

(continued from previous page)

```
HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to connect to all the addresses returned as a result of the name resolution, and sends traffic to the first one connected successfully.

```
# Echo server program
import socket
import sys

HOST = None               # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data: break
        conn.send(data)
```

```
# Echo client program
import socket
import sys

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
```

(continues on next page)

(continued from previous page)

```
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
with s:
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

The next example shows how to write a very simple network sniffer with raw sockets on Windows. The example requires administrator privileges to modify the interface:

```
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packets
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a packet
print(s.recvfrom(65565))

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

The next example shows how to use the socket interface to communicate to a CAN network using the raw socket protocol. To use CAN with the broadcast manager protocol instead, open a socket with:

```
socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_BCM)
```

After binding (`CAN_RAW`) or connecting (*`CAN_BCM`*) the socket, you can use the *`socket.send()`* and *`socket.recv()`* operations (and their counterparts) on the socket object as usual.

This last example might require special privileges:

```
import socket
import struct
```

(continues on next page)

(continued from previous page)

```
# CAN frame packing/unpacking (see 'struct can_frame' in <linux/can.h>)

can_frame_fmt = "=IB3x8s"
can_frame_size = struct.calcsize(can_frame_fmt)

def build_can_frame(can_id, data):
    can_dlc = len(data)
    data = data.ljust(8, b'\x00')
    return struct.pack(can_frame_fmt, can_id, can_dlc, data)

def dissect_can_frame(frame):
    can_id, can_dlc, data = struct.unpack(can_frame_fmt, frame)
    return (can_id, can_dlc, data[:can_dlc])


# create a raw socket and bind it to the 'vcan0' interface
s = socket.socket(socket.AF_CAN, socket.SOCK_RAW, socket.CAN_RAW)
s.bind(('vcan0',))

while True:
    cf, addr = s.recvfrom(can_frame_size)

    print('Received: can_id=%x, can_dlc=%x, data=%s' % dissect_can_frame(cf))

    try:
        s.send(cf)
    except OSError:
        print('Error sending CAN frame')

    try:
        s.send(build_can_frame(0x01, b'\x01\x02\x03'))
    except OSError:
        print('Error sending CAN frame')
```

Running an example several times with too small delay between executions, could lead to this error:

```
OSError: [Errno 98] Address already in use
```

This is because the previous execution has left the socket in a `TIME_WAIT` state, and can't be immediately reused.

There is a *socket* flag to set, in order to prevent this, `socket.SO_REUSEADDR`:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
```

the `SO_REUSEADDR` flag tells the kernel to reuse a local socket in `TIME_WAIT` state, without waiting for its natural timeout to expire.

See also

For an introduction to socket programming (in C), see the following papers:

- *An Introductory 4.3BSD Interprocess Communication Tutorial*, by Stuart Sechrest

- *An Advanced 4.3BSD Interprocess Communication Tutorial*, by Samuel J. Leffler et al,

both in the UNIX Programmer's Manual, Supplementary Documents 1 (sections PS1:7 and PS1:8). The platform-specific reference material for the various socket-related system calls are also a valuable source of information on the details of socket semantics. For Unix, refer to the manual pages; for Windows, see the WinSock (or Winsock 2) specification. For IPv6-ready APIs, readers may want to refer to **RFC 3493** titled Basic Socket Interface Extensions for IPv6.

18.3 `ssl` — TLS/SSL wrapper for socket objects

Source code: Lib/ssl.py

This module provides access to Transport Layer Security (often known as "Secure Sockets Layer") encryption and peer authentication facilities for network sockets, both client-side and server-side. This module uses the OpenSSL library. It is available on all modern Unix systems, Windows, macOS, and probably additional platforms, as long as OpenSSL is installed on that platform.

Note

Some behavior may be platform dependent, since calls are made to the operating system socket APIs. The installed version of OpenSSL may also cause variations in behavior. For example, TLSv1.3 comes with OpenSSL version 1.1.1.

Warning

Don't use this module without reading the *Security considerations*. Doing so may lead to a false sense of security, as the default settings of the ssl module are not necessarily appropriate for your application.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

This section documents the objects and functions in the `ssl` module; for more general information about TLS, SSL, and certificates, the reader is referred to the documents in the "See Also" section at the bottom.

This module provides a class, *`ssl.SSLSocket`*, which is derived from the *`socket.socket`* type, and provides a socket-like wrapper that also encrypts and decrypts the data going over the socket with SSL. It supports additional methods such as `getpeercert()`, which retrieves the certificate of the other side of the connection, `cipher()`, which retrieves the cipher being used for the secure connection or `get_verified_chain()`, `get_unverified_chain()` which retrieves certificate chain.

For more sophisticated applications, the *`ssl.SSLContext`* class helps manage settings and certificates, which can then be inherited by SSL sockets created through the *`SSLContext.wrap_socket()`* method.

Changed in version 3.5.3: Updated to support linking with OpenSSL 1.1.0

Changed in version 3.6: OpenSSL 0.9.8, 1.0.0 and 1.0.1 are deprecated and no longer supported. In the future the ssl module will require at least OpenSSL 1.0.2 or 1.1.0.

Changed in version 3.10: **PEP 644** has been implemented. The ssl module requires OpenSSL 1.1.1 or newer.

Use of deprecated constants and functions result in deprecation warnings.

18.3.1 Functions, Constants, and Exceptions

Socket creation

Instances of *SSLSocket* must be created using the *SSLContext.wrap_socket()* method. The helper function *create_default_context()* returns a new context with secure default settings.

Client socket example with default context and IPv4/IPv6 dual stack:

```
import socket
import ssl

hostname = 'www.python.org'
context = ssl.create_default_context()

with socket.create_connection((hostname, 443)) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

Client socket example with custom context and IPv4:

```
hostname = 'www.python.org'
# PROTOCOL_TLS_CLIENT requires valid cert chain and hostname
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.load_verify_locations('path/to/cabundle.pem')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

Server socket example listening on localhost IPv4:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain('/path/to/certchain.pem', '/path/to/private.key')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    sock.bind(('127.0.0.1', 8443))
    sock.listen(5)
    with context.wrap_socket(sock, server_side=True) as ssock:
        conn, addr = ssock.accept()
        ...
```

Context creation

A convenience function helps create *SSLContext* objects for common purposes.

ssl.**create_default_context**(*purpose=Purpose.SERVER_AUTH*, *cafile=None*, *capath=None*, *cadata=None*)

Return a new *SSLContext* object with default settings for the given *purpose*. The settings are chosen by the *ssl* module, and usually represent a higher security level than when calling the *SSLContext* constructor directly.

cafile, *capath*, *cadata* represent optional CA certificates to trust for certificate verification, as in *SSLContext.load_verify_locations()*. If all three are *None*, this function can choose to trust the system's default CA certificates instead.

The settings are: *PROTOCOL_TLS_CLIENT* or *PROTOCOL_TLS_SERVER*, *OP_NO_SSLv2*, and *OP_NO_SSLv3* with high encryption cipher suites without RC4 and without unauthenticated cipher suites.

Passing *SERVER_AUTH* as *purpose* sets *verify_mode* to *CERT_REQUIRED* and either loads CA certificates (when at least one of *cafile*, *capath* or *cadata* is given) or uses *SSLContext.load_default_certs()* to load default CA certificates.

When *keylog_filename* is supported and the environment variable SSLKEYLOGFILE is set, *create_default_context()* enables key logging.

The default settings for this context include *VERIFY_X509_PARTIAL_CHAIN* and *VERIFY_X509_STRICT*. These make the underlying OpenSSL implementation behave more like a conforming implementation of **RFC 5280**, in exchange for a small amount of incompatibility with older X.509 certificates.

Note

The protocol, options, cipher and other settings may change to more restrictive values anytime without prior deprecation. The values represent a fair balance between compatibility and security.

If your application needs specific settings, you should create a *SSLContext* and apply the settings yourself.

Note

If you find that when certain older clients or servers attempt to connect with a *SSLContext* created by this function that they get an error stating "Protocol or cipher suite mismatch", it may be that they only support SSL3.0 which this function excludes using the *OP_NO_SSLv3*. SSL3.0 is widely considered to be completely broken. If you still wish to continue to use this function but still allow SSL 3.0 connections you can re-enable them using:

```
ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
ctx.options &= ~ssl.OP_NO_SSLv3
```

Note

This context enables *VERIFY_X509_STRICT* by default, which may reject pre-**RFC 5280** or malformed certificates that the underlying OpenSSL implementation otherwise would accept. While disabling this is not recommended, you can do so using:

```
ctx = ssl.create_default_context()
ctx.verify_flags &= ~ssl.VERIFY_X509_STRICT
```

Added in version 3.4.

Changed in version 3.4.4: RC4 was dropped from the default cipher string.

Changed in version 3.6: ChaCha20/Poly1305 was added to the default cipher string.

3DES was dropped from the default cipher string.

Changed in version 3.8: Support for key logging to SSLKEYLOGFILE was added.

Changed in version 3.10: The context now uses *PROTOCOL_TLS_CLIENT* or *PROTOCOL_TLS_SERVER* protocol instead of generic *PROTOCOL_TLS*.

Changed in version 3.13: The context now uses *VERIFY_X509_PARTIAL_CHAIN* and *VERIFY_X509_STRICT* in its default verify flags.

Exceptions

exception ssl.**SSLError**

Raised to signal an error from the underlying SSL implementation (currently provided by the OpenSSL library). This signifies some problem in the higher-level encryption and authentication layer that's superimposed on the underlying network connection. This error is a subtype of *OSError*. The error code and message of *SSLError* instances are provided by the OpenSSL library.

Changed in version 3.3: *SSLError* used to be a subtype of *socket.error*.

library

A string mnemonic designating the OpenSSL submodule in which the error occurred, such as SSL, PEM or X509. The range of possible values depends on the OpenSSL version.

Added in version 3.3.

reason

A string mnemonic designating the reason this error occurred, for example CERTIFICATE_VERIFY_FAILED. The range of possible values depends on the OpenSSL version.

Added in version 3.3.

exception ssl.**SSLZeroReturnError**

A subclass of *SSLError* raised when trying to read or write and the SSL connection has been closed cleanly. Note that this doesn't mean that the underlying transport (read TCP) has been closed.

Added in version 3.3.

exception ssl.**SSLWantReadError**

A subclass of *SSLError* raised by a *non-blocking SSL socket* when trying to read or write data, but more data needs to be received on the underlying TCP transport before the request can be fulfilled.

Added in version 3.3.

exception ssl.**SSLWantWriteError**

A subclass of *SSLError* raised by a *non-blocking SSL socket* when trying to read or write data, but more data needs to be sent on the underlying TCP transport before the request can be fulfilled.

Added in version 3.3.

exception ssl.**SSLSyscallError**

A subclass of *SSLError* raised when a system error was encountered while trying to fulfill an operation on a SSL socket. Unfortunately, there is no easy way to inspect the original errno number.

Added in version 3.3.

exception ssl.**SSLEOFError**

A subclass of *SSLError* raised when the SSL connection has been terminated abruptly. Generally, you shouldn't try to reuse the underlying transport when this error is encountered.

Added in version 3.3.

exception ssl.**SSLCertVerificationError**

A subclass of *SSLError* raised when certificate validation has failed.

Added in version 3.7.

verify_code

A numeric error number that denotes the verification error.

verify_message

A human readable string of the verification error.

exception ssl.**CertificateError**

An alias for *SSLCertVerificationError*.

Changed in version 3.7: The exception is now an alias for *SSLCertVerificationError*.

Random generation

ssl.**RAND_bytes**(*num*)

Return *num* cryptographically strong pseudo-random bytes. Raises an *SSLError* if the PRNG has not been seeded with enough data or if the operation is not supported by the current RAND method. *RAND_status()* can be used to check the status of the PRNG and *RAND_add()* can be used to seed the PRNG.

For almost all applications *os.urandom()* is preferable.

Read the Wikipedia article, Cryptographically secure pseudorandom number generator (CSPRNG), to get the requirements of a cryptographically strong generator.

Added in version 3.3.

ssl.**RAND_status**()

Return `True` if the SSL pseudo-random number generator has been seeded with 'enough' randomness, and `False` otherwise. You can use `ssl.RAND_egd()` and *ssl.RAND_add()* to increase the randomness of the pseudo-random number generator.

ssl.**RAND_add**(*bytes*, *entropy*)

Mix the given *bytes* into the SSL pseudo-random number generator. The parameter *entropy* (a float) is a lower bound on the entropy contained in string (so you can always use `0.0`). See **RFC 1750** for more information on sources of entropy.

Changed in version 3.5: Writable *bytes-like object* is now accepted.

Certificate handling

ssl.**cert_time_to_seconds**(*cert_time*)

Return the time in seconds since the Epoch, given the `cert_time` string representing the "notBefore" or "notAfter" date from a certificate in `"%b %d %H:%M:%S %Y %Z"` strptime format (C locale).

Here's an example:

```
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan  5 09:34:43 2018 GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

"notBefore" or "notAfter" dates must use GMT (**RFC 5280**).

Changed in version 3.5: Interpret the input time as a time in UTC as specified by 'GMT' timezone in the input string. Local timezone was used previously. Return an integer (no fractions of a second in the input format)

ssl.**get_server_certificate**(*addr*, *ssl_version=PROTOCOL_TLS_CLIENT*, *ca_certs=None*[, *timeout*])

Given the address `addr` of an SSL-protected server, as a (*hostname*, *port-number*) pair, fetches the server's certificate, and returns it as a PEM-encoded string. If `ssl_version` is specified, uses that version of the SSL protocol to attempt to connect to the server. If *ca_certs* is specified, it should be a file containing a list of root certificates, the same format as used for the *cafile* parameter in *SSLContext.load_verify_locations()*. The call will attempt to validate the server certificate against that set of root certificates, and will fail if the validation attempt fails. A timeout can be specified with the `timeout` parameter.

Changed in version 3.3: This function is now IPv6-compatible.

Changed in version 3.5: The default *ssl_version* is changed from *PROTOCOL_SSLv3* to *PROTOCOL_TLS* for maximum compatibility with modern servers.

Changed in version 3.10: The *timeout* parameter was added.

ssl.**DER_cert_to_PEM_cert**(*DER_cert_bytes*)

Given a certificate as a DER-encoded blob of bytes, returns a PEM-encoded string version of the same certificate.

ssl.**PEM_cert_to_DER_cert**(*PEM_cert_string*)

Given a certificate as an ASCII PEM string, returns a DER-encoded sequence of bytes for that same certificate.

ssl.**get_default_verify_paths**()

Returns a named tuple with paths to OpenSSL's default cafile and capath. The paths are the same as used by *SSLContext.set_default_verify_paths()*. The return value is a *named tuple* `DefaultVerifyPaths`:

- `cafile` - resolved path to cafile or `None` if the file doesn't exist,
- `capath` - resolved path to capath or `None` if the directory doesn't exist,
- `openssl_cafile_env` - OpenSSL's environment key that points to a cafile,
- `openssl_cafile` - hard coded path to a cafile,
- `openssl_capath_env` - OpenSSL's environment key that points to a capath,
- `openssl_capath` - hard coded path to a capath directory

Added in version 3.4.

ssl.**enum_certificates**(*store_name*)

Retrieve certificates from Windows' system cert store. *store_name* may be one of `CA`, `ROOT` or `MY`. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either `x509_asn` for X.509 ASN.1 data or `pkcs_7_asn` for PKCS#7 ASN.1 data. Trust specifies the purpose of the certificate as a set of OIDS or exactly `True` if the certificate is trustworthy for all purposes.

Example:

```
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

Availability: Windows.

Added in version 3.4.

ssl.**enum_crls**(*store_name*)

Retrieve CRLs from Windows' system cert store. *store_name* may be one of `CA`, `ROOT` or `MY`. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either `x509_asn` for X.509 ASN.1 data or `pkcs_7_asn` for PKCS#7 ASN.1 data.

Availability: Windows.

Added in version 3.4.

Constants

All constants are now *enum.IntEnum* or *enum.IntFlag* collections.

Added in version 3.6.

ssl.**CERT_NONE**

Possible value for *SSLContext.verify_mode*. Except for *PROTOCOL_TLS_CLIENT*, it is the default mode. With client-side sockets, just about any cert is accepted. Validation errors, such as untrusted or expired cert, are ignored and do not abort the TLS/SSL handshake.

In server mode, no certificate is requested from the client, so the client does not send any for client cert authentication.

See the discussion of *Security considerations* below.

ssl.**CERT_OPTIONAL**

Possible value for *SSLContext.verify_mode*. In client mode, *CERT_OPTIONAL* has the same meaning as *CERT_REQUIRED*. It is recommended to use *CERT_REQUIRED* for client-side sockets instead.

In server mode, a client certificate request is sent to the client. The client may either ignore the request or send a certificate in order perform TLS client cert authentication. If the client chooses to send a certificate, it is verified. Any verification error immediately aborts the TLS handshake.

Use of this setting requires a valid set of CA certificates to be passed to *SSLContext.load_verify_locations()*.

ssl.**CERT_REQUIRED**

Possible value for *SSLContext.verify_mode*. In this mode, certificates are required from the other side of the socket connection; an *SSLError* will be raised if no certificate is provided, or if its validation fails. This mode is **not** sufficient to verify a certificate in client mode as it does not match hostnames. *check_hostname* must be enabled as well to verify the authenticity of a cert. *PROTOCOL_TLS_CLIENT* uses *CERT_REQUIRED* and enables *check_hostname* by default.

With server socket, this mode provides mandatory TLS client cert authentication. A client certificate request is sent to the client and the client must provide a valid and trusted certificate.

Use of this setting requires a valid set of CA certificates to be passed to *SSLContext.load_verify_locations()*.

class ssl.**VerifyMode**

enum.IntEnum collection of CERT_* constants.

Added in version 3.6.

ssl.**VERIFY_DEFAULT**

Possible value for *SSLContext.verify_flags*. In this mode, certificate revocation lists (CRLs) are not checked. By default OpenSSL does neither require nor verify CRLs.

Added in version 3.4.

ssl.**VERIFY_CRL_CHECK_LEAF**

Possible value for *SSLContext.verify_flags*. In this mode, only the peer cert is checked but none of the intermediate CA certificates. The mode requires a valid CRL that is signed by the peer cert's issuer (its direct ancestor CA). If no proper CRL has been loaded with *SSLContext.load_verify_locations*, validation will fail.

Added in version 3.4.

ssl.**VERIFY_CRL_CHECK_CHAIN**

Possible value for *SSLContext.verify_flags*. In this mode, CRLs of all certificates in the peer cert chain are checked.

Added in version 3.4.

ssl.**VERIFY_X509_STRICT**

Possible value for *SSLContext.verify_flags* to disable workarounds for broken X.509 certificates.

Added in version 3.4.

ssl.**VERIFY_ALLOW_PROXY_CERTS**

Possible value for *SSLContext.verify_flags* to enables proxy certificate verification.

Added in version 3.10.

ssl.**VERIFY_X509_TRUSTED_FIRST**

Possible value for *SSLContext.verify_flags*. It instructs OpenSSL to prefer trusted certificates when building the trust chain to validate a certificate. This flag is enabled by default.

Added in version 3.4.4.

ssl.**VERIFY_X509_PARTIAL_CHAIN**

Possible value for *SSLContext.verify_flags*. It instructs OpenSSL to accept intermediate CAs in the trust store to be treated as trust-anchors, in the same way as the self-signed root CA certificates. This makes it possible to trust certificates issued by an intermediate CA without having to trust its ancestor root CA.

Added in version 3.10.

class ssl.**VerifyFlags**

enum.IntFlag collection of VERIFY_* constants.

Added in version 3.6.

ssl.**PROTOCOL_TLS**

Selects the highest protocol version that both the client and server support. Despite the name, this option can select both "SSL" and "TLS" protocols.

Added in version 3.6.

Deprecated since version 3.10: TLS clients and servers require different default settings for secure communication. The generic TLS protocol constant is deprecated in favor of *PROTOCOL_TLS_CLIENT* and *PROTOCOL_TLS_SERVER*.

ssl.**PROTOCOL_TLS_CLIENT**

Auto-negotiate the highest protocol version that both the client and server support, and configure the context client-side connections. The protocol enables *CERT_REQUIRED* and *check_hostname* by default.

Added in version 3.6.

ssl.**PROTOCOL_TLS_SERVER**

Auto-negotiate the highest protocol version that both the client and server support, and configure the context server-side connections.

Added in version 3.6.

ssl.**PROTOCOL_SSLv23**

Alias for *PROTOCOL_TLS*.

Deprecated since version 3.6: Use *PROTOCOL_TLS* instead.

ssl.**PROTOCOL_SSLv3**

Selects SSL version 3 as the channel encryption protocol.

This protocol is not available if OpenSSL is compiled with the `no-ssl3` option.

> **Warning**
>
> SSL version 3 is insecure. Its use is highly discouraged.

Deprecated since version 3.6: OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS_SERVER* or *PROTOCOL_TLS_CLIENT* with *SSLContext.minimum_version* and *SSLContext.maximum_version* instead.

ssl.**PROTOCOL_TLSv1**

Selects TLS version 1.0 as the channel encryption protocol.

Deprecated since version 3.6: OpenSSL has deprecated all version specific protocols.

ssl.**PROTOCOL_TLSv1_1**

Selects TLS version 1.1 as the channel encryption protocol. Available only with openssl version 1.0.1+.

Added in version 3.4.

Deprecated since version 3.6: OpenSSL has deprecated all version specific protocols.

ssl.**PROTOCOL_TLSv1_2**

Selects TLS version 1.2 as the channel encryption protocol. Available only with openssl version 1.0.1+.

Added in version 3.4.

Deprecated since version 3.6: OpenSSL has deprecated all version specific protocols.

ssl.**OP_ALL**

Enables workarounds for various bugs present in other SSL implementations. This option is set by default. It does not necessarily set the same flags as OpenSSL's `SSL_OP_ALL` constant.

Added in version 3.2.

ssl.**OP_NO_SSLv2**

Prevents an SSLv2 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing SSLv2 as the protocol version.

Added in version 3.2.

Deprecated since version 3.6: SSLv2 is deprecated

ssl.**OP_NO_SSLv3**

Prevents an SSLv3 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing SSLv3 as the protocol version.

Added in version 3.2.

Deprecated since version 3.6: SSLv3 is deprecated

ssl.**OP_NO_TLSv1**

Prevents a TLSv1 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1 as the protocol version.

Added in version 3.2.

Deprecated since version 3.7: The option is deprecated since OpenSSL 1.1.0, use the new *SSLContext.minimum_version* and *SSLContext.maximum_version* instead.

ssl.**OP_NO_TLSv1_1**

Prevents a TLSv1.1 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1.1 as the protocol version. Available only with openssl version 1.0.1+.

Added in version 3.4.

Deprecated since version 3.7: The option is deprecated since OpenSSL 1.1.0.

ssl.**OP_NO_TLSv1_2**

Prevents a TLSv1.2 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1.2 as the protocol version. Available only with openssl version 1.0.1+.

Added in version 3.4.

Deprecated since version 3.7: The option is deprecated since OpenSSL 1.1.0.

ssl.**OP_NO_TLSv1_3**

Prevents a TLSv1.3 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1.3 as the protocol version. TLS 1.3 is available with OpenSSL 1.1.1 or later. When Python has been compiled against an older version of OpenSSL, the flag defaults to *0*.

Added in version 3.6.3.

Deprecated since version 3.7: The option is deprecated since OpenSSL 1.1.0. It was added to 2.7.15 and 3.6.3 for backwards compatibility with OpenSSL 1.0.2.

ssl.**OP_NO_RENEGOTIATION**

Disable all renegotiation in TLSv1.2 and earlier. Do not send HelloRequest messages, and ignore renegotiation requests via ClientHello.

This option is only available with OpenSSL 1.1.0h and later.

Added in version 3.7.

ssl.**OP_CIPHER_SERVER_PREFERENCE**

Use the server's cipher ordering preference, rather than the client's. This option has no effect on client sockets and SSLv2 server sockets.

Added in version 3.3.

ssl.**OP_SINGLE_DH_USE**

Prevents reuse of the same DH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

Added in version 3.3.

ssl.**OP_SINGLE_ECDH_USE**

Prevents reuse of the same ECDH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

Added in version 3.3.

ssl.**OP_ENABLE_MIDDLEBOX_COMPAT**

Send dummy Change Cipher Spec (CCS) messages in TLS 1.3 handshake to make a TLS 1.3 connection look more like a TLS 1.2 connection.

This option is only available with OpenSSL 1.1.1 and later.

Added in version 3.8.

ssl.**OP_NO_COMPRESSION**

Disable compression on the SSL channel. This is useful if the application protocol supports its own compression scheme.

Added in version 3.3.

class ssl.**Options**

enum.IntFlag collection of OP_* constants.

ssl.**OP_NO_TICKET**

Prevent client side from requesting a session ticket.

Added in version 3.6.

ssl.**OP_IGNORE_UNEXPECTED_EOF**

Ignore unexpected shutdown of TLS connections.

This option is only available with OpenSSL 3.0.0 and later.

Added in version 3.10.

ssl.**OP_ENABLE_KTLS**

Enable the use of the kernel TLS. To benefit from the feature, OpenSSL must have been compiled with support for it, and the negotiated cipher suites and extensions must be supported by it (a list of supported ones may vary by platform and kernel version).

Note that with enabled kernel TLS some cryptographic operations are performed by the kernel directly and not via any available OpenSSL Providers. This might be undesirable if, for example, the application requires all cryptographic operations to be performed by the FIPS provider.

This option is only available with OpenSSL 3.0.0 and later.

Added in version 3.12.

ssl.**OP_LEGACY_SERVER_CONNECT**

Allow legacy insecure renegotiation between OpenSSL and unpatched servers only.

Added in version 3.12.

ssl.**HAS_ALPN**

Whether the OpenSSL library has built-in support for the *Application-Layer Protocol Negotiation* TLS extension as described in **RFC 7301**.

Added in version 3.5.

ssl.**HAS_NEVER_CHECK_COMMON_NAME**

Whether the OpenSSL library has built-in support not checking subject common name and *SSLContext.hostname_checks_common_name* is writeable.

Added in version 3.7.

ssl.**HAS_ECDH**

Whether the OpenSSL library has built-in support for the Elliptic Curve-based Diffie-Hellman key exchange. This should be true unless the feature was explicitly disabled by the distributor.

Added in version 3.3.

ssl.**HAS_SNI**

Whether the OpenSSL library has built-in support for the *Server Name Indication* extension (as defined in **RFC 6066**).

Added in version 3.2.

ssl.**HAS_NPN**

Whether the OpenSSL library has built-in support for the *Next Protocol Negotiation* as described in the Application Layer Protocol Negotiation. When true, you can use the *SSLContext.set_npn_protocols()* method to advertise which protocols you want to support.

Added in version 3.3.

ssl.**HAS_SSLv2**

Whether the OpenSSL library has built-in support for the SSL 2.0 protocol.

Added in version 3.7.

ssl.**HAS_SSLv3**

Whether the OpenSSL library has built-in support for the SSL 3.0 protocol.

Added in version 3.7.

ssl.**HAS_TLSv1**

Whether the OpenSSL library has built-in support for the TLS 1.0 protocol.

Added in version 3.7.

ssl.**HAS_TLSv1_1**

Whether the OpenSSL library has built-in support for the TLS 1.1 protocol.

Added in version 3.7.

ssl.**HAS_TLSv1_2**

Whether the OpenSSL library has built-in support for the TLS 1.2 protocol.

Added in version 3.7.

ssl.**HAS_TLSv1_3**

Whether the OpenSSL library has built-in support for the TLS 1.3 protocol.

Added in version 3.7.

ssl.**HAS_PSK**

Whether the OpenSSL library has built-in support for TLS-PSK.

Added in version 3.13.

ssl.**CHANNEL_BINDING_TYPES**

List of supported TLS channel binding types. Strings in this list can be used as arguments to *SSLSocket.get_channel_binding()*.

Added in version 3.3.

ssl.**OPENSSL_VERSION**

The version string of the OpenSSL library loaded by the interpreter:

```
>>> ssl.OPENSSL_VERSION
'OpenSSL 1.0.2k  26 Jan 2017'
```

Added in version 3.2.

ssl.**OPENSSL_VERSION_INFO**

A tuple of five integers representing version information about the OpenSSL library:

```
>>> ssl.OPENSSL_VERSION_INFO
(1, 0, 2, 11, 15)
```

Added in version 3.2.

ssl.**OPENSSL_VERSION_NUMBER**

The raw version number of the OpenSSL library, as a single integer:

```
>>> ssl.OPENSSL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x100020bf'
```

Added in version 3.2.

ssl.**ALERT_DESCRIPTION_HANDSHAKE_FAILURE**

ssl.**ALERT_DESCRIPTION_INTERNAL_ERROR**

ALERT_DESCRIPTION_*

Alert Descriptions from **RFC 5246** and others. The IANA TLS Alert Registry contains this list and references to the RFCs where their meaning is defined.

Used as the return value of the callback function in *SSLContext.set_servername_callback()*.

Added in version 3.4.

class ssl.**AlertDescription**

enum.IntEnum collection of ALERT_DESCRIPTION_* constants.

Added in version 3.6.

Purpose.**SERVER_AUTH**

Option for *create_default_context()* and *SSLContext.load_default_certs()*. This value indicates that the context may be used to authenticate web servers (therefore, it will be used to create client-side sockets).

Added in version 3.4.

Purpose.**CLIENT_AUTH**

Option for *create_default_context()* and *SSLContext.load_default_certs()*. This value indicates that the context may be used to authenticate web clients (therefore, it will be used to create server-side sockets).

Added in version 3.4.

class ssl.**SSLErrorNumber**

enum.IntEnum collection of SSL_ERROR_* constants.

Added in version 3.6.

class ssl.**TLSVersion**

enum.IntEnum collection of SSL and TLS versions for *SSLContext.maximum_version* and *SSLContext.minimum_version*.

Added in version 3.7.

TLSVersion.**MINIMUM_SUPPORTED**

TLSVersion.**MAXIMUM_SUPPORTED**

The minimum or maximum supported SSL or TLS version. These are magic constants. Their values don't reflect the lowest and highest available TLS/SSL versions.

TLSVersion.**SSLv3**

TLSVersion.**TLSv1**

TLSVersion.**TLSv1_1**

TLSVersion.**TLSv1_2**

TLSVersion.**TLSv1_3**

SSL 3.0 to TLS 1.3.

Deprecated since version 3.10: All *TLSVersion* members except *TLSVersion.TLSv1_2* and *TLSVersion.TLSv1_3* are deprecated.

18.3.2 SSL Sockets

class ssl.**SSLSocket**(*socket.socket*)

SSL sockets provide the following methods of *Socket Objects*:

- *accept()*
- *bind()*
- *close()*
- *connect()*
- *detach()*
- *fileno()*
- *getpeername()*, *getsockname()*
- *getsockopt()*, *setsockopt()*
- *gettimeout()*, *settimeout()*, *setblocking()*
- *listen()*
- *makefile()*
- *recv()*, *recv_into()* (but passing a non-zero `flags` argument is not allowed)
- *send()*, *sendall()* (with the same limitation)
- *sendfile()* (but *os.sendfile* will be used for plain-text sockets only, else *send()* will be used)

- *shutdown()*

However, since the SSL (and TLS) protocol has its own framing atop of TCP, the SSL sockets abstraction can, in certain respects, diverge from the specification of normal, OS-level sockets. See especially the *notes on non-blocking sockets*.

Instances of *SSLSocket* must be created using the *SSLContext.wrap_socket()* method.

Changed in version 3.5: The `sendfile()` method was added.

Changed in version 3.5: The `shutdown()` does not reset the socket timeout each time bytes are received or sent. The socket timeout is now the maximum total duration of the shutdown.

Deprecated since version 3.6: It is deprecated to create a *SSLSocket* instance directly, use *SSLContext.wrap_socket()* to wrap a socket.

Changed in version 3.7: *SSLSocket* instances must to created with *wrap_socket()*. In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

Changed in version 3.10: Python now uses `SSL_read_ex` and `SSL_write_ex` internally. The functions support reading and writing of data larger than 2 GB. Writing zero-length data no longer fails with a protocol violation error.

SSL sockets also have the following additional methods and attributes:

`SSLSocket.`**`read`**(*len=1024*, *buffer=None*)

Read up to *len* bytes of data from the SSL socket and return the result as a `bytes` instance. If *buffer* is specified, then read into the buffer instead, and return the number of bytes read.

Raise *SSLWantReadError* or *SSLWantWriteError* if the socket is *non-blocking* and the read would block.

As at any time a re-negotiation is possible, a call to *read()* can also cause write operations.

Changed in version 3.5: The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration to read up to *len* bytes.

Deprecated since version 3.6: Use `recv()` instead of *read()*.

`SSLSocket.`**`write`**(*buf*)

Write *buf* to the SSL socket and return the number of bytes written. The *buf* argument must be an object supporting the buffer interface.

Raise *SSLWantReadError* or *SSLWantWriteError* if the socket is *non-blocking* and the write would block.

As at any time a re-negotiation is possible, a call to *write()* can also cause read operations.

Changed in version 3.5: The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration to write *buf*.

Deprecated since version 3.6: Use `send()` instead of *write()*.

Note

The *read()* and *write()* methods are the low-level methods that read and write unencrypted, application-level data and decrypt/encrypt it to encrypted, wire-level data. These methods require an active SSL connection, i.e. the handshake was completed and *SSLSocket.unwrap()* was not called.

Normally you should use the socket API methods like *recv()* and *send()* instead of these methods.

SSLSocket.**do_handshake**()

Perform the SSL setup handshake.

Changed in version 3.4: The handshake method also performs match_hostname() when the *check_hostname* attribute of the socket's *context* is true.

Changed in version 3.5: The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration of the handshake.

Changed in version 3.7: Hostname or IP address is matched by OpenSSL during handshake. The function match_hostname() is no longer used. In case OpenSSL refuses a hostname or IP address, the handshake is aborted early and a TLS alert message is sent to the peer.

SSLSocket.**getpeercert**(*binary_form=False*)

If there is no certificate for the peer on the other end of the connection, return None. If the SSL handshake hasn't been done yet, raise *ValueError*.

If the binary_form parameter is *False*, and a certificate was received from the peer, this method returns a *dict* instance. If the certificate was not validated, the dict is empty. If the certificate was validated, it returns a dict with several keys, amongst them subject (the principal for which the certificate was issued) and issuer (the principal issuing the certificate). If a certificate contains an instance of the *Subject Alternative Name* extension (see **RFC 3280**), there will also be a subjectAltName key in the dictionary.

The subject and issuer fields are tuples containing the sequence of relative distinguished names (RDNs) given in the certificate's data structure for the respective fields, and each RDN is a sequence of name-value pairs. Here is a real-world example:

```
{'issuer': ((('countryName', 'IL'),),
            (('organizationName', 'StartCom Ltd.'),),
            (('organizationalUnitName',
              'Secure Digital Certificate Signing'),),
            (('commonName',
              'StartCom Class 2 Primary Intermediate Server CA'),)),
 'notAfter': 'Nov 22 08:15:19 2013 GMT',
 'notBefore': 'Nov 21 03:09:52 2011 GMT',
 'serialNumber': '95F0',
 'subject': ((('description', '571208-SLe257oHY9fVQ07Z'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'San Francisco'),),
             (('organizationName', 'Electronic Frontier Foundation, Inc.'),),
             (('commonName', '*.eff.org'),),
             (('emailAddress', 'hostmaster@eff.org'),)),
 'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org')),
 'version': 3}
```

If the binary_form parameter is *True*, and a certificate was provided, this method returns the DER-encoded form of the entire certificate as a sequence of bytes, or *None* if the peer did not provide a certificate. Whether the peer provides a certificate depends on the SSL socket's role:

- for a client SSL socket, the server will always provide a certificate, regardless of whether validation was required;
- for a server SSL socket, the client will only provide a certificate when requested by the server; therefore *getpeercert()* will return *None* if you used *CERT_NONE* (rather than *CERT_OPTIONAL* or *CERT_REQUIRED*).

See also *SSLContext.check_hostname*.

Changed in version 3.2: The returned dictionary includes additional items such as issuer and notBefore.

Changed in version 3.4: *ValueError* is raised when the handshake isn't done. The returned dictionary includes additional X509v3 extension items such as `crlDistributionPoints`, `caIssuers` and `OCSP` URIs.

Changed in version 3.9: IPv6 address strings no longer have a trailing new line.

`SSLSocket.`**`get_verified_chain`**`()`

Returns verified certificate chain provided by the other end of the SSL channel as a list of DER-encoded bytes. If certificate verification was disabled method acts the same as *get_unverified_chain()*.

Added in version 3.13.

`SSLSocket.`**`get_unverified_chain`**`()`

Returns raw certificate chain provided by the other end of the SSL channel as a list of DER-encoded bytes.

Added in version 3.13.

`SSLSocket.`**`cipher`**`()`

Returns a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used. If no connection has been established, returns `None`.

`SSLSocket.`**`shared_ciphers`**`()`

Return the list of ciphers available in both the client and server. Each entry of the returned list is a three-value tuple containing the name of the cipher, the version of the SSL protocol that defines its use, and the number of secret bits the cipher uses. *shared_ciphers()* returns `None` if no connection has been established or the socket is a client socket.

Added in version 3.5.

`SSLSocket.`**`compression`**`()`

Return the compression algorithm being used as a string, or `None` if the connection isn't compressed.

If the higher-level protocol supports its own compression mechanism, you can use *OP_NO_COMPRESSION* to disable SSL-level compression.

Added in version 3.3.

`SSLSocket.`**`get_channel_binding`**`(`*cb_type='tls-unique'*`)`

Get channel binding data for current connection, as a bytes object. Returns `None` if not connected or the handshake has not been completed.

The *cb_type* parameter allow selection of the desired channel binding type. Valid channel binding types are listed in the *CHANNEL_BINDING_TYPES* list. Currently only the 'tls-unique' channel binding, defined by **RFC 5929**, is supported. *ValueError* will be raised if an unsupported channel binding type is requested.

Added in version 3.3.

`SSLSocket.`**`selected_alpn_protocol`**`()`

Return the protocol that was selected during the TLS handshake. If *SSLContext.set_alpn_protocols()* was not called, if the other party does not support ALPN, if this socket does not support any of the client's proposed protocols, or if the handshake has not happened yet, `None` is returned.

Added in version 3.5.

`SSLSocket.`**`selected_npn_protocol`**`()`

Return the higher-level protocol that was selected during the TLS/SSL handshake. If *SSLContext.set_npn_protocols()* was not called, or if the other party does not support NPN, or if the handshake has not yet happened, this will return `None`.

Added in version 3.3.

Deprecated since version 3.10: NPN has been superseded by ALPN

SSLSocket.**unwrap**()

Performs the SSL shutdown handshake, which removes the TLS layer from the underlying socket, and returns the underlying socket object. This can be used to go from encrypted operation over a connection to unencrypted. The returned socket should always be used for further communication with the other side of the connection, rather than the original socket.

SSLSocket.**verify_client_post_handshake**()

Requests post-handshake authentication (PHA) from a TLS 1.3 client. PHA can only be initiated for a TLS 1.3 connection from a server-side socket, after the initial TLS handshake and with PHA enabled on both sides, see *SSLContext.post_handshake_auth*.

The method does not perform a cert exchange immediately. The server-side sends a CertificateRequest during the next write event and expects the client to respond with a certificate on the next read event.

If any precondition isn't met (e.g. not TLS 1.3, PHA not enabled), an *SSLError* is raised.

> **Note**
>
> Only available with OpenSSL 1.1.1 and TLS 1.3 enabled. Without TLS 1.3 support, the method raises *NotImplementedError*.

Added in version 3.8.

SSLSocket.**version**()

Return the actual SSL protocol version negotiated by the connection as a string, or `None` if no secure connection is established. As of this writing, possible return values include `"SSLv2"`, `"SSLv3"`, `"TLSv1"`, `"TLSv1.1"` and `"TLSv1.2"`. Recent OpenSSL versions may define more return values.

Added in version 3.5.

SSLSocket.**pending**()

Returns the number of already decrypted bytes available for read, pending on the connection.

SSLSocket.**context**

The *SSLContext* object this SSL socket is tied to.

Added in version 3.2.

SSLSocket.**server_side**

A boolean which is `True` for server-side sockets and `False` for client-side sockets.

Added in version 3.2.

SSLSocket.**server_hostname**

Hostname of the server: *str* type, or `None` for server-side socket or if the hostname was not specified in the constructor.

Added in version 3.2.

Changed in version 3.7: The attribute is now always ASCII text. When `server_hostname` is an internationalized domain name (IDN), this attribute now stores the A-label form (`"xn--pythn-mua.org"`), rather than the U-label form (`"pythön.org"`).

SSLSocket.**session**

The *SSLSession* for this SSL connection. The session is available for client and server side sockets after the TLS handshake has been performed. For client sockets the session can be set before *do_handshake()* has been called to reuse a session.

Added in version 3.6.

SSLSocket.**session_reused**

Added in version 3.6.

18.3.3 SSL Contexts

Added in version 3.2.

An SSL context holds various data longer-lived than single SSL connections, such as SSL configuration options, certificate(s) and private key(s). It also manages a cache of SSL sessions for server-side sockets, in order to speed up repeated connections from the same clients.

class ssl.**SSLContext**(*protocol=None*)

Create a new SSL context. You may pass *protocol* which must be one of the PROTOCOL_* constants defined in this module. The parameter specifies which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server's choice. Most of the versions are not interoperable with the other versions. If not specified, the default is *PROTOCOL_TLS*; it provides the most compatibility with other versions.

Here's a table showing which versions in a client (down the side) can connect to which versions in a server (along the top):

client / **server**	**SSLv2**	**SSLv3**	**TLS**[3]	**TLSv1**	**TLSv1.1**	**TLSv1.2**
SSLv2	yes	no	no[1]	no	no	no
SSLv3	no	yes	no[2]	no	no	no
TLS (SSLv23)[3]	no[1]	no[2]	yes	yes	yes	yes
TLSv1	no	no	yes	yes	no	no
TLSv1.1	no	no	yes	no	yes	no
TLSv1.2	no	no	yes	no	no	yes

See also

create_default_context() lets the *ssl* module choose security settings for a given purpose.

Changed in version 3.6: The context is created with secure default values. The options *OP_NO_COMPRESSION*, *OP_CIPHER_SERVER_PREFERENCE*, *OP_SINGLE_DH_USE*, *OP_SINGLE_ECDH_USE*, *OP_NO_SSLv2*, and *OP_NO_SSLv3* (except for *PROTOCOL_SSLv3*) are set by default. The initial cipher suite list contains only HIGH ciphers, no NULL ciphers and no MD5 ciphers.

Deprecated since version 3.10: *SSLContext* without protocol argument is deprecated. The context class will either require *PROTOCOL_TLS_CLIENT* or *PROTOCOL_TLS_SERVER* protocol in the future.

Changed in version 3.10: The default cipher suites now include only secure AES and ChaCha20 ciphers with forward secrecy and security level 2. RSA and DH keys with less than 2048 bits and ECC keys with less than 224 bits are prohibited. *PROTOCOL_TLS*, *PROTOCOL_TLS_CLIENT*, and *PROTOCOL_TLS_SERVER* use TLS 1.2 as minimum TLS version.

[3] TLS 1.3 protocol will be available with *PROTOCOL_TLS* in OpenSSL >= 1.1.1. There is no dedicated PROTOCOL constant for just TLS 1.3.
[1] *SSLContext* disables SSLv2 with *OP_NO_SSLv2* by default.
[2] *SSLContext* disables SSLv3 with *OP_NO_SSLv3* by default.

> **Note**
>
> *SSLContext* only supports limited mutation once it has been used by a connection. Adding new certificates to the internal trust store is allowed, but changing ciphers, verification settings, or mTLS certificates may result in surprising behavior.

> **Note**
>
> *SSLContext* is designed to be shared and used by multiple connections. Thus, it is thread-safe as long as it is not reconfigured after being used by a connection.

SSLContext objects have the following methods and attributes:

SSLContext.**cert_store_stats**()

Get statistics about quantities of loaded X.509 certificates, count of X.509 certificates flagged as CA certificates and certificate revocation lists as dictionary.

Example for a context with one CA cert and one other cert:

```
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

Added in version 3.4.

SSLContext.**load_cert_chain**(*certfile*, *keyfile=None*, *password=None*)

Load a private key and the corresponding certificate. The *certfile* string must be the path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate's authenticity. The *keyfile* string, if present, must point to a file containing the private key. Otherwise the private key will be taken from *certfile* as well. See the discussion of *Certificates* for more information on how the certificate is stored in the *certfile*.

The *password* argument may be a function to call to get the password for decrypting the private key. It will only be called if the private key is encrypted and a password is necessary. It will be called with no arguments, and it should return a string, bytes, or bytearray. If the return value is a string it will be encoded as UTF-8 before using it to decrypt the key. Alternatively a string, bytes, or bytearray value may be supplied directly as the *password* argument. It will be ignored if the private key is not encrypted and no password is needed.

If the *password* argument is not specified and a password is required, OpenSSL's built-in password prompting mechanism will be used to interactively prompt the user for a password.

An *SSLError* is raised if the private key doesn't match with the certificate.

Changed in version 3.3: New optional argument *password*.

SSLContext.**load_default_certs**(*purpose=Purpose.SERVER_AUTH*)

Load a set of default "certification authority" (CA) certificates from default locations. On Windows it loads CA certs from the `CA` and `ROOT` system stores. On all systems it calls *SSLContext.set_default_verify_paths()*. In the future the method may load CA certificates from other locations, too.

The *purpose* flag specifies what kind of CA certificates are loaded. The default settings *Purpose.SERVER_AUTH* loads certificates, that are flagged and trusted for TLS web server authentication (client side sockets). *Purpose.CLIENT_AUTH* loads CA certificates for client certificate verification on the server side.

Added in version 3.4.

SSLContext.**load_verify_locations**(*cafile=None*, *capath=None*, *cadata=None*)

Load a set of "certification authority" (CA) certificates used to validate other peers' certificates when *verify_mode* is other than *CERT_NONE*. At least one of *cafile* or *capath* must be specified.

This method can also load certification revocation lists (CRLs) in PEM or DER format. In order to make use of CRLs, *SSLContext.verify_flags* must be configured properly.

The *cafile* string, if present, is the path to a file of concatenated CA certificates in PEM format. See the discussion of *Certificates* for more information about how to arrange the certificates in this file.

The *capath* string, if present, is the path to a directory containing several CA certificates in PEM format, following an OpenSSL specific layout.

The *cadata* object, if present, is either an ASCII string of one or more PEM-encoded certificates or a *bytes-like object* of DER-encoded certificates. Like with *capath* extra lines around PEM-encoded certificates are ignored but at least one certificate must be present.

Changed in version 3.4: New optional argument *cadata*

SSLContext.**get_ca_certs**(*binary_form=False*)

Get a list of loaded "certification authority" (CA) certificates. If the `binary_form` parameter is *False* each list entry is a dict like the output of *SSLSocket.getpeercert()*. Otherwise the method returns a list of DER-encoded certificates. The returned list does not contain certificates from *capath* unless a certificate was requested and loaded by a SSL connection.

> **Note**
>
> Certificates in a capath directory aren't loaded unless they have been used at least once.

Added in version 3.4.

SSLContext.**get_ciphers**()

Get a list of enabled ciphers. The list is in order of cipher priority. See *SSLContext.set_ciphers()*.

Example:

```
>>> ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> ctx.set_ciphers('ECDHE+AESGCM:!ECDSA')
>>> ctx.get_ciphers()
[{'aead': True,
  'alg_bits': 256,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'digest': None,
  'id': 50380848,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1.2',
  'strength_bits': 256,
  'symmetric': 'aes-256-gcm'},
 {'aead': True,
  'alg_bits': 128,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'digest': None,
```

(continues on next page)

(continued from previous page)

```
  'id': 50380847,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1.2',
  'strength_bits': 128,
  'symmetric': 'aes-128-gcm'}]
```

Added in version 3.6.

SSLContext.**set_default_verify_paths**()

Load a set of default "certification authority" (CA) certificates from a filesystem path defined when building the OpenSSL library. Unfortunately, there's no easy way to know whether this method succeeds: no error is returned if no certificates are to be found. When the OpenSSL library is provided as part of the operating system, though, it is likely to be configured properly.

SSLContext.**set_ciphers**(*ciphers*)

Set the available ciphers for sockets created with this context. It should be a string in the OpenSSL cipher list format. If no cipher can be selected (because compile-time options or other configuration forbids use of all the specified ciphers), an *SSLError* will be raised.

Note

when connected, the *SSLSocket.cipher()* method of SSL sockets will give the currently selected cipher.

TLS 1.3 cipher suites cannot be disabled with *set_ciphers()*.

SSLContext.**set_alpn_protocols**(*protocols*)

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of ASCII strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to **RFC 7301**. After a successful handshake, the *SSLSocket.selected_alpn_protocol()* method will return the agreed-upon protocol.

This method will raise *NotImplementedError* if *HAS_ALPN* is `False`.

Added in version 3.5.

SSLContext.**set_npn_protocols**(*protocols*)

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to the Application Layer Protocol Negotiation. After a successful handshake, the *SSLSocket.selected_npn_protocol()* method will return the agreed-upon protocol.

This method will raise *NotImplementedError* if *HAS_NPN* is `False`.

Added in version 3.3.

Deprecated since version 3.10: NPN has been superseded by ALPN

SSLContext.**sni_callback**

Register a callback function that will be called after the TLS Client Hello handshake message has been received by the SSL/TLS server when the TLS client specifies a server name indication. The server name indication mechanism is specified in **RFC 6066** section 3 - Server Name Indication.

Only one callback can be set per `SSLContext`. If *sni_callback* is set to `None` then the callback is disabled. Calling this function a subsequent time will disable the previously registered callback.

The callback function will be called with three arguments; the first being the *ssl.SSLSocket*, the second is a string that represents the server name that the client is intending to communicate (or *None* if the TLS Client Hello does not contain a server name) and the third argument is the original *SSLContext*. The server name argument is text. For internationalized domain name, the server name is an IDN A-label (`"xn--pythn-mua.org"`).

A typical use of this callback is to change the *ssl.SSLSocket*'s *SSLSocket.context* attribute to a new object of type *SSLContext* representing a certificate chain that matches the server name.

Due to the early negotiation phase of the TLS connection, only limited methods and attributes are usable like *SSLSocket.selected_alpn_protocol()* and *SSLSocket.context*. The *SSLSocket.getpeercert()*, *SSLSocket.get_verified_chain()*, *SSLSocket.get_unverified_chain()* *SSLSocket.cipher()* and *SSLSocket.compression()* methods require that the TLS connection has progressed beyond the TLS Client Hello and therefore will not return meaningful values nor can they be called safely.

The *sni_callback* function must return `None` to allow the TLS negotiation to continue. If a TLS failure is required, a constant *ALERT_DESCRIPTION_** can be returned. Other return values will result in a TLS fatal error with *ALERT_DESCRIPTION_INTERNAL_ERROR*.

If an exception is raised from the *sni_callback* function the TLS connection will terminate with a fatal TLS alert message *ALERT_DESCRIPTION_HANDSHAKE_FAILURE*.

This method will raise *NotImplementedError* if the OpenSSL library had OPENSSL_NO_TLSEXT defined when it was built.

Added in version 3.7.

`SSLContext.`**`set_servername_callback`**(*server_name_callback*)

This is a legacy API retained for backwards compatibility. When possible, you should use *sni_callback* instead. The given *server_name_callback* is similar to *sni_callback*, except that when the server hostname is an IDN-encoded internationalized domain name, the *server_name_callback* receives a decoded U-label (`"pythön.org"`).

If there is a decoding error on the server name, the TLS connection will terminate with an *ALERT_DESCRIPTION_INTERNAL_ERROR* fatal TLS alert message to the client.

Added in version 3.4.

`SSLContext.`**`load_dh_params`**(*dhfile*)

Load the key generation parameters for Diffie-Hellman (DH) key exchange. Using DH key exchange improves forward secrecy at the expense of computational resources (both on the server and on the client). The *dhfile* parameter should be the path to a file containing DH parameters in PEM format.

This setting doesn't apply to client sockets. You can also use the *OP_SINGLE_DH_USE* option to further improve security.

Added in version 3.3.

`SSLContext.`**`set_ecdh_curve`**(*curve_name*)

Set the curve name for Elliptic Curve-based Diffie-Hellman (ECDH) key exchange. ECDH is significantly faster than regular DH while arguably as secure. The *curve_name* parameter should be a string describing a well-known elliptic curve, for example `prime256v1` for a widely supported curve.

This setting doesn't apply to client sockets. You can also use the *OP_SINGLE_ECDH_USE* option to further improve security.

This method is not available if *HAS_ECDH* is `False`.

Added in version 3.3.

> **See also**
>
> **SSL/TLS & Perfect Forward Secrecy**
> Vincent Bernat.

SSLContext.**wrap_socket**(*sock*, *server_side=False*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *server_hostname=None*, *session=None*)

Wrap an existing Python socket *sock* and return an instance of *SSLContext.sslsocket_class* (default *SSLSocket*). The returned SSL socket is tied to the context, its settings and certificates. *sock* must be a *SOCK_STREAM* socket; other socket types are unsupported.

The parameter `server_side` is a boolean which identifies whether server-side or client-side behavior is desired from this socket.

For client-side sockets, the context construction is lazy; if the underlying socket isn't connected yet, the context construction will be performed after `connect()` is called on the socket. For server-side sockets, if the socket has no remote peer, it is assumed to be a listening socket, and the server-side SSL wrapping is automatically performed on client connections accepted via the `accept()` method. The method may raise *SSLError*.

On client connections, the optional parameter *server_hostname* specifies the hostname of the service which we are connecting to. This allows a single server to host multiple SSL-based services with distinct certificates, quite similarly to HTTP virtual hosts. Specifying *server_hostname* will raise a *ValueError* if *server_side* is true.

The parameter `do_handshake_on_connect` specifies whether to do the SSL handshake automatically after doing a `socket.connect()`, or whether the application program will call it explicitly, by invoking the *SSLSocket.do_handshake()* method. Calling *SSLSocket.do_handshake()* explicitly gives the program control over the blocking behavior of the socket I/O involved in the handshake.

The parameter `suppress_ragged_eofs` specifies how the `SSLSocket.recv()` method should signal unexpected EOF from the other end of the connection. If specified as *True* (the default), it returns a normal EOF (an empty bytes object) in response to unexpected EOF errors raised from the underlying socket; if *False*, it will raise the exceptions back to the caller.

session, see *session*.

To wrap an *SSLSocket* in another *SSLSocket*, use *SSLContext.wrap_bio()*.

Changed in version 3.5: Always allow a server_hostname to be passed, even if OpenSSL does not have SNI.

Changed in version 3.6: *session* argument was added.

Changed in version 3.7: The method returns an instance of *SSLContext.sslsocket_class* instead of hard-coded *SSLSocket*.

SSLContext.**sslsocket_class**

The return type of *SSLContext.wrap_socket()*, defaults to *SSLSocket*. The attribute can be overridden on instance of class in order to return a custom subclass of *SSLSocket*.

Added in version 3.7.

SSLContext.**wrap_bio**(*incoming*, *outgoing*, *server_side=False*, *server_hostname=None*, *session=None*)

Wrap the BIO objects *incoming* and *outgoing* and return an instance of *SSLContext.sslobject_class* (default *SSLObject*). The SSL routines will read input data from the incoming BIO and write data to the outgoing BIO.

The *server_side*, *server_hostname* and *session* parameters have the same meaning as in *SSLContext.wrap_socket()*.

Changed in version 3.6: *session* argument was added.

Changed in version 3.7: The method returns an instance of *SSLContext.sslobject_class* instead of hard-coded *SSLObject*.

SSLContext.**sslobject_class**

The return type of *SSLContext.wrap_bio()*, defaults to *SSLObject*. The attribute can be overridden on instance of class in order to return a custom subclass of *SSLObject*.

Added in version 3.7.

SSLContext.**session_stats**()

Get statistics about the SSL sessions created or managed by this context. A dictionary is returned which maps the names of each piece of information to their numeric values. For example, here is the total number of hits and misses in the session cache since the context was created:

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

SSLContext.**check_hostname**

Whether to match the peer cert's hostname in *SSLSocket.do_handshake()*. The context's *verify_mode* must be set to *CERT_OPTIONAL* or *CERT_REQUIRED*, and you must pass *server_hostname* to *wrap_socket()* in order to match the hostname. Enabling hostname checking automatically sets *verify_mode* from *CERT_NONE* to *CERT_REQUIRED*. It cannot be set back to *CERT_NONE* as long as hostname checking is enabled. The *PROTOCOL_TLS_CLIENT* protocol enables hostname checking by default. With other protocols, hostname checking must be enabled explicitly.

Example:

```
import socket, ssl

context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

Added in version 3.4.

Changed in version 3.7: *verify_mode* is now automatically changed to *CERT_REQUIRED* when hostname checking is enabled and *verify_mode* is *CERT_NONE*. Previously the same operation would have failed with a *ValueError*.

SSLContext.**keylog_filename**

Write TLS keys to a keylog file, whenever key material is generated or received. The keylog file is designed for debugging purposes only. The file format is specified by NSS and used by many traffic analyzers such as Wireshark. The log file is opened in append-only mode. Writes are synchronized between threads, but not between processes.

Added in version 3.8.

SSLContext.**maximum_version**

A *TLSVersion* enum member representing the highest supported TLS version. The value defaults to *TLSVersion.MAXIMUM_SUPPORTED*. The attribute is read-only for protocols other than *PROTOCOL_TLS*, *PROTOCOL_TLS_CLIENT*, and *PROTOCOL_TLS_SERVER*.

The attributes *maximum_version*, *minimum_version* and *SSLContext.options* all affect the supported SSL and TLS versions of the context. The implementation does not prevent invalid combination. For exam-

ple a context with *OP_NO_TLSv1_2* in *options* and *maximum_version* set to *TLSVersion.TLSv1_2* will not be able to establish a TLS 1.2 connection.

Added in version 3.7.

SSLContext.**minimum_version**

Like *SSLContext.maximum_version* except it is the lowest supported version or *TLSVersion.MINIMUM_SUPPORTED*.

Added in version 3.7.

SSLContext.**num_tickets**

Control the number of TLS 1.3 session tickets of a *PROTOCOL_TLS_SERVER* context. The setting has no impact on TLS 1.0 to 1.2 connections.

Added in version 3.8.

SSLContext.**options**

An integer representing the set of SSL options enabled on this context. The default value is *OP_ALL*, but you can specify other options such as *OP_NO_SSLv2* by ORing them together.

Changed in version 3.6: *SSLContext.options* returns *Options* flags:

```
>>> ssl.create_default_context().options
<Options.OP_ALL|OP_NO_SSLv3|OP_NO_SSLv2|OP_NO_COMPRESSION: 2197947391>
```

Deprecated since version 3.7: All OP_NO_SSL* and OP_NO_TLS* options have been deprecated since Python 3.7. Use *SSLContext.minimum_version* and *SSLContext.maximum_version* instead.

SSLContext.**post_handshake_auth**

Enable TLS 1.3 post-handshake client authentication. Post-handshake auth is disabled by default and a server can only request a TLS client certificate during the initial handshake. When enabled, a server may request a TLS client certificate at any time after the handshake.

When enabled on client-side sockets, the client signals the server that it supports post-handshake authentication.

When enabled on server-side sockets, *SSLContext.verify_mode* must be set to *CERT_OPTIONAL* or *CERT_REQUIRED*, too. The actual client cert exchange is delayed until *SSLSocket.verify_client_post_handshake()* is called and some I/O is performed.

Added in version 3.8.

SSLContext.**protocol**

The protocol version chosen when constructing the context. This attribute is read-only.

SSLContext.**hostname_checks_common_name**

Whether *check_hostname* falls back to verify the cert's subject common name in the absence of a subject alternative name extension (default: true).

Added in version 3.7.

Changed in version 3.10: The flag had no effect with OpenSSL before version 1.1.1l. Python 3.8.9, 3.9.3, and 3.10 include workarounds for previous versions.

SSLContext.**security_level**

An integer representing the security level for the context. This attribute is read-only.

Added in version 3.10.

SSLContext.**verify_flags**

The flags for certificate verification operations. You can set flags like *VERIFY_CRL_CHECK_LEAF* by ORing them together. By default OpenSSL does neither require nor verify certificate revocation lists (CRLs).

Added in version 3.4.

Changed in version 3.6: *SSLContext.verify_flags* returns *VerifyFlags* flags:

```
>>> ssl.create_default_context().verify_flags
<VerifyFlags.VERIFY_X509_TRUSTED_FIRST: 32768>
```

SSLContext.**verify_mode**

Whether to try to verify other peers' certificates and how to behave if verification fails. This attribute must be one of *CERT_NONE*, *CERT_OPTIONAL* or *CERT_REQUIRED*.

Changed in version 3.6: *SSLContext.verify_mode* returns *VerifyMode* enum:

```
>>> ssl.create_default_context().verify_mode
<VerifyMode.CERT_REQUIRED: 2>
```

SSLContext.**set_psk_client_callback**(*callback*)

Enables TLS-PSK (pre-shared key) authentication on a client-side connection.

In general, certificate based authentication should be preferred over this method.

The parameter `callback` is a callable object with the signature: `def callback(hint: str | None) -> tuple[str | None, bytes]`. The `hint` parameter is an optional identity hint sent by the server. The return value is a tuple in the form (client-identity, psk). Client-identity is an optional string which may be used by the server to select a corresponding PSK for the client. The string must be less than or equal to `256` octets when UTF-8 encoded. PSK is a *bytes-like object* representing the pre-shared key. Return a zero length PSK to reject the connection.

Setting `callback` to *None* removes any existing callback.

Note

When using TLS 1.3:

- the `hint` parameter is always *None*.
- client-identity must be a non-empty string.

Example usage:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.check_hostname = False
context.verify_mode = ssl.CERT_NONE
context.maximum_version = ssl.TLSVersion.TLSv1_2
context.set_ciphers('PSK')

# A simple lambda:
psk = bytes.fromhex('c0ffee')
context.set_psk_client_callback(lambda hint: (None, psk))

# A table using the hint from the server:
psk_table = { 'ServerId_1': bytes.fromhex('c0ffee'),
              'ServerId_2': bytes.fromhex('facade')
}
```

(continues on next page)

(continued from previous page)

```
def callback(hint):
    return 'ClientId_1', psk_table.get(hint, b'')
context.set_psk_client_callback(callback)
```

This method will raise *NotImplementedError* if *HAS_PSK* is `False`.

Added in version 3.13.

`SSLContext.`**`set_psk_server_callback`**(*callback*, *identity_hint=None*)

Enables TLS-PSK (pre-shared key) authentication on a server-side connection.

In general, certificate based authentication should be preferred over this method.

The parameter `callback` is a callable object with the signature: `def callback(identity: str | None) -> bytes`. The `identity` parameter is an optional identity sent by the client which can be used to select a corresponding PSK. The return value is a *bytes-like object* representing the pre-shared key. Return a zero length PSK to reject the connection.

Setting `callback` to *None* removes any existing callback.

The parameter `identity_hint` is an optional identity hint string sent to the client. The string must be less than or equal to `256` octets when UTF-8 encoded.

Note

When using TLS 1.3 the `identity_hint` parameter is not sent to the client.

Example usage:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.maximum_version = ssl.TLSVersion.TLSv1_2
context.set_ciphers('PSK')

# A simple lambda:
psk = bytes.fromhex('c0ffee')
context.set_psk_server_callback(lambda identity: psk)

# A table using the identity of the client:
psk_table = { 'ClientId_1': bytes.fromhex('c0ffee'),
              'ClientId_2': bytes.fromhex('facade')
}
def callback(identity):
    return psk_table.get(identity, b'')
context.set_psk_server_callback(callback, 'ServerId_1')
```

This method will raise *NotImplementedError* if *HAS_PSK* is `False`.

Added in version 3.13.

18.3.4 Certificates

Certificates in general are part of a public-key / private-key system. In this system, each *principal*, (which may be a machine, or a person, or an organization) is assigned a unique two-part encryption key. One part of the key is public, and is called the *public key*; the other part is kept secret, and is called the *private key*. The two parts are related, in that if you encrypt a message with one of the parts, you can decrypt it with the other part, and **only** with the other part.

A certificate contains information about two principals. It contains the name of a *subject*, and the subject's public key. It also contains a statement by a second principal, the *issuer*, that the subject is who they claim to be, and that this is indeed the subject's public key. The issuer's statement is signed with the issuer's private key, which only the issuer knows. However, anyone can verify the issuer's statement by finding the issuer's public key, decrypting the statement with it, and comparing it to the other information in the certificate. The certificate also contains information about the time period over which it is valid. This is expressed as two fields, called "notBefore" and "notAfter".

In the Python use of certificates, a client or server can use a certificate to prove who they are. The other side of a network connection can also be required to produce a certificate, and that certificate can be validated to the satisfaction of the client or server that requires such validation. The connection attempt can be set to raise an exception if the validation fails. Validation is done automatically, by the underlying OpenSSL framework; the application need not concern itself with its mechanics. But the application does usually need to provide sets of certificates to allow this process to take place.

Python uses files to contain certificates. They should be formatted as "PEM" (see **RFC 1422**), which is a base-64 encoded form wrapped with a header line and a footer line:

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Certificate chains

The Python files which contain certificates can contain a sequence of certificates, sometimes called a *certificate chain*. This chain should start with the specific certificate for the principal who "is" the client or server, and then the certificate for the issuer of that certificate, and then the certificate for the issuer of *that* certificate, and so on up the chain till you get to a certificate which is *self-signed*, that is, a certificate which has the same subject and issuer, sometimes called a *root certificate*. The certificates should just be concatenated together in the certificate file. For example, suppose we had a three certificate chain, from our server certificate to the certificate of the certification authority that signed our server certificate, to the root certificate of the agency which issued the certification authority's certificate:

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

CA certificates

If you are going to require validation of the other side of the connection's certificate, you need to provide a "CA certs" file, filled with the certificate chains for each issuer you are willing to trust. Again, this file just contains these chains concatenated together. For validation, Python will use the first chain it finds in the file which matches. The platform's certificates file can be used by calling *SSLContext.load_default_certs()*, this is done automatically with *create_default_context()*.

Combined key and certificate

Often the private key is stored in the same file as the certificate; in this case, only the `certfile` parameter to *SSLContext.load_cert_chain()* needs to be passed. If the private key is stored with the certificate, it should come before the first certificate in the certificate chain:

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Self-signed certificates

If you are going to create a server that provides SSL-encrypted connection services, you will need to acquire a certificate for that service. There are many ways of acquiring appropriate certificates, such as buying one from a certification authority. Another common practice is to generate a self-signed certificate. The simplest way to do this is with the OpenSSL package, using something like the following:

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
.......++++++
..............................++++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

The disadvantage of a self-signed certificate is that it is its own root certificate, and no one else will have it in their cache of known (and trusted) root certificates.

18.3.5 Examples

Testing for SSL support

To test for the presence of SSL support in a Python installation, user code should use the following idiom:

```
try:
    import ssl
except ImportError:
    pass
else:
    ...  # do something that requires SSL support
```

Client-side operation

This example creates a SSL context with the recommended security settings for client sockets, including automatic certificate verification:

```
>>> context = ssl.create_default_context()
```

If you prefer to tune security settings yourself, you might create a context from scratch (but beware that you might not get the settings right):

```
>>> context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(this snippet assumes your operating system places a bundle of all CA certificates in `/etc/ssl/certs/ca-bundle.crt`; if not, you'll get an error and have to adjust the location)

The *PROTOCOL_TLS_CLIENT* protocol configures the context for cert validation and hostname verification. *verify_mode* is set to *CERT_REQUIRED* and *check_hostname* is set to `True`. All other protocols create SSL contexts with insecure defaults.

When you use the context to connect to a server, *CERT_REQUIRED* and *check_hostname* validate the server certificate: it ensures that the server certificate was signed with one of the CA certificates, checks the signature for correctness, and verifies other properties like validity and identity of the hostname:

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                            server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

You may then fetch the certificate:

```
>>> cert = conn.getpeercert()
```

Visual inspection shows that the certificate does identify the desired service (that is, the HTTPS host `www.python.org`):

```
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationServerCA.crt
↪',),
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-server-g1.crl',
                           'http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'DigiCert Inc'),),
```

(continues on next page)

(continued from previous page)

```
           (('organizationalUnitName', 'www.digicert.com'),),
           (('commonName', 'DigiCert SHA2 Extended Validation Server CA'),)),
 'notAfter': 'Sep  9 12:00:00 2016 GMT',
 'notBefore': 'Sep  5 00:00:00 2014 GMT',
 'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
 'subject': ((('businessCategory', 'Private Organization'),),
             (('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
             (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
             (('serialNumber', '3359300'),),
             (('streetAddress', '16 Allen Rd'),),
             (('postalCode', '03894-4801'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'NH'),),
             (('localityName', 'Wolfeboro'),),
             (('organizationName', 'Python Software Foundation'),),
             (('commonName', 'www.python.org'),)),
 'subjectAltName': (('DNS', 'www.python.org'),
                    ('DNS', 'python.org'),
                    ('DNS', 'pypi.org'),
                    ('DNS', 'docs.python.org'),
                    ('DNS', 'testpypi.org'),
                    ('DNS', 'bugs.python.org'),
                    ('DNS', 'wiki.python.org'),
                    ('DNS', 'hg.python.org'),
                    ('DNS', 'mail.python.org'),
                    ('DNS', 'packaging.python.org'),
                    ('DNS', 'pythonhosted.org'),
                    ('DNS', 'www.pythonhosted.org'),
                    ('DNS', 'test.pythonhosted.org'),
                    ('DNS', 'us.pycon.org'),
                    ('DNS', 'id.python.org')),
 'version': 3}
```

Now the SSL channel is established and the certificate verified, you can proceed to talk with the server:

```
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
 b'Connection: close',
 b'',
 b'']
```

See the discussion of *Security considerations* below.

Server-side operation

For server operation, typically you'll need to have a server certificate, and private key, each in a file. You'll first create a context holding the key and the certificate, so that clients can check your authenticity. Then you'll open a socket, bind it to a port, call `listen()` on it, and start waiting for clients to connect:

```
import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="mycertfile", keyfile="mykeyfile")

bindsocket = socket.socket()
bindsocket.bind(('myaddr.example.com', 10023))
bindsocket.listen(5)
```

When a client connects, you'll call `accept()` on the socket to get the new socket from the other end, and use the context's *SSLContext.wrap_socket()* method to create a server-side SSL socket for the connection:

```
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

Then you'll read data from the `connstream` and do something with it till you are finished with the client (or the client is finished with you):

```
def deal_with_client(connstream):
    data = connstream.recv(1024)
    # empty data means the client is finished with us
    while data:
        if not do_something(connstream, data):
            # we'll assume do_something returns False
            # when we're finished with client
            break
        data = connstream.recv(1024)
    # finished with client
```

And go back to listening for new client connections (of course, a real server would probably handle each client connection in a separate thread, or put the sockets in *non-blocking mode* and use an event loop).

18.3.6 Notes on non-blocking sockets

SSL sockets behave slightly different than regular sockets in non-blocking mode. When working with non-blocking sockets, there are thus several things you need to be aware of:

- Most *SSLSocket* methods will raise either *SSLWantWriteError* or *SSLWantReadError* instead of *BlockingIOError* if an I/O operation would block. *SSLWantReadError* will be raised if a read operation on the underlying socket is necessary, and *SSLWantWriteError* for a write operation on the underlying socket. Note that attempts to *write* to an SSL socket may require *reading* from the underlying socket first, and attempts to *read* from the SSL socket may require a prior *write* to the underlying socket.

  Changed in version 3.5: In earlier Python versions, the `SSLSocket.send()` method returned zero instead of raising *SSLWantWriteError* or *SSLWantReadError*.

- Calling *select()* tells you that the OS-level socket can be read from (or written to), but it does not imply that there is sufficient data at the upper SSL layer. For example, only part of an SSL frame might have arrived. Therefore, you must be ready to handle `SSLSocket.recv()` and `SSLSocket.send()` failures, and retry after another call to *select()*.
- Conversely, since the SSL layer has its own framing, a SSL socket may still have data available for reading without *select()* being aware of it. Therefore, you should first call `SSLSocket.recv()` to drain any potentially available data, and then only block on a *select()* call if still necessary.

  (of course, similar provisions apply when using other primitives such as *poll()*, or those in the *selectors* module)
- The SSL handshake itself will be non-blocking: the *SSLSocket.do_handshake()* method has to be retried until it returns successfully. Here is a synopsis using *select()* to wait for the socket's readiness:

```
while True:
    try:
        sock.do_handshake()
        break
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
    except ssl.SSLWantWriteError:
        select.select([], [sock], [])
```

See also

The *asyncio* module supports *non-blocking SSL sockets* and provides a higher level API. It polls for events using the *selectors* module and handles *SSLWantWriteError*, *SSLWantReadError* and *BlockingIOError* exceptions. It runs the SSL handshake asynchronously as well.

18.3.7 Memory BIO Support

Added in version 3.5.

Ever since the SSL module was introduced in Python 2.6, the *SSLSocket* class has provided two related but distinct areas of functionality:

- SSL protocol handling
- Network IO

The network IO API is identical to that provided by *socket.socket*, from which *SSLSocket* also inherits. This allows an SSL socket to be used as a drop-in replacement for a regular socket, making it very easy to add SSL support to an existing application.

Combining SSL protocol handling and network IO usually works well, but there are some cases where it doesn't. An example is async IO frameworks that want to use a different IO multiplexing model than the "select/poll on a file descriptor" (readiness based) model that is assumed by *socket.socket* and by the internal OpenSSL socket IO routines. This is mostly relevant for platforms like Windows where this model is not efficient. For this purpose, a reduced scope variant of *SSLSocket* called *SSLObject* is provided.

class ssl.**SSLObject**

A reduced-scope variant of *SSLSocket* representing an SSL protocol instance that does not contain any network IO methods. This class is typically used by framework authors that want to implement asynchronous IO for SSL through memory buffers.

This class implements an interface on top of a low-level SSL object as implemented by OpenSSL. This object captures the state of an SSL connection but does not provide any network IO itself. IO needs to be performed through separate "BIO" objects which are OpenSSL's IO abstraction layer.

This class has no public constructor. An *SSLObject* instance must be created using the *wrap_bio()* method. This method will create the *SSLObject* instance and bind it to a pair of BIOs. The *incoming* BIO is used to pass data from Python to the SSL protocol instance, while the *outgoing* BIO is used to pass data the other way around.

The following methods are available:

- *context*
- *server_side*
- *server_hostname*
- *session*
- *session_reused*
- *read()*
- *write()*
- *getpeercert()*
- *get_verified_chain()*
- *get_unverified_chain()*
- *selected_alpn_protocol()*
- *selected_npn_protocol()*
- *cipher()*
- *shared_ciphers()*
- *compression()*
- *pending()*
- *do_handshake()*
- *verify_client_post_handshake()*
- *unwrap()*
- *get_channel_binding()*
- *version()*

When compared to *SSLSocket*, this object lacks the following features:

- Any form of network IO; `recv()` and `send()` read and write only to the underlying *MemoryBIO* buffers.
- There is no *do_handshake_on_connect* machinery. You must always manually call *do_handshake()* to start the handshake.
- There is no handling of *suppress_ragged_eofs*. All end-of-file conditions that are in violation of the protocol are reported via the *SSLEOFError* exception.
- The method *unwrap()* call does not return anything, unlike for an SSL socket where it returns the underlying socket.
- The *server_name_callback* callback passed to *SSLContext.set_servername_callback()* will get an *SSLObject* instance instead of a *SSLSocket* instance as its first parameter.

Some notes related to the use of *SSLObject*:

- All IO on an *SSLObject* is *non-blocking*. This means that for example *read()* will raise an *SSLWantReadError* if it needs more data than the incoming BIO has available.

Changed in version 3.7: *SSLObject* instances must be created with *wrap_bio()*. In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

An SSLObject communicates with the outside world using memory buffers. The class *MemoryBIO* provides a memory buffer that can be used for this purpose. It wraps an OpenSSL memory BIO (Basic IO) object:

class ssl.**MemoryBIO**

A memory buffer that can be used to pass data between Python and an SSL protocol instance.

pending

Return the number of bytes currently in the memory buffer.

eof

A boolean indicating whether the memory BIO is current at the end-of-file position.

read(*n=-1*)

Read up to *n* bytes from the memory buffer. If *n* is not specified or negative, all bytes are returned.

write(*buf*)

Write the bytes from *buf* to the memory BIO. The *buf* argument must be an object supporting the buffer protocol.

The return value is the number of bytes written, which is always equal to the length of *buf*.

write_eof()

Write an EOF marker to the memory BIO. After this method has been called, it is illegal to call *write()*. The attribute *eof* will become true after all data currently in the buffer has been read.

18.3.8 SSL session

Added in version 3.6.

class ssl.**SSLSession**

Session object used by *session*.

id

time

timeout

ticket_lifetime_hint

has_ticket

18.3.9 Security considerations

Best defaults

For **client use**, if you don't have any special requirements for your security policy, it is highly recommended that you use the *create_default_context()* function to create your SSL context. It will load the system's trusted CA certificates, enable certificate validation and hostname checking, and try to choose reasonably secure protocol and cipher settings.

For example, here is how you would use the *smtplib.SMTP* class to create a trusted, secure connection to a SMTP server:

```
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

If a client certificate is needed for the connection, it can be added with *SSLContext.load_cert_chain()*.

By contrast, if you create the SSL context by calling the *SSLContext* constructor yourself, it will not have certificate validation nor hostname checking enabled by default. If you do so, please read the paragraphs below to achieve a good security level.

Manual settings

Verifying certificates

When calling the *SSLContext* constructor directly, *CERT_NONE* is the default. Since it does not authenticate the other peer, it can be insecure, especially in client mode where most of time you would like to ensure the authenticity of the server you're talking to. Therefore, when in client mode, it is highly recommended to use *CERT_REQUIRED*. However, it is in itself not sufficient; you also have to check that the server certificate, which can be obtained by calling *SSLSocket.getpeercert()*, matches the desired service. For many protocols and applications, the service can be identified by the hostname. This common check is automatically performed when *SSLContext.check_hostname* is enabled.

Changed in version 3.7: Hostname matchings is now performed by OpenSSL. Python no longer uses `match_hostname()`.

In server mode, if you want to authenticate your clients using the SSL layer (rather than using a higher-level authentication mechanism), you'll also have to specify *CERT_REQUIRED* and similarly check the client certificate.

Protocol versions

SSL versions 2 and 3 are considered insecure and are therefore dangerous to use. If you want maximum compatibility between clients and servers, it is recommended to use *PROTOCOL_TLS_CLIENT* or *PROTOCOL_TLS_SERVER* as the protocol version. SSLv2 and SSLv3 are disabled by default.

```
>>> client_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> client_context.minimum_version = ssl.TLSVersion.TLSv1_3
>>> client_context.maximum_version = ssl.TLSVersion.TLSv1_3
```

The SSL context created above will only allow TLSv1.3 and later (if supported by your system) connections to a server. *PROTOCOL_TLS_CLIENT* implies certificate validation and hostname checks by default. You have to load certificates into the context.

Cipher selection

If you have advanced security requirements, fine-tuning of the ciphers enabled when negotiating a SSL session is possible through the *SSLContext.set_ciphers()* method. Starting from Python 3.2.3, the ssl module disables certain weak ciphers by default, but you may want to further restrict the cipher choice. Be sure to read OpenSSL's documentation about the cipher list format. If you want to check which ciphers are enabled by a given cipher list, use *SSLContext.get_ciphers()* or the `openssl ciphers` command on your system.

Multi-processing

If using this module as part of a multi-processed application (using, for example the *multiprocessing* or *concurrent.futures* modules), be aware that OpenSSL's internal random number generator does not properly handle forked processes. Applications must change the PRNG state of the parent process if they use any SSL feature with *os.fork()*. Any successful call of *RAND_add()* or *RAND_bytes()* is sufficient.

18.3.10 TLS 1.3

Added in version 3.7.

The TLS 1.3 protocol behaves slightly differently than previous version of TLS/SSL. Some new TLS 1.3 features are not yet available.

- TLS 1.3 uses a disjunct set of cipher suites. All AES-GCM and ChaCha20 cipher suites are enabled by default. The method *SSLContext.set_ciphers()* cannot enable or disable any TLS 1.3 ciphers yet, but *SSLContext.get_ciphers()* returns them.
- Session tickets are no longer sent as part of the initial handshake and are handled differently. *SSLSocket.session* and *SSLSession* are not compatible with TLS 1.3.
- Client-side certificates are also no longer verified during the initial handshake. A server can request a certificate at any time. Clients process certificate requests while they send or receive application data from the server.
- TLS 1.3 features like early data, deferred TLS client cert request, signature algorithm configuration, and rekeying are not supported yet.

See also

Class *socket.socket*
Documentation of underlying *socket* class

SSL/TLS Strong Encryption: An Introduction
Intro from the Apache HTTP Server documentation

RFC 1422: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management
Steve Kent

RFC 4086: Randomness Requirements for Security
Donald E., Jeffrey I. Schiller

RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile
D. Cooper

RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2
T. Dierks et. al.

RFC 6066: Transport Layer Security (TLS) Extensions
D. Eastlake

IANA TLS: Transport Layer Security (TLS) Parameters
IANA

RFC 7525: Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)
IETF

Mozilla's Server Side TLS recommendations
Mozilla

18.4 `select` — Waiting for I/O completion

This module provides access to the `select()` and `poll()` functions available in most operating systems, `devpoll()` available on Solaris and derivatives, `epoll()` available on Linux 2.5+ and `kqueue()` available on most BSD. Note that on Windows, it only works for sockets; on other operating systems, it also works for other file types (in particular, on Unix, it works on pipes). It cannot be used on regular files to determine whether a file has grown since it was last read.

Note

The *`selectors`* module allows high-level and efficient I/O multiplexing, built upon the *`select`* module primitives. Users are encouraged to use the *`selectors`* module instead, unless they want precise control over the OS-level primitives used.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

The module defines the following:

exception `select.error`

A deprecated alias of *`OSError`*.

Changed in version 3.3: Following **PEP 3151**, this class was made an alias of *`OSError`*.

`select.`**`devpoll`**`()`

(Only supported on Solaris and derivatives.) Returns a `/dev/poll` polling object; see section */dev/poll Polling Objects* below for the methods supported by devpoll objects.

`devpoll()` objects are linked to the number of file descriptors allowed at the time of instantiation. If your program reduces this value, `devpoll()` will fail. If your program increases this value, `devpoll()` may return an incomplete list of active file descriptors.

The new file descriptor is *non-inheritable*.

Added in version 3.3.

Changed in version 3.4: The new file descriptor is now non-inheritable.

select.**epoll**(*sizehint=-1*, *flags=0*)

(Only supported on Linux 2.5.44 and newer.) Return an edge polling object, which can be used as Edge or Level Triggered interface for I/O events.

sizehint informs epoll about the expected number of events to be registered. It must be positive, or `-1` to use the default. It is only used on older systems where `epoll_create1()` is not available; otherwise it has no effect (though its value is still checked).

flags is deprecated and completely ignored. However, when supplied, its value must be `0` or `select.EPOLL_CLOEXEC`, otherwise `OSError` is raised.

See the *Edge and Level Trigger Polling (epoll) Objects* section below for the methods supported by epolling objects.

`epoll` objects support the context management protocol: when used in a `with` statement, the new file descriptor is automatically closed at the end of the block.

The new file descriptor is *non-inheritable*.

Changed in version 3.3: Added the *flags* parameter.

Changed in version 3.4: Support for the `with` statement was added. The new file descriptor is now non-inheritable.

Deprecated since version 3.4: The *flags* parameter. `select.EPOLL_CLOEXEC` is used by default now. Use *os.set_inheritable()* to make the file descriptor inheritable.

select.**poll**()

(Not supported by all operating systems.) Returns a polling object, which supports registering and unregistering file descriptors, and then polling them for I/O events; see section *Polling Objects* below for the methods supported by polling objects.

select.**kqueue**()

(Only supported on BSD.) Returns a kernel queue object; see section *Kqueue Objects* below for the methods supported by kqueue objects.

The new file descriptor is *non-inheritable*.

Changed in version 3.4: The new file descriptor is now non-inheritable.

select.**kevent**(*ident*, *filter=KQ_FILTER_READ*, *flags=KQ_EV_ADD*, *fflags=0*, *data=0*, *udata=0*)

(Only supported on BSD.) Returns a kernel event object; see section *Kevent Objects* below for the methods supported by kevent objects.

select.**select**(*rlist*, *wlist*, *xlist*[, *timeout*])

This is a straightforward interface to the Unix `select()` system call. The first three arguments are iterables of 'waitable objects': either integers representing file descriptors or objects with a parameterless method named *fileno()* returning such an integer:

- *rlist*: wait until ready for reading
- *wlist*: wait until ready for writing
- *xlist*: wait for an "exceptional condition" (see the manual page for what your system considers such a condition)

Empty iterables are allowed, but acceptance of three empty iterables is platform-dependent. (It is known to work on Unix but not on Windows.) The optional *timeout* argument specifies a time-out as a floating-point number in seconds. When the *timeout* argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

The return value is a triple of lists of objects that are ready: subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.

Among the acceptable object types in the iterables are Python *file objects* (e.g. `sys.stdin`, or objects returned by *open()* or *os.popen()*), socket objects returned by *socket.socket()*. You may also define a *wrapper*

class yourself, as long as it has an appropriate *fileno()* method (that really returns a file descriptor, not just a random integer).

> **Note**
>
> File objects on Windows are not acceptable, but sockets are. On Windows, the underlying `select()` function is provided by the WinSock library, and does not handle file descriptors that don't originate from WinSock.

Changed in version 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

`select.`**`PIPE_BUF`**

The minimum number of bytes which can be written without blocking to a pipe when the pipe has been reported as ready for writing by *select()*, `poll()` or another interface in this module. This doesn't apply to other kind of file-like objects such as sockets.

This value is guaranteed by POSIX to be at least 512.

Availability: Unix

Added in version 3.2.

18.4.1 `/dev/poll` Polling Objects

Solaris and derivatives have `/dev/poll`. While `select()` is *O*(*highest file descriptor*) and `poll()` is *O*(*number of file descriptors*), `/dev/poll` is *O*(*active file descriptors*).

`/dev/poll` behaviour is very close to the standard `poll()` object.

`devpoll.`**`close`**`()`

Close the file descriptor of the polling object.

Added in version 3.4.

`devpoll.`**`closed`**

`True` if the polling object is closed.

Added in version 3.4.

`devpoll.`**`fileno`**`()`

Return the file descriptor number of the polling object.

Added in version 3.4.

`devpoll.`**`register`**`(`*fd*[, *eventmask*]`)`

Register a file descriptor with the polling object. Future calls to the *poll()* method will then check whether the file descriptor has any pending I/O events. *fd* can be either an integer, or an object with a *fileno()* method that returns an integer. File objects implement `fileno()`, so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for. The constants are the same that with `poll()` object. The default value is a combination of the constants `POLLIN`, `POLLPRI`, and `POLLOUT`.

> **Warning**
>
> Registering a file descriptor that's already registered is not an error, but the result is undefined. The appropriate action is to unregister or modify it first. This is an important difference compared with `poll()`.

devpoll.**modify**(*fd*[, *eventmask*])

This method does an *unregister()* followed by a *register()*. It is (a bit) more efficient that doing the same explicitly.

devpoll.**unregister**(*fd*)

Remove a file descriptor being tracked by a polling object. Just like the *register()* method, *fd* can be an integer or an object with a *fileno()* method that returns an integer.

Attempting to remove a file descriptor that was never registered is safely ignored.

devpoll.**poll**([*timeout*])

Polls the set of registered file descriptors, and returns a possibly empty list containing `(fd, event)` 2-tuples for the descriptors that have events or errors to report. *fd* is the file descriptor, and *event* is a bitmask with bits set for the reported events for that descriptor — `POLLIN` for waiting input, `POLLOUT` to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If *timeout* is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If *timeout* is omitted, -1, or *None*, the call will block until there is an event for this poll object.

Changed in version 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

18.4.2 Edge and Level Trigger Polling (epoll) Objects

https://linux.die.net/man/4/epoll

eventmask

Constant	Meaning
`EPOLLIN`	Available for read
`EPOLLOUT`	Available for write
`EPOLLPRI`	Urgent data for read
`EPOLLERR`	Error condition happened on the assoc. fd
`EPOLLHUP`	Hang up happened on the assoc. fd
`EPOLLET`	Set Edge Trigger behavior, the default is Level Trigger behavior
`EPOLLONESHOT`	Set one-shot behavior. After one event is pulled out, the fd is internally disabled
`EPOLLEXCLUSIVE`	Wake only one epoll object when the associated fd has an event. The default (if this flag is not set) is to wake all epoll objects polling on a fd.
`EPOLLRDHUP`	Stream socket peer closed connection or shut down writing half of connection.
`EPOLLRDNORM`	Equivalent to `EPOLLIN`
`EPOLLRDBAND`	Priority data band can be read.
`EPOLLWRNORM`	Equivalent to `EPOLLOUT`
`EPOLLWRBAND`	Priority data may be written.
`EPOLLMSG`	Ignored.

Added in version 3.6: `EPOLLEXCLUSIVE` was added. It's only supported by Linux Kernel 4.5 or later.

epoll.**close**()

Close the control file descriptor of the epoll object.

epoll.**closed**

True if the epoll object is closed.

epoll.**fileno**()

Return the file descriptor number of the control fd.

epoll.**fromfd**(*fd*)

Create an epoll object from a given file descriptor.

epoll.**register**(*fd*[, *eventmask*])

Register a fd descriptor with the epoll object.

epoll.**modify**(*fd*, *eventmask*)

Modify a registered file descriptor.

epoll.**unregister**(*fd*)

Remove a registered file descriptor from the epoll object.

Changed in version 3.9: The method no longer ignores the *EBADF* error.

epoll.**poll**(*timeout=None*, *maxevents=-1*)

Wait for events. timeout in seconds (float)

Changed in version 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

18.4.3 Polling Objects

The poll() system call, supported on most Unix systems, provides better scalability for network servers that service many, many clients at the same time. poll() scales better because the system call only requires listing the file descriptors of interest, while select() builds a bitmap, turns on bits for the fds of interest, and then afterward the whole bitmap has to be linearly scanned again. select() is *O*(*highest file descriptor*), while poll() is *O*(*number of file descriptors*).

poll.**register**(*fd*[, *eventmask*])

Register a file descriptor with the polling object. Future calls to the *poll()* method will then check whether the file descriptor has any pending I/O events. *fd* can be either an integer, or an object with a *fileno()* method that returns an integer. File objects implement fileno(), so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for, and can be a combination of the constants POLLIN, POLLPRI, and POLLOUT, described in the table below. If not specified, the default value used will check for all 3 types of events.

Constant	Meaning
POLLIN	There is data to read
POLLPRI	There is urgent data to read
POLLOUT	Ready for output: writing will not block
POLLERR	Error condition of some sort
POLLHUP	Hung up
POLLRDHUP	Stream socket peer closed connection, or shut down writing half of connection
POLLNVAL	Invalid request: descriptor not open

Registering a file descriptor that's already registered is not an error, and has the same effect as registering the descriptor exactly once.

poll.**modify**(*fd*, *eventmask*)

Modifies an already registered fd. This has the same effect as `register(fd, eventmask)`. Attempting to modify a file descriptor that was never registered causes an *OSError* exception with errno ENOENT to be raised.

poll.**unregister**(*fd*)

Remove a file descriptor being tracked by a polling object. Just like the *register()* method, *fd* can be an integer or an object with a *fileno()* method that returns an integer.

Attempting to remove a file descriptor that was never registered causes a *KeyError* exception to be raised.

poll.**poll**([*timeout*])

Polls the set of registered file descriptors, and returns a possibly empty list containing `(fd, event)` 2-tuples for the descriptors that have events or errors to report. *fd* is the file descriptor, and *event* is a bitmask with bits set for the reported events for that descriptor — `POLLIN` for waiting input, `POLLOUT` to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If *timeout* is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If *timeout* is omitted, negative, or *None*, the call will block until there is an event for this poll object.

Changed in version 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

18.4.4 Kqueue Objects

kqueue.**close**()

Close the control file descriptor of the kqueue object.

kqueue.**closed**

`True` if the kqueue object is closed.

kqueue.**fileno**()

Return the file descriptor number of the control fd.

kqueue.**fromfd**(*fd*)

Create a kqueue object from a given file descriptor.

kqueue.**control**(*changelist*, *max_events*[, *timeout*]) → eventlist

Low level interface to kevent

- changelist must be an iterable of kevent objects or `None`
- max_events must be 0 or a positive integer
- timeout in seconds (floats possible); the default is `None`, to wait forever

Changed in version 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

18.4.5 Kevent Objects

https://man.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

`kevent.`**`ident`**

Value used to identify the event. The interpretation depends on the filter but it's usually the file descriptor. In the constructor ident can either be an int or an object with a *`fileno()`* method. kevent stores the integer internally.

`kevent.`**`filter`**

Name of the kernel filter.

Constant	Meaning
`KQ_FILTER_READ`	Takes a descriptor and returns whenever there is data available to read
`KQ_FILTER_WRITE`	Takes a descriptor and returns whenever there is data available to write
`KQ_FILTER_AIO`	AIO requests
`KQ_FILTER_VNODE`	Returns when one or more of the requested events watched in *fflag* occurs
`KQ_FILTER_PROC`	Watch for events on a process id
`KQ_FILTER_NETDEV`	Watch for events on a network device [not available on macOS]
`KQ_FILTER_SIGNAL`	Returns whenever the watched signal is delivered to the process
`KQ_FILTER_TIMER`	Establishes an arbitrary timer

`kevent.`**`flags`**

Filter action.

Constant	Meaning
`KQ_EV_ADD`	Adds or modifies an event
`KQ_EV_DELETE`	Removes an event from the queue
`KQ_EV_ENABLE`	Permitscontrol() to returns the event
`KQ_EV_DISABLE`	Disablesevent
`KQ_EV_ONESHOT`	Removes event after first occurrence
`KQ_EV_CLEAR`	Reset the state after an event is retrieved
`KQ_EV_SYSFLAGS`	internal event
`KQ_EV_FLAG1`	internal event
`KQ_EV_EOF`	Filter specific EOF condition
`KQ_EV_ERROR`	See return values

`kevent.`**`fflags`**

Filter specific flags.

`KQ_FILTER_READ` and `KQ_FILTER_WRITE` filter flags:

Constant	Meaning
`KQ_NOTE_LOWAT`	low water mark of a socket buffer

`KQ_FILTER_VNODE` filter flags:

Constant	Meaning
KQ_NOTE_DELETE	*unlink()* was called
KQ_NOTE_WRITE	a write occurred
KQ_NOTE_EXTEND	the file was extended
KQ_NOTE_ATTRIB	an attribute was changed
KQ_NOTE_LINK	the link count has changed
KQ_NOTE_RENAME	the file was renamed
KQ_NOTE_REVOKE	access to the file was revoked

KQ_FILTER_PROC filter flags:

Constant	Meaning
KQ_NOTE_EXIT	the process has exited
KQ_NOTE_FORK	the process has called *fork()*
KQ_NOTE_EXEC	the process has executed a new process
KQ_NOTE_PCTRLMASK	internal filter flag
KQ_NOTE_PDATAMASK	internal filter flag
KQ_NOTE_TRACK	follow a process across *fork()*
KQ_NOTE_CHILD	returned on the child process for *NOTE_TRACK*
KQ_NOTE_TRACKERR	unable to attach to a child

KQ_FILTER_NETDEV filter flags (not available on macOS):

Constant	Meaning
KQ_NOTE_LINKUP	link is up
KQ_NOTE_LINKDOWN	link is down
KQ_NOTE_LINKINV	link state is invalid

kevent.**data**

Filter specific data.

kevent.**udata**

User defined value.

18.5 selectors — High-level I/O multiplexing

Added in version 3.4.

Source code: Lib/selectors.py

18.5.1 Introduction

This module allows high-level and efficient I/O multiplexing, built upon the `select` module primitives. Users are encouraged to use this module instead, unless they want precise control over the OS-level primitives used.

It defines a `BaseSelector` abstract base class, along with several concrete implementations (`KqueueSelector`, `EpollSelector`...), that can be used to wait for I/O readiness notification on multiple file objects. In the following, "file object" refers to any object with a `fileno()` method, or a raw file descriptor. See *file object*.

`DefaultSelector` is an alias to the most efficient implementation available on the current platform: this should be the default choice for most users.

Note

The type of file objects supported depends on the platform: on Windows, sockets are supported, but not pipes, whereas on Unix, both are supported (some other types may be supported as well, such as fifos or special file devices).

See also

`select`
Low-level I/O multiplexing module.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

18.5.2 Classes

Classes hierarchy:

```
BaseSelector
+-- SelectSelector
+-- PollSelector
+-- EpollSelector
+-- DevpollSelector
+-- KqueueSelector
```

In the following, *events* is a bitwise mask indicating which I/O events should be waited for on a given file object. It can be a combination of the modules constants below:

Constant	Meaning
`selectors.`**`EVENT_READ`**	Available for read
`selectors.`**`EVENT_WRITE`**	Available for write

class selectors.**SelectorKey**

A *SelectorKey* is a *namedtuple* used to associate a file object to its underlying file descriptor, selected event mask and attached data. It is returned by several *BaseSelector* methods.

fileobj

File object registered.

fd

Underlying file descriptor.

events

Events that must be waited for on this file object.

data

Optional opaque data associated to this file object: for example, this could be used to store a per-client session ID.

class selectors.**BaseSelector**

A *BaseSelector* is used to wait for I/O event readiness on multiple file objects. It supports file stream registration, unregistration, and a method to wait for I/O events on those streams, with an optional timeout. It's an abstract base class, so cannot be instantiated. Use *DefaultSelector* instead, or one of *SelectSelector*, *KqueueSelector* etc. if you want to specifically use an implementation, and your platform supports it. *BaseSelector* and its concrete implementations support the *context manager* protocol.

abstractmethod register(*fileobj*, *events*, *data=None*)

Register a file object for selection, monitoring it for I/O events.

fileobj is the file object to monitor. It may either be an integer file descriptor or an object with a `fileno()` method. *events* is a bitwise mask of events to monitor. *data* is an opaque object.

This returns a new *SelectorKey* instance, or raises a *ValueError* in case of invalid event mask or file descriptor, or *KeyError* if the file object is already registered.

abstractmethod unregister(*fileobj*)

Unregister a file object from selection, removing it from monitoring. A file object shall be unregistered prior to being closed.

fileobj must be a file object previously registered.

This returns the associated *SelectorKey* instance, or raises a *KeyError* if *fileobj* is not registered. It will raise *ValueError* if *fileobj* is invalid (e.g. it has no `fileno()` method or its `fileno()` method has an invalid return value).

modify(*fileobj*, *events*, *data=None*)

Change a registered file object's monitored events or attached data.

This is equivalent to `BaseSelector.unregister(fileobj)` followed by `BaseSelector.register(fileobj, events, data)`, except that it can be implemented more efficiently.

This returns a new *SelectorKey* instance, or raises a *ValueError* in case of invalid event mask or file descriptor, or *KeyError* if the file object is not registered.

abstractmethod select(*timeout=None*)

Wait until some registered file objects become ready, or the timeout expires.

If `timeout > 0`, this specifies the maximum wait time, in seconds. If `timeout <= 0`, the call won't block, and will report the currently ready file objects. If *timeout* is `None`, the call will block until a monitored file object becomes ready.

This returns a list of `(key, events)` tuples, one for each ready file object.

key is the *SelectorKey* instance corresponding to a ready file object. *events* is a bitmask of events ready on this file object.

> **Note**
>
> This method can return before any file object becomes ready or the timeout has elapsed if the current process receives a signal: in this case, an empty list will be returned.

Changed in version 3.5: The selector is now retried with a recomputed timeout when interrupted by a signal if the signal handler did not raise an exception (see **PEP 475** for the rationale), instead of returning an empty list of events before the timeout.

close()

Close the selector.

This must be called to make sure that any underlying resource is freed. The selector shall not be used once it has been closed.

get_key(*fileobj*)

Return the key associated with a registered file object.

This returns the *SelectorKey* instance associated to this file object, or raises *KeyError* if the file object is not registered.

abstractmethod get_map()

Return a mapping of file objects to selector keys.

This returns a *Mapping* instance mapping registered file objects to their associated *SelectorKey* instance.

class selectors.**DefaultSelector**

The default selector class, using the most efficient implementation available on the current platform. This should be the default choice for most users.

class selectors.**SelectSelector**

select.select()-based selector.

class selectors.**PollSelector**

select.poll()-based selector.

class selectors.**EpollSelector**

select.epoll()-based selector.

fileno()

This returns the file descriptor used by the underlying *select.epoll()* object.

class selectors.**DevpollSelector**

select.devpoll()-based selector.

fileno()

This returns the file descriptor used by the underlying *select.devpoll()* object.

Added in version 3.5.

class selectors.**KqueueSelector**

select.kqueue()-based selector.

fileno()

This returns the file descriptor used by the underlying *select.kqueue()* object.

18.5.3 Examples

Here is a simple echo server implementation:

```
import selectors
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept()  # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)

def read(conn, mask):
    data = conn.recv(1000)  # Should be ready
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # Hope it won't block
    else:
        print('closing', conn)
        sel.unregister(conn)
        conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

18.6 `signal` — Set handlers for asynchronous events

Source code: Lib/signal.py

This module provides mechanisms to use signal handlers in Python.

18.6.1 General rules

The *signal.signal()* function allows defining custom handlers to be executed when a signal is received. A small number of default handlers are installed: *SIGPIPE* is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions) and *SIGINT* is translated into a *KeyboardInterrupt* exception if the parent process has not changed it.

A handler for a particular signal, once set, remains installed until it is explicitly reset (Python emulates the BSD style interface regardless of the underlying implementation), with the exception of the handler for *SIGCHLD*, which follows the underlying implementation.

On WebAssembly platforms, signals are emulated and therefore behave differently. Several functions and signals are not available on these platforms.

Execution of Python signal handlers

A Python signal handler does not get executed inside the low-level (C) signal handler. Instead, the low-level signal handler sets a flag which tells the *virtual machine* to execute the corresponding Python signal handler at a later point(for example at the next *bytecode* instruction). This has consequences:

- It makes little sense to catch synchronous errors like `SIGFPE` or `SIGSEGV` that are caused by an invalid operation in C code. Python will return from the signal handler to the C code, which is likely to raise the same signal again, causing Python to apparently hang. From Python 3.3 onwards, you can use the `faulthandler` module to report on synchronous errors.
- A long-running calculation implemented purely in C (such as regular expression matching on a large body of text) may run uninterrupted for an arbitrary amount of time, regardless of any signals received. The Python signal handlers will be called when the calculation finishes.
- If the handler raises an exception, it will be raised "out of thin air" in the main thread. See the *note below* for a discussion.

Signals and threads

Python signal handlers are always executed in the main Python thread of the main interpreter, even if the signal was received in another thread. This means that signals can't be used as a means of inter-thread communication. You can use the synchronization primitives from the `threading` module instead.

Besides, only the main thread of the main interpreter is allowed to set a new signal handler.

18.6.2 Module contents

Changed in version 3.5: signal (SIG*), handler (`SIG_DFL`, `SIG_IGN`) and sigmask (`SIG_BLOCK`, `SIG_UNBLOCK`, `SIG_SETMASK`) related constants listed below were turned into `enums` (`Signals`, `Handlers` and `Sigmasks` respectively). `getsignal()`, `pthread_sigmask()`, `sigpending()` and `sigwait()` functions return human-readable `enums` as `Signals` objects.

The signal module defines three enums:

class signal.**Signals**

`enum.IntEnum` collection of SIG* constants and the CTRL_* constants.

Added in version 3.5.

class signal.**Handlers**

`enum.IntEnum` collection the constants `SIG_DFL` and `SIG_IGN`.

Added in version 3.5.

class signal.**Sigmasks**

`enum.IntEnum` collection the constants `SIG_BLOCK`, `SIG_UNBLOCK` and `SIG_SETMASK`.

Availability: Unix.

See the man page `sigprocmask(2)` and `pthread_sigmask(3)` for further information.

Added in version 3.5.

The variables defined in the `signal` module are:

signal.**SIG_DFL**

This is one of two standard signal handling options; it will simply perform the default function for the signal. For example, on most systems the default action for SIGQUIT is to dump core and exit, while the default action for *SIGCHLD* is to simply ignore it.

signal.**SIG_IGN**

This is another standard signal handler, which will simply ignore the given signal.

signal.**SIGABRT**

Abort signal from *abort(3)*.

signal.**SIGALRM**

Timer signal from *alarm(2)*.

Availability: Unix.

signal.**SIGBREAK**

Interrupt from keyboard (CTRL + BREAK).

Availability: Windows.

signal.**SIGBUS**

Bus error (bad memory access).

Availability: Unix.

signal.**SIGCHLD**

Child process stopped or terminated.

Availability: Unix.

signal.**SIGCLD**

Alias to *SIGCHLD*.

Availability: not macOS.

signal.**SIGCONT**

Continue the process if it is currently stopped

Availability: Unix.

signal.**SIGFPE**

Floating-point exception. For example, division by zero.

> **See also**
>
> *ZeroDivisionError* is raised when the second argument of a division or modulo operation is zero.

signal.**SIGHUP**

Hangup detected on controlling terminal or death of controlling process.

Availability: Unix.

signal.**SIGILL**

Illegal instruction.

signal.**SIGINT**
Interrupt from keyboard (CTRL + C).

Default action is to raise *KeyboardInterrupt*.

signal.**SIGKILL**
Kill signal.

It cannot be caught, blocked, or ignored.

Availability: Unix.

signal.**SIGPIPE**
Broken pipe: write to pipe with no readers.

Default action is to ignore the signal.

Availability: Unix.

signal.**SIGSEGV**
Segmentation fault: invalid memory reference.

signal.**SIGSTKFLT**

Stack fault on coprocessor. The Linux kernel does not raise this signal: it can only be raised in user space.

Availability: Linux.

On architectures where the signal is available. See the man page *signal(7)* for further information.

Added in version 3.11.

signal.**SIGTERM**
Termination signal.

signal.**SIGUSR1**
User-defined signal 1.

Availability: Unix.

signal.**SIGUSR2**
User-defined signal 2.

Availability: Unix.

signal.**SIGWINCH**
Window resize signal.

Availability: Unix.

SIG*
All the signal numbers are defined symbolically. For example, the hangup signal is defined as *signal.SIGHUP*; the variable names are identical to the names used in C programs, as found in <signal.h>. The Unix man page for 'signal()' lists the existing signals (on some systems this is *signal(2)*, on others the list is in *signal(7)*). Note that not all systems define the same set of signal names; only those names defined by the system are defined by this module.

signal.**CTRL_C_EVENT**
The signal corresponding to the Ctrl+C keystroke event. This signal can only be used with *os.kill()*.

Availability: Windows.

Added in version 3.2.

signal.**CTRL_BREAK_EVENT**

The signal corresponding to the `Ctrl+Break` keystroke event. This signal can only be used with *os.kill()*.

Availability: Windows.

Added in version 3.2.

signal.**NSIG**

One more than the number of the highest signal number. Use *valid_signals()* to get valid signal numbers.

signal.**ITIMER_REAL**

Decrements interval timer in real time, and delivers *SIGALRM* upon expiration.

signal.**ITIMER_VIRTUAL**

Decrements interval timer only when the process is executing, and delivers SIGVTALRM upon expiration.

signal.**ITIMER_PROF**

Decrements interval timer both when the process executes and when the system is executing on behalf of the process. Coupled with ITIMER_VIRTUAL, this timer is usually used to profile the time spent by the application in user and kernel space. SIGPROF is delivered upon expiration.

signal.**SIG_BLOCK**

A possible value for the *how* parameter to *pthread_sigmask()* indicating that signals are to be blocked.

Added in version 3.3.

signal.**SIG_UNBLOCK**

A possible value for the *how* parameter to *pthread_sigmask()* indicating that signals are to be unblocked.

Added in version 3.3.

signal.**SIG_SETMASK**

A possible value for the *how* parameter to *pthread_sigmask()* indicating that the signal mask is to be replaced.

Added in version 3.3.

The *signal* module defines one exception:

exception signal.**ItimerError**

Raised to signal an error from the underlying *setitimer()* or *getitimer()* implementation. Expect this error if an invalid interval timer or a negative time is passed to *setitimer()*. This error is a subtype of *OSError*.

Added in version 3.3: This error used to be a subtype of *IOError*, which is now an alias of *OSError*.

The *signal* module defines the following functions:

signal.**alarm**(*time*)

If *time* is non-zero, this function requests that a *SIGALRM* signal be sent to the process in *time* seconds. Any previously scheduled alarm is canceled (only one alarm can be scheduled at any time). The returned value is then the number of seconds before any previously set alarm was to have been delivered. If *time* is zero, no alarm is scheduled, and any scheduled alarm is canceled. If the return value is zero, no alarm is currently scheduled.

Availability: Unix.

See the man page *alarm(2)* for further information.

signal.**getsignal**(*signalnum*)

Return the current signal handler for the signal *signalnum*. The returned value may be a callable Python object, or one of the special values *signal.SIG_IGN*, *signal.SIG_DFL* or *None*. Here, *signal.SIG_IGN* means that the signal was previously ignored, *signal.SIG_DFL* means that the default way of handling the signal was previously in use, and None means that the previous signal handler was not installed from Python.

signal.**strsignal**(*signalnum*)

Returns the description of signal *signalnum*, such as "Interrupt" for *SIGINT*. Returns *None* if *signalnum* has no description. Raises *ValueError* if *signalnum* is invalid.

Added in version 3.8.

signal.**valid_signals**()

Return the set of valid signal numbers on this platform. This can be less than range(1, NSIG) if some signals are reserved by the system for internal use.

Added in version 3.8.

signal.**pause**()

Cause the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing.

Availability: Unix.

See the man page *signal(2)* for further information.

See also *sigwait()*, *sigwaitinfo()*, *sigtimedwait()* and *sigpending()*.

signal.**raise_signal**(*signum*)

Sends a signal to the calling process. Returns nothing.

Added in version 3.8.

signal.**pidfd_send_signal**(*pidfd*, *sig*, *siginfo=None*, *flags=0*)

Send signal *sig* to the process referred to by file descriptor *pidfd*. Python does not currently support the *siginfo* parameter; it must be None. The *flags* argument is provided for future extensions; no flag values are currently defined.

See the *pidfd_send_signal(2)* man page for more information.

Availability: Linux >= 5.1

Added in version 3.9.

signal.**pthread_kill**(*thread_id*, *signalnum*)

Send the signal *signalnum* to the thread *thread_id*, another thread in the same process as the caller. The target thread can be executing any code (Python or not). However, if the target thread is executing the Python interpreter, the Python signal handlers will be *executed by the main thread of the main interpreter*. Therefore, the only point of sending a signal to a particular Python thread would be to force a running system call to fail with *InterruptedError*.

Use *threading.get_ident()* or the *ident* attribute of *threading.Thread* objects to get a suitable value for *thread_id*.

If *signalnum* is 0, then no signal is sent, but error checking is still performed; this can be used to check if the target thread is still running.

Raises an *auditing event* signal.pthread_kill with arguments thread_id, signalnum.

Availability: Unix.

See the man page *pthread_kill(3)* for further information.

See also *os.kill()*.

Added in version 3.3.

signal.**pthread_sigmask**(*how*, *mask*)

Fetch and/or change the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller. Return the old signal mask as a set of signals.

The behavior of the call is dependent on the value of *how*, as follows.

- *SIG_BLOCK*: The set of blocked signals is the union of the current set and the *mask* argument.
- *SIG_UNBLOCK*: The signals in *mask* are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.
- *SIG_SETMASK*: The set of blocked signals is set to the *mask* argument.

mask is a set of signal numbers (e.g. {*signal.SIGINT*, *signal.SIGTERM*}). Use *valid_signals()* for a full mask including all signals.

For example, `signal.pthread_sigmask(signal.SIG_BLOCK, [])` reads the signal mask of the calling thread.

SIGKILL and SIGSTOP cannot be blocked.

Availability: Unix.

See the man page *sigprocmask(2)* and *pthread_sigmask(3)* for further information.

See also *pause()*, *sigpending()* and *sigwait()*.

Added in version 3.3.

signal.**setitimer**(*which*, *seconds*, *interval=0.0*)

Sets given interval timer (one of *signal.ITIMER_REAL*, *signal.ITIMER_VIRTUAL* or *signal.ITIMER_PROF*) specified by *which* to fire after *seconds* (float is accepted, different from *alarm()*) and after that every *interval* seconds (if *interval* is non-zero). The interval timer specified by *which* can be cleared by setting *seconds* to zero.

When an interval timer fires, a signal is sent to the process. The signal sent is dependent on the timer being used; *signal.ITIMER_REAL* will deliver *SIGALRM*, *signal.ITIMER_VIRTUAL* sends SIGVTALRM, and *signal.ITIMER_PROF* will deliver SIGPROF.

The old values are returned as a tuple: (delay, interval).

Attempting to pass an invalid interval timer will cause an *ItimerError*.

Availability: Unix.

signal.**getitimer**(*which*)

Returns current value of a given interval timer specified by *which*.

Availability: Unix.

signal.**set_wakeup_fd**(*fd*, *, *warn_on_full_buffer=True*)

Set the wakeup file descriptor to *fd*. When a signal is received, the signal number is written as a single byte into the fd. This can be used by a library to wakeup a poll or select call, allowing the signal to be fully processed.

The old wakeup fd is returned (or -1 if file descriptor wakeup was not enabled). If *fd* is -1, file descriptor wakeup is disabled. If not -1, *fd* must be non-blocking. It is up to the library to remove any bytes from *fd* before calling poll or select again.

When threads are enabled, this function can only be called from *the main thread of the main interpreter*; attempting to call it from other threads will cause a *ValueError* exception to be raised.

There are two common ways to use this function. In both approaches, you use the fd to wake up when a signal arrives, but then they differ in how they determine *which* signal or signals have arrived.

In the first approach, we read the data out of the fd's buffer, and the byte values give you the signal numbers. This is simple, but in rare cases it can run into a problem: generally the fd will have a limited amount of buffer space, and if too many signals arrive too quickly, then the buffer may become full, and some signals may be lost. If you use this approach, then you should set `warn_on_full_buffer=True`, which will at least cause a warning to be printed to stderr when signals are lost.

In the second approach, we use the wakeup fd *only* for wakeups, and ignore the actual byte values. In this case, all we care about is whether the fd's buffer is empty or non-empty; a full buffer doesn't indicate a problem at all. If you use this approach, then you should set `warn_on_full_buffer=False`, so that your users are not confused by spurious warning messages.

Changed in version 3.5: On Windows, the function now also supports socket handles.

Changed in version 3.7: Added `warn_on_full_buffer` parameter.

`signal.`**`siginterrupt`**(*signalnum*, *flag*)

Change system call restart behaviour: if *flag* is `False`, system calls will be restarted when interrupted by signal *signalnum*, otherwise system calls will be interrupted. Returns nothing.

Availability: Unix.

See the man page `siginterrupt(3)` for further information.

Note that installing a signal handler with `signal()` will reset the restart behaviour to interruptible by implicitly calling `siginterrupt()` with a true *flag* value for the given signal.

`signal.`**`signal`**(*signalnum*, *handler*)

Set the handler for signal *signalnum* to the function *handler*. *handler* can be a callable Python object taking two arguments (see below), or one of the special values `signal.SIG_IGN` or `signal.SIG_DFL`. The previous signal handler will be returned (see the description of `getsignal()` above). (See the Unix man page `signal(2)` for further information.)

When threads are enabled, this function can only be called from *the main thread of the main interpreter*; attempting to call it from other threads will cause a `ValueError` exception to be raised.

The *handler* is called with two arguments: the signal number and the current stack frame (`None` or a frame object; for a description of frame objects, see the description in the type hierarchy or see the attribute descriptions in the `inspect` module).

On Windows, `signal()` can only be called with `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, `SIGTERM`, or `SIGBREAK`. A `ValueError` will be raised in any other case. Note that not all systems define the same set of signal names; an `AttributeError` will be raised if a signal name is not defined as `SIG*` module level constant.

`signal.`**`sigpending`**()

Examine the set of signals that are pending for delivery to the calling thread (i.e., the signals which have been raised while blocked). Return the set of the pending signals.

Availability: Unix.

See the man page `sigpending(2)` for further information.

See also `pause()`, `pthread_sigmask()` and `sigwait()`.

Added in version 3.3.

`signal.`**`sigwait`**(*sigset*)

Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal (removes it from the pending list of signals), and returns the signal number.

Availability: Unix.

See the man page `sigwait(3)` for further information.

See also *pause()*, *pthread_sigmask()*, *sigpending()*, *sigwaitinfo()* and *sigtimedwait()*.

Added in version 3.3.

signal.**sigwaitinfo**(*sigset*)

Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal and removes it from the pending list of signals. If one of the signals in *sigset* is already pending for the calling thread, the function will return immediately with information about that signal. The signal handler is not called for the delivered signal. The function raises an *InterruptedError* if it is interrupted by a signal that is not in *sigset*.

The return value is an object representing the data contained in the `siginfo_t` structure, namely: `si_signo`, `si_code`, `si_errno`, `si_pid`, `si_uid`, `si_status`, `si_band`.

Availability: Unix.

See the man page *sigwaitinfo(2)* for further information.

See also *pause()*, *sigwait()* and *sigtimedwait()*.

Added in version 3.3.

Changed in version 3.5: The function is now retried if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see **PEP 475** for the rationale).

signal.**sigtimedwait**(*sigset*, *timeout*)

Like *sigwaitinfo()*, but takes an additional *timeout* argument specifying a timeout. If *timeout* is specified as `0`, a poll is performed. Returns *None* if a timeout occurs.

Availability: Unix.

See the man page *sigtimedwait(2)* for further information.

See also *pause()*, *sigwait()* and *sigwaitinfo()*.

Added in version 3.3.

Changed in version 3.5: The function is now retried with the recomputed *timeout* if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see **PEP 475** for the rationale).

18.6.3 Examples

Here is a minimal example program. It uses the *alarm()* function to limit the time spent waiting to open a file; this is useful if the file is for a serial device that may not be turned on, which would normally cause the *os.open()* to hang indefinitely. The solution is to set a 5-second alarm before opening the file; if the operation takes too long, the alarm signal will be sent, and the handler raises an exception.

```
import signal, os

def handler(signum, frame):
    signame = signal.Signals(signum).name
    print(f'Signal handler called with signal {signame} ({signum})')
    raise OSError("Couldn't open device!")

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)
```

(continues on next page)

(continued from previous page)

```
signal.alarm(0)          # Disable the alarm
```

18.6.4 Note on SIGPIPE

Piping output of your program to tools like *head(1)* will cause a *SIGPIPE* signal to be sent to your process when the receiver of its standard output closes early. This results in an exception like `BrokenPipeError: [Errno 32] Broken pipe`. To handle this case, wrap your entry point to catch this exception as follows:

```
import os
import sys

def main():
    try:
        # simulate large output (your code replaces this loop)
        for x in range(10000):
            print("y")
        # flush output here to force SIGPIPE to be triggered
        # while inside this try block.
        sys.stdout.flush()
    except BrokenPipeError:
        # Python flushes standard streams on exit; redirect remaining output
        # to devnull to avoid another BrokenPipeError at shutdown
        devnull = os.open(os.devnull, os.O_WRONLY)
        os.dup2(devnull, sys.stdout.fileno())
        sys.exit(1)  # Python exits with error code 1 on EPIPE

if __name__ == '__main__':
    main()
```

Do not set *SIGPIPE*'s disposition to *SIG_DFL* in order to avoid *BrokenPipeError*. Doing that would cause your program to exit unexpectedly whenever any socket connection is interrupted while your program is still writing to it.

18.6.5 Note on Signal Handlers and Exceptions

If a signal handler raises an exception, the exception will be propagated to the main thread and may be raised after any *bytecode* instruction. Most notably, a *KeyboardInterrupt* may appear at any point during execution. Most Python code, including the standard library, cannot be made robust against this, and so a *KeyboardInterrupt* (or any other exception resulting from a signal handler) may on rare occasions put the program in an unexpected state.

To illustrate this issue, consider the following code:

```
class SpamContext:
    def __init__(self):
        self.lock = threading.Lock()

    def __enter__(self):
        # If KeyboardInterrupt occurs here, everything is fine
        self.lock.acquire()
        # If KeyboardInterrupt occurs here, __exit__ will not be called
        ...
        # KeyboardInterrupt could occur just before the function returns
```

(continues on next page)

(continued from previous page)

```
    def __exit__(self, exc_type, exc_val, exc_tb):
        ...
        self.lock.release()
```

For many programs, especially those that merely want to exit on *KeyboardInterrupt*, this is not a problem, but applications that are complex or require high reliability should avoid raising exceptions from signal handlers. They should also avoid catching *KeyboardInterrupt* as a means of gracefully shutting down. Instead, they should install their own *SIGINT* handler. Below is an example of an HTTP server that avoids *KeyboardInterrupt*:

```
import signal
import socket
from selectors import DefaultSelector, EVENT_READ
from http.server import HTTPServer, SimpleHTTPRequestHandler

interrupt_read, interrupt_write = socket.socketpair()

def handler(signum, frame):
    print('Signal handler called with signal', signum)
    interrupt_write.send(b'\0')
signal.signal(signal.SIGINT, handler)

def serve_forever(httpd):
    sel = DefaultSelector()
    sel.register(interrupt_read, EVENT_READ)
    sel.register(httpd, EVENT_READ)

    while True:
        for key, _ in sel.select():
            if key.fileobj == interrupt_read:
                interrupt_read.recv(1)
                return
            if key.fileobj == httpd:
                httpd.handle_request()

print("Serving on port 8000")
httpd = HTTPServer(('', 8000), SimpleHTTPRequestHandler)
serve_forever(httpd)
print("Shutdown...")
```

18.7 mmap — Memory-mapped file support

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

Memory-mapped file objects behave like both *bytearray* and like *file objects*. You can use mmap objects in most places where *bytearray* are expected; for example, you can use the *re* module to search through a memory-mapped file. You can also change a single byte by doing `obj[index] = 97`, or change a subsequence by assigning to a slice: `obj[i1:i2] = b'...'`. You can also read and write data starting at the current file position, and `seek()` through the file to different positions.

A memory-mapped file is created by the *mmap* constructor, which is different on Unix and on Windows. In either case you must provide a file descriptor for a file opened for update. If you wish to map an existing Python file object, use

its *fileno()* method to obtain the correct value for the *fileno* parameter. Otherwise, you can open the file using the *os.open()* function, which returns a file descriptor directly (the file still needs to be closed when done).

> **Note**
>
> If you want to create a memory-mapping for a writable, buffered file, you should *flush()* the file first. This is necessary to ensure that local modifications to the buffers are actually available to the mapping.

For both the Unix and Windows versions of the constructor, *access* may be specified as an optional keyword parameter. *access* accepts one of four values: ACCESS_READ, ACCESS_WRITE, or ACCESS_COPY to specify read-only, write-through or copy-on-write memory respectively, or ACCESS_DEFAULT to defer to *prot*. *access* can be used on both Unix and Windows. If *access* is not specified, Windows mmap returns a write-through mapping. The initial memory values for all three access types are taken from the specified file. Assignment to an ACCESS_READ memory map raises a *TypeError* exception. Assignment to an ACCESS_WRITE memory map affects both memory and the underlying file. Assignment to an ACCESS_COPY memory map affects memory but does not update the underlying file.

Changed in version 3.7: Added ACCESS_DEFAULT constant.

To map anonymous memory, -1 should be passed as the fileno along with the length.

class mmap.**mmap**(*fileno*, *length*, *tagname=None*, *access=ACCESS_DEFAULT*, *offset=0*)

(Windows version) Maps *length* bytes from the file specified by the file handle *fileno*, and creates a mmap object. If *length* is larger than the current size of the file, the file is extended to contain *length* bytes. If *length* is 0, the maximum length of the map is the current size of the file, except that if the file is empty Windows raises an exception (you cannot create an empty mapping on Windows).

tagname, if specified and not None, is a string giving a tag name for the mapping. Windows allows you to have many different mappings against the same file. If you specify the name of an existing tag, that tag is opened, otherwise a new tag of this name is created. If this parameter is omitted or None, the mapping is created without a name. Avoiding the use of the *tagname* parameter will assist in keeping your code portable between Unix and Windows.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of the ALLOCATIONGRANULARITY.

Raises an *auditing event* mmap.__new__ with arguments fileno, length, access, offset.

class mmap.**mmap**(*fileno*, *length*, *flags=MAP_SHARED*, *prot=PROT_WRITE | PROT_READ*, *access=ACCESS_DEFAULT*, *offset=0*, *, *trackfd=True*)

(Unix version) Maps *length* bytes from the file specified by the file descriptor *fileno*, and returns a mmap object. If *length* is 0, the maximum length of the map will be the current size of the file when *mmap* is called.

flags specifies the nature of the mapping. *MAP_PRIVATE* creates a private copy-on-write mapping, so changes to the contents of the mmap object will be private to this process, and *MAP_SHARED* creates a mapping that's shared with all other processes mapping the same areas of the file. The default value is *MAP_SHARED*. Some systems have additional possible flags with the full list specified in *MAP_* constants*.

prot, if specified, gives the desired memory protection; the two most useful values are PROT_READ and PROT_WRITE, to specify that the pages may be read or written. *prot* defaults to PROT_READ | PROT_WRITE.

access may be specified in lieu of *flags* and *prot* as an optional keyword parameter. It is an error to specify both *flags*, *prot* and *access*. See the description of *access* above for information on how to use this parameter.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of ALLOCATIONGRANULARITY which is equal to PAGESIZE on Unix systems.

If *trackfd* is `False`, the file descriptor specified by *fileno* will not be duplicated, and the resulting `mmap` object will not be associated with the map's underlying file. This means that the *size()* and *resize()* methods will fail. This mode is useful to limit the number of open file descriptors.

To ensure validity of the created memory mapping the file specified by the descriptor *fileno* is internally automatically synchronized with the physical backing store on macOS.

Changed in version 3.13: The *trackfd* parameter was added.

This example shows a simple way of using *mmap*:

```
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write(b"Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print(mm.readline())  # prints b"Hello Python!\n"
    # read content via slice notation
    print(mm[:5])  # prints b"Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = b" world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print(mm.readline())  # prints b"Hello  world!\n"
    # close the map
    mm.close()
```

mmap can also be used as a context manager in a `with` statement:

```
import mmap

with mmap.mmap(-1, 13) as mm:
    mm.write(b"Hello world!")
```

Added in version 3.2: Context manager support.

The next example demonstrates how to create an anonymous map and exchange data between the parent and child processes:

```
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write(b"Hello world!")

pid = os.fork()

if pid == 0:  # In a child process
    mm.seek(0)
    print(mm.readline())

    mm.close()
```

Raises an *auditing event* `mmap.__new__` with arguments `fileno`, `length`, `access`, `offset`.

Memory-mapped file objects support the following methods:

close()

Closes the mmap. Subsequent calls to other methods of the object will result in a ValueError exception being raised. This will not close the open file.

closed

`True` if the file is closed.

Added in version 3.2.

find(*sub*[, *start*[, *end*]])

Returns the lowest index in the object where the subsequence *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns `-1` on failure.

Changed in version 3.5: Writable *bytes-like object* is now accepted.

flush([*offset*[, *size*]])

Flushes changes made to the in-memory copy of a file back to disk. Without use of this call there is no guarantee that changes are written back before the object is destroyed. If *offset* and *size* are specified, only changes to the given range of bytes will be flushed to disk; otherwise, the whole extent of the mapping is flushed. *offset* must be a multiple of the `PAGESIZE` or `ALLOCATIONGRANULARITY`.

`None` is returned to indicate success. An exception is raised when the call failed.

Changed in version 3.8: Previously, a nonzero value was returned on success; zero was returned on error under Windows. A zero value was returned on success; an exception was raised on error under Unix.

madvise(*option*[, *start*[, *length*]])

Send advice *option* to the kernel about the memory region beginning at *start* and extending *length* bytes. *option* must be one of the *MADV_* constants* available on the system. If *start* and *length* are omitted, the entire mapping is spanned. On some systems (including Linux), *start* must be a multiple of the `PAGESIZE`.

Availability: Systems with the `madvise()` system call.

Added in version 3.8.

move(*dest*, *src*, *count*)

Copy the *count* bytes starting at offset *src* to the destination index *dest*. If the mmap was created with `ACCESS_READ`, then calls to move will raise a *TypeError* exception.

read([*n*])

Return a *bytes* containing up to *n* bytes starting from the current file position. If the argument is omitted, `None` or negative, return all bytes from the current file position to the end of the mapping. The file position is updated to point after the bytes that were returned.

Changed in version 3.3: Argument can be omitted or `None`.

read_byte()

Returns a byte at the current file position as an integer, and advances the file position by 1.

readline()

Returns a single line, starting at the current file position and up to the next newline. The file position is updated to point after the bytes that were returned.

resize(*newsize*)

Resizes the map and the underlying file, if any.

Resizing a map created with *access* of `ACCESS_READ` or `ACCESS_COPY`, will raise a *TypeError* exception. Resizing a map created with with *trackfd* set to `False`, will raise a *ValueError* exception.

On Windows: Resizing the map will raise an *OSError* if there are other maps against the same named file. Resizing an anonymous map (ie against the pagefile) will silently create a new map with the original data copied over up to the length of the new size.

Changed in version 3.11: Correctly fails if attempting to resize when another map is held Allows resize against an anonymous map on Windows

rfind(*sub*[, *start*[, *end*]])

Returns the highest index in the object where the subsequence *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns `-1` on failure.

Changed in version 3.5: Writable *bytes-like object* is now accepted.

seek(*pos*[, *whence*])

Set the file's current position. *whence* argument is optional and defaults to `os.SEEK_SET` or `0` (absolute file positioning); other values are `os.SEEK_CUR` or `1` (seek relative to the current position) and `os.SEEK_END` or `2` (seek relative to the file's end).

Changed in version 3.13: Return the new absolute position instead of `None`.

seekable()

Return whether the file supports seeking, and the return value is always `True`.

Added in version 3.13.

size()

Return the length of the file, which can be larger than the size of the memory-mapped area.

tell()

Returns the current position of the file pointer.

write(*bytes*)

Write the bytes in *bytes* into memory at the current position of the file pointer and return the number of bytes written (never less than `len(bytes)`, since if the write fails, a *ValueError* will be raised). The file position is updated to point after the bytes that were written. If the mmap was created with `ACCESS_READ`, then writing to it will raise a *TypeError* exception.

Changed in version 3.5: Writable *bytes-like object* is now accepted.

Changed in version 3.6: The number of bytes written is now returned.

write_byte(*byte*)

Write the integer *byte* into memory at the current position of the file pointer; the file position is advanced by `1`. If the mmap was created with `ACCESS_READ`, then writing to it will raise a *TypeError* exception.

18.7.1 MADV_* Constants

mmap.**MADV_NORMAL**

mmap.**MADV_RANDOM**

mmap.**MADV_SEQUENTIAL**

mmap.**MADV_WILLNEED**

mmap.**MADV_DONTNEED**

mmap.**MADV_REMOVE**

mmap.**MADV_DONTFORK**

mmap.**MADV_DOFORK**

mmap.**MADV_HWPOISON**

mmap.**MADV_MERGEABLE**
mmap.**MADV_UNMERGEABLE**
mmap.**MADV_SOFT_OFFLINE**
mmap.**MADV_HUGEPAGE**
mmap.**MADV_NOHUGEPAGE**
mmap.**MADV_DONTDUMP**
mmap.**MADV_DODUMP**
mmap.**MADV_FREE**
mmap.**MADV_NOSYNC**
mmap.**MADV_AUTOSYNC**
mmap.**MADV_NOCORE**
mmap.**MADV_CORE**
mmap.**MADV_PROTECT**
mmap.**MADV_FREE_REUSABLE**
mmap.**MADV_FREE_REUSE**

These options can be passed to *mmap.madvise()*. Not every option will be present on every system.

Availability: Systems with the madvise() system call.

Added in version 3.8.

18.7.2 MAP_* Constants

mmap.**MAP_SHARED**
mmap.**MAP_PRIVATE**
mmap.**MAP_32BIT**
mmap.**MAP_ALIGNED_SUPER**
mmap.**MAP_ANON**
mmap.**MAP_ANONYMOUS**
mmap.**MAP_CONCEAL**
mmap.**MAP_DENYWRITE**
mmap.**MAP_EXECUTABLE**
mmap.**MAP_HASSEMAPHORE**
mmap.**MAP_JIT**
mmap.**MAP_NOCACHE**
mmap.**MAP_NOEXTEND**
mmap.**MAP_NORESERVE**
mmap.**MAP_POPULATE**
mmap.**MAP_RESILIENT_CODESIGN**
mmap.**MAP_RESILIENT_MEDIA**
mmap.**MAP_STACK**
mmap.**MAP_TPRO**
mmap.**MAP_TRANSLATED_ALLOW_EXECUTE**

mmap.**MAP_UNIX03**

These are the various flags that can be passed to *mmap.mmap()*. *MAP_ALIGNED_SUPER* is only available at FreeBSD and *MAP_CONCEAL* is only available at OpenBSD. Note that some options might not be present on some systems.

Changed in version 3.10: Added *MAP_POPULATE* constant.

Added in version 3.11: Added *MAP_STACK* constant.

Added in version 3.12: Added *MAP_ALIGNED_SUPER* and *MAP_CONCEAL* constants.

Added in version 3.13: Added *MAP_32BIT*, *MAP_HASSEMAPHORE*, *MAP_JIT*, *MAP_NOCACHE*, *MAP_NOEXTEND*, *MAP_NORESERVE*, *MAP_RESILIENT_CODESIGN*, *MAP_RESILIENT_MEDIA*, *MAP_TPRO*, *MAP_TRANSLATED_ALLOW_EXECUTE*, and *MAP_UNIX03* constants.

CHAPTER

NINETEEN

INTERNET DATA HANDLING

This chapter describes modules which support handling data formats commonly used on the internet.

19.1 `email` — An email and MIME handling package

Source code: Lib/email/__init__.py

The `email` package is a library for managing email messages. It is specifically *not* designed to do any sending of email messages to SMTP (**RFC 2821**), NNTP, or other servers; those are functions of modules such as `smtplib`. The `email` package attempts to be as RFC-compliant as possible, supporting **RFC 5322** and **RFC 6532**, as well as such MIME-related RFCs as **RFC 2045**, **RFC 2046**, **RFC 2047**, **RFC 2183**, and **RFC 2231**.

The overall structure of the email package can be divided into three major components, plus a fourth component that controls the behavior of the other components.

The central component of the package is an "object model" that represents email messages. An application interacts with the package primarily through the object model interface defined in the `message` sub-module. The application can use this API to ask questions about an existing email, to construct a new email, or to add or remove email subcomponents that themselves use the same object model interface. That is, following the nature of email messages and their MIME subcomponents, the email object model is a tree structure of objects that all provide the `EmailMessage` API.

The other two major components of the package are the `parser` and the `generator`. The parser takes the serialized version of an email message (a stream of bytes) and converts it into a tree of `EmailMessage` objects. The generator takes an `EmailMessage` and turns it back into a serialized byte stream. (The parser and generator also handle streams of text characters, but this usage is discouraged as it is too easy to end up with messages that are not valid in one way or another.)

The control component is the `policy` module. Every `EmailMessage`, every `generator`, and every `parser` has an associated `policy` object that controls its behavior. Usually an application only needs to specify the policy when an `EmailMessage` is created, either by directly instantiating an `EmailMessage` to create a new email, or by parsing an input stream using a `parser`. But the policy can be changed when the message is serialized using a `generator`. This allows, for example, a generic email message to be parsed from disk, but to serialize it using standard SMTP settings when sending it to an email server.

The email package does its best to hide the details of the various governing RFCs from the application. Conceptually the application should be able to treat the email message as a structured tree of unicode text and binary attachments, without having to worry about how these are represented when serialized. In practice, however, it is often necessary to be aware of at least some of the rules governing MIME messages and their structure, specifically the names and nature of the MIME "content types" and how they identify multipart documents. For the most part this knowledge should only be required for more complex applications, and even then it should only be the high level structure in question, and not the details of how those structures are represented. Since MIME content types are used widely in modern internet software (not just email), this will be a familiar concept to many programmers.

The following sections describe the functionality of the *email* package. We start with the *message* object model, which is the primary interface an application will use, and follow that with the *parser* and *generator* components. Then we cover the *policy* controls, which completes the treatment of the main components of the library.

The next three sections cover the exceptions the package may raise and the defects (non-compliance with the RFCs) that the *parser* may detect. Then we cover the *headerregistry* and the *contentmanager* sub-components, which provide tools for doing more detailed manipulation of headers and payloads, respectively. Both of these components contain features relevant to consuming and producing non-trivial messages, but also document their extensibility APIs, which will be of interest to advanced applications.

Following those is a set of examples of using the fundamental parts of the APIs covered in the preceding sections.

The foregoing represent the modern (unicode friendly) API of the email package. The remaining sections, starting with the *Message* class, cover the legacy *compat32* API that deals much more directly with the details of how email messages are represented. The *compat32* API does *not* hide the details of the RFCs from the application, but for applications that need to operate at that level, they can be useful tools. This documentation is also relevant for applications that are still using the *compat32* API for backward compatibility reasons.

Changed in version 3.6: Docs reorganized and rewritten to promote the new *EmailMessage*/*EmailPolicy* API.

Contents of the *email* package documentation:

19.1.1 email.message: Representing an email message

Source code: Lib/email/message.py

Added in version 3.6:[1]

The central class in the *email* package is the *EmailMessage* class, imported from the *email.message* module. It is the base class for the *email* object model. *EmailMessage* provides the core functionality for setting and querying header fields, for accessing message bodies, and for creating or modifying structured messages.

An email message consists of *headers* and a *payload* (which is also referred to as the *content*). Headers are **RFC 5322** or **RFC 6532** style field names and values, where the field name and value are separated by a colon. The colon is not part of either the field name or the field value. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as *multipart/** or *message/rfc822*.

The conceptual model provided by an *EmailMessage* object is that of an ordered dictionary of headers coupled with a *payload* that represents the **RFC 5322** body of the message, which might be a list of sub-`EmailMessage` objects. In addition to the normal dictionary methods for accessing the header names and values, there are methods for accessing specialized information from the headers (for example the MIME content type), for operating on the payload, for generating a serialized version of the message, and for recursively walking over the object tree.

The *EmailMessage* dictionary-like interface is indexed by the header names, which must be ASCII values. The values of the dictionary are strings with some extra methods. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. The keys are ordered, but unlike a real dict, there can be duplicates. Additional methods are provided for working with headers that have duplicate keys.

The *payload* is either a string or bytes object, in the case of simple message objects, or a list of *EmailMessage* objects, for MIME container documents such as *multipart/** and *message/rfc822* message objects.

[1] Originally added in 3.4 as a *provisional module*. Docs for legacy message class moved to *email.message.Message: Representing an email message using the compat32 API*.

class email.message.**EmailMessage**(*policy=default*)

If *policy* is specified use the rules it specifies to update and serialize the representation of the message. If *policy* is not set, use the *default* policy, which follows the rules of the email RFCs except for line endings (instead of the RFC mandated `\r\n`, it uses the Python standard `\n` line endings). For more information see the *policy* documentation.

as_string(*unixfrom=False*, *maxheaderlen=None*, *policy=None*)

Return the entire message flattened as a string. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. For backward compatibility with the base *Message* class *maxheaderlen* is accepted, but defaults to `None`, which means that by default the line length is controlled by the *max_line_length* of the policy. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the *Generator*.

Flattening the message may trigger changes to the *EmailMessage* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not be the most useful way to serialize messages in your application, especially if you are dealing with multiple messages. See *email.generator.Generator* for a more flexible API for serializing messages. Note also that this method is restricted to producing messages serialized as "7 bit clean" when *utf8* is `False`, which is the default.

Changed in version 3.6: the default behavior when *maxheaderlen* is not specified was changed from defaulting to 0 to defaulting to the value of *max_line_length* from the policy.

__str__()

Equivalent to `as_string(policy=self.policy.clone(utf8=True))`. Allows `str(msg)` to produce a string containing the serialized message in a readable format.

Changed in version 3.4: the method was changed to use `utf8=True`, thus producing an **RFC 6531**-like message representation, instead of being a direct alias for *as_string()*.

as_bytes(*unixfrom=False*, *policy=None*)

Return the entire message flattened as a bytes object. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the *BytesGenerator*.

Flattening the message may trigger changes to the *EmailMessage* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not be the most useful way to serialize messages in your application, especially if you are dealing with multiple messages. See *email.generator.BytesGenerator* for a more flexible API for serializing messages.

__bytes__()

Equivalent to *as_bytes()*. Allows `bytes(msg)` to produce a bytes object containing the serialized message.

is_multipart()

Return `True` if the message's payload is a list of sub-*EmailMessage* objects, otherwise return `False`. When *is_multipart()* returns `False`, the payload should be a string object (which might be a CTE encoded binary payload). Note that *is_multipart()* returning `True` does not necessarily mean that "msg.get_content_maintype() == 'multipart'" will return the `True`. For example, `is_multipart` will return `True` when the *EmailMessage* is of type `message/rfc822`.

set_unixfrom(*unixfrom*)

Set the message's envelope header to *unixfrom*, which should be a string. (See *mboxMessage* for a brief description of this header.)

get_unixfrom()

Return the message's envelope header. Defaults to `None` if the envelope header was never set.

The following methods implement the mapping-like interface for accessing the message's headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by *keys()*, but in an *EmailMessage* object, headers are always returned in the order they appeared in the original message, or in which they were added to the message later. Any header deleted and then re-added is always appended to the end of the header list.

These semantic differences are intentional and are biased toward convenience in the most common use cases.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

__len__()

Return the total number of headers, including duplicates.

__contains__(*name*)

Return `True` if the message object has a field named *name*. Matching is done without regard to case and *name* does not include the trailing colon. Used for the `in` operator. For example:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)

Return the value of the named header field. *name* does not include the colon field separator. If the header is missing, `None` is returned; a *KeyError* is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the *get_all()* method to get the values of all the extant headers named *name*.

Using the standard (non-`compat32`) policies, the returned value is an instance of a subclass of *email.headerregistry.BaseHeader*.

__setitem__(*name*, *val*)

Add a header to the message with field name *name* and value *val*. The field is appended to the end of the message's existing headers.

Note that this does *not* overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name *name*, delete the field first, e.g.:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

If the *policy* defines certain headers to be unique (as the standard policies do), this method may raise a *ValueError* when an attempt is made to assign a value to such a header when one already exists. This behavior is intentional for consistency's sake, but do not depend on it as we may choose to make such assignments do an automatic deletion of the existing header in the future.

__delitem__(*name*)

Delete all occurrences of the field with name *name* from the message's headers. No exception is raised if the named field isn't present in the headers.

keys()

Return a list of all the message's header field names.

values()

Return a list of all the message's field values.

items()

Return a list of 2-tuples containing all the message's field headers and values.

get(*name*, *failobj=None*)

Return the value of the named header field. This is identical to `__getitem__()` except that optional *failobj* is returned if the named header is missing (*failobj* defaults to `None`).

Here are some additional useful header related methods:

get_all(*name*, *failobj=None*)

Return a list of all the values for the field named *name*. If there are no such named headers in the message, *failobj* is returned (defaults to `None`).

add_header(*_name*, *_value*, ***_params*)

Extended header setting. This method is similar to *`__setitem__()`* except that additional header parameters can be provided as keyword arguments. *_name* is the header field to add and *_value* is the *primary* value for the header.

For each item in the keyword argument dictionary *_params*, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as `key="value"` unless the value is `None`, in which case only the key will be added.

If the value contains non-ASCII characters, the charset and language may be explicitly controlled by specifying the value as a three tuple in the format `(CHARSET, LANGUAGE, VALUE)`, where `CHARSET` is a string naming the charset to be used to encode the value, `LANGUAGE` can usually be set to `None` or the empty string (see **RFC 2231** for other possibilities), and `VALUE` is the string value containing non-ASCII code points. If a three tuple is not passed and the value contains non-ASCII characters, it is automatically encoded in **RFC 2231** format using a `CHARSET` of `utf-8` and a `LANGUAGE` of `None`.

Here is an example:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.gif"
```

An example of the extended interface with non-ASCII characters:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

replace_header(*_name*, *_value*)

Replace a header. Replace the first header found in the message that matches *_name*, retaining header order and field name case of the original header. If no matching header is found, raise a *`KeyError`*.

get_content_type()

Return the message's content type, coerced to lower case of the form *`maintype/subtype`*. If there is no *`Content-Type`* header in the message return the value returned by *`get_default_type()`*. If the *`Content-Type`* header is invalid, return `text/plain`.

(According to **RFC 2045**, messages always have a default type, *`get_content_type()`* will always return a value. **RFC 2045** defines a message's default type to be *`text/plain`* unless it appears inside a *`multipart/digest`* container, in which case it would be *`message/rfc822`*. If the *`Content-Type`* header has an invalid type specification, **RFC 2045** mandates that the default type be *`text/plain`*.)

get_content_maintype()

Return the message's main content type. This is the *maintype* part of the string returned by *get_content_type()*.

get_content_subtype()

Return the message's sub-content type. This is the *subtype* part of the string returned by *get_content_type()*.

get_default_type()

Return the default content type. Most messages have a default content type of *text/plain*, except for messages that are subparts of *multipart/digest* containers. Such subparts have a default content type of *message/rfc822*.

set_default_type(*ctype*)

Set the default content type. *ctype* should either be *text/plain* or *message/rfc822*, although this is not enforced. The default content type is not stored in the *Content-Type* header, so it only affects the return value of the get_content_type methods when no *Content-Type* header is present in the message.

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)

Set a parameter in the *Content-Type* header. If the parameter already exists in the header, replace its value with *value*. When *header* is Content-Type (the default) and the header does not yet exist in the message, add it, set its value to *text/plain*, and append the new parameter value. Optional *header* specifies an alternative header to *Content-Type*.

If the value contains non-ASCII characters, the charset and language may be explicitly specified using the optional *charset* and *language* parameters. Optional *language* specifies the **RFC 2231** language, defaulting to the empty string. Both *charset* and *language* should be strings. The default is to use the utf8 *charset* and None for the *language*.

If *replace* is False (the default) the header is moved to the end of the list of headers. If *replace* is True, the header will be updated in place.

Use of the *requote* parameter with *EmailMessage* objects is deprecated.

Note that existing parameter values of headers may be accessed through the *params* attribute of the header value (for example, msg['Content-Type'].params['charset']).

Changed in version 3.4: replace keyword was added.

del_param(*param*, *header='content-type'*, *requote=True*)

Remove the given parameter completely from the *Content-Type* header. The header will be re-written in place without the parameter or its value. Optional *header* specifies an alternative to *Content-Type*.

Use of the *requote* parameter with *EmailMessage* objects is deprecated.

get_filename(*failobj=None*)

Return the value of the filename parameter of the *Content-Disposition* header of the message. If the header does not have a filename parameter, this method falls back to looking for the name parameter on the *Content-Type* header. If neither is found, or the header is missing, then *failobj* is returned. The returned string will always be unquoted as per *email.utils.unquote()*.

get_boundary(*failobj=None*)

Return the value of the boundary parameter of the *Content-Type* header of the message, or *failobj* if either the header is missing, or has no boundary parameter. The returned string will always be unquoted as per *email.utils.unquote()*.

set_boundary(*boundary*)

Set the `boundary` parameter of the *Content-Type* header to *boundary*. *set_boundary()* will always quote *boundary* if necessary. A *HeaderParseError* is raised if the message object has no *Content-Type* header.

Note that using this method is subtly different from deleting the old *Content-Type* header and adding a new one with the new boundary via *add_header()*, because *set_boundary()* preserves the order of the *Content-Type* header in the list of headers.

get_content_charset(*failobj=None*)

Return the `charset` parameter of the *Content-Type* header, coerced to lower case. If there is no *Content-Type* header, or if that header has no `charset` parameter, *failobj* is returned.

get_charsets(*failobj=None*)

Return a list containing the character set names in the message. If the message is a *multipart*, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the `charset` parameter in the *Content-Type* header for the represented subpart. If the subpart has no *Content-Type* header, no `charset` parameter, or is not of the *text* main MIME type, then that item in the returned list will be *failobj*.

is_attachment()

Return `True` if there is a *Content-Disposition* header and its (case insensitive) value is `attachment`, `False` otherwise.

Changed in version 3.4.2: is_attachment is now a method instead of a property, for consistency with *is_multipart()*.

get_content_disposition()

Return the lowercased value (without parameters) of the message's *Content-Disposition* header if it has one, or `None`. The possible values for this method are *inline*, *attachment* or `None` if the message follows **RFC 2183**.

Added in version 3.5.

The following methods relate to interrogating and manipulating the content (payload) of the message.

walk()

The *walk()* method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use *walk()* as the iterator in a `for` loop; each iteration returns the next subpart.

Here's an example that prints the MIME type of every part of a multipart message structure:

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

`walk` iterates over the subparts of any part where *is_multipart()* returns `True`, even though `msg.get_content_maintype() == 'multipart'` may return `False`. We can see this in our example by making use of the `_structure` debug helper function:

```
>>> from email.iterators import _structure
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

Here the `message` parts are not `multiparts`, but they do contain subparts. `is_multipart()` returns `True` and `walk` descends into the subparts.

get_body (*preferencelist=('related', 'html', 'plain')*)

Return the MIME part that is the best candidate to be the "body" of the message.

preferencelist must be a sequence of strings from the set `related`, `html`, and `plain`, and indicates the order of preference for the content type of the part returned.

Start looking for candidate matches with the object on which the `get_body` method is called.

If `related` is not included in *preferencelist*, consider the root part (or subpart of the root part) of any related encountered as a candidate if the (sub-)part matches a preference.

When encountering a `multipart/related`, check the `start` parameter and if a part with a matching *Content-ID* is found, consider only it when looking for candidate matches. Otherwise consider only the first (default root) part of the `multipart/related`.

If a part has a *Content-Disposition* header, only consider the part a candidate match if the value of the header is `inline`.

If none of the candidates matches any of the preferences in *preferencelist*, return `None`.

Notes: (1) For most applications the only *preferencelist* combinations that really make sense are `('plain',)`, `('html', 'plain')`, and the default `('related', 'html', 'plain')`. (2) Because matching starts with the object on which `get_body` is called, calling `get_body` on a `multipart/related` will return the object itself unless *preferencelist* has a non-default value. (3) Messages (or message parts) that do not specify a *Content-Type* or whose *Content-Type* header is invalid will be treated as if they are of type `text/plain`, which may occasionally cause `get_body` to return unexpected results.

iter_attachments()

Return an iterator over all of the immediate sub-parts of the message that are not candidate "body" parts. That is, skip the first occurrence of each of `text/plain`, `text/html`, `multipart/related`, or `multipart/alternative` (unless they are explicitly marked as attachments via *Content-Disposition: attachment*), and return all remaining parts. When applied directly to a `multipart/related`, return an iterator over the all the related parts except the root part (ie: the part pointed to by the `start` parameter, or the first part if there is no `start` parameter or the `start` parameter doesn't match the *Content-ID* of any of the parts). When applied directly to a `multipart/alternative` or a non-`multipart`, return an empty iterator.

iter_parts()

Return an iterator over all of the immediate sub-parts of the message, which will be empty for a non-multipart. (See also *walk()*.)

get_content(**args*, *content_manager=None*, ***kw*)

Call the *get_content()* method of the *content_manager*, passing self as the message object, and passing along any other arguments or keywords as additional arguments. If *content_manager* is not specified, use the content_manager specified by the current *policy*.

set_content(**args*, *content_manager=None*, ***kw*)

Call the *set_content()* method of the *content_manager*, passing self as the message object, and passing along any other arguments or keywords as additional arguments. If *content_manager* is not specified, use the content_manager specified by the current *policy*.

make_related(*boundary=None*)

Convert a non-multipart message into a multipart/related message, moving any existing *Content-* headers and payload into a (new) first part of the multipart. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

make_alternative(*boundary=None*)

Convert a non-multipart or a multipart/related into a multipart/alternative, moving any existing *Content-* headers and payload into a (new) first part of the multipart. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

make_mixed(*boundary=None*)

Convert a non-multipart, a multipart/related, or a multipart-alternative into a multipart/mixed, moving any existing *Content-* headers and payload into a (new) first part of the multipart. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

add_related(**args*, *content_manager=None*, ***kw*)

If the message is a multipart/related, create a new message object, pass all of the arguments to its *set_content()* method, and *attach()* it to the multipart. If the message is a non-multipart, call *make_related()* and then proceed as above. If the message is any other type of multipart, raise a *TypeError*. If *content_manager* is not specified, use the content_manager specified by the current *policy*. If the added part has no *Content-Disposition* header, add one with the value inline.

add_alternative(**args*, *content_manager=None*, ***kw*)

If the message is a multipart/alternative, create a new message object, pass all of the arguments to its *set_content()* method, and *attach()* it to the multipart. If the message is a non-multipart or multipart/related, call *make_alternative()* and then proceed as above. If the message is any other type of multipart, raise a *TypeError*. If *content_manager* is not specified, use the content_manager specified by the current *policy*.

add_attachment(**args*, *content_manager=None*, ***kw*)

If the message is a multipart/mixed, create a new message object, pass all of the arguments to its *set_content()* method, and *attach()* it to the multipart. If the message is a non-multipart, multipart/related, or multipart/alternative, call *make_mixed()* and then proceed as above. If *content_manager* is not specified, use the content_manager specified by the current *policy*. If the added part has no *Content-Disposition* header, add one with the value attachment. This method can be used both for explicit attachments (*Content-Disposition: attachment*) and inline attachments (*Content-Disposition: inline*), by passing appropriate options to the content_manager.

clear()

Remove the payload and all of the headers.

clear_content()

Remove the payload and all of the *!Content−* headers, leaving all other headers intact and in their original order.

EmailMessage objects have the following instance attributes:

preamble

The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The *preamble* attribute contains this leading extra-armor text for MIME documents. When the *Parser* discovers some text after the headers but before the first boundary string, it assigns this text to the message's *preamble* attribute. When the *Generator* is writing out the plain text representation of a MIME message, and it finds the message has a *preamble* attribute, it will write this text in the area between the headers and the first boundary. See *email.parser* and *email.generator* for details.

Note that if the message object has no preamble, the *preamble* attribute will be None.

epilogue

The *epilogue* attribute acts the same way as the *preamble* attribute, except that it contains text that appears between the last boundary and the end of the message. As with the *preamble*, if there is no epilog text this attribute will be None.

defects

The *defects* attribute contains a list of all the problems found when parsing this message. See *email.errors* for a detailed description of the possible parsing defects.

class email.message.**MIMEPart**(*policy=default*)

This class represents a subpart of a MIME message. It is identical to *EmailMessage*, except that no *MIME-Version* headers are added when *set_content()* is called, since sub-parts do not need their own *MIME-Version* headers.

19.1.2 email.parser: Parsing email messages

Source code: Lib/email/parser.py

Message object structures can be created in one of two ways: they can be created from whole cloth by creating an *EmailMessage* object, adding headers using the dictionary interface, and adding payload(s) using *set_content()* and related methods, or they can be created by parsing a serialized representation of the email message.

The *email* package provides a standard parser that understands most email document structures, including MIME documents. You can pass the parser a bytes, string or file object, and the parser will return to you the root *EmailMessage* instance of the object structure. For simple, non-MIME messages the payload of this root object will likely be a string containing the text of the message. For MIME messages, the root object will return True from its *is_multipart()* method, and the subparts can be accessed via the payload manipulation methods, such as *get_body()*, *iter_parts()*, and *walk()*.

There are actually two parser interfaces available for use, the *Parser* API and the incremental *FeedParser* API. The *Parser* API is most useful if you have the entire text of the message in memory, or if the entire message lives in a file on the file system. *FeedParser* is more appropriate when you are reading the message from a stream which might

block waiting for more input (such as reading an email message from a socket). The *FeedParser* can consume and parse the message incrementally, and only returns the root object when you close the parser.

Note that the parser can be extended in limited ways, and of course you can implement your own parser completely from scratch. All of the logic that connects the *email* package's bundled parser and the *EmailMessage* class is embodied in the *Policy* class, so a custom parser can create message object trees any way it finds necessary by implementing custom versions of the appropriate `Policy` methods.

FeedParser API

The *BytesFeedParser*, imported from the `email.feedparser` module, provides an API that is conducive to incremental parsing of email messages, such as would be necessary when reading the text of an email message from a source that can block (such as a socket). The *BytesFeedParser* can of course be used to parse an email message fully contained in a *bytes-like object*, string, or file, but the *BytesParser* API may be more convenient for such use cases. The semantics and results of the two parser APIs are identical.

The *BytesFeedParser*'s API is simple; you create an instance, feed it a bunch of bytes until there's no more to feed it, then close the parser to retrieve the root message object. The *BytesFeedParser* is extremely accurate when parsing standards-compliant messages, and it does a very good job of parsing non-compliant messages, providing information about how a message was deemed broken. It will populate a message object's *defects* attribute with a list of any problems it found in a message. See the *email.errors* module for the list of defects that it can find.

Here is the API for the *BytesFeedParser*:

class email.parser.**BytesFeedParser**(*_factory=None*, *, *policy=policy.compat32*)

Create a *BytesFeedParser* instance. Optional *_factory* is a no-argument callable; if not specified use the *message_factory* from the *policy*. Call *_factory* whenever a new message object is needed.

If *policy* is specified use the rules it specifies to update the representation of the message. If *policy* is not set, use the *compat32* policy, which maintains backward compatibility with the Python 3.2 version of the email package and provides *Message* as the default factory. All other policies provide *EmailMessage* as the default *_factory*. For more information on what else *policy* controls, see the *policy* documentation.

Note: **The policy keyword should always be specified**; The default will change to *email.policy.default* in a future version of Python.

Added in version 3.2.

Changed in version 3.3: Added the *policy* keyword.

Changed in version 3.6: *_factory* defaults to the policy `message_factory`.

feed(*data*)

Feed the parser some more data. *data* should be a *bytes-like object* containing one or more lines. The lines can be partial and the parser will stitch such partial lines together properly. The lines can have any of the three common line endings: carriage return, newline, or carriage return and newline (they can even be mixed).

close()

Complete the parsing of all previously fed data and return the root message object. It is undefined what happens if *feed()* is called after this method has been called.

class email.parser.**FeedParser**(*_factory=None*, *, *policy=policy.compat32*)

Works like *BytesFeedParser* except that the input to the *feed()* method must be a string. This is of limited utility, since the only way for such a message to be valid is for it to contain only ASCII text or, if `utf8` is `True`, no binary attachments.

Changed in version 3.3: Added the *policy* keyword.

Parser API

The `BytesParser` class, imported from the `email.parser` module, provides an API that can be used to parse a message when the complete contents of the message are available in a *bytes-like object* or file. The `email.parser` module also provides `Parser` for parsing strings, and header-only parsers, `BytesHeaderParser` and `HeaderParser`, which can be used if you're only interested in the headers of the message. `BytesHeaderParser` and `HeaderParser` can be much faster in these situations, since they do not attempt to parse the message body, instead setting the payload to the raw body.

class email.parser.**BytesParser**(*_class=None*, ***, *policy=policy.compat32*)

Create a `BytesParser` instance. The *_class* and *policy* arguments have the same meaning and semantics as the *_factory* and *policy* arguments of `BytesFeedParser`.

Note: **The policy keyword should always be specified**; The default will change to `email.policy.default` in a future version of Python.

Changed in version 3.3: Removed the *strict* argument that was deprecated in 2.4. Added the *policy* keyword.

Changed in version 3.6: *_class* defaults to the policy `message_factory`.

parse(*fp*, *headersonly=False*)

Read all the data from the binary file-like object *fp*, parse the resulting bytes, and return the message object. *fp* must support both the `readline()` and the `read()` methods.

The bytes contained in *fp* must be formatted as a block of **RFC 5322** (or, if `utf8` is `True`, **RFC 6532**) style headers and header continuation lines, optionally preceded by an envelope header. The header block is terminated either by the end of the data or by a blank line. Following the header block is the body of the message (which may contain MIME-encoded subparts, including subparts with a `Content-Transfer-Encoding` of `8bit`).

Optional *headersonly* is a flag specifying whether to stop parsing after reading the headers or not. The default is `False`, meaning it parses the entire contents of the file.

parsebytes(*bytes*, *headersonly=False*)

Similar to the `parse()` method, except it takes a *bytes-like object* instead of a file-like object. Calling this method on a *bytes-like object* is equivalent to wrapping *bytes* in a `BytesIO` instance first and calling `parse()`.

Optional *headersonly* is as with the `parse()` method.

Added in version 3.2.

class email.parser.**BytesHeaderParser**(*_class=None*, ***, *policy=policy.compat32*)

Exactly like `BytesParser`, except that *headersonly* defaults to `True`.

Added in version 3.3.

class email.parser.**Parser**(*_class=None*, ***, *policy=policy.compat32*)

This class is parallel to `BytesParser`, but handles string input.

Changed in version 3.3: Removed the *strict* argument. Added the *policy* keyword.

Changed in version 3.6: *_class* defaults to the policy `message_factory`.

parse(*fp*, *headersonly=False*)

Read all the data from the text-mode file-like object *fp*, parse the resulting text, and return the root message object. *fp* must support both the `readline()` and the `read()` methods on file-like objects.

Other than the text mode requirement, this method operates like `BytesParser.parse()`.

parsestr(*text*, *headersonly=False*)

Similar to the *parse()* method, except it takes a string object instead of a file-like object. Calling this method on a string is equivalent to wrapping *text* in a *StringIO* instance first and calling *parse()*.

Optional *headersonly* is as with the *parse()* method.

class email.parser.**HeaderParser**(*_class=None*, ***, *policy=policy.compat32*)

Exactly like *Parser*, except that *headersonly* defaults to `True`.

Since creating a message object structure from a string or a file object is such a common task, four functions are provided as a convenience. They are available in the top-level *email* package namespace.

email.**message_from_bytes**(*s*, *_class=None*, ***, *policy=policy.compat32*)

Return a message object structure from a *bytes-like object*. This is equivalent to `BytesParser().parsebytes(s)`. Optional *_class* and *policy* are interpreted as with the *BytesParser* class constructor.

Added in version 3.2.

Changed in version 3.3: Removed the *strict* argument. Added the *policy* keyword.

email.**message_from_binary_file**(*fp*, *_class=None*, ***, *policy=policy.compat32*)

Return a message object structure tree from an open binary *file object*. This is equivalent to `BytesParser().parse(fp)`. *_class* and *policy* are interpreted as with the *BytesParser* class constructor.

Added in version 3.2.

Changed in version 3.3: Removed the *strict* argument. Added the *policy* keyword.

email.**message_from_string**(*s*, *_class=None*, ***, *policy=policy.compat32*)

Return a message object structure from a string. This is equivalent to `Parser().parsestr(s)`. *_class* and *policy* are interpreted as with the *Parser* class constructor.

Changed in version 3.3: Removed the *strict* argument. Added the *policy* keyword.

email.**message_from_file**(*fp*, *_class=None*, ***, *policy=policy.compat32*)

Return a message object structure tree from an open *file object*. This is equivalent to `Parser().parse(fp)`. *_class* and *policy* are interpreted as with the *Parser* class constructor.

Changed in version 3.3: Removed the *strict* argument. Added the *policy* keyword.

Changed in version 3.6: *_class* defaults to the policy `message_factory`.

Here's an example of how you might use *message_from_bytes()* at an interactive Python prompt:

```
>>> import email
>>> msg = email.message_from_bytes(myBytes)
```

Additional notes

Here are some notes on the parsing semantics:

- Most non-*multipart* type messages are parsed as a single message object with a string payload. These objects will return `False` for *is_multipart()*, and *iter_parts()* will yield an empty list.
- All *multipart* type messages will be parsed as a container message object with a list of sub-message objects for their payload. The outer container message will return `True` for *is_multipart()*, and *iter_parts()* will yield a list of subparts.
- Most messages with a content type of *message/** (such as *message/delivery-status* and *message/rfc822*) will also be parsed as container object containing a list payload of length 1. Their *is_multipart()* method will return `True`. The single element yielded by *iter_parts()* will be a sub-message object.

- Some non-standards-compliant messages may not be internally consistent about their *multipart*-edness. Such messages may have a *Content-Type* header of type *multipart*, but their *is_multipart()* method may return `False`. If such messages were parsed with the *FeedParser*, they will have an instance of the `MultipartInvariantViolationDefect` class in their *defects* attribute list. See *email.errors* for details.

19.1.3 `email.generator`: Generating MIME documents

Source code: Lib/email/generator.py

One of the most common tasks is to generate the flat (serialized) version of the email message represented by a message object structure. You will need to do this if you want to send your message via *smtplib.SMTP.sendmail()*, or print the message on the console. Taking a message object structure and producing a serialized representation is the job of the generator classes.

As with the *email.parser* module, you aren't limited to the functionality of the bundled generator; you could write one from scratch yourself. However the bundled generator knows how to generate most email in a standards-compliant way, should handle MIME and non-MIME email messages just fine, and is designed so that the bytes-oriented parsing and generation operations are inverses, assuming the same non-transforming *policy* is used for both. That is, parsing the serialized byte stream via the *BytesParser* class and then regenerating the serialized byte stream using *BytesGenerator* should produce output identical to the input[1]. (On the other hand, using the generator on an *EmailMessage* constructed by program may result in changes to the *EmailMessage* object as defaults are filled in.)

The *Generator* class can be used to flatten a message into a text (as opposed to binary) serialized representation, but since Unicode cannot represent binary data directly, the message is of necessity transformed into something that contains only ASCII characters, using the standard email RFC Content Transfer Encoding techniques for encoding email messages for transport over channels that are not "8 bit clean".

To accommodate reproducible processing of SMIME-signed messages *Generator* disables header folding for message parts of type `multipart/signed` and all subparts.

class email.generator.**BytesGenerator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, ***, *policy=None*)

Return a *BytesGenerator* object that will write any message provided to the *flatten()* method, or any surrogateescape encoded text provided to the *write()* method, to the *file-like object* *outfp*. *outfp* must support a `write` method that accepts binary data.

If optional *mangle_from_* is `True`, put a > character in front of any line in the body that starts with the exact string `"From "`, that is `From` followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the *mangle_from_* setting of the *policy* (which is `True` for the *compat32* policy and `False` for all others). *mangle_from_* is intended for use when messages are stored in Unix mbox format (see *mailbox* and WHY THE CONTENT-LENGTH FORMAT IS BAD).

If *maxheaderlen* is not `None`, refold any header lines that are longer than *maxheaderlen*, or if `0`, do not rewrap any headers. If *manheaderlen* is `None` (the default), wrap headers and other message lines according to the *policy* settings.

If *policy* is specified, use that policy to control message generation. If *policy* is `None` (the default), use the policy associated with the *Message* or *EmailMessage* object passed to `flatten` to control the message generation. See *email.policy* for details on what *policy* controls.

Added in version 3.2.

[1] This statement assumes that you use the appropriate setting for `unixfrom`, and that there are no *email.policy* settings calling for automatic adjustments (for example, *refold_source* must be `none`, which is *not* the default). It is also not 100% true, since if the message does not conform to the RFC standards occasionally information about the exact original text is lost during parsing error recovery. It is a goal to fix these latter edge cases when possible.

Changed in version 3.3: Added the *policy* keyword.

Changed in version 3.6: The default behavior of the *mangle_from_* and *maxheaderlen* parameters is to follow the policy.

flatten(*msg*, *unixfrom=False*, *linesep=None*)

Print the textual representation of the message object structure rooted at *msg* to the output file specified when the `BytesGenerator` instance was created.

If the `policy` option `cte_type` is `8bit` (the default), copy any headers in the original parsed message that have not been modified to the output with any bytes with the high bit set reproduced as in the original, and preserve the non-ASCII `Content-Transfer-Encoding` of any body parts that have them. If `cte_type` is `7bit`, convert the bytes with the high bit set as needed using an ASCII-compatible `Content-Transfer-Encoding`. That is, transform parts with non-ASCII `Content-Transfer-Encoding` (`Content-Transfer-Encoding: 8bit`) to an ASCII compatible `Content-Transfer-Encoding`, and encode RFC-invalid non-ASCII bytes in headers using the MIME `unknown-8bit` character set, thus rendering them RFC-compliant.

If *unixfrom* is `True`, print the envelope header delimiter used by the Unix mailbox format (see `mailbox`) before the first of the **RFC 5322** headers of the root message object. If the root object has no envelope header, craft a standard one. The default is `False`. Note that for subparts, no envelope header is ever printed.

If *linesep* is not `None`, use it as the separator character between all the lines of the flattened message. If *linesep* is `None` (the default), use the value specified in the *policy*.

clone(*fp*)

Return an independent clone of this `BytesGenerator` instance with the exact same option settings, and *fp* as the new *outfp*.

write(*s*)

Encode *s* using the `ASCII` codec and the `surrogateescape` error handler, and pass it to the *write* method of the *outfp* passed to the `BytesGenerator`'s constructor.

As a convenience, `EmailMessage` provides the methods `as_bytes()` and `bytes(aMessage)` (a.k.a. `__bytes__()`), which simplify the generation of a serialized binary representation of a message object. For more detail, see `email.message`.

Because strings cannot represent binary data, the `Generator` class must convert any binary data in any message it flattens to an ASCII compatible format, by converting them to an ASCII compatible `Content-Transfer_Encoding`. Using the terminology of the email RFCs, you can think of this as `Generator` serializing to an I/O stream that is not "8 bit clean". In other words, most applications will want to be using `BytesGenerator`, and not `Generator`.

class email.generator.**Generator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, ***, *policy=None*)

Return a `Generator` object that will write any message provided to the `flatten()` method, or any text provided to the `write()` method, to the *file-like object* *outfp*. *outfp* must support a `write` method that accepts string data.

If optional *mangle_from_* is `True`, put a > character in front of any line in the body that starts with the exact string `"From "`, that is `From` followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the `mangle_from_` setting of the *policy* (which is `True` for the `compat32` policy and `False` for all others). *mangle_from_* is intended for use when messages are stored in Unix mbox format (see `mailbox` and WHY THE CONTENT-LENGTH FORMAT IS BAD).

If *maxheaderlen* is not `None`, refold any header lines that are longer than *maxheaderlen*, or if `0`, do not rewrap any headers. If *manheaderlen* is `None` (the default), wrap headers and other message lines according to the *policy* settings.

If *policy* is specified, use that policy to control message generation. If *policy* is `None` (the default), use the policy associated with the `Message` or `EmailMessage` object passed to `flatten` to control the message generation. See `email.policy` for details on what *policy* controls.

Changed in version 3.3: Added the *policy* keyword.

Changed in version 3.6: The default behavior of the *mangle_from_* and *maxheaderlen* parameters is to follow the policy.

flatten(*msg*, *unixfrom=False*, *linesep=None*)

Print the textual representation of the message object structure rooted at *msg* to the output file specified when the `Generator` instance was created.

If the `policy` option `cte_type` is `8bit`, generate the message as if the option were set to `7bit`. (This is required because strings cannot represent non-ASCII bytes.) Convert any bytes with the high bit set as needed using an ASCII-compatible `Content-Transfer-Encoding`. That is, transform parts with non-ASCII `Content-Transfer-Encoding` (`Content-Transfer-Encoding: 8bit`) to an ASCII compatible `Content-Transfer-Encoding`, and encode RFC-invalid non-ASCII bytes in headers using the MIME `unknown-8bit` character set, thus rendering them RFC-compliant.

If *unixfrom* is `True`, print the envelope header delimiter used by the Unix mailbox format (see `mailbox`) before the first of the **RFC 5322** headers of the root message object. If the root object has no envelope header, craft a standard one. The default is `False`. Note that for subparts, no envelope header is ever printed.

If *linesep* is not `None`, use it as the separator character between all the lines of the flattened message. If *linesep* is `None` (the default), use the value specified in the *policy*.

Changed in version 3.2: Added support for re-encoding `8bit` message bodies, and the *linesep* argument.

clone(*fp*)

Return an independent clone of this `Generator` instance with the exact same options, and *fp* as the new *outfp*.

write(*s*)

Write *s* to the *write* method of the *outfp* passed to the `Generator`'s constructor. This provides just enough file-like API for `Generator` instances to be used in the `print()` function.

As a convenience, `EmailMessage` provides the methods `as_string()` and `str(aMessage)` (a.k.a. `__str__()`), which simplify the generation of a formatted string representation of a message object. For more detail, see `email.message`.

The `email.generator` module also provides a derived class, `DecodedGenerator`, which is like the `Generator` base class, except that non-`text` parts are not serialized, but are instead represented in the output stream by a string derived from a template filled in with information about the part.

class email.generator.**DecodedGenerator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *fmt=None*, ***, *policy=None*)

Act like `Generator`, except that for any subpart of the message passed to `Generator.flatten()`, if the subpart is of main type `text`, print the decoded payload of the subpart, and if the main type is not `text`, instead of printing it fill in the string *fmt* using information from the part and print the resulting filled-in string.

To fill in *fmt*, execute `fmt % part_info`, where `part_info` is a dictionary composed of the following keys and values:

- `type` – Full MIME type of the non-`text` part
- `maintype` – Main MIME type of the non-`text` part
- `subtype` – Sub-MIME type of the non-`text` part
- `filename` – Filename of the non-`text` part
- `description` – Description associated with the non-`text` part
- `encoding` – Content transfer encoding of the non-`text` part

If *fmt* is `None`, use the following default *fmt*:

"[Non-text (%(type)s) part of message omitted, filename %(filename)s]"

Optional *_mangle_from_* and *maxheaderlen* are as with the `Generator` base class.

19.1.4 `email.policy`: Policy Objects

Added in version 3.3.

Source code: Lib/email/policy.py

The `email` package's prime focus is the handling of email messages as described by the various email and MIME RFCs. However, the general format of email messages (a block of header fields each consisting of a name followed by a colon followed by a value, the whole block followed by a blank line and an arbitrary 'body'), is a format that has found utility outside of the realm of email. Some of these uses conform fairly closely to the main email RFCs, some do not. Even when working with email, there are times when it is desirable to break strict compliance with the RFCs, such as generating emails that interoperate with email servers that do not themselves follow the standards, or that implement extensions you want to use in ways that violate the standards.

Policy objects give the email package the flexibility to handle all these disparate use cases.

A `Policy` object encapsulates a set of attributes and methods that control the behavior of various components of the email package during use. `Policy` instances can be passed to various classes and methods in the email package to alter the default behavior. The settable values and their defaults are described below.

There is a default policy used by all classes in the email package. For all of the `parser` classes and the related convenience functions, and for the `Message` class, this is the `Compat32` policy, via its corresponding pre-defined instance `compat32`. This policy provides for complete backward compatibility (in some cases, including bug compatibility) with the pre-Python3.3 version of the email package.

This default value for the *policy* keyword to `EmailMessage` is the `EmailPolicy` policy, via its pre-defined instance `default`.

When a `Message` or `EmailMessage` object is created, it acquires a policy. If the message is created by a `parser`, a policy passed to the parser will be the policy used by the message it creates. If the message is created by the program, then the policy can be specified when it is created. When a message is passed to a `generator`, the generator uses the policy from the message by default, but you can also pass a specific policy to the generator that will override the one stored on the message object.

The default value for the *policy* keyword for the `email.parser` classes and the parser convenience functions **will be changing** in a future version of Python. Therefore you should **always specify explicitly which policy you want to use** when calling any of the classes and functions described in the `parser` module.

The first part of this documentation covers the features of `Policy`, an *abstract base class* that defines the features that are common to all policy objects, including `compat32`. This includes certain hook methods that are called internally by the email package, which a custom policy could override to obtain different behavior. The second part describes the concrete classes `EmailPolicy` and `Compat32`, which implement the hooks that provide the standard behavior and the backward compatible behavior and features, respectively.

`Policy` instances are immutable, but they can be cloned, accepting the same keyword arguments as the class constructor and returning a new `Policy` instance that is a copy of the original but with the specified attributes values changed.

As an example, the following code could be used to read an email message from a file on disk and pass it to the system `sendmail` program on a Unix system:

```
>>> from email import message_from_binary_file
>>> from email.generator import BytesGenerator
>>> from email import policy
>>> from subprocess import Popen, PIPE
>>> with open('mymsg.txt', 'rb') as f:
...     msg = message_from_binary_file(f, policy=policy.default)
...
>>> p = Popen(['sendmail', msg['To'].addresses[0]], stdin=PIPE)
>>> g = BytesGenerator(p.stdin, policy=msg.policy.clone(linesep='\r\n'))
>>> g.flatten(msg)
>>> p.stdin.close()
>>> rc = p.wait()
```

Here we are telling *BytesGenerator* to use the RFC correct line separator characters when creating the binary string to feed into `sendmail's stdin`, where the default policy would use `\n` line separators.

Some email package methods accept a *policy* keyword argument, allowing the policy to be overridden for that method. For example, the following code uses the *as_bytes()* method of the *msg* object from the previous example and writes the message to a file using the native line separators for the platform on which it is running:

```
>>> import os
>>> with open('converted.txt', 'wb') as f:
...     f.write(msg.as_bytes(policy=msg.policy.clone(linesep=os.linesep)))
17
```

Policy objects can also be combined using the addition operator, producing a policy object whose settings are a combination of the non-default values of the summed objects:

```
>>> compat_SMTP = policy.compat32.clone(linesep='\r\n')
>>> compat_strict = policy.compat32.clone(raise_on_defect=True)
>>> compat_strict_SMTP = compat_SMTP + compat_strict
```

This operation is not commutative; that is, the order in which the objects are added matters. To illustrate:

```
>>> policy100 = policy.compat32.clone(max_line_length=100)
>>> policy80 = policy.compat32.clone(max_line_length=80)
>>> apolicy = policy100 + policy80
>>> apolicy.max_line_length
80
>>> apolicy = policy80 + policy100
>>> apolicy.max_line_length
100
```

class email.policy.**Policy**(***kw*)

This is the *abstract base class* for all policy classes. It provides default implementations for a couple of trivial methods, as well as the implementation of the immutability property, the *clone()* method, and the constructor semantics.

The constructor of a policy class can be passed various keyword arguments. The arguments that may be specified are any non-method properties on this class, plus any additional non-method properties on the concrete class. A value specified in the constructor will override the default value for the corresponding attribute.

This class defines the following properties, and thus values for the following may be passed in the constructor of any policy class:

max_line_length

The maximum length of any line in the serialized output, not counting the end of line character(s). Default is 78, per **RFC 5322**. A value of `0` or *None* indicates that no line wrapping should be done at all.

linesep

The string to be used to terminate lines in serialized output. The default is `\n` because that's the internal end-of-line discipline used by Python, though `\r\n` is required by the RFCs.

cte_type

Controls the type of Content Transfer Encodings that may be or are required to be used. The possible values are:

`7bit`	all data must be "7 bit clean" (ASCII-only). This means that where necessary data will be encoded using either quoted-printable or base64 encoding.
`8bit`	data is not constrained to be 7 bit clean. Data in headers is still required to be ASCII-only and so will be encoded (see *`fold_binary()`* and *`utf8`* below for exceptions), but body parts may use the `8bit` CTE.

A `cte_type` value of `8bit` only works with `BytesGenerator`, not `Generator`, because strings cannot contain binary data. If a `Generator` is operating under a policy that specifies `cte_type=8bit`, it will act as if `cte_type` is `7bit`.

raise_on_defect

If *`True`*, any defects encountered will be raised as errors. If *`False`* (the default), defects will be passed to the *`register_defect()`* method.

mangle_from_

If *`True`*, lines starting with *"From "* in the body are escaped by putting a > in front of them. This parameter is used when the message is being serialized by a generator. Default: *`False`*.

Added in version 3.5.

message_factory

A factory function for constructing a new empty message object. Used by the parser when building messages. Defaults to `None`, in which case *`Message`* is used.

Added in version 3.6.

The following *`Policy`* method is intended to be called by code using the email library to create policy instances with custom settings:

clone(***kw*)

Return a new *`Policy`* instance whose attributes have the same values as the current instance, except where those attributes are given new values by the keyword arguments.

The remaining *`Policy`* methods are called by the email package code, and are not intended to be called by an application using the email package. A custom policy must implement all of these methods.

handle_defect(*obj*, *defect*)

Handle a *defect* found on *obj*. When the email package calls this method, *defect* will always be a subclass of `Defect`.

The default implementation checks the *`raise_on_defect`* flag. If it is `True`, *defect* is raised as an exception. If it is `False` (the default), *obj* and *defect* are passed to *`register_defect()`*.

register_defect(*obj*, *defect*)

Register a *defect* on *obj*. In the email package, *defect* will always be a subclass of `Defect`.

The default implementation calls the `append` method of the `defects` attribute of *obj*. When the email package calls *`handle_defect`*, *obj* will normally have a `defects` attribute that has an `append` method. Custom object types used with the email package (for example, custom `Message` objects) should also provide such an attribute, otherwise defects in parsed messages will raise unexpected errors.

header_max_count(*name*)

Return the maximum allowed number of headers named *name*.

Called when a header is added to an *EmailMessage* or *Message* object. If the returned value is not `0` or `None`, and there are already a number of headers with the name *name* greater than or equal to the value returned, a *ValueError* is raised.

Because the default behavior of `Message.__setitem__` is to append the value to the list of headers, it is easy to create duplicate headers without realizing it. This method allows certain headers to be limited in the number of instances of that header that may be added to a `Message` programmatically. (The limit is not observed by the parser, which will faithfully produce as many headers as exist in the message being parsed.)

The default implementation returns `None` for all header names.

header_source_parse(*sourcelines*)

The email package calls this method with a list of strings, each string ending with the line separation characters found in the source being parsed. The first line includes the field header name and separator. All whitespace in the source is preserved. The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the parsed header.

If an implementation wishes to retain compatibility with the existing email package policies, *name* should be the case preserved name (all characters up to the '`:`' separator), while *value* should be the unfolded value (all line separator characters removed, but whitespace kept intact), stripped of leading whitespace.

sourcelines may contain surrogateescaped binary data.

There is no default implementation

header_store_parse(*name*, *value*)

The email package calls this method with the name and value provided by the application program when the application program is modifying a `Message` programmatically (as opposed to a `Message` created by a parser). The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the header.

If an implementation wishes to retain compatibility with the existing email package policies, the *name* and *value* should be strings or string subclasses that do not change the content of the passed in arguments.

There is no default implementation

header_fetch_parse(*name*, *value*)

The email package calls this method with the *name* and *value* currently stored in the `Message` when that header is requested by the application program, and whatever the method returns is what is passed back to the application as the value of the header being retrieved. Note that there may be more than one header with the same name stored in the `Message`; the method is passed the specific name and value of the header destined to be returned to the application.

value may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the value returned by the method.

There is no default implementation

fold(*name*, *value*)

The email package calls this method with the *name* and *value* currently stored in the `Message` for a given header. The method should return a string that represents that header "folded" correctly (according to the policy settings) by composing the *name* with the *value* and inserting *linesep* characters at the appropriate places. See **RFC 5322** for a discussion of the rules for folding email headers.

value may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the string returned by the method.

fold_binary(*name*, *value*)

The same as *fold()*, except that the returned value should be a bytes object rather than a string.

value may contain surrogateescaped binary data. These could be converted back into binary data in the returned bytes object.

class email.policy.**EmailPolicy**(**kw*)

This concrete *Policy* provides behavior that is intended to be fully compliant with the current email RFCs. These include (but are not limited to) **RFC 5322**, **RFC 2047**, and the current MIME RFCs.

This policy adds new header parsing and folding algorithms. Instead of simple strings, headers are `str` subclasses with attributes that depend on the type of the field. The parsing and folding algorithm fully implement **RFC 2047** and **RFC 5322**.

The default value for the *message_factory* attribute is *EmailMessage*.

In addition to the settable attributes listed above that apply to all policies, this policy adds the following additional attributes:

Added in version 3.6:[1]

utf8

If `False`, follow **RFC 5322**, supporting non-ASCII characters in headers by encoding them as "encoded words". If `True`, follow **RFC 6532** and use `utf-8` encoding for headers. Messages formatted in this way may be passed to SMTP servers that support the `SMTPUTF8` extension (**RFC 6531**).

refold_source

If the value for a header in the `Message` object originated from a *parser* (as opposed to being set by a program), this attribute indicates whether or not a generator should refold that value when transforming the message back into serialized form. The possible values are:

`none`	all source values use original folding
`long`	source values that have any line that is longer than `max_line_length` will be refolded
`all`	all values are refolded.

The default is `long`.

header_factory

A callable that takes two arguments, `name` and `value`, where `name` is a header field name and `value` is an unfolded header field value, and returns a string subclass that represents that header. A default `header_factory` (see *headerregistry*) is provided that supports custom parsing for the various address and date **RFC 5322** header field types, and the major MIME header field stypes. Support for additional custom parsing will be added in the future.

content_manager

An object with at least two methods: get_content and set_content. When the *get_content()* or *set_content()* method of an *EmailMessage* object is called, it calls the corresponding method of this object, passing it the message object as its first argument, and any arguments or keywords that were passed to it as additional arguments. By default `content_manager` is set to *raw_data_manager*.

Added in version 3.4.

The class provides the following concrete implementations of the abstract methods of *Policy*:

[1] Originally added in 3.3 as a *provisional feature*.

header_max_count(*name*)

Returns the value of the *max_count* attribute of the specialized class used to represent the header with the given name.

header_source_parse(*sourcelines*)

The name is parsed as everything up to the ':' and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

header_store_parse(*name*, *value*)

The name is returned unchanged. If the input value has a `name` attribute and it matches *name* ignoring case, the value is returned unchanged. Otherwise the *name* and *value* are passed to `header_factory`, and the resulting header object is returned as the value. In this case a `ValueError` is raised if the input value contains CR or LF characters.

header_fetch_parse(*name*, *value*)

If the value has a `name` attribute, it is returned to unmodified. Otherwise the *name*, and the *value* with any CR or LF characters removed, are passed to the `header_factory`, and the resulting header object is returned. Any surrogateescaped bytes get turned into the unicode unknown-character glyph.

fold(*name*, *value*)

Header folding is controlled by the *refold_source* policy setting. A value is considered to be a 'source value' if and only if it does not have a `name` attribute (having a `name` attribute means it is a header object of some sort). If a source value needs to be refolded according to the policy, it is converted into a header object by passing the *name* and the *value* with any CR and LF characters removed to the `header_factory`. Folding of a header object is done by calling its `fold` method with the current policy.

Source values are split into lines using *splitlines()*. If the value is not to be refolded, the lines are rejoined using the `linesep` from the policy and returned. The exception is lines containing non-ascii binary data. In that case the value is refolded regardless of the `refold_source` setting, which causes the binary data to be CTE encoded using the `unknown-8bit` charset.

fold_binary(*name*, *value*)

The same as *fold()* if *cte_type* is `7bit`, except that the returned value is bytes.

If *cte_type* is `8bit`, non-ASCII binary data is converted back into bytes. Headers with binary data are not refolded, regardless of the `refold_header` setting, since there is no way to know whether the binary data consists of single byte characters or multibyte characters.

The following instances of *EmailPolicy* provide defaults suitable for specific application domains. Note that in the future the behavior of these instances (in particular the `HTTP` instance) may be adjusted to conform even more closely to the RFCs relevant to their domains.

email.policy.**default**

An instance of `EmailPolicy` with all defaults unchanged. This policy uses the standard Python `\n` line endings rather than the RFC-correct `\r\n`.

email.policy.**SMTP**

Suitable for serializing messages in conformance with the email RFCs. Like `default`, but with `linesep` set to `\r\n`, which is RFC compliant.

email.policy.**SMTPUTF8**

The same as `SMTP` except that *utf8* is `True`. Useful for serializing messages to a message store without using encoded words in the headers. Should only be used for SMTP transmission if the sender or recipient addresses have non-ASCII characters (the *smtplib.SMTP.send_message()* method handles this automatically).

email.policy.**HTTP**

Suitable for serializing headers with for use in HTTP traffic. Like `SMTP` except that `max_line_length` is set to `None` (unlimited).

email.policy.**strict**

Convenience instance. The same as `default` except that `raise_on_defect` is set to `True`. This allows any policy to be made strict by writing:

```
somepolicy + policy.strict
```

With all of these *EmailPolicies*, the effective API of the email package is changed from the Python 3.2 API in the following ways:

- Setting a header on a *Message* results in that header being parsed and a header object created.
- Fetching a header value from a *Message* results in that header being parsed and a header object created and returned.
- Any header object, or any header that is refolded due to the policy settings, is folded using an algorithm that fully implements the RFC folding algorithms, including knowing where encoded words are required and allowed.

From the application view, this means that any header obtained through the *EmailMessage* is a header object with extra attributes, whose string value is the fully decoded unicode value of the header. Likewise, a header may be assigned a new value, or a new header created, using a unicode string, and the policy will take care of converting the unicode string into the correct RFC encoded form.

The header objects and their attributes are described in *headerregistry*.

class email.policy.**Compat32**(***kw*)

This concrete *Policy* is the backward compatibility policy. It replicates the behavior of the email package in Python 3.2. The *policy* module also defines an instance of this class, *compat32*, that is used as the default policy. Thus the default behavior of the email package is to maintain compatibility with Python 3.2.

The following attributes have values that are different from the *Policy* default:

mangle_from_

The default is `True`.

The class provides the following concrete implementations of the abstract methods of *Policy*:

header_source_parse(*sourcelines*)

The name is parsed as everything up to the '`:`' and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

header_store_parse(*name*, *value*)

The name and value are returned unmodified.

header_fetch_parse(*name*, *value*)

If the value contains binary data, it is converted into a *Header* object using the `unknown-8bit` charset. Otherwise it is returned unmodified.

fold(*name*, *value*)

Headers are folded using the *Header* folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the `max_line_length`. Non-ASCII binary data are CTE encoded using the `unknown-8bit` charset.

fold_binary(*name*, *value*)

Headers are folded using the *Header* folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the `max_line_length`. If `cte_type` is `7bit`, non-ascii binary data is CTE encoded using the `unknown-8bit` charset. Otherwise the original source header is used, with its existing line breaks and any (RFC invalid) binary data it may contain.

`email.policy.`**compat32**

An instance of *Compat32*, providing backward compatibility with the behavior of the email package in Python 3.2.

19.1.5 `email.errors`: Exception and Defect classes

Source code: Lib/email/errors.py

The following exception classes are defined in the *email.errors* module:

exception `email.errors.`**MessageError**

This is the base class for all exceptions that the *email* package can raise. It is derived from the standard *Exception* class and defines no additional methods.

exception `email.errors.`**MessageParseError**

This is the base class for exceptions raised by the *Parser* class. It is derived from *MessageError*. This class is also used internally by the parser used by *headerregistry*.

exception `email.errors.`**HeaderParseError**

Raised under some error conditions when parsing the **RFC 5322** headers of a message, this class is derived from *MessageParseError*. The *set_boundary()* method will raise this error if the content type is unknown when the method is called. *Header* may raise this error for certain base64 decoding errors, and when an attempt is made to create a header that appears to contain an embedded header (that is, there is what is supposed to be a continuation line that has no leading whitespace and looks like a header).

exception `email.errors.`**BoundaryError**

Deprecated and no longer used.

exception `email.errors.`**MultipartConversionError**

Raised when a payload is added to a *Message* object using `add_payload()`, but the payload is already a scalar and the message's *Content-Type* main type is not either *multipart* or missing. *MultipartConversionError* multiply inherits from *MessageError* and the built-in *TypeError*.

Since `Message.add_payload()` is deprecated, this exception is rarely raised in practice. However the exception may also be raised if the *attach()* method is called on an instance of a class derived from *MIMENonMultipart* (e.g. *MIMEImage*).

exception `email.errors.`**MessageDefect**

This is the base class for all defects found when parsing email messages. It is derived from *ValueError*.

exception `email.errors.`**HeaderDefect**

This is the base class for all defects found when parsing email headers. It is derived from *MessageDefect*.

Here is the list of the defects that the *FeedParser* can find while parsing messages. Note that the defects are added to the message where the problem was found, so for example, if a message nested inside a *multipart/alternative* had a malformed header, that nested message object would have a defect, but the containing messages would not.

All defect classes are subclassed from *email.errors.MessageDefect*.

- `NoBoundaryInMultipartDefect` – A message claimed to be a multipart, but had no *`boundary`* parameter.
- `StartBoundaryNotFoundDefect` – The start boundary claimed in the *`Content-Type`* header was never found.
- `CloseBoundaryNotFoundDefect` – A start boundary was found, but no corresponding close boundary was ever found.

  Added in version 3.3.
- `FirstHeaderLineIsContinuationDefect` – The message had a continuation line as its first header line.
- `MisplacedEnvelopeHeaderDefect` - A "Unix From" header was found in the middle of a header block.
- `MissingHeaderBodySeparatorDefect` - A line was found while parsing headers that had no leading white space but contained no ':'. Parsing continues assuming that the line represents the first line of the body.

  Added in version 3.3.
- `MalformedHeaderDefect` – A header was found that was missing a colon, or was otherwise malformed.

  Deprecated since version 3.3: This defect has not been used for several Python versions.
- `MultipartInvariantViolationDefect` – A message claimed to be a *`multipart`*, but no subparts were found. Note that when a message has this defect, its *`is_multipart()`* method may return `False` even though its content type claims to be *`multipart`*.
- `InvalidBase64PaddingDefect` – When decoding a block of base64 encoded bytes, the padding was not correct. Enough padding is added to perform the decode, but the resulting decoded bytes may be invalid.
- `InvalidBase64CharactersDefect` – When decoding a block of base64 encoded bytes, characters outside the base64 alphabet were encountered. The characters are ignored, but the resulting decoded bytes may be invalid.
- `InvalidBase64LengthDefect` – When decoding a block of base64 encoded bytes, the number of non-padding base64 characters was invalid (1 more than a multiple of 4). The encoded block was kept as-is.
- `InvalidDateDefect` – When decoding an invalid or unparsable date field. The original value is kept as-is.

19.1.6 `email.headerregistry`: Custom Header Objects

Source code: Lib/email/headerregistry.py

Added in version 3.6:[1]

Headers are represented by customized subclasses of *`str`*. The particular class used to represent a given header is determined by the *`header_factory`* of the *`policy`* in effect when the headers are created. This section documents the particular `header_factory` implemented by the email package for handling **RFC 5322** compliant email messages, which not only provides customized header objects for various header types, but also provides an extension mechanism for applications to add their own custom header types.

When using any of the policy objects derived from *`EmailPolicy`*, all headers are produced by *`HeaderRegistry`* and have *`BaseHeader`* as their last base class. Each header class has an additional base class that is determined by the type of the header. For example, many headers have the class *`UnstructuredHeader`* as their other base class. The specialized second class for a header is determined by the name of the header, using a lookup table stored in the *`HeaderRegistry`*. All of this is managed transparently for the typical application program, but interfaces are provided for modifying the default behavior for use by more complex applications.

[1] Originally added in 3.3 as a *provisional module*

The sections below first document the header base classes and their attributes, followed by the API for modifying the behavior of *HeaderRegistry*, and finally the support classes used to represent the data parsed from structured headers.

class email.headerregistry.**BaseHeader**(*name*, *value*)

name and *value* are passed to BaseHeader from the *header_factory* call. The string value of any header object is the *value* fully decoded to unicode.

This base class defines the following read-only properties:

name

The name of the header (the portion of the field before the ':'). This is exactly the value passed in the *header_factory* call for *name*; that is, case is preserved.

defects

A tuple of *HeaderDefect* instances reporting any RFC compliance problems found during parsing. The email package tries to be complete about detecting compliance issues. See the *errors* module for a discussion of the types of defects that may be reported.

max_count

The maximum number of headers of this type that can have the same name. A value of None means unlimited. The BaseHeader value for this attribute is None; it is expected that specialized header classes will override this value as needed.

BaseHeader also provides the following method, which is called by the email library code and should not in general be called by application programs:

fold(*, *policy*)

Return a string containing *linesep* characters as required to correctly fold the header according to *policy*. A *cte_type* of 8bit will be treated as if it were 7bit, since headers may not contain arbitrary binary data. If *utf8* is False, non-ASCII data will be **RFC 2047** encoded.

BaseHeader by itself cannot be used to create a header object. It defines a protocol that each specialized header cooperates with in order to produce the header object. Specifically, BaseHeader requires that the specialized class provide a *classmethod()* named parse. This method is called as follows:

```
parse(string, kwds)
```

kwds is a dictionary containing one pre-initialized key, defects. defects is an empty list. The parse method should append any detected defects to this list. On return, the kwds dictionary *must* contain values for at least the keys decoded and defects. decoded should be the string value for the header (that is, the header value fully decoded to unicode). The parse method should assume that *string* may contain content-transfer-encoded parts, but should correctly handle all valid unicode characters as well so that it can parse un-encoded header values.

BaseHeader's __new__ then creates the header instance, and calls its init method. The specialized class only needs to provide an init method if it wishes to set additional attributes beyond those provided by BaseHeader itself. Such an init method should look like this:

```
def init(self, /, *args, **kw):
    self._myattr = kw.pop('myattr')
    super().init(*args, **kw)
```

That is, anything extra that the specialized class puts in to the kwds dictionary should be removed and handled, and the remaining contents of kw (and args) passed to the BaseHeader init method.

class email.headerregistry.**UnstructuredHeader**

An "unstructured" header is the default type of header in **RFC 5322**. Any header that does not have a specified syntax is treated as unstructured. The classic example of an unstructured header is the *Subject* header.

In **RFC 5322**, an unstructured header is a run of arbitrary text in the ASCII character set. **RFC 2047**, however, has an **RFC 5322** compatible mechanism for encoding non-ASCII text as ASCII characters within a header value. When a *value* containing encoded words is passed to the constructor, the `UnstructuredHeader` parser converts such encoded words into unicode, following the **RFC 2047** rules for unstructured text. The parser uses heuristics to attempt to decode certain non-compliant encoded words. Defects are registered in such cases, as well as defects for issues such as invalid characters within the encoded words or the non-encoded text.

This header type provides no additional attributes.

class email.headerregistry.**DateHeader**

RFC 5322 specifies a very specific format for dates within email headers. The `DateHeader` parser recognizes that date format, as well as recognizing a number of variant forms that are sometimes found "in the wild".

This header type provides the following additional attributes:

datetime

If the header value can be recognized as a valid date of one form or another, this attribute will contain a *datetime* instance representing that date. If the timezone of the input date is specified as `-0000` (indicating it is in UTC but contains no information about the source timezone), then *datetime* will be a naive *datetime*. If a specific timezone offset is found (including `+0000`), then *datetime* will contain an aware `datetime` that uses *datetime.timezone* to record the timezone offset.

The `decoded` value of the header is determined by formatting the `datetime` according to the **RFC 5322** rules; that is, it is set to:

```
email.utils.format_datetime(self.datetime)
```

When creating a `DateHeader`, *value* may be *datetime* instance. This means, for example, that the following code is valid and does what one would expect:

```
msg['Date'] = datetime(2011, 7, 15, 21)
```

Because this is a naive `datetime` it will be interpreted as a UTC timestamp, and the resulting value will have a timezone of `-0000`. Much more useful is to use the *localtime()* function from the *utils* module:

```
msg['Date'] = utils.localtime()
```

This example sets the date header to the current time and date using the current timezone offset.

class email.headerregistry.**AddressHeader**

Address headers are one of the most complex structured header types. The `AddressHeader` class provides a generic interface to any address header.

This header type provides the following additional attributes:

groups

A tuple of *Group* objects encoding the addresses and groups found in the header value. Addresses that are not part of a group are represented in this list as single-address `Groups` whose *display_name* is `None`.

addresses

A tuple of *Address* objects encoding all of the individual addresses from the header value. If the header value contains any groups, the individual addresses from the group are included in the list at the point where the group occurs in the value (that is, the list of addresses is "flattened" into a one dimensional list).

The `decoded` value of the header will have all encoded words decoded to unicode. *idna* encoded domain names are also decoded to unicode. The `decoded` value is set by *joining* the *str* value of the elements of the `groups` attribute with `', '`.

A list of *Address* and *Group* objects in any combination may be used to set the value of an address header. `Group` objects whose `display_name` is `None` will be interpreted as single addresses, which allows an address list to be copied with groups intact by using the list obtained from the `groups` attribute of the source header.

class email.headerregistry.**SingleAddressHeader**

A subclass of *AddressHeader* that adds one additional attribute:

address

The single address encoded by the header value. If the header value actually contains more than one address (which would be a violation of the RFC under the default *policy*), accessing this attribute will result in a *ValueError*.

Many of the above classes also have a `Unique` variant (for example, `UniqueUnstructuredHeader`). The only difference is that in the `Unique` variant, *max_count* is set to 1.

class email.headerregistry.**MIMEVersionHeader**

There is really only one valid value for the *MIME-Version* header, and that is `1.0`. For future proofing, this header class supports other valid version numbers. If a version number has a valid value per **RFC 2045**, then the header object will have non-`None` values for the following attributes:

version

The version number as a string, with any whitespace and/or comments removed.

major

The major version number as an integer

minor

The minor version number as an integer

class email.headerregistry.**ParameterizedMIMEHeader**

MIME headers all start with the prefix 'Content-'. Each specific header has a certain value, described under the class for that header. Some can also take a list of supplemental parameters, which have a common format. This class serves as a base for all the MIME headers that take parameters.

params

A dictionary mapping parameter names to parameter values.

class email.headerregistry.**ContentTypeHeader**

A *ParameterizedMIMEHeader* class that handles the *Content-Type* header.

content_type

The content type string, in the form `maintype/subtype`.

maintype

subtype

class email.headerregistry.**ContentDispositionHeader**

A *ParameterizedMIMEHeader* class that handles the *Content-Disposition* header.

content_disposition

`inline` and `attachment` are the only valid values in common use.

class email.headerregistry.**ContentTransferEncoding**

Handles the *Content-Transfer-Encoding* header.

cte

Valid values are `7bit`, `8bit`, `base64`, and `quoted-printable`. See **RFC 2045** for more information.

class email.headerregistry.**HeaderRegistry**(*base_class=BaseHeader*, *default_class=UnstructuredHeader*, *use_default_map=True*)

This is the factory used by *EmailPolicy* by default. `HeaderRegistry` builds the class used to create a header instance dynamically, using *base_class* and a specialized class retrieved from a registry that it holds. When a given header name does not appear in the registry, the class specified by *default_class* is used as the specialized class. When *use_default_map* is `True` (the default), the standard mapping of header names to classes is copied in to the registry during initialization. *base_class* is always the last class in the generated class's `__bases__` list.

The default mappings are:

subject
UniqueUnstructuredHeader

date
UniqueDateHeader

resent-date
DateHeader

orig-date
UniqueDateHeader

sender
UniqueSingleAddressHeader

resent-sender
SingleAddressHeader

to
UniqueAddressHeader

resent-to
AddressHeader

cc
UniqueAddressHeader

resent-cc
AddressHeader

bcc
UniqueAddressHeader

resent-bcc
AddressHeader

from
UniqueAddressHeader

resent-from
AddressHeader

reply-to
UniqueAddressHeader

mime-version
MIMEVersionHeader

content-type
ContentTypeHeader

content-disposition
ContentDispositionHeader

content-transfer-encoding
ContentTransferEncodingHeader

message-id
MessageIDHeader

`HeaderRegistry` has the following methods:

map_to_type(*self*, *name*, *cls*)

name is the name of the header to be mapped. It will be converted to lower case in the registry. *cls* is the specialized class to be used, along with *base_class*, to create the class used to instantiate headers that match *name*.

__getitem__(*name*)

Construct and return a class to handle creating a *name* header.

__call__(*name*, *value*)

Retrieves the specialized header associated with *name* from the registry (using *default_class* if *name* does not appear in the registry) and composes it with *base_class* to produce a class, calls the constructed class's constructor, passing it the same argument list, and finally returns the class instance created thereby.

The following classes are the classes used to represent data parsed from structured headers and can, in general, be used by an application program to construct structured values to assign to specific headers.

class email.headerregistry.**Address**(*display_name=''*, *username=''*, *domain=''*, *addr_spec=None*)

The class used to represent an email address. The general form of an address is:

```
[display_name] <username@domain>
```

or:

```
username@domain
```

where each part must conform to specific syntax rules spelled out in **RFC 5322**.

As a convenience *addr_spec* can be specified instead of *username* and *domain*, in which case *username* and *domain* will be parsed from the *addr_spec*. An *addr_spec* must be a properly RFC quoted string; if it is not `Address` will raise an error. Unicode characters are allowed and will be property encoded when serialized. However, per the RFCs, unicode is *not* allowed in the username portion of the address.

display_name

The display name portion of the address, if any, with all quoting removed. If the address does not have a display name, this attribute will be an empty string.

username

The `username` portion of the address, with all quoting removed.

domain

The `domain` portion of the address.

addr_spec

The `username@domain` portion of the address, correctly quoted for use as a bare address (the second form shown above). This attribute is not mutable.

__str__()

The str value of the object is the address quoted according to **RFC 5322** rules, but with no Content Transfer Encoding of any non-ASCII characters.

To support SMTP (**RFC 5321**), Address handles one special case: if username and domain are both the empty string (or None), then the string value of the Address is <>.

class email.headerregistry.**Group**(*display_name=None*, *addresses=None*)

The class used to represent an address group. The general form of an address group is:

```
display_name: [address-list];
```

As a convenience for processing lists of addresses that consist of a mixture of groups and single addresses, a Group may also be used to represent single addresses that are not part of a group by setting *display_name* to None and providing a list of the single address as *addresses*.

display_name

The display_name of the group. If it is None and there is exactly one Address in addresses, then the Group represents a single address that is not in a group.

addresses

A possibly empty tuple of *Address* objects representing the addresses in the group.

__str__()

The str value of a Group is formatted according to **RFC 5322**, but with no Content Transfer Encoding of any non-ASCII characters. If display_name is none and there is a single Address in the addresses list, the str value will be the same as the str of that single Address.

19.1.7 email.contentmanager: Managing MIME Content

Source code: Lib/email/contentmanager.py

Added in version 3.6:[1]

class email.contentmanager.**ContentManager**

Base class for content managers. Provides the standard registry mechanisms to register converters between MIME content and other representations, as well as the get_content and set_content dispatch methods.

get_content(*msg*, **args*, ***kw*)

Look up a handler function based on the mimetype of *msg* (see next paragraph), call it, passing through all arguments, and return the result of the call. The expectation is that the handler will extract the payload from *msg* and return an object that encodes information about the extracted data.

To find the handler, look for the following keys in the registry, stopping with the first one found:

- the string representing the full MIME type (maintype/subtype)
- the string representing the maintype
- the empty string

If none of these keys produce a handler, raise a *KeyError* for the full MIME type.

[1] Originally added in 3.4 as a *provisional module*

set_content(*msg*, *obj*, **args*, ***kw*)

If the `maintype` is `multipart`, raise a *TypeError*; otherwise look up a handler function based on the type of *obj* (see next paragraph), call *clear_content()* on the *msg*, and call the handler function, passing through all arguments. The expectation is that the handler will transform and store *obj* into *msg*, possibly making other changes to *msg* as well, such as adding various MIME headers to encode information needed to interpret the stored data.

To find the handler, obtain the type of *obj* (`typ = type(obj)`), and look for the following keys in the registry, stopping with the first one found:

- the type itself (`typ`)
- the type's fully qualified name (`typ.__module__ + '.' + typ.__qualname__`).
- the type's qualname (`typ.__qualname__`)
- the type's name (`typ.__name__`).

If none of the above match, repeat all of the checks above for each of the types in the *MRO* (`typ.__mro__`). Finally, if no other key yields a handler, check for a handler for the key `None`. If there is no handler for `None`, raise a *KeyError* for the fully qualified name of the type.

Also add a *MIME-Version* header if one is not present (see also *MIMEPart*).

add_get_handler(*key*, *handler*)

Record the function *handler* as the handler for *key*. For the possible values of *key*, see *get_content()*.

add_set_handler(*typekey*, *handler*)

Record *handler* as the function to call when an object of a type matching *typekey* is passed to *set_content()*. For the possible values of *typekey*, see *set_content()*.

Content Manager Instances

Currently the email package provides only one concrete content manager, *raw_data_manager*, although more may be added in the future. *raw_data_manager* is the *content_manager* provided by *EmailPolicy* and its derivatives.

`email.contentmanager.`**raw_data_manager**

This content manager provides only a minimum interface beyond that provided by *Message* itself: it deals only with text, raw byte strings, and *Message* objects. Nevertheless, it provides significant advantages compared to the base API: `get_content` on a text part will return a unicode string without the application needing to manually decode it, `set_content` provides a rich set of options for controlling the headers added to a part and controlling the content transfer encoding, and it enables the use of the various `add_` methods, thereby simplifying the creation of multipart messages.

`email.contentmanager.`**get_content**(*msg*, *errors='replace'*)

Return the payload of the part as either a string (for `text` parts), an *EmailMessage* object (for `message/rfc822` parts), or a `bytes` object (for all other non-multipart types). Raise a *KeyError* if called on a `multipart`. If the part is a `text` part and *errors* is specified, use it as the error handler when decoding the payload to unicode. The default error handler is `replace`.

`email.contentmanager.`**set_content**(*msg*, *<'str'>*, *subtype="plain"*, *charset='utf-8'*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)

`email.contentmanager.`**set_content**(*msg*, *<'bytes'>*, *maintype*, *subtype*, *cte="base64"*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)

email.contentmanager.**set_content**(*msg*, *<'EmailMessage'>*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)

Add headers and payload to *msg*:

Add a *Content-Type* header with a `maintype/subtype` value.

- For `str`, set the MIME `maintype` to `text`, and set the subtype to *subtype* if it is specified, or `plain` if it is not.
- For `bytes`, use the specified *maintype* and *subtype*, or raise a *TypeError* if they are not specified.
- For *EmailMessage* objects, set the maintype to `message`, and set the subtype to *subtype* if it is specified or `rfc822` if it is not. If *subtype* is `partial`, raise an error (`bytes` objects must be used to construct `message/partial` parts).

If *charset* is provided (which is valid only for `str`), encode the string to bytes using the specified character set. The default is `utf-8`. If the specified *charset* is a known alias for a standard MIME charset name, use the standard charset instead.

If *cte* is set, encode the payload using the specified content transfer encoding, and set the *Content-Transfer-Encoding* header to that value. Possible values for *cte* are `quoted-printable`, `base64`, `7bit`, `8bit`, and `binary`. If the input cannot be encoded in the specified encoding (for example, specifying a *cte* of `7bit` for an input that contains non-ASCII values), raise a *ValueError*.

- For `str` objects, if *cte* is not set use heuristics to determine the most compact encoding.
- For *EmailMessage*, per **RFC 2046**, raise an error if a *cte* of `quoted-printable` or `base64` is requested for *subtype* `rfc822`, and for any *cte* other than `7bit` for *subtype* `external-body`. For `message/rfc822`, use `8bit` if *cte* is not specified. For all other values of *subtype*, use `7bit`.

> **Note**
>
> A *cte* of `binary` does not actually work correctly yet. The `EmailMessage` object as modified by `set_content` is correct, but *BytesGenerator* does not serialize it correctly.

If *disposition* is set, use it as the value of the *Content-Disposition* header. If not specified, and *filename* is specified, add the header with the value `attachment`. If *disposition* is not specified and *filename* is also not specified, do not add the header. The only valid values for *disposition* are `attachment` and `inline`.

If *filename* is specified, use it as the value of the `filename` parameter of the *Content-Disposition* header.

If *cid* is specified, add a *Content-ID* header with *cid* as its value.

If *params* is specified, iterate its `items` method and use the resulting `(key, value)` pairs to set additional parameters on the *Content-Type* header.

If *headers* is specified and is a list of strings of the form `headername: headervalue` or a list of `header` objects (distinguished from strings by having a `name` attribute), add the headers to *msg*.

19.1.8 email: Examples

Here are a few examples of how to use the *email* package to read, write, and send simple email messages, as well as more complex MIME messages.

First, let's see how to create and send a simple text message (both the text content and the addresses may contain unicode characters):

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.message import EmailMessage

# Open the plain text file whose name is in textfile for reading.
with open(textfile) as fp:
    # Create a text/plain message
    msg = EmailMessage()
    msg.set_content(fp.read())

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = f'The contents of {textfile}'
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server.
s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()
```

Parsing **RFC 822** headers can easily be done by the using the classes from the *parser* module:

```
# Import the email modules we'll need
#from email.parser import BytesParser
from email.parser import Parser
from email.policy import default

# If the e-mail headers are in a file, uncomment these two lines:
# with open(messagefile, 'rb') as fp:
#     headers = BytesParser(policy=default).parse(fp)

#  Or for parsing headers in a string (this is an uncommon operation), use:
headers = Parser(policy=default).parsestr(
        'From: Foo Bar <user@example.com>\n'
        'To: <someone_else@example.com>\n'
        'Subject: Test message\n'
        '\n'
        'Body would go here\n')

#  Now the header items can be accessed as a dictionary:
print('To: {}'.format(headers['to']))
print('From: {}'.format(headers['from']))
print('Subject: {}'.format(headers['subject']))

# You can also access the parts of the addresses:
print('Recipient username: {}'.format(headers['to'].addresses[0].username))
print('Sender name: {}'.format(headers['from'].addresses[0].display_name))
```

Here's an example of how to send a MIME message containing a bunch of family pictures that may be residing in a directory:

```
# Import smtplib for the actual sending function.
import smtplib

# Here are the email package modules we'll need.
from email.message import EmailMessage

# Create the container email message.
msg = EmailMessage()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = ', '.join(family)
msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

# Open the files in binary mode.  You can also omit the subtype
# if you want MIMEImage to guess it.
for file in pngfiles:
    with open(file, 'rb') as fp:
        img_data = fp.read()
    msg.add_attachment(img_data, maintype='image',
                                 subtype='png')

# Send the email via our own SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

Here's an example of how to send the entire contents of a directory as an email message:[1]

```
#!/usr/bin/env python3

"""Send the contents of a directory as a MIME message."""

import os
import smtplib
# For guessing MIME type based on file name extension
import mimetypes

from argparse import ArgumentParser

from email.message import EmailMessage
from email.policy import SMTP


def main():
    parser = ArgumentParser(description="""\
Send the contents of a directory as a MIME message.
Unless the -o option is given, the email is sent by forwarding to your local
SMTP server, which then does the normal delivery process.  Your local machine
must be running an SMTP server.
""")
    parser.add_argument('-d', '--directory',
                        help="""Mail the contents of the specified directory,
```

(continues on next page)

[1] Thanks to Matthew Dixon Cowles for the original inspiration and examples.

(continued from previous page)

```
                        otherwise use the current directory.  Only the regular
                        files in the directory are sent, and we don't recurse to
                        subdirectories.""")
    parser.add_argument('-o', '--output',
                        metavar='FILE',
                        help="""Print the composed message to FILE instead of
                        sending the message to the SMTP server.""")
    parser.add_argument('-s', '--sender', required=True,
                        help='The value of the From: header (required)')
    parser.add_argument('-r', '--recipient', required=True,
                        action='append', metavar='RECIPIENT',
                        default=[], dest='recipients',
                        help='A To: header value (at least one required)')
    args = parser.parse_args()
    directory = args.directory
    if not directory:
        directory = '.'
    # Create the message
    msg = EmailMessage()
    msg['Subject'] = f'Contents of directory {os.path.abspath(directory)}'
    msg['To'] = ', '.join(args.recipients)
    msg['From'] = args.sender
    msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        # Guess the content type based on the file's extension.  Encoding
        # will be ignored, although we should check for simple things like
        # gzip'd or compressed files.
        ctype, encoding = mimetypes.guess_file_type(path)
        if ctype is None or encoding is not None:
            # No guess could be made, or the file is encoded (compressed), so
            # use a generic bag-of-bits type.
            ctype = 'application/octet-stream'
        maintype, subtype = ctype.split('/', 1)
        with open(path, 'rb') as fp:
            msg.add_attachment(fp.read(),
                               maintype=maintype,
                               subtype=subtype,
                               filename=filename)
    # Now send or store the message
    if args.output:
        with open(args.output, 'wb') as fp:
            fp.write(msg.as_bytes(policy=SMTP))
    else:
        with smtplib.SMTP('localhost') as s:
            s.send_message(msg)


if __name__ == '__main__':
    main()
```

Here's an example of how to unpack a MIME message like the one above, into a directory of files:

```
#!/usr/bin/env python3

"""Unpack a MIME message into a directory of files."""

import os
import email
import mimetypes

from email.policy import default

from argparse import ArgumentParser


def main():
    parser = ArgumentParser(description="""\
Unpack a MIME message into a directory of files.
""")
    parser.add_argument('-d', '--directory', required=True,
                        help="""Unpack the MIME message into the named
                        directory, which will be created if it doesn't already
                        exist.""")
    parser.add_argument('msgfile')
    args = parser.parse_args()

    with open(args.msgfile, 'rb') as fp:
        msg = email.message_from_binary_file(fp, policy=default)

    try:
        os.mkdir(args.directory)
    except FileExistsError:
        pass

    counter = 1
    for part in msg.walk():
        # multipart/* are just containers
        if part.get_content_maintype() == 'multipart':
            continue
        # Applications should really sanitize the given filename so that an
        # email message can't be used to overwrite important files
        filename = part.get_filename()
        if not filename:
            ext = mimetypes.guess_extension(part.get_content_type())
            if not ext:
                # Use a generic bag-of-bits extension
                ext = '.bin'
            filename = f'part-{counter:03d}{ext}'
        counter += 1
        with open(os.path.join(args.directory, filename), 'wb') as fp:
            fp.write(part.get_payload(decode=True))


if __name__ == '__main__':
    main()
```

Here's an example of how to create an HTML message with an alternative plain text version. To make things a bit more interesting, we include a related image in the html part, and we save a copy of what we are going to send to disk, as well as sending it.

```
#!/usr/bin/env python3

import smtplib

from email.message import EmailMessage
from email.headerregistry import Address
from email.utils import make_msgid

# Create the base text message.
msg = EmailMessage()
msg['Subject'] = "Pourquoi pas des asperges pour ce midi ?"
msg['From'] = Address("Pepé Le Pew", "pepe", "example.com")
msg['To'] = (Address("Penelope Pussycat", "penelope", "example.com"),
             Address("Fabrette Pussycat", "fabrette", "example.com"))
msg.set_content("""\
Salut!

Cette recette [1] sera sûrement un très bon repas.

[1] http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718

--Pepé
""")

# Add the html version.  This converts the message into a multipart/alternative
# container, with the original text message as the first part and the new html
# message as the second part.
asparagus_cid = make_msgid()
msg.add_alternative("""\
<html>
  <head></head>
  <body>
    <p>Salut!</p>
    <p>Cette
        <a href="http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718">
            recette
        </a> sera sûrement un très bon repas.
    </p>
    <img src="cid:{asparagus_cid}" />
  </body>
</html>
""".format(asparagus_cid=asparagus_cid[1:-1]), subtype='html')
# note that we needed to peel the <> off the msgid for use in the html.

# Now add the related image to the html part.
with open("roasted-asparagus.jpg", 'rb') as img:
    msg.get_payload()[1].add_related(img.read(), 'image', 'jpeg',
                                     cid=asparagus_cid)

# Make a local copy of what we are going to send.
with open('outgoing.msg', 'wb') as f:
    f.write(bytes(msg))

# Send the message via local SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

If we were sent the message from the last example, here is one way we could process it:

```
import os
import sys
import tempfile
import mimetypes
import webbrowser

# Import the email modules we'll need
from email import policy
from email.parser import BytesParser


def magic_html_parser(html_text, partfiles):
    """Return safety-sanitized html linked to partfiles.

    Rewrite the href="cid:...." attributes to point to the filenames in partfiles.
    Though not trivial, this should be possible using html.parser.
    """
    raise NotImplementedError("Add the magic needed")


# In a real program you'd get the filename from the arguments.
with open('outgoing.msg', 'rb') as fp:
    msg = BytesParser(policy=policy.default).parse(fp)

# Now the header items can be accessed as a dictionary, and any non-ASCII will
# be converted to unicode:
print('To:', msg['to'])
print('From:', msg['from'])
print('Subject:', msg['subject'])

# If we want to print a preview of the message content, we can extract whatever
# the least formatted payload is and print the first three lines.  Of course,
# if the message has no plain text part printing the first three lines of html
# is probably useless, but this is just a conceptual example.
simplest = msg.get_body(preferencelist=('plain', 'html'))
print()
print(''.join(simplest.get_content().splitlines(keepends=True)[:3]))

ans = input("View full message?")
if ans.lower()[0] == 'n':
    sys.exit()

# We can extract the richest alternative in order to display it:
richest = msg.get_body()
partfiles = {}
if richest['content-type'].maintype == 'text':
    if richest['content-type'].subtype == 'plain':
        for line in richest.get_content().splitlines():
            print(line)
        sys.exit()
    elif richest['content-type'].subtype == 'html':
        body = richest
    else:
        print("Don't know how to display {}".format(richest.get_content_type()))
        sys.exit()
elif richest['content-type'].content_type == 'multipart/related':
    body = richest.get_body(preferencelist=('html'))
```

(continues on next page)

(continued from previous page)

```
    for part in richest.iter_attachments():
        fn = part.get_filename()
        if fn:
            extension = os.path.splitext(part.get_filename())[1]
        else:
            extension = mimetypes.guess_extension(part.get_content_type())
        with tempfile.NamedTemporaryFile(suffix=extension, delete=False) as f:
            f.write(part.get_content())
            # again strip the <> to go from email form of cid to html form.
            partfiles[part['content-id'][1:-1]] = f.name
else:
    print("Don't know how to display {}".format(richest.get_content_type()))
    sys.exit()
with tempfile.NamedTemporaryFile(mode='w', delete=False) as f:
    f.write(magic_html_parser(body.get_content(), partfiles))
webbrowser.open(f.name)
os.remove(f.name)
for fn in partfiles.values():
    os.remove(fn)

# Of course, there are lots of email messages that could break this simple
# minded program, but it will handle the most common ones.
```

Up to the prompt, the output from the above is:

```
To: Penelope Pussycat <penelope@example.com>, Fabrette Pussycat <fabrette@example.com>
From: Pepé Le Pew <pepe@example.com>
Subject: Pourquoi pas des asperges pour ce midi ?

Salut!

Cette recette [1] sera sûrement un très bon repas.
```

Legacy API:

19.1.9 `email.message.Message`: Representing an email message using the `compat32` API

The *Message* class is very similar to the *EmailMessage* class, without the methods added by that class, and with the default behavior of certain other methods being slightly different. We also document here some methods that, while supported by the *EmailMessage* class, are not recommended unless you are dealing with legacy code.

The philosophy and structure of the two classes is otherwise the same.

This document describes the behavior under the default (for *Message*) policy *Compat32*. If you are going to use another policy, you should be using the *EmailMessage* class instead.

An email message consists of *headers* and a *payload*. Headers must be **RFC 5322** style names and values, where the field name and value are separated by a colon. The colon is not part of either the field name or the field value. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as *multipart/** or *message/rfc822*.

The conceptual model provided by a *Message* object is that of an ordered dictionary of headers with additional methods for accessing both specialized information from the headers, for accessing the payload, for generating a serialized version

of the message, and for recursively walking over the object tree. Note that duplicate headers are supported but special methods must be used to access them.

The *Message* pseudo-dictionary is indexed by the header names, which must be ASCII values. The values of the dictionary are strings that are supposed to contain only ASCII characters; there is some special handling for non-ASCII input, but it doesn't always produce the correct results. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. There may also be a single envelope header, also known as the *Unix-From* header or the `From_` header. The *payload* is either a string or bytes, in the case of simple message objects, or a list of *Message* objects, for MIME container documents (e.g. *multipart/** and *message/rfc822*).

Here are the methods of the *Message* class:

class email.message.**Message**(*policy=compat32*)

If *policy* is specified (it must be an instance of a *policy* class) use the rules it specifies to update and serialize the representation of the message. If *policy* is not set, use the *compat32* policy, which maintains backward compatibility with the Python 3.2 version of the email package. For more information see the *policy* documentation.

Changed in version 3.3: The *policy* keyword argument was added.

as_string(*unixfrom=False*, *maxheaderlen=0*, *policy=None*)

Return the entire message flattened as a string. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. For backward compatibility reasons, *maxheaderlen* defaults to `0`, so if you want a different value you must override it explicitly (the value specified for *max_line_length* in the policy will be ignored by this method). The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the `Generator`.

Flattening the message may trigger changes to the *Message* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required by the Unix mbox format. For more flexibility, instantiate a *Generator* instance and use its *flatten()* method directly. For example:

```
from io import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

If the message object contains binary data that is not encoded according to RFC standards, the non-compliant data will be replaced by unicode "unknown character" code points. (See also *as_bytes()* and *BytesGenerator*.)

Changed in version 3.4: the *policy* keyword argument was added.

__str__()

Equivalent to *as_string()*. Allows `str(msg)` to produce a string containing the formatted message.

as_bytes(*unixfrom=False*, *policy=None*)

Return the entire message flattened as a bytes object. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the `BytesGenerator`.

Flattening the message may trigger changes to the *Message* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required by the Unix mbox format. For more flexibility, instantiate a *`BytesGenerator`* instance and use its *`flatten()`* method directly. For example:

```
from io import BytesIO
from email.generator import BytesGenerator
fp = BytesIO()
g = BytesGenerator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

Added in version 3.4.

__bytes__()

Equivalent to *`as_bytes()`*. Allows `bytes(msg)` to produce a bytes object containing the formatted message.

Added in version 3.4.

is_multipart()

Return `True` if the message's payload is a list of sub-*`Message`* objects, otherwise return `False`. When *`is_multipart()`* returns `False`, the payload should be a string object (which might be a CTE encoded binary payload). (Note that *`is_multipart()`* returning `True` does not necessarily mean that "msg.get_content_maintype() == 'multipart'" will return the `True`. For example, `is_multipart` will return `True` when the *`Message`* is of type `message/rfc822`.)

set_unixfrom(*unixfrom*)

Set the message's envelope header to *unixfrom*, which should be a string.

get_unixfrom()

Return the message's envelope header. Defaults to `None` if the envelope header was never set.

attach(*payload*)

Add the given *payload* to the current payload, which must be `None` or a list of *`Message`* objects before the call. After the call, the payload will always be a list of *`Message`* objects. If you want to set the payload to a scalar object (e.g. a string), use *`set_payload()`* instead.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by *`set_content()`* and the related `make` and `add` methods.

get_payload(*i=None*, *decode=False*)

Return the current payload, which will be a list of *`Message`* objects when *`is_multipart()`* is `True`, or a string when *`is_multipart()`* is `False`. If the payload is a list and you mutate the list object, you modify the message's payload in place.

With optional argument *i*, *`get_payload()`* will return the *i*-th element of the payload, counting from zero, if *`is_multipart()`* is `True`. An *`IndexError`* will be raised if *i* is less than 0 or greater than or equal to the number of items in the payload. If the payload is a string (i.e. *`is_multipart()`* is `False`) and *i* is given, a *`TypeError`* is raised.

Optional *decode* is a flag indicating whether the payload should be decoded or not, according to the *`Content-Transfer-Encoding`* header. When `True` and the message is not a multipart, the payload will be decoded if this header's value is `quoted-printable` or `base64`. If some other encoding is used, or *`Content-Transfer-Encoding`* header is missing, the payload is returned as-is (undecoded). In all cases the returned value is binary data. If the message is a multipart and the *decode* flag is `True`, then `None` is returned. If the payload is base64 and it was not perfectly formed (missing padding, characters outside the base64 alphabet), then an appropriate defect will be added to the message's defect property (`InvalidBase64PaddingDefect` or `InvalidBase64CharactersDefect`, respectively).

When *decode* is `False` (the default) the body is returned as a string without decoding the *Content-Transfer-Encoding*. However, for a *Content-Transfer-Encoding* of 8bit, an attempt is made to decode the original bytes using the `charset` specified by the *Content-Type* header, using the `replace` error handler. If no `charset` is specified, or if the `charset` given is not recognized by the email package, the body is decoded using the default ASCII charset.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by *get_content()* and *iter_parts()*.

set_payload(*payload*, *charset=None*)

Set the entire message object's payload to *payload*. It is the client's responsibility to ensure the payload invariants. Optional *charset* sets the message's default character set; see *set_charset()* for details.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by *set_content()*.

set_charset(*charset*)

Set the character set of the payload to *charset*, which can either be a *Charset* instance (see *email.charset*), a string naming a character set, or `None`. If it is a string, it will be converted to a *Charset* instance. If *charset* is `None`, the `charset` parameter will be removed from the *Content-Type* header (the message will not be otherwise modified). Anything else will generate a *TypeError*.

If there is no existing *MIME-Version* header one will be added. If there is no existing *Content-Type* header, one will be added with a value of *text/plain*. Whether the *Content-Type* header already exists or not, its `charset` parameter will be set to *charset.output_charset*. If *charset.input_charset* and *charset.output_charset* differ, the payload will be re-encoded to the *output_charset*. If there is no existing *Content-Transfer-Encoding* header, then the payload will be transfer-encoded, if needed, using the specified *Charset*, and a header with the appropriate value will be added. If a *Content-Transfer-Encoding* header already exists, the payload is assumed to already be correctly encoded using that *Content-Transfer-Encoding* and is not modified.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the *charset* parameter of the `email.emailmessage.EmailMessage.set_content()` method.

get_charset()

Return the *Charset* instance associated with the message's payload.

This is a legacy method. On the `EmailMessage` class it always returns `None`.

The following methods implement a mapping-like interface for accessing the message's **RFC 2822** headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by *keys()*, but in a *Message* object, headers are always returned in the order they appeared in the original message, or were added to the message later. Any header deleted and then re-added are always appended to the end of the header list.

These semantic differences are intentional and are biased toward maximal convenience.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

In a model generated from bytes, any header values that (in contravention of the RFCs) contain non-ASCII bytes will, when retrieved through this interface, be represented as *Header* objects with a charset of `unknown-8bit`.

__len__()

Return the total number of headers, including duplicates.

__contains__(*name*)

Return `True` if the message object has a field named *name*. Matching is done case-insensitively and *name* should not include the trailing colon. Used for the `in` operator, e.g.:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)

Return the value of the named header field. *name* should not include the colon field separator. If the header is missing, `None` is returned; a *KeyError* is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the *get_all()* method to get the values of all the extant named headers.

__setitem__(*name*, *val*)

Add a header to the message with field name *name* and value *val*. The field is appended to the end of the message's existing fields.

Note that this does *not* overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name *name*, delete the field first, e.g.:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

__delitem__(*name*)

Delete all occurrences of the field with name *name* from the message's headers. No exception is raised if the named field isn't present in the headers.

keys()

Return a list of all the message's header field names.

values()

Return a list of all the message's field values.

items()

Return a list of 2-tuples containing all the message's field headers and values.

get(*name*, *failobj=None*)

Return the value of the named header field. This is identical to `__getitem__()` except that optional *failobj* is returned if the named header is missing (defaults to `None`).

Here are some additional useful methods:

get_all(*name*, *failobj=None*)

Return a list of all the values for the field named *name*. If there are no such named headers in the message, *failobj* is returned (defaults to `None`).

add_header(*_name*, *_value*, ***_params*)

Extended header setting. This method is similar to *__setitem__()* except that additional header parameters can be provided as keyword arguments. *_name* is the header field to add and *_value* is the *primary* value for the header.

For each item in the keyword argument dictionary *_params*, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as `key="value"` unless the value is `None`, in which case only the key will be added. If the value contains non-ASCII characters, it can be specified as a three tuple in the format `(CHARSET, LANGUAGE, VALUE)`, where `CHARSET` is a string naming the charset to be used to encode the value, `LANGUAGE` can usually be set to `None` or the empty string (see **RFC 2231** for other possibilities), and `VALUE` is the string value containing non-ASCII code points. If a three tuple is not passed and the value contains non-ASCII characters, it is automatically encoded in **RFC 2231** format using a `CHARSET` of `utf-8` and a `LANGUAGE` of `None`.

Here's an example:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.gif"
```

An example with non-ASCII characters:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

Which produces

```
Content-Disposition: attachment; filename*="iso-8859-1''Fu%DFballer.ppt"
```

replace_header(*_name*, *_value*)

Replace a header. Replace the first header found in the message that matches *_name*, retaining header order and field name case. If no matching header was found, a *KeyError* is raised.

get_content_type()

Return the message's content type. The returned string is coerced to lower case of the form *maintype/subtype*. If there was no *Content-Type* header in the message the default type as given by *get_default_type()* will be returned. Since according to **RFC 2045**, messages always have a default type, *get_content_type()* will always return a value.

RFC 2045 defines a message's default type to be *text/plain* unless it appears inside a *multipart/digest* container, in which case it would be *message/rfc822*. If the *Content-Type* header has an invalid type specification, **RFC 2045** mandates that the default type be *text/plain*.

get_content_maintype()

Return the message's main content type. This is the *maintype* part of the string returned by *get_content_type()*.

get_content_subtype()

Return the message's sub-content type. This is the *subtype* part of the string returned by *get_content_type()*.

get_default_type()

Return the default content type. Most messages have a default content type of *text/plain*, except for messages that are subparts of *multipart/digest* containers. Such subparts have a default content type of *message/rfc822*.

set_default_type(*ctype*)

Set the default content type. *ctype* should either be *text/plain* or *message/rfc822*, although this is not enforced. The default content type is not stored in the *Content-Type* header.

get_params(*failobj=None*, *header='content-type'*, *unquote=True*)

Return the message's *Content-Type* parameters, as a list. The elements of the returned list are 2-tuples of key/value pairs, as split on the `'='` sign. The left hand side of the `'='` is the key, while the right hand side is the value. If there is no `'='` sign in the parameter the value is the empty string, otherwise the value is as described in *get_param()* and is unquoted if optional *unquote* is `True` (the default).

Optional *failobj* is the object to return if there is no *Content-Type* header. Optional *header* is the header to search instead of *Content-Type*.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the *params* property of the individual header objects returned by the header access methods.

get_param(*param*, *failobj=None*, *header='content-type'*, *unquote=True*)

Return the value of the *Content-Type* header's parameter *param* as a string. If the message has no *Content-Type* header or if there is no such parameter, then *failobj* is returned (defaults to `None`).

Optional *header* if given, specifies the message header to use instead of *Content-Type*.

Parameter keys are always compared case insensitively. The return value can either be a string, or a 3-tuple if the parameter was **RFC 2231** encoded. When it's a 3-tuple, the elements of the value are of the form `(CHARSET, LANGUAGE, VALUE)`. Note that both `CHARSET` and `LANGUAGE` can be `None`, in which case you should consider `VALUE` to be encoded in the `us-ascii` charset. You can usually ignore `LANGUAGE`.

If your application doesn't care whether the parameter was encoded as in **RFC 2231**, you can collapse the parameter value by calling *email.utils.collapse_rfc2231_value()*, passing in the return value from *get_param()*. This will return a suitably decoded Unicode string when the value is a tuple, or the original string unquoted if it isn't. For example:

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

In any case, the parameter value (either the returned string, or the `VALUE` item in the 3-tuple) is always unquoted, unless *unquote* is set to `False`.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the *params* property of the individual header objects returned by the header access methods.

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)

Set a parameter in the *Content-Type* header. If the parameter already exists in the header, its value will be replaced with *value*. If the *Content-Type* header as not yet been defined for this message, it will be set to *text/plain* and the new parameter value will be appended as per **RFC 2045**.

Optional *header* specifies an alternative header to *Content-Type*, and all parameters will be quoted as necessary unless optional *requote* is `False` (the default is `True`).

If optional *charset* is specified, the parameter will be encoded according to **RFC 2231**. Optional *language* specifies the RFC 2231 language, defaulting to the empty string. Both *charset* and *language* should be strings.

If *replace* is `False` (the default) the header is moved to the end of the list of headers. If *replace* is `True`, the header will be updated in place.

Changed in version 3.4: `replace` keyword was added.

del_param(*param*, *header='content-type'*, *requote=True*)

Remove the given parameter completely from the *Content-Type* header. The header will be re-written in place without the parameter or its value. All values will be quoted as necessary unless *requote* is `False` (the default is `True`). Optional *header* specifies an alternative to *Content-Type*.

set_type(*type*, *header='Content-Type'*, *requote=True*)

Set the main type and subtype for the *Content-Type* header. *type* must be a string in the form *maintype/subtype*, otherwise a *ValueError* is raised.

This method replaces the *Content-Type* header, keeping all the parameters in place. If *requote* is `False`, this leaves the existing header's quoting as is, otherwise the parameters will be quoted (the default).

An alternative header can be specified in the *header* argument. When the *Content-Type* header is set a *MIME-Version* header is also added.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the `make_` and `add_` methods.

get_filename(*failobj=None*)

Return the value of the `filename` parameter of the *Content-Disposition* header of the message. If the header does not have a `filename` parameter, this method falls back to looking for the `name` parameter on the *Content-Type* header. If neither is found, or the header is missing, then *failobj* is returned. The returned string will always be unquoted as per *email.utils.unquote()*.

get_boundary(*failobj=None*)

Return the value of the `boundary` parameter of the *Content-Type* header of the message, or *failobj* if either the header is missing, or has no `boundary` parameter. The returned string will always be unquoted as per *email.utils.unquote()*.

set_boundary(*boundary*)

Set the `boundary` parameter of the *Content-Type* header to *boundary*. *set_boundary()* will always quote *boundary* if necessary. A *HeaderParseError* is raised if the message object has no *Content-Type* header.

Note that using this method is subtly different than deleting the old *Content-Type* header and adding a new one with the new boundary via *add_header()*, because *set_boundary()* preserves the order of the *Content-Type* header in the list of headers. However, it does *not* preserve any continuation lines which may have been present in the original *Content-Type* header.

get_content_charset(*failobj=None*)

Return the `charset` parameter of the *Content-Type* header, coerced to lower case. If there is no *Content-Type* header, or if that header has no `charset` parameter, *failobj* is returned.

Note that this method differs from *get_charset()* which returns the *Charset* instance for the default encoding of the message body.

get_charsets(*failobj=None*)

Return a list containing the character set names in the message. If the message is a *multipart*, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the `charset` parameter in the *Content-Type* header for the represented subpart. However, if the subpart has no *Content-Type* header, no `charset` parameter, or is not of the *text* main MIME type, then that item in the returned list will be *failobj*.

get_content_disposition()

Return the lowercased value (without parameters) of the message's *Content-Disposition* header if it has one, or `None`. The possible values for this method are *inline*, *attachment* or `None` if the message follows **RFC 2183**.

Added in version 3.5.

walk()

The *walk()* method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use *walk()* as the iterator in a `for` loop; each iteration returns the next subpart.

Here's an example that prints the MIME type of every part of a multipart message structure:

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

`walk` iterates over the subparts of any part where *`is_multipart()`* returns `True`, even though `msg.get_content_maintype() == 'multipart'` may return `False`. We can see this in our example by making use of the `_structure` debug helper function:

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

Here the `message` parts are not `multiparts`, but they do contain subparts. `is_multipart()` returns `True` and `walk` descends into the subparts.

`Message` objects can also optionally contain two instance attributes, which can be used when generating the plain text of a MIME message.

preamble

The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The *preamble* attribute contains this leading extra-armor text for MIME documents. When the *`Parser`* discovers some text after the headers but before the first boundary string, it assigns this text to the message's *preamble* attribute. When the *`Generator`* is writing out the plain text representation of a MIME message, and it finds the message has a *preamble* attribute, it will write this text in the area between the headers and the first boundary. See *`email.parser`* and *`email.generator`* for details.

Note that if the message object has no preamble, the *preamble* attribute will be `None`.

epilogue

The *epilogue* attribute acts the same way as the *preamble* attribute, except that it contains text that appears between the last boundary and the end of the message.

You do not need to set the epilogue to the empty string in order for the *`Generator`* to print a newline at the end of the file.

defects

The *defects* attribute contains a list of all the problems found when parsing this message. See *`email.errors`* for a detailed description of the possible parsing defects.

19.1.10 email.mime: Creating email and MIME objects from scratch

Source code: Lib/email/mime/

This module is part of the legacy (Compat32) email API. Its functionality is partially replaced by the *contentmanager* in the new API, but in certain applications these classes may still be useful, even in non-legacy code.

Ordinarily, you get a message object structure by passing a file or some text to a parser, which parses the text and returns the root message object. However you can also build a complete message structure from scratch, or even individual *Message* objects by hand. In fact, you can also take an existing structure and add new *Message* objects, move them around, etc. This makes a very convenient interface for slicing-and-dicing MIME messages.

You can create a new object structure by creating *Message* instances, adding attachments and all the appropriate headers manually. For MIME messages though, the *email* package provides some convenient subclasses to make things easier.

Here are the classes:

class email.mime.base.**MIMEBase**(*_maintype*, *_subtype*, *, *policy=compat32*, **_params*)

Module: *email.mime.base*

This is the base class for all the MIME-specific subclasses of *Message*. Ordinarily you won't create instances specifically of *MIMEBase*, although you could. *MIMEBase* is provided primarily as a convenient base class for more specific MIME-aware subclasses.

_maintype is the *Content-Type* major type (e.g. *text* or *image*), and *_subtype* is the *Content-Type* minor type (e.g. *plain* or *gif*). *_params* is a parameter key/value dictionary and is passed directly to *Message.add_header*.

If *policy* is specified, (defaults to the *compat32* policy) it will be passed to *Message*.

The *MIMEBase* class always adds a *Content-Type* header (based on *_maintype*, *_subtype*, and *_params*), and a *MIME-Version* header (always set to 1.0).

Changed in version 3.6: Added *policy* keyword-only parameter.

class email.mime.nonmultipart.**MIMENonMultipart**

Module: *email.mime.nonmultipart*

A subclass of *MIMEBase*, this is an intermediate base class for MIME messages that are not *multipart*. The primary purpose of this class is to prevent the use of the *attach()* method, which only makes sense for *multipart* messages. If *attach()* is called, a *MultipartConversionError* exception is raised.

class email.mime.multipart.**MIMEMultipart**(*_subtype='mixed'*, *boundary=None*, *_subparts=None*, *, *policy=compat32*, **_params*)

Module: *email.mime.multipart*

A subclass of *MIMEBase*, this is an intermediate base class for MIME messages that are *multipart*. Optional *_subtype* defaults to *mixed*, but can be used to specify the subtype of the message. A *Content-Type* header of *multipart/_subtype* will be added to the message object. A *MIME-Version* header will also be added.

Optional *boundary* is the multipart boundary string. When None (the default), the boundary is calculated when needed (for example, when the message is serialized).

_subparts is a sequence of initial subparts for the payload. It must be possible to convert this sequence to a list. You can always attach new subparts to the message by using the *Message.attach* method.

Optional *policy* argument defaults to *compat32*.

Additional parameters for the *Content-Type* header are taken from the keyword arguments, or passed into the *_params* argument, which is a keyword dictionary.

Changed in version 3.6: Added *policy* keyword-only parameter.

class email.mime.application.**MIMEApplication**(*_data*, *_subtype='octet-stream'*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

Module: *email.mime.application*

A subclass of *MIMENonMultipart*, the *MIMEApplication* class is used to represent MIME message objects of major type *application*. *_data* contains the bytes for the raw application data. Optional *_subtype* specifies the MIME subtype and defaults to *octet-stream*.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the data for transport. This callable takes one argument, which is the *MIMEApplication* instance. It should use *get_payload()* and *set_payload()* to change the payload to encoded form. It should also add any *Content-Transfer-Encoding* or other headers to the message object as necessary. The default encoding is base64. See the *email.encoders* module for a list of the built-in encoders.

Optional *policy* argument defaults to *compat32*.

_params are passed straight through to the base class constructor.

Changed in version 3.6: Added *policy* keyword-only parameter.

class email.mime.audio.**MIMEAudio**(*_audiodata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

Module: *email.mime.audio*

A subclass of *MIMENonMultipart*, the *MIMEAudio* class is used to create MIME message objects of major type *audio*. *_audiodata* contains the bytes for the raw audio data. If this data can be decoded as au, wav, aiff, or aifc, then the subtype will be automatically included in the *Content-Type* header. Otherwise you can explicitly specify the audio subtype via the *_subtype* argument. If the minor type could not be guessed and *_subtype* was not given, then *TypeError* is raised.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the audio data for transport. This callable takes one argument, which is the *MIMEAudio* instance. It should use *get_payload()* and *set_payload()* to change the payload to encoded form. It should also add any *Content-Transfer-Encoding* or other headers to the message object as necessary. The default encoding is base64. See the *email.encoders* module for a list of the built-in encoders.

Optional *policy* argument defaults to *compat32*.

_params are passed straight through to the base class constructor.

Changed in version 3.6: Added *policy* keyword-only parameter.

class email.mime.image.**MIMEImage**(*_imagedata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

Module: *email.mime.image*

A subclass of *MIMENonMultipart*, the *MIMEImage* class is used to create MIME message objects of major type *image*. *_imagedata* contains the bytes for the raw image data. If this data type can be detected (jpeg, png, gif, tiff, rgb, pbm, pgm, ppm, rast, xbm, bmp, webp, and exr attempted), then the subtype will be automatically included in the *Content-Type* header. Otherwise you can explicitly specify the image subtype via the *_subtype* argument. If the minor type could not be guessed and *_subtype* was not given, then *TypeError* is raised.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the image data for transport. This callable takes one argument, which is the *MIMEImage* instance. It should use *get_payload()* and *set_payload()* to change the payload to encoded form. It should also add any *Content-Transfer-Encoding* or other headers to the message object as necessary. The default encoding is base64. See the *email.encoders* module for a list of the built-in encoders.

Optional *policy* argument defaults to *compat32*.

_params are passed straight through to the *MIMEBase* constructor.

Changed in version 3.6: Added *policy* keyword-only parameter.

class email.mime.message.**MIMEMessage**(*_msg*, *_subtype='rfc822'*, ***, *policy=compat32*)

Module: *email.mime.message*

A subclass of *MIMENonMultipart*, the *MIMEMessage* class is used to create MIME objects of main type *message*. *_msg* is used as the payload, and must be an instance of class *Message* (or a subclass thereof), otherwise a *TypeError* is raised.

Optional *_subtype* sets the subtype of the message; it defaults to *rfc822*.

Optional *policy* argument defaults to *compat32*.

Changed in version 3.6: Added *policy* keyword-only parameter.

class email.mime.text.**MIMEText**(*_text*, *_subtype='plain'*, *_charset=None*, ***, *policy=compat32*)

Module: *email.mime.text*

A subclass of *MIMENonMultipart*, the *MIMEText* class is used to create MIME objects of major type *text*. *_text* is the string for the payload. *_subtype* is the minor type and defaults to *plain*. *_charset* is the character set of the text and is passed as an argument to the *MIMENonMultipart* constructor; it defaults to us-ascii if the string contains only ascii code points, and utf-8 otherwise. The *_charset* parameter accepts either a string or a *Charset* instance.

Unless the *_charset* argument is explicitly set to None, the MIMEText object created will have both a *Content-Type* header with a charset parameter, and a *Content-Transfer-Encoding* header. This means that a subsequent set_payload call will not result in an encoded payload, even if a charset is passed in the set_payload command. You can "reset" this behavior by deleting the Content-Transfer-Encoding header, after which a set_payload call will automatically encode the new payload (and add a new *Content-Transfer-Encoding* header).

Optional *policy* argument defaults to *compat32*.

Changed in version 3.5: *_charset* also accepts *Charset* instances.

Changed in version 3.6: Added *policy* keyword-only parameter.

19.1.11 email.header: Internationalized headers

Source code: Lib/email/header.py

This module is part of the legacy (Compat32) email API. In the current API encoding and decoding of headers is handled transparently by the dictionary-like API of the *EmailMessage* class. In addition to uses in legacy code, this module can be useful in applications that need to completely control the character sets used when encoding headers.

The remaining text in this section is the original documentation of the module.

RFC 2822 is the base standard that describes the format of email messages. It derives from the older **RFC 822** standard which came into widespread use at a time when most email was composed of ASCII characters only. **RFC 2822** is a specification written assuming email contains only 7-bit ASCII characters.

Of course, as email has been deployed worldwide, it has become internationalized, such that language specific character sets can now be used in email messages. The base standard still requires email messages to be transferred using only 7-bit ASCII characters, so a slew of RFCs have been written describing how to encode email containing non-ASCII characters

into RFC 2822-compliant format. These RFCs include RFC 2045, RFC 2046, RFC 2047, and RFC 2231. The *email* package supports these standards in its *email.header* and *email.charset* modules.

If you want to include non-ASCII characters in your email headers, say in the *Subject* or *To* fields, you should use the *Header* class and assign the field in the *Message* object to an instance of *Header* instead of using a string for the header value. Import the *Header* class from the *email.header* module. For example:

```
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> msg.as_string()
'Subject: =?iso-8859-1?q?p=F6stal?=\n\n'
```

Notice here how we wanted the *Subject* field to contain a non-ASCII character? We did this by creating a *Header* instance and passing in the character set that the byte string was encoded in. When the subsequent *Message* instance was flattened, the *Subject* field was properly RFC 2047 encoded. MIME-aware mail readers would show this header using the embedded ISO-8859-1 character.

Here is the *Header* class description:

class email.header.**Header**(*s=None*, *charset=None*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*, *errors='strict'*)

Create a MIME-compliant header that can contain strings in different character sets.

Optional *s* is the initial header value. If `None` (the default), the initial header value is not set. You can later append to the header with *append()* method calls. *s* may be an instance of *bytes* or *str*, but see the *append()* documentation for semantics.

Optional *charset* serves two purposes: it has the same meaning as the *charset* argument to the *append()* method. It also sets the default character set for all subsequent *append()* calls that omit the *charset* argument. If *charset* is not provided in the constructor (the default), the `us-ascii` character set is used both as *s*'s initial charset and as the default for subsequent *append()* calls.

The maximum line length can be specified explicitly via *maxlinelen*. For splitting the first line to a shorter value (to account for the field header which isn't included in *s*, e.g. *Subject*) pass in the name of the field in *header_name*. The default *maxlinelen* is 78, and the default value for *header_name* is `None`, meaning it is not taken into account for the first line of a long, split header.

Optional *continuation_ws* must be RFC 2822-compliant folding whitespace, and is usually either a space or a hard tab character. This character will be prepended to continuation lines. *continuation_ws* defaults to a single space character.

Optional *errors* is passed straight through to the *append()* method.

append(*s*, *charset=None*, *errors='strict'*)

Append the string *s* to the MIME header.

Optional *charset*, if given, should be a *Charset* instance (see *email.charset*) or the name of a character set, which will be converted to a *Charset* instance. A value of `None` (the default) means that the *charset* given in the constructor is used.

s may be an instance of *bytes* or *str*. If it is an instance of *bytes*, then *charset* is the encoding of that byte string, and a *UnicodeError* will be raised if the string cannot be decoded with that character set.

If *s* is an instance of *str*, then *charset* is a hint specifying the character set of the characters in the string.

In either case, when producing an RFC 2822-compliant header using RFC 2047 rules, the string will be encoded using the output codec of the charset. If the string cannot be encoded using the output codec, a UnicodeError will be raised.

Optional *errors* is passed as the errors argument to the decode call if *s* is a byte string.

encode(*splitchars=';, \t'*, *maxlinelen=None*, *linesep='\n'*)

Encode a message header into an RFC-compliant format, possibly wrapping long lines and encapsulating non-ASCII parts in base64 or quoted-printable encodings.

Optional *splitchars* is a string containing characters which should be given extra weight by the splitting algorithm during normal header wrapping. This is in very rough support of **RFC 2822**'s 'higher level syntactic breaks': split points preceded by a splitchar are preferred during line splitting, with the characters preferred in the order in which they appear in the string. Space and tab may be included in the string to indicate whether preference should be given to one over the other as a split point when other split chars do not appear in the line being split. Splitchars does not affect **RFC 2047** encoded lines.

maxlinelen, if given, overrides the instance's value for the maximum line length.

linesep specifies the characters used to separate the lines of the folded header. It defaults to the most useful value for Python application code (`\n`), but `\r\n` can be specified in order to produce headers with RFC-compliant line separators.

Changed in version 3.2: Added the *linesep* argument.

The *Header* class also provides a number of methods to support standard operators and built-in functions.

__str__()

Returns an approximation of the *Header* as a string, using an unlimited line length. All pieces are converted to unicode using the specified encoding and joined together appropriately. Any pieces with a charset of `'unknown-8bit'` are decoded as ASCII using the `'replace'` error handler.

Changed in version 3.2: Added handling for the `'unknown-8bit'` charset.

__eq__(*other*)

This method allows you to compare two *Header* instances for equality.

__ne__(*other*)

This method allows you to compare two *Header* instances for inequality.

The *email.header* module also provides the following convenient functions.

email.header.**decode_header**(*header*)

Decode a message header value without converting the character set. The header value is in *header*.

This function returns a list of `(decoded_string, charset)` pairs containing each of the decoded parts of the header. *charset* is `None` for non-encoded parts of the header, otherwise a lower case string containing the name of the character set specified in the encoded string.

Here's an example:

```
>>> from email.header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[(b'p\xf6stal', 'iso-8859-1')]
```

email.header.**make_header**(*decoded_seq*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*)

Create a *Header* instance from a sequence of pairs as returned by *decode_header()*.

decode_header() takes a header value string and returns a sequence of pairs of the format `(decoded_string, charset)` where *charset* is the name of the character set.

This function takes one of those sequence of pairs and returns a *Header* instance. Optional *maxlinelen*, *header_name*, and *continuation_ws* are as in the *Header* constructor.

19.1.12 email.charset: Representing character sets

Source code: Lib/email/charset.py

This module is part of the legacy (Compat32) email API. In the new API only the aliases table is used.

The remaining text in this section is the original documentation of the module.

This module provides a class *Charset* for representing character sets and character set conversions in email messages, as well as a character set registry and several convenience methods for manipulating this registry. Instances of *Charset* are used in several other modules within the *email* package.

Import this class from the *email.charset* module.

class email.charset.**Charset**(*input_charset=DEFAULT_CHARSET*)

Map character sets to their email properties.

This class provides information about the requirements imposed on email for a specific character set. It also provides convenience routines for converting between character sets, given the availability of the applicable codecs. Given a character set, it will do its best to provide information on how to use that character set in an email message in an RFC-compliant way.

Certain character sets must be encoded with quoted-printable or base64 when used in email headers or bodies. Certain character sets must be converted outright, and are not allowed in email.

Optional *input_charset* is as described below; it is always coerced to lower case. After being alias normalized it is also used as a lookup into the registry of character sets to find out the header encoding, body encoding, and output conversion codec to be used for the character set. For example, if *input_charset* is iso-8859-1, then headers and bodies will be encoded using quoted-printable and no output conversion codec is necessary. If *input_charset* is euc-jp, then headers will be encoded with base64, bodies will not be encoded, but output text will be converted from the euc-jp character set to the iso-2022-jp character set.

Charset instances have the following data attributes:

input_charset

The initial character set specified. Common aliases are converted to their *official* email names (e.g. latin_1 is converted to iso-8859-1). Defaults to 7-bit us-ascii.

header_encoding

If the character set must be encoded before it can be used in an email header, this attribute will be set to charset.QP (for quoted-printable), charset.BASE64 (for base64 encoding), or charset.SHORTEST for the shortest of QP or BASE64 encoding. Otherwise, it will be None.

body_encoding

Same as *header_encoding*, but describes the encoding for the mail message's body, which indeed may be different than the header encoding. charset.SHORTEST is not allowed for *body_encoding*.

output_charset

Some character sets must be converted before they can be used in email headers or bodies. If the *input_charset* is one of them, this attribute will contain the name of the character set output will be converted to. Otherwise, it will be None.

input_codec

The name of the Python codec used to convert the *input_charset* to Unicode. If no conversion codec is necessary, this attribute will be None.

output_codec

The name of the Python codec used to convert Unicode to the *output_charset*. If no conversion codec is necessary, this attribute will have the same value as the *input_codec*.

`Charset` instances also have the following methods:

get_body_encoding()

Return the content transfer encoding used for body encoding.

This is either the string `quoted-printable` or `base64` depending on the encoding used, or it is a function, in which case you should call the function with a single argument, the Message object being encoded. The function should then set the `Content-Transfer-Encoding` header itself to whatever is appropriate.

Returns the string `quoted-printable` if *body_encoding* is `QP`, returns the string `base64` if *body_encoding* is `BASE64`, and returns the string `7bit` otherwise.

get_output_charset()

Return the output character set.

This is the *output_charset* attribute if that is not `None`, otherwise it is *input_charset*.

header_encode(*string*)

Header-encode the string *string*.

The type of encoding (base64 or quoted-printable) will be based on the *header_encoding* attribute.

header_encode_lines(*string*, *maxlengths*)

Header-encode a *string* by converting it first to bytes.

This is similar to `header_encode()` except that the string is fit into maximum line lengths as given by the argument *maxlengths*, which must be an iterator: each element returned from this iterator will provide the next maximum line length.

body_encode(*string*)

Body-encode the string *string*.

The type of encoding (base64 or quoted-printable) will be based on the *body_encoding* attribute.

The `Charset` class also provides a number of methods to support standard operations and built-in functions.

__str__()

Returns *input_charset* as a string coerced to lower case. `__repr__()` is an alias for `__str__()`.

__eq__(*other*)

This method allows you to compare two `Charset` instances for equality.

__ne__(*other*)

This method allows you to compare two `Charset` instances for inequality.

The `email.charset` module also provides the following functions for adding new entries to the global character set, alias, and codec registries:

email.charset.**add_charset**(*charset*, *header_enc=None*, *body_enc=None*, *output_charset=None*)

Add character properties to the global registry.

charset is the input character set, and must be the canonical name of a character set.

Optional *header_enc* and *body_enc* is either `charset.QP` for quoted-printable, `charset.BASE64` for base64 encoding, `charset.SHORTEST` for the shortest of quoted-printable or base64 encoding, or `None` for no encoding. `SHORTEST` is only valid for *header_enc*. The default is `None` for no encoding.

Optional *output_charset* is the character set that the output should be in. Conversions will proceed from input charset, to Unicode, to the output charset when the method `Charset.convert()` is called. The default is to output in the same character set as the input.

Both *input_charset* and *output_charset* must have Unicode codec entries in the module's character set-to-codec mapping; use *add_codec()* to add codecs the module does not know about. See the *codecs* module's documentation for more information.

The global character set registry is kept in the module global dictionary `CHARSETS`.

`email.charset.`**`add_alias`**(*alias*, *canonical*)

Add a character set alias. *alias* is the alias name, e.g. `latin-1`. *canonical* is the character set's canonical name, e.g. `iso-8859-1`.

The global charset alias registry is kept in the module global dictionary `ALIASES`.

`email.charset.`**`add_codec`**(*charset*, *codecname*)

Add a codec that map characters in the given character set to and from Unicode.

charset is the canonical name of a character set. *codecname* is the name of a Python codec, as appropriate for the second argument to the *str*'s *encode()* method.

19.1.13 `email.encoders`: Encoders

Source code: Lib/email/encoders.py

This module is part of the legacy (`Compat32`) email API. In the new API the functionality is provided by the *cte* parameter of the *set_content()* method.

This module is deprecated in Python 3. The functions provided here should not be called explicitly since the *MIMEText* class sets the content type and CTE header using the *_subtype* and *_charset* values passed during the instantiation of that class.

The remaining text in this section is the original documentation of the module.

When creating *Message* objects from scratch, you often need to encode the payloads for transport through compliant mail servers. This is especially true for *image/** and *text/** type messages containing binary data.

The *email* package provides some convenient encoders in its *encoders* module. These encoders are actually used by the *MIMEAudio* and *MIMEImage* class constructors to provide default encodings. All encoder functions take exactly one argument, the message object to encode. They usually extract the payload, encode it, and reset the payload to this newly encoded value. They should also set the *Content-Transfer-Encoding* header as appropriate.

Note that these functions are not meaningful for a multipart message. They must be applied to individual subparts instead, and will raise a *TypeError* if passed a message whose type is multipart.

Here are the encoding functions provided:

`email.encoders.`**`encode_quopri`**(*msg*)

Encodes the payload into quoted-printable form and sets the *Content-Transfer-Encoding* header to `quoted-printable`[1]. This is a good encoding to use when most of your payload is normal printable data, but contains a few unprintable characters.

[1] Note that encoding with *encode_quopri()* also encodes all tabs and space characters in the data.

email.encoders.**encode_base64**(*msg*)

Encodes the payload into base64 form and sets the *Content-Transfer-Encoding* header to base64. This is a good encoding to use when most of your payload is unprintable data since it is a more compact form than quoted-printable. The drawback of base64 encoding is that it renders the text non-human readable.

email.encoders.**encode_7or8bit**(*msg*)

This doesn't actually modify the message's payload, but it does set the *Content-Transfer-Encoding* header to either 7bit or 8bit as appropriate, based on the payload data.

email.encoders.**encode_noop**(*msg*)

This does nothing; it doesn't even set the *Content-Transfer-Encoding* header.

19.1.14 email.utils: Miscellaneous utilities

Source code: Lib/email/utils.py

There are a couple of useful utilities provided in the *email.utils* module:

email.utils.**localtime**(*dt=None*)

Return local time as an aware datetime object. If called without arguments, return current time. Otherwise *dt* argument should be a *datetime* instance, and it is converted to the local time zone according to the system time zone database. If *dt* is naive (that is, dt.tzinfo is None), it is assumed to be in local time. The *isdst* parameter is ignored.

Added in version 3.3.

Deprecated since version 3.12, will be removed in version 3.14: The *isdst* parameter.

email.utils.**make_msgid**(*idstring=None*, *domain=None*)

Returns a string suitable for an **RFC 2822**-compliant *Message-ID* header. Optional *idstring* if given, is a string used to strengthen the uniqueness of the message id. Optional *domain* if given provides the portion of the msgid after the '@'. The default is the local hostname. It is not normally necessary to override this default, but may be useful certain cases, such as a constructing distributed system that uses a consistent domain name across multiple hosts.

Changed in version 3.2: Added the *domain* keyword.

The remaining functions are part of the legacy (Compat32) email API. There is no need to directly use these with the new API, since the parsing and formatting they provide is done automatically by the header parsing machinery of the new API.

email.utils.**quote**(*str*)

Return a new string with backslashes in *str* replaced by two backslashes, and double quotes replaced by backslash-double quote.

email.utils.**unquote**(*str*)

Return a new string which is an *unquoted* version of *str*. If *str* ends and begins with double quotes, they are stripped off. Likewise if *str* ends and begins with angle brackets, they are stripped off.

email.utils.**parseaddr**(*address*, *, *strict=True*)

Parse address – which should be the value of some address-containing field such as *To* or *Cc* – into its constituent *realname* and *email address* parts. Returns a tuple of that information, unless the parse fails, in which case a 2-tuple of ('', '') is returned.

If *strict* is true, use a strict parser which rejects malformed inputs.

Changed in version 3.13: Add *strict* optional parameter and reject malformed inputs by default.

email.utils.**formataddr**(*pair*, *charset='utf-8'*)

The inverse of *parseaddr()*, this takes a 2-tuple of the form `(realname, email_address)` and returns the string value suitable for a *To* or *Cc* header. If the first element of *pair* is false, then the second element is returned unmodified.

Optional *charset* is the character set that will be used in the **RFC 2047** encoding of the `realname` if the `realname` contains non-ASCII characters. Can be an instance of *str* or a *Charset*. Defaults to `utf-8`.

Changed in version 3.3: Added the *charset* option.

email.utils.**getaddresses**(*fieldvalues*, *, *strict=True*)

This method returns a list of 2-tuples of the form returned by `parseaddr()`. *fieldvalues* is a sequence of header field values as might be returned by *Message.get_all*.

If *strict* is true, use a strict parser which rejects malformed inputs.

Here's a simple example that gets all the recipients of a message:

```
from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

Changed in version 3.13: Add *strict* optional parameter and reject malformed inputs by default.

email.utils.**parsedate**(*date*)

Attempts to parse a date according to the rules in **RFC 2822**. however, some mailers don't follow that format as specified, so *parsedate()* tries to guess correctly in such cases. *date* is a string containing an **RFC 2822** date, such as `"Mon, 20 Nov 1995 19:12:08 -0500"`. If it succeeds in parsing the date, *parsedate()* returns a 9-tuple that can be passed directly to *time.mktime()*; otherwise `None` will be returned. Note that indexes 6, 7, and 8 of the result tuple are not usable.

email.utils.**parsedate_tz**(*date*)

Performs the same function as *parsedate()*, but returns either `None` or a 10-tuple; the first 9 elements make up a tuple that can be passed directly to *time.mktime()*, and the tenth is the offset of the date's timezone from UTC (which is the official term for Greenwich Mean Time)[1]. If the input string has no timezone, the last element of the tuple returned is `0`, which represents UTC. Note that indexes 6, 7, and 8 of the result tuple are not usable.

email.utils.**parsedate_to_datetime**(*date*)

The inverse of *format_datetime()*. Performs the same function as *parsedate()*, but on success returns a *datetime*; otherwise `ValueError` is raised if *date* contains an invalid value such as an hour greater than 23 or a timezone offset not between -24 and 24 hours. If the input date has a timezone of `-0000`, the `datetime` will be a naive `datetime`, and if the date is conforming to the RFCs it will represent a time in UTC but with no indication of the actual source timezone of the message the date comes from. If the input date has any other valid timezone offset, the `datetime` will be an aware `datetime` with the corresponding a *timezone tzinfo*.

Added in version 3.3.

email.utils.**mktime_tz**(*tuple*)

Turn a 10-tuple as returned by *parsedate_tz()* into a UTC timestamp (seconds since the Epoch). If the timezone item in the tuple is `None`, assume local time.

[1] Note that the sign of the timezone offset is the opposite of the sign of the `time.timezone` variable for the same timezone; the latter variable follows the POSIX standard while this module follows **RFC 2822**.

email.utils.**formatdate**(*timeval=None*, *localtime=False*, *usegmt=False*)

Returns a date string as per **RFC 2822**, e.g.:

```
Fri, 09 Nov 2001 01:08:47 -0000
```

Optional *timeval* if given is a floating-point time value as accepted by *time.gmtime()* and *time.localtime()*, otherwise the current time is used.

Optional *localtime* is a flag that when `True`, interprets *timeval*, and returns a date relative to the local timezone instead of UTC, properly taking daylight savings time into account. The default is `False` meaning UTC is used.

Optional *usegmt* is a flag that when `True`, outputs a date string with the timezone as an ascii string `GMT`, rather than a numeric `-0000`. This is needed for some protocols (such as HTTP). This only applies when *localtime* is `False`. The default is `False`.

email.utils.**format_datetime**(*dt*, *usegmt=False*)

Like `formatdate`, but the input is a *datetime* instance. If it is a naive datetime, it is assumed to be "UTC with no information about the source timezone", and the conventional `-0000` is used for the timezone. If it is an aware `datetime`, then the numeric timezone offset is used. If it is an aware timezone with offset zero, then *usegmt* may be set to `True`, in which case the string `GMT` is used instead of the numeric timezone offset. This provides a way to generate standards conformant HTTP date headers.

Added in version 3.3.

email.utils.**decode_rfc2231**(*s*)

Decode the string *s* according to **RFC 2231**.

email.utils.**encode_rfc2231**(*s*, *charset=None*, *language=None*)

Encode the string *s* according to **RFC 2231**. Optional *charset* and *language*, if given is the character set name and language name to use. If neither is given, *s* is returned as-is. If *charset* is given but *language* is not, the string is encoded using the empty string for *language*.

email.utils.**collapse_rfc2231_value**(*value*, *errors='replace'*, *fallback_charset='us-ascii'*)

When a header parameter is encoded in **RFC 2231** format, *Message.get_param* may return a 3-tuple containing the character set, language, and value. *collapse_rfc2231_value()* turns this into a unicode string. Optional *errors* is passed to the *errors* argument of *str*'s *encode()* method; it defaults to `'replace'`. Optional *fallback_charset* specifies the character set to use if the one in the **RFC 2231** header is not known by Python; it defaults to `'us-ascii'`.

For convenience, if the *value* passed to *collapse_rfc2231_value()* is not a tuple, it should be a string and it is returned unquoted.

email.utils.**decode_params**(*params*)

Decode parameters list according to **RFC 2231**. *params* is a sequence of 2-tuples containing elements of the form `(content-type, string-value)`.

19.1.15 `email.iterators`: Iterators

Source code: Lib/email/iterators.py

Iterating over a message object tree is fairly easy with the *Message.walk* method. The *email.iterators* module provides some useful higher level iterations over message object trees.

email.iterators.**body_line_iterator**(*msg*, *decode=False*)

This iterates over all the payloads in all the subparts of *msg*, returning the string payloads line-by-line. It skips over all the subpart headers, and it skips over any subpart with a payload that isn't a Python string. This is somewhat equivalent to reading the flat text representation of the message from a file using `readline()`, skipping over all the intervening headers.

Optional *decode* is passed through to `Message.get_payload`.

email.iterators.**typed_subpart_iterator**(*msg*, *maintype='text'*, *subtype=None*)

This iterates over all the subparts of *msg*, returning only those subparts that match the MIME type specified by *maintype* and *subtype*.

Note that *subtype* is optional; if omitted, then subpart MIME type matching is done only with the main type. *maintype* is optional too; it defaults to `text`.

Thus, by default `typed_subpart_iterator()` returns each subpart that has a MIME type of `text/*`.

The following function has been added as a useful debugging tool. It should *not* be considered part of the supported public interface for the package.

email.iterators.**_structure**(*msg*, *fp=None*, *level=0*, *include_default=False*)

Prints an indented representation of the content types of the message object structure. For example:

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
    text/plain
    text/plain
    multipart/digest
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
    text/plain
```

Optional *fp* is a file-like object to print the output to. It must be suitable for Python's `print()` function. *level* is used internally. *include_default*, if true, prints the default type as well.

See also

Module *smtplib*
SMTP (Simple Mail Transport Protocol) client

Module *poplib*
POP (Post Office Protocol) client

Module *imaplib*
IMAP (Internet Message Access Protocol) client

Module *mailbox*
Tools for creating, reading, and managing collections of messages on disk using a variety standard formats.

19.2 `json` — JSON encoder and decoder

Source code: Lib/json/__init__.py

JSON (JavaScript Object Notation), specified by **RFC 7159** (which obsoletes **RFC 4627**) and by ECMA-404, is a lightweight data interchange format inspired by JavaScript object literal syntax (although it is not a strict subset of JavaScript[1]).

> **Warning**
>
> Be cautious when parsing JSON data from untrusted sources. A malicious JSON string may cause the decoder to consume considerable CPU and memory resources. Limiting the size of data to be parsed is recommended.

json exposes an API familiar to users of the standard library *marshal* and *pickle* modules.

Encoding basic Python object hierarchies:

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

Compact encoding:

```
>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
```

Pretty printing:

```
>>> import json
>>> print(json.dumps({'6': 7, '4': 5}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

Specializing JSON object encoding:

[1] As noted in the errata for RFC 7159, JSON permits literal U+2028 (LINE SEPARATOR) and U+2029 (PARAGRAPH SEPARATOR) characters in strings, whereas JavaScript (as of ECMAScript Edition 5.1) does not.

```
>>> import json
>>> def custom_json(obj):
...     if isinstance(obj, complex):
...         return {'__complex__': True, 'real': obj.real, 'imag': obj.imag}
...     raise TypeError(f'Cannot serialize object of {type(obj)}')
...
>>> json.dumps(1 + 2j, default=custom_json)
'{"__complex__": true, "real": 1.0, "imag": 2.0}'
```

Decoding JSON:

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
>>> from io import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
['streaming API']
```

Specializing JSON object decoding:

```
>>> import json
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...     object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal)
Decimal('1.1')
```

Extending *JSONEncoder*:

```
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # Let the base class default method raise the TypeError
...         return super().default(obj)
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[2.0', ', 1.0', ']']
```

Using *json.tool* from the shell to validate and pretty-print:

```
$ echo '{"json":"obj"}' | python -m json.tool
{
```

(continues on next page)

(continued from previous page)

```
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

See *Command Line Interface* for detailed documentation.

Note

JSON is a subset of YAML 1.2. The JSON produced by this module's default settings (in particular, the default *separators* value) is also a subset of YAML 1.0 and 1.1. This module can thus also be used as a YAML serializer.

Note

This module's encoders and decoders preserve input and output order by default. Order is only lost if the underlying containers are unordered.

19.2.1 Basic Usage

`json.`**`dump`**(*obj*, *fp*, ***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Serialize *obj* as a JSON formatted stream to *fp* (a `.write()`-supporting *file-like object*) using this *conversion table*.

If *skipkeys* is true (default: `False`), then dict keys that are not of a basic type (*str*, *int*, *float*, *bool*, `None`) will be skipped instead of raising a *TypeError*.

The *json* module always produces *str* objects, not *bytes* objects. Therefore, `fp.write()` must support *str* input.

If *ensure_ascii* is true (the default), the output is guaranteed to have all incoming non-ASCII characters escaped. If *ensure_ascii* is false, these characters will be output as-is.

If *check_circular* is false (default: `True`), then the circular reference check for container types will be skipped and a circular reference will result in a *RecursionError* (or worse).

If *allow_nan* is false (default: `True`), then it will be a *ValueError* to serialize out of range *float* values (`nan`, `inf`, `-inf`) in strict compliance of the JSON specification. If *allow_nan* is true, their JavaScript equivalents (`NaN`, `Infinity`, `-Infinity`) will be used.

If *indent* is a non-negative integer or string, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0, negative, or `""` will only insert newlines. `None` (the default) selects the most compact representation. Using a positive integer indent indents that many spaces per level. If *indent* is a string (such as `"\t"`), that string is used to indent each level.

Changed in version 3.2: Allow strings for *indent* in addition to integers.

If specified, *separators* should be an `(item_separator, key_separator)` tuple. The default is `(', ', ': ')` if *indent* is `None` and `(',', ': ')` otherwise. To get the most compact JSON representation, you should specify `(',', ':')` to eliminate whitespace.

Changed in version 3.4: Use `(',', ': ')` as default if *indent* is not `None`.

If specified, *default* should be a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`. If not specified, `TypeError` is raised.

If *sort_keys* is true (default: `False`), then the output of dictionaries will be sorted by key.

To use a custom `JSONEncoder` subclass (e.g. one that overrides the `default()` method to serialize additional types), specify it with the *cls* kwarg; otherwise `JSONEncoder` is used.

Changed in version 3.6: All optional parameters are now *keyword-only*.

> **Note**
>
> Unlike `pickle` and `marshal`, JSON is not a framed protocol, so trying to serialize multiple objects with repeated calls to `dump()` using the same *fp* will result in an invalid JSON file.

`json.`**`dumps`**(*obj*, *, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Serialize *obj* to a JSON formatted `str` using this *conversion table*. The arguments have the same meaning as in `dump()`.

> **Note**
>
> Keys in key/value pairs of JSON are always of the type `str`. When a dictionary is converted into JSON, all the keys of the dictionary are coerced to strings. As a result of this, if a dictionary is converted into JSON and then back into a dictionary, the dictionary may not equal the original one. That is, `loads(dumps(x)) != x` if x has non-string keys.

`json.`**`load`**(*fp*, *, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

Deserialize *fp* (a `.read()`-supporting *text file* or *binary file* containing a JSON document) to a Python object using this *conversion table*.

object_hook is an optional function that will be called with the result of any object literal decoded (a `dict`). The return value of *object_hook* will be used instead of the `dict`. This feature can be used to implement custom decoders (e.g. JSON-RPC class hinting).

object_pairs_hook is an optional function that will be called with the result of any object literal decoded with an ordered list of pairs. The return value of *object_pairs_hook* will be used instead of the `dict`. This feature can be used to implement custom decoders. If *object_hook* is also defined, the *object_pairs_hook* takes priority.

Changed in version 3.1: Added support for *object_pairs_hook*.

parse_float, if specified, will be called with the string of every JSON float to be decoded. By default, this is equivalent to `float(num_str)`. This can be used to use another datatype or parser for JSON floats (e.g. `decimal.Decimal`).

parse_int, if specified, will be called with the string of every JSON int to be decoded. By default, this is equivalent to `int(num_str)`. This can be used to use another datatype or parser for JSON integers (e.g. `float`).

Changed in version 3.11: The default *parse_int* of `int()` now limits the maximum length of the integer string via the interpreter's *integer string conversion length limitation* to help avoid denial of service attacks.

parse_constant, if specified, will be called with one of the following strings: `'-Infinity'`, `'Infinity'`, `'NaN'`. This can be used to raise an exception if invalid JSON numbers are encountered.

Changed in version 3.1: *parse_constant* doesn't get called on 'null', 'true', 'false' anymore.

To use a custom *JSONDecoder* subclass, specify it with the `cls` kwarg; otherwise *JSONDecoder* is used. Additional keyword arguments will be passed to the constructor of the class.

If the data being deserialized is not a valid JSON document, a *JSONDecodeError* will be raised.

Changed in version 3.6: All optional parameters are now *keyword-only*.

Changed in version 3.6: *fp* can now be a *binary file*. The input encoding should be UTF-8, UTF-16 or UTF-32.

`json.`**`loads`**(*s*, ***, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

Deserialize *s* (a *str*, *bytes* or *bytearray* instance containing a JSON document) to a Python object using this *conversion table*.

The other arguments have the same meaning as in *load()*.

If the data being deserialized is not a valid JSON document, a *JSONDecodeError* will be raised.

Changed in version 3.6: *s* can now be of type *bytes* or *bytearray*. The input encoding should be UTF-8, UTF-16 or UTF-32.

Changed in version 3.9: The keyword argument *encoding* has been removed.

19.2.2 Encoders and Decoders

class `json.`**`JSONDecoder`**(***, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

Simple JSON decoder.

Performs the following translations in decoding by default:

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

It also understands `NaN`, `Infinity`, and `-Infinity` as their corresponding `float` values, which is outside the JSON spec.

object_hook, if specified, will be called with the result of every JSON object decoded and its return value will be used in place of the given *dict*. This can be used to provide custom deserializations (e.g. to support JSON-RPC class hinting).

object_pairs_hook, if specified will be called with the result of every JSON object decoded with an ordered list of pairs. The return value of *object_pairs_hook* will be used instead of the *dict*. This feature can be used to implement custom decoders. If *object_hook* is also defined, the *object_pairs_hook* takes priority.

Changed in version 3.1: Added support for *object_pairs_hook*.

parse_float, if specified, will be called with the string of every JSON float to be decoded. By default, this is equivalent to `float(num_str)`. This can be used to use another datatype or parser for JSON floats (e.g. *decimal.Decimal*).

parse_int, if specified, will be called with the string of every JSON int to be decoded. By default, this is equivalent to `int(num_str)`. This can be used to use another datatype or parser for JSON integers (e.g. *float*).

parse_constant, if specified, will be called with one of the following strings: `'-Infinity'`, `'Infinity'`, `'NaN'`. This can be used to raise an exception if invalid JSON numbers are encountered.

If *strict* is false (`True` is the default), then control characters will be allowed inside strings. Control characters in this context are those with character codes in the 0–31 range, including `'\t'` (tab), `'\n'`, `'\r'` and `'\0'`.

If the data being deserialized is not a valid JSON document, a *JSONDecodeError* will be raised.

Changed in version 3.6: All parameters are now *keyword-only*.

decode(*s*)

Return the Python representation of *s* (a *str* instance containing a JSON document).

JSONDecodeError will be raised if the given JSON document is not valid.

raw_decode(*s*)

Decode a JSON document from *s* (a *str* beginning with a JSON document) and return a 2-tuple of the Python representation and the index in *s* where the document ended.

This can be used to decode a JSON document from a string that may have extraneous data at the end.

class json.**JSONEncoder**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Extensible JSON encoder for Python data structures.

Supports the following objects and types by default:

Python	JSON
dict	object
list, tuple	array
str	string
int, float, int- & float-derived Enums	number
True	true
False	false
None	null

Changed in version 3.4: Added support for int- and float-derived Enum classes.

To extend this to recognize other objects, subclass and implement a *default()* method with another method that returns a serializable object for `o` if possible, otherwise it should call the superclass implementation (to raise *TypeError*).

If *skipkeys* is false (the default), a *TypeError* will be raised when trying to encode keys that are not *str*, *int*, *float* or `None`. If *skipkeys* is true, such items are simply skipped.

If *ensure_ascii* is true (the default), the output is guaranteed to have all incoming non-ASCII characters escaped. If *ensure_ascii* is false, these characters will be output as-is.

If *check_circular* is true (the default), then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause a *RecursionError*). Otherwise, no such check takes place.

If *allow_nan* is true (the default), then `NaN`, `Infinity`, and `-Infinity` will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a *ValueError* to encode such floats.

If *sort_keys* is true (default: `False`), then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If *indent* is a non-negative integer or string, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0, negative, or `""` will only insert newlines. `None` (the default) selects the most compact representation. Using a positive integer indent indents that many spaces per level. If *indent* is a string (such as `"\t"`), that string is used to indent each level.

Changed in version 3.2: Allow strings for *indent* in addition to integers.

If specified, *separators* should be an `(item_separator, key_separator)` tuple. The default is `(', ', ': ')` if *indent* is `None` and `(',', ': ')` otherwise. To get the most compact JSON representation, you should specify `(',', ':')` to eliminate whitespace.

Changed in version 3.4: Use `(',', ': ')` as default if *indent* is not `None`.

If specified, *default* should be a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a *TypeError*. If not specified, *TypeError* is raised.

Changed in version 3.6: All parameters are now *keyword-only*.

default(*o*)

Implement this method in a subclass such that it returns a serializable object for *o*, or calls the base implementation (to raise a *TypeError*).

For example, to support arbitrary iterators, you could implement *default()* like this:

```
def default(self, o):
   try:
       iterable = iter(o)
   except TypeError:
       pass
   else:
       return list(iterable)
   # Let the base class default method raise the TypeError
   return super().default(o)
```

encode(*o*)

Return a JSON string representation of a Python data structure, *o*. For example:

```
>>> json.JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*)

Encode the given object, *o*, and yield each string representation as available. For example:

```
for chunk in json.JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

19.2.3 Exceptions

exception json.**JSONDecodeError**(*msg*, *doc*, *pos*)

Subclass of *ValueError* with the following additional attributes:

msg

The unformatted error message.

doc

The JSON document being parsed.

pos

The start index of *doc* where parsing failed.

lineno

The line corresponding to *pos*.

colno

The column corresponding to *pos*.

Added in version 3.5.

19.2.4 Standard Compliance and Interoperability

The JSON format is specified by **RFC 7159** and by ECMA-404. This section details this module's level of compliance with the RFC. For simplicity, *JSONEncoder* and *JSONDecoder* subclasses, and parameters other than those explicitly mentioned, are not considered.

This module does not comply with the RFC in a strict fashion, implementing some extensions that are valid JavaScript but not valid JSON. In particular:

- Infinite and NaN number values are accepted and output;
- Repeated names within an object are accepted, and only the value of the last name-value pair is used.

Since the RFC permits RFC-compliant parsers to accept input texts that are not RFC-compliant, this module's deserializer is technically RFC-compliant under default settings.

Character Encodings

The RFC requires that JSON be represented using either UTF-8, UTF-16, or UTF-32, with UTF-8 being the recommended default for maximum interoperability.

As permitted, though not required, by the RFC, this module's serializer sets *ensure_ascii*=*True* by default, thus escaping the output so that the resulting strings only contain ASCII characters.

Other than the *ensure_ascii* parameter, this module is defined strictly in terms of conversion between Python objects and *Unicode strings*, and thus does not otherwise directly address the issue of character encodings.

The RFC prohibits adding a byte order mark (BOM) to the start of a JSON text, and this module's serializer does not add a BOM to its output. The RFC permits, but does not require, JSON deserializers to ignore an initial BOM in their input. This module's deserializer raises a *ValueError* when an initial BOM is present.

The RFC does not explicitly forbid JSON strings which contain byte sequences that don't correspond to valid Unicode characters (e.g. unpaired UTF-16 surrogates), but it does note that they may cause interoperability problems. By default, this module accepts and outputs (when present in the original *str*) code points for such sequences.

Infinite and NaN Number Values

The RFC does not permit the representation of infinite or NaN number values. Despite that, by default, this module accepts and outputs `Infinity`, `-Infinity`, and `NaN` as if they were valid JSON number literal values:

```
>>> # Neither of these calls raises an exception, but the results are not valid JSON
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
```

In the serializer, the *allow_nan* parameter can be used to alter this behavior. In the deserializer, the *parse_constant* parameter can be used to alter this behavior.

Repeated Names Within an Object

The RFC specifies that the names within a JSON object should be unique, but does not mandate how repeated names in JSON objects should be handled. By default, this module does not raise an exception; instead, it ignores all but the last name-value pair for a given name:

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{'x': 3}
```

The *object_pairs_hook* parameter can be used to alter this behavior.

Top-level Non-Object, Non-Array Values

The old version of JSON specified by the obsolete **RFC 4627** required that the top-level value of a JSON text must be either a JSON object or array (Python *dict* or *list*), and could not be a JSON null, boolean, number, or string value. **RFC 7159** removed that restriction, and this module does not and has never implemented that restriction in either its serializer or its deserializer.

Regardless, for maximum interoperability, you may wish to voluntarily adhere to the restriction yourself.

Implementation Limitations

Some JSON deserializer implementations may set limits on:

- the size of accepted JSON texts
- the maximum level of nesting of JSON objects and arrays
- the range and precision of JSON numbers
- the content and maximum length of JSON strings

This module does not impose any such limits beyond those of the relevant Python datatypes themselves or the Python interpreter itself.

When serializing to JSON, beware any such limitations in applications that may consume your JSON. In particular, it is common for JSON numbers to be deserialized into IEEE 754 double precision numbers and thus subject to that representation's range and precision limitations. This is especially relevant when serializing Python *int* values of extremely large magnitude, or when serializing instances of "exotic" numerical types such as *decimal.Decimal*.

19.2.5 Command Line Interface

Source code: Lib/json/tool.py

The *json.tool* module provides a simple command line interface to validate and pretty-print JSON objects.

If the optional `infile` and `outfile` arguments are not specified, *sys.stdin* and *sys.stdout* will be used respectively:

```
$ echo '{"json": "obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

Changed in version 3.5: The output is now in the same order as the input. Use the *--sort-keys* option to sort the output of dictionaries alphabetically by key.

Command line options

infile

The JSON file to be validated or pretty-printed:

```
$ python -m json.tool mp_films.json
[
    {
        "title": "And Now for Something Completely Different",
        "year": 1971
    },
    {
        "title": "Monty Python and the Holy Grail",
        "year": 1975
    }
]
```

If *infile* is not specified, read from *sys.stdin*.

outfile

Write the output of the *infile* to the given *outfile*. Otherwise, write it to *sys.stdout*.

--sort-keys

Sort the output of dictionaries alphabetically by key.

Added in version 3.5.

--no-ensure-ascii

Disable escaping of non-ascii characters, see *json.dumps()* for more information.

Added in version 3.9.

--json-lines

Parse every input line as separate JSON object.

Added in version 3.8.

--indent, --tab, --no-indent, --compact

Mutually exclusive options for whitespace control.

Added in version 3.9.

-h, --help

Show the help message.

19.3 mailbox — Manipulate mailboxes in various formats

Source code: Lib/mailbox.py

This module defines two classes, *Mailbox* and *Message*, for accessing and manipulating on-disk mailboxes and the messages they contain. Mailbox offers a dictionary-like mapping from keys to messages. Message extends the *email.message* module's *Message* class with format-specific state and behavior. Supported mailbox formats are Maildir, mbox, MH, Babyl, and MMDF.

See also

Module *email*
Represent and manipulate messages.

19.3.1 Mailbox objects

class mailbox.**Mailbox**

A mailbox, which may be inspected and modified.

The Mailbox class defines an interface and is not intended to be instantiated. Instead, format-specific subclasses should inherit from Mailbox and your code should instantiate a particular subclass.

The Mailbox interface is dictionary-like, with small keys corresponding to messages. Keys are issued by the Mailbox instance with which they will be used and are only meaningful to that Mailbox instance. A key continues to identify a message even if the corresponding message is modified, such as by replacing it with another message.

Messages may be added to a Mailbox instance using the set-like method *add()* and removed using a del statement or the set-like methods *remove()* and *discard()*.

Mailbox interface semantics differ from dictionary semantics in some noteworthy ways. Each time a message is requested, a new representation (typically a *Message* instance) is generated based upon the current state of the mailbox. Similarly, when a message is added to a Mailbox instance, the provided message representation's contents are copied. In neither case is a reference to the message representation kept by the Mailbox instance.

The default Mailbox *iterator* iterates over message representations, not keys as the default *dictionary* iterator does. Moreover, modification of a mailbox during iteration is safe and well-defined. Messages added to the mailbox after an iterator is created will not be seen by the iterator. Messages removed from the mailbox before the iterator

yields them will be silently skipped, though using a key from an iterator may result in a `KeyError` exception if the corresponding message is subsequently removed.

> **Warning**
>
> Be very cautious when modifying mailboxes that might be simultaneously changed by some other process. The safest mailbox format to use for such tasks is `Maildir`; try to avoid using single-file formats such as `mbox` for concurrent writing. If you're modifying a mailbox, you *must* lock it by calling the `lock()` and `unlock()` methods *before* reading any messages in the file or making any changes by adding or deleting a message. Failing to lock the mailbox runs the risk of losing messages or corrupting the entire mailbox.

`Mailbox` instances have the following methods:

add(*message*)

Add *message* to the mailbox and return the key that has been assigned to it.

Parameter *message* may be a `Message` instance, an `email.message.Message` instance, a string, a byte string, or a file-like object (which should be open in binary mode). If *message* is an instance of the appropriate format-specific `Message` subclass (e.g., if it's an `mboxMessage` instance and this is an `mbox` instance), its format-specific information is used. Otherwise, reasonable defaults for format-specific information are used.

Changed in version 3.2: Support for binary input was added.

remove(*key*)

__delitem__(*key*)

discard(*key*)

Delete the message corresponding to *key* from the mailbox.

If no such message exists, a `KeyError` exception is raised if the method was called as `remove()` or `__delitem__()` but no exception is raised if the method was called as `discard()`. The behavior of `discard()` may be preferred if the underlying mailbox format supports concurrent modification by other processes.

__setitem__(*key*, *message*)

Replace the message corresponding to *key* with *message*. Raise a `KeyError` exception if no message already corresponds to *key*.

As with `add()`, parameter *message* may be a `Message` instance, an `email.message.Message` instance, a string, a byte string, or a file-like object (which should be open in binary mode). If *message* is an instance of the appropriate format-specific `Message` subclass (e.g., if it's an `mboxMessage` instance and this is an `mbox` instance), its format-specific information is used. Otherwise, the format-specific information of the message that currently corresponds to *key* is left unchanged.

iterkeys()

Return an *iterator* over all keys

keys()

The same as `iterkeys()`, except that a `list` is returned rather than an *iterator*

itervalues()

__iter__()

Return an *iterator* over representations of all messages. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

> **Note**
>
> The behavior of `__iter__()` is unlike that of dictionaries, which iterate over keys.

values()

The same as `itervalues()`, except that a `list` is returned rather than an *iterator*

iteritems()

Return an *iterator* over (*key*, *message*) pairs, where *key* is a key and *message* is a message representation. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

items()

The same as `iteritems()`, except that a `list` of pairs is returned rather than an *iterator* of pairs.

get(*key*, *default=None*)

__getitem__(*key*)

Return a representation of the message corresponding to *key*. If no such message exists, *default* is returned if the method was called as `get()` and a `KeyError` exception is raised if the method was called as `__getitem__()`. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

get_message(*key*)

Return a representation of the message corresponding to *key* as an instance of the appropriate format-specific `Message` subclass, or raise a `KeyError` exception if no such message exists.

get_bytes(*key*)

Return a byte representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists.

Added in version 3.2.

get_string(*key*)

Return a string representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists. The message is processed through `email.message.Message` to convert it to a 7bit clean representation.

get_file(*key*)

Return a *file-like* representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists. The file-like object behaves as if open in binary mode. This file should be closed once it is no longer needed.

Changed in version 3.2: The file object really is a *binary file*; previously it was incorrectly returned in text mode. Also, the *file-like object* now supports the *context manager* protocol: you can use a `with` statement to automatically close it.

> **Note**
>
> Unlike other representations of messages, *file-like* representations are not necessarily independent of the `Mailbox` instance that created them or of the underlying mailbox. More specific documentation is provided by each subclass.

__contains__(*key*)

Return `True` if *key* corresponds to a message, `False` otherwise.

__len__()

Return a count of messages in the mailbox.

clear()

Delete all messages from the mailbox.

pop(*key*, *default=None*)

Return a representation of the message corresponding to *key* and delete the message. If no such message exists, return *default*. The message is represented as an instance of the appropriate format-specific *Message* subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

popitem()

Return an arbitrary (*key*, *message*) pair, where *key* is a key and *message* is a message representation, and delete the corresponding message. If the mailbox is empty, raise a *KeyError* exception. The message is represented as an instance of the appropriate format-specific *Message* subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

update(*arg*)

Parameter *arg* should be a *key*-to-*message* mapping or an iterable of (*key*, *message*) pairs. Updates the mailbox so that, for each given *key* and *message*, the message corresponding to *key* is set to *message* as if by using *__setitem__()*. As with *__setitem__()*, each *key* must already correspond to a message in the mailbox or else a *KeyError* exception will be raised, so in general it is incorrect for *arg* to be a `Mailbox` instance.

> **Note**
>
> Unlike with dictionaries, keyword arguments are not supported.

flush()

Write any pending changes to the filesystem. For some *Mailbox* subclasses, changes are always written immediately and `flush()` does nothing, but you should still make a habit of calling this method.

lock()

Acquire an exclusive advisory lock on the mailbox so that other processes know not to modify it. An *ExternalClashError* is raised if the lock is not available. The particular locking mechanisms used depend upon the mailbox format. You should *always* lock the mailbox before making any modifications to its contents.

unlock()

Release the lock on the mailbox, if any.

close()

Flush the mailbox, unlock it if necessary, and close any open files. For some `Mailbox` subclasses, this method does nothing.

`Maildir` objects

class mailbox.**Maildir**(*dirname*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in Maildir format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *MaildirMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

If *create* is `True` and the *dirname* path exists, it will be treated as an existing maildir without attempting to verify its directory layout.

It is for historical reasons that *dirname* is named as such rather than *path*.

Maildir is a directory-based mailbox format invented for the qmail mail transfer agent and now widely supported by other programs. Messages in a Maildir mailbox are stored in separate files within a common directory structure. This design allows Maildir mailboxes to be accessed and modified by multiple unrelated programs without data corruption, so file locking is unnecessary.

Maildir mailboxes contain three subdirectories, namely: `tmp`, `new`, and `cur`. Messages are created momentarily in the `tmp` subdirectory and then moved to the `new` subdirectory to finalize delivery. A mail user agent may subsequently move the message to the `cur` subdirectory and store information about the state of the message in a special "info" section appended to its file name.

Folders of the style introduced by the Courier mail transfer agent are also supported. Any subdirectory of the main mailbox is considered a folder if `'.'` is the first character in its name. Folder names are represented by `Maildir` without the leading `'.'`. Each folder is itself a Maildir mailbox but should not contain other folders. Instead, a logical nesting is indicated using `'.'` to delimit levels, e.g., "Archived.2005.07".

colon

The Maildir specification requires the use of a colon (`':'`) in certain message file names. However, some operating systems do not permit this character in file names, If you wish to use a Maildir-like format on such an operating system, you should specify another character to use instead. The exclamation point (`'!'`) is a popular choice. For example:

```
import mailbox
mailbox.Maildir.colon = '!'
```

The `colon` attribute may also be set on a per-instance basis.

Changed in version 3.13: *Maildir* now ignores files with a leading dot.

`Maildir` instances have all of the methods of *Mailbox* in addition to the following:

list_folders()

Return a list of the names of all folders.

get_folder(*folder*)

Return a `Maildir` instance representing the folder whose name is *folder*. A *NoSuchMailboxError* exception is raised if the folder does not exist.

add_folder(*folder*)

Create a folder whose name is *folder* and return a `Maildir` instance representing it.

remove_folder(*folder*)

Delete the folder whose name is *folder*. If the folder contains any messages, a *NotEmptyError* exception will be raised and the folder will not be deleted.

clean()

Delete temporary files from the mailbox that have not been accessed in the last 36 hours. The Maildir specification says that mail-reading programs should do this occasionally.

get_flags(*key*)

Return as a string the flags that are set on the message corresponding to *key*. This is the same as `get_message(key).get_flags()` but much faster, because it does not open the message file. Use this method when iterating over the keys to determine which messages are interesting to get.

If you do have a *MaildirMessage* object, use its *get_flags()* method instead, because changes made by the message's *set_flags()*, *add_flag()* and *remove_flag()* methods are not reflected here until the mailbox's *__setitem__()* method is called.

Added in version 3.13.

set_flags(*key*, *flags*)

On the message corresponding to *key*, set the flags specified by *flags* and unset all others. Calling `some_mailbox.set_flags(key, flags)` is similar to

```
one_message = some_mailbox.get_message(key)
one_message.set_flags(flags)
some_mailbox[key] = one_message
```

but faster, because it does not open the message file.

If you do have a *MaildirMessage* object, use its *set_flags()* method instead, because changes made with this mailbox method will not be visible to the message object's method, *get_flags()*.

Added in version 3.13.

add_flag(*key*, *flag*)

On the message corresponding to *key*, set the flags specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

Considerations for using this method versus the message object's *add_flag()* method are similar to those for *set_flags()*; see the discussion there.

Added in version 3.13.

remove_flag(*key*, *flag*)

On the message corresponding to *key*, unset the flags specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* may be a string of more than one character.

Considerations for using this method versus the message object's *remove_flag()* method are similar to those for *set_flags()*; see the discussion there.

Added in version 3.13.

get_info(*key*)

Return a string containing the info for the message corresponding to *key*. This is the same as `get_message(key).get_info()` but much faster, because it does not open the message file. Use this method when iterating over the keys to determine which messages are interesting to get.

If you do have a *MaildirMessage* object, use its *get_info()* method instead, because changes made by the message's *set_info()* method are not reflected here until the mailbox's *__setitem__()* method is called.

Added in version 3.13.

set_info(*key*, *info*)

Set the info of the message corresponding to *key* to *info*. Calling `some_mailbox.set_info(key, flags)` is similar to

```
one_message = some_mailbox.get_message(key)
one_message.set_info(info)
some_mailbox[key] = one_message
```

but faster, because it does not open the message file.

If you do have a *MaildirMessage* object, use its *set_info()* method instead, because changes made with this mailbox method will not be visible to the message object's method, *get_info()*.

Added in version 3.13.

Some *Mailbox* methods implemented by `Maildir` deserve special remarks:

add(*message*)

__setitem__(*key*, *message*)

update(*arg*)

> **Warning**
>
> These methods generate unique file names based upon the current process ID. When using multiple threads, undetected name clashes may occur and cause corruption of the mailbox unless threads are coordinated to avoid using these methods to manipulate the same mailbox simultaneously.

flush()

All changes to Maildir mailboxes are immediately applied, so this method does nothing.

lock()

unlock()

Maildir mailboxes do not support (or require) locking, so these methods do nothing.

close()

`Maildir` instances do not keep any open files and the underlying mailboxes do not support locking, so this method does nothing.

get_file(*key*)

Depending upon the host platform, it may not be possible to modify or remove the underlying message while the returned file remains open.

> **See also**
>
> **maildir man page from Courier**
> A specification of the format. Describes a common extension for supporting folders.
>
> **Using maildir format**
> Notes on Maildir by its inventor. Includes an updated name-creation scheme and details on "info" semantics.

mbox objects

class mailbox.**mbox**(*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in mbox format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *mboxMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

The mbox format is the classic format for storing mail on Unix systems. All messages in an mbox mailbox are stored in a single file with the beginning of each message indicated by a line whose first five characters are "From ".

Several variations of the mbox format exist to address perceived shortcomings in the original. In the interest of compatibility, `mbox` implements the original format, which is sometimes referred to as *mboxo*. This means that the *Content-Length* header, if present, is ignored and that any occurrences of "From " at the beginning of a line in a message body are transformed to ">From " when storing the message, although occurrences of ">From " are not transformed to "From " when reading the message.

Some *Mailbox* methods implemented by `mbox` deserve special remarks:

get_file(*key*)

Using the file after calling *flush()* or *close()* on the `mbox` instance may yield unpredictable results or raise an exception.

lock()

unlock()

Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.

See also

mbox man page from tin
A specification of the format, with details on locking.

Configuring Netscape Mail on Unix: Why The Content-Length Format is Bad
An argument for using the original mbox format rather than a variation.

"mbox" is a family of several mutually incompatible mailbox formats
A history of mbox variations.

MH objects

class mailbox.**MH**(*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in MH format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *MHMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

MH is a directory-based mailbox format invented for the MH Message Handling System, a mail user agent. Each message in an MH mailbox resides in its own file. An MH mailbox may contain other MH mailboxes (called *folders*) in addition to messages. Folders may be nested indefinitely. MH mailboxes also support *sequences*, which are named lists used to logically group messages without moving them to sub-folders. Sequences are defined in a file called `.mh_sequences` in each folder.

The MH class manipulates MH mailboxes, but it does not attempt to emulate all of **mh**'s behaviors. In particular, it does not modify and is not affected by the `context` or `.mh_profile` files that are used by **mh** to store its state and configuration.

MH instances have all of the methods of *Mailbox* in addition to the following:

Changed in version 3.13: Supported folders that don't contain a `.mh_sequences` file.

list_folders()

Return a list of the names of all folders.

get_folder(*folder*)

Return an MH instance representing the folder whose name is *folder*. A *NoSuchMailboxError* exception is raised if the folder does not exist.

add_folder(*folder*)

Create a folder whose name is *folder* and return an MH instance representing it.

remove_folder(*folder*)

Delete the folder whose name is *folder*. If the folder contains any messages, a *NotEmptyError* exception will be raised and the folder will not be deleted.

get_sequences()

Return a dictionary of sequence names mapped to key lists. If there are no sequences, the empty dictionary is returned.

set_sequences(*sequences*)

Re-define the sequences that exist in the mailbox based upon *sequences*, a dictionary of names mapped to key lists, like returned by *get_sequences()*.

pack()

Rename messages in the mailbox as necessary to eliminate gaps in numbering. Entries in the sequences list are updated correspondingly.

> **Note**
>
> Already-issued keys are invalidated by this operation and should not be subsequently used.

Some *Mailbox* methods implemented by MH deserve special remarks:

remove(*key*)

__delitem__(*key*)

discard(*key*)

These methods immediately delete the message. The MH convention of marking a message for deletion by prepending a comma to its name is not used.

lock()

unlock()

Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls. For MH mailboxes, locking the mailbox means locking the `.mh_sequences` file and, only for the duration of any operations that affect them, locking individual message files.

get_file(*key*)

Depending upon the host platform, it may not be possible to remove the underlying message while the returned file remains open.

flush()

All changes to MH mailboxes are immediately applied, so this method does nothing.

close()

`MH` instances do not keep any open files, so this method is equivalent to *`unlock()`*.

See also

nmh - Message Handling System
Home page of **nmh**, an updated version of the original **mh**.

MH & nmh: Email for Users & Programmers
A GPL-licensed book on **mh** and **nmh**, with some information on the mailbox format.

`Babyl` objects

class mailbox.**Babyl**(*path*, *factory=None*, *create=True*)

A subclass of *`Mailbox`* for mailboxes in Babyl format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *`BabylMessage`* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

Babyl is a single-file mailbox format used by the Rmail mail user agent included with Emacs. The beginning of a message is indicated by a line containing the two characters Control-Underscore (`'\037'`) and Control-L (`'\014'`). The end of a message is indicated by the start of the next message or, in the case of the last message, a line containing a Control-Underscore (`'\037'`) character.

Messages in a Babyl mailbox have two sets of headers, original headers and so-called visible headers. Visible headers are typically a subset of the original headers that have been reformatted or abridged to be more attractive. Each message in a Babyl mailbox also has an accompanying list of *labels*, or short strings that record extra information about the message, and a list of all user-defined labels found in the mailbox is kept in the Babyl options section.

`Babyl` instances have all of the methods of *`Mailbox`* in addition to the following:

get_labels()

Return a list of the names of all user-defined labels used in the mailbox.

Note

The actual messages are inspected to determine which labels exist in the mailbox rather than consulting the list of labels in the Babyl options section, but the Babyl section is updated whenever the mailbox is modified.

Some *`Mailbox`* methods implemented by `Babyl` deserve special remarks:

get_file(*key*)

In Babyl mailboxes, the headers of a message are not stored contiguously with the body of the message. To generate a file-like representation, the headers and body are copied together into an *`io.BytesIO`* instance, which has an API identical to that of a file. As a result, the file-like object is truly independent of the underlying mailbox but does not save memory compared to a string representation.

lock()

unlock()

Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.

See also

Format of Version 5 Babyl Files
A specification of the Babyl format.

Reading Mail with Rmail
The Rmail manual, with some information on Babyl semantics.

`MMDF` objects

class mailbox.**MMDF**(*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in MMDF format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *MMDFMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

MMDF is a single-file mailbox format invented for the Multichannel Memorandum Distribution Facility, a mail transfer agent. Each message is in the same form as an mbox message but is bracketed before and after by lines containing four Control-A (`'\001'`) characters. As with the mbox format, the beginning of each message is indicated by a line whose first five characters are "From ", but additional occurrences of "From " are not transformed to ">From " when storing messages because the extra message separator lines prevent mistaking such occurrences for the starts of subsequent messages.

Some *Mailbox* methods implemented by `MMDF` deserve special remarks:

get_file(*key*)

Using the file after calling *flush()* or *close()* on the `MMDF` instance may yield unpredictable results or raise an exception.

lock()

unlock()

Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.

See also

mmdf man page from tin
A specification of MMDF format from the documentation of tin, a newsreader.

MMDF
A Wikipedia article describing the Multichannel Memorandum Distribution Facility.

19.3.2 `Message` objects

class mailbox.**Message**(*message=None*)

A subclass of the *email.message* module's *Message*. Subclasses of `mailbox.Message` add mailbox-format-specific state and behavior.

If *message* is omitted, the new instance is created in a default, empty state. If *message* is an *email.message.Message* instance, its contents are copied; furthermore, any format-specific information is converted insofar as possible if *message* is a `Message` instance. If *message* is a string, a byte string, or a file, it should contain an **RFC 2822**-compliant message, which is read and parsed. Files should be open in binary mode, but text mode files are accepted for backward compatibility.

The format-specific state and behaviors offered by subclasses vary, but in general it is only the properties that are not specific to a particular mailbox that are supported (although presumably the properties are specific to a particular mailbox format). For example, file offsets for single-file mailbox formats and file names for directory-based mailbox formats are not retained, because they are only applicable to the original mailbox. But state such as whether a message has been read by the user or marked as important is retained, because it applies to the message itself.

There is no requirement that `Message` instances be used to represent messages retrieved using *Mailbox* instances. In some situations, the time and memory required to generate `Message` representations might not be acceptable. For such situations, `Mailbox` instances also offer string and file-like representations, and a custom message factory may be specified when a `Mailbox` instance is initialized.

`MaildirMessage` objects

class mailbox.**MaildirMessage**(*message=None*)

A message with Maildir-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

Typically, a mail user agent application moves all of the messages in the `new` subdirectory to the `cur` subdirectory after the first time the user opens and closes the mailbox, recording that the messages are old whether or not they've actually been read. Each message in `cur` has an "info" section added to its file name to store information about its state. (Some mail readers may also add an "info" section to messages in `new`.) The "info" section may take one of two forms: it may contain "2," followed by a list of standardized flags (e.g., "2,FR") or it may contain "1," followed by so-called experimental information. Standard flags for Maildir messages are as follows:

Flag	Meaning	Explanation
D	Draft	Under composition
F	Flagged	Marked as important
P	Passed	Forwarded, resent, or bounced
R	Replied	Replied to
S	Seen	Read
T	Trashed	Marked for subsequent deletion

`MaildirMessage` instances offer the following methods:

get_subdir()

Return either "new" (if the message should be stored in the `new` subdirectory) or "cur" (if the message should be stored in the `cur` subdirectory).

> **Note**
>
> A message is typically moved from `new` to `cur` after its mailbox has been accessed, whether or not the message has been read. A message `msg` has been read if `"S" in msg.get_flags()` is `True`.

set_subdir(*subdir*)

Set the subdirectory the message should be stored in. Parameter *subdir* must be either "new" or "cur".

get_flags()

Return a string specifying the flags that are currently set. If the message complies with the standard Maildir format, the result is the concatenation in alphabetical order of zero or one occurrence of each of `'D'`, `'F'`, `'P'`, `'R'`, `'S'`, and `'T'`. The empty string is returned if no flags are set or if "info" contains experimental semantics.

set_flags(*flags*)

Set the flags specified by *flags* and unset all others.

add_flag(*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character. The current "info" is overwritten whether or not it contains experimental information rather than flags.

remove_flag(*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character. If "info" contains experimental information rather than flags, the current "info" is not modified.

get_date()

Return the delivery date of the message as a floating-point number representing seconds since the epoch.

set_date(*date*)

Set the delivery date of the message to *date*, a floating-point number representing seconds since the epoch.

get_info()

Return a string containing the "info" for a message. This is useful for accessing and modifying "info" that is experimental (i.e., not a list of flags).

set_info(*info*)

Set "info" to *info*, which should be a string.

When a `MaildirMessage` instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

Resulting state	*mboxMessage* or *MMDFMessage* state
"cur" subdirectory	O flag
F flag	F flag
R flag	A flag
S flag	R flag
T flag	D flag

When a `MaildirMessage` instance is created based upon an *MHMessage* instance, the following conversions take place:

Resulting state	*MHMessage* state
“cur” subdirectory	“unseen” sequence
“cur” subdirectory and S flag	no “unseen” sequence
F flag	“flagged” sequence
R flag	“replied” sequence

When a `MaildirMessage` instance is created based upon a *BabylMessage* instance, the following conversions take place:

Resulting state	*BabylMessage* state
“cur” subdirectory	“unseen” label
“cur” subdirectory and S flag	no “unseen” label
P flag	“forwarded” or “resent” label
R flag	“answered” label
T flag	“deleted” label

`mboxMessage` objects

class mailbox.**mboxMessage**(*message=None*)

A message with mbox-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

Messages in an mbox mailbox are stored together in a single file. The sender’s envelope address and the time of delivery are typically stored in a line beginning with “From “ that is used to indicate the start of a message, though there is considerable variation in the exact format of this data among mbox implementations. Flags that indicate the state of the message, such as whether it has been read or marked as important, are typically stored in *Status* and *X-Status* headers.

Conventional flags for mbox messages are as follows:

Flag	Meaning	Explanation
R	Read	Read
O	Old	Previously detected by MUA
D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

The “R” and “O” flags are stored in the *Status* header, and the “D”, “F”, and “A” flags are stored in the *X-Status* header. The flags and headers typically appear in the order mentioned.

`mboxMessage` instances offer the following methods:

get_from()

Return a string representing the “From “ line that marks the start of the message in an mbox mailbox. The leading “From “ and the trailing newline are excluded.

set_from(*from_*, *time_=None*)

Set the “From “ line to *from_*, which should be specified without a leading “From “ or trailing newline. For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a *time.struct_time* instance, a tuple suitable for passing to *time.strftime()*, or `True` (to use *time.gmtime()*).

get_flags()

Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

set_flags(*flags*)

Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

add_flag(*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

remove_flag(*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an `mboxMessage` instance is created based upon a *`MaildirMessage`* instance, a "From " line is generated based upon the *`MaildirMessage`* instance's delivery date, and the following conversions take place:

Resulting state	*`MaildirMessage`* state
R flag	S flag
O flag	"cur" subdirectory
D flag	T flag
F flag	F flag
A flag	R flag

When an `mboxMessage` instance is created based upon an *`MHMessage`* instance, the following conversions take place:

Resulting state	*`MHMessage`* state
R flag and O flag	no "unseen" sequence
O flag	"unseen" sequence
F flag	"flagged" sequence
A flag	"replied" sequence

When an `mboxMessage` instance is created based upon a *`BabylMessage`* instance, the following conversions take place:

Resulting state	*`BabylMessage`* state
R flag and O flag	no "unseen" label
O flag	"unseen" label
D flag	"deleted" label
A flag	"answered" label

When a `mboxMessage` instance is created based upon an *`MMDFMessage`* instance, the "From " line is copied and all flags directly correspond:

Resulting state	*MMDFMessage* state
R flag	R flag
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

MHMessage objects

class mailbox.**MHMessage**(*message=None*)

A message with MH-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

MH messages do not support marks or flags in the traditional sense, but they do support sequences, which are logical groupings of arbitrary messages. Some mail reading programs (although not the standard **mh** and **nmh**) use sequences in much the same way flags are used with other formats, as follows:

Sequence	Explanation
unseen	Not read, but previously detected by MUA
replied	Replied to
flagged	Marked as important

MHMessage instances offer the following methods:

get_sequences()

Return a list of the names of sequences that include this message.

set_sequences(*sequences*)

Set the list of sequences that include this message.

add_sequence(*sequence*)

Add *sequence* to the list of sequences that include this message.

remove_sequence(*sequence*)

Remove *sequence* from the list of sequences that include this message.

When an MHMessage instance is created based upon a *MaildirMessage* instance, the following conversions take place:

Resulting state	*MaildirMessage* state
“unseen” sequence	no S flag
“replied” sequence	R flag
“flagged” sequence	F flag

When an MHMessage instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

Resulting state	*mboxMessage* or *MMDFMessage* state
“unseen” sequence	no R flag
“replied” sequence	A flag
“flagged” sequence	F flag

When an MHMessage instance is created based upon a *BabylMessage* instance, the following conversions take place:

Resulting state	*BabylMessage* state
“unseen” sequence	“unseen” label
“replied” sequence	“answered” label

BabylMessage objects

class mailbox.**BabylMessage**(*message=None*)

A message with Babyl-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

Certain message labels, called *attributes*, are defined by convention to have special meanings. The attributes are as follows:

Label	Explanation
unseen	Not read, but previously detected by MUA
deleted	Marked for subsequent deletion
filed	Copied to another file or mailbox
answered	Replied to
forwarded	Forwarded
edited	Modified by the user
resent	Resent

By default, Rmail displays only visible headers. The BabylMessage class, though, uses the original headers because they are more complete. Visible headers may be accessed explicitly if desired.

BabylMessage instances offer the following methods:

get_labels()

Return a list of labels on the message.

set_labels(*labels*)

Set the list of labels on the message to *labels*.

add_label(*label*)

Add *label* to the list of labels on the message.

remove_label(*label*)

Remove *label* from the list of labels on the message.

get_visible()

Return a *Message* instance whose headers are the message’s visible headers and whose body is empty.

set_visible(*visible*)

Set the message's visible headers to be the same as the headers in *message*. Parameter *visible* should be a *Message* instance, an *email.message.Message* instance, a string, or a file-like object (which should be open in text mode).

update_visible()

When a BabylMessage instance's original headers are modified, the visible headers are not automatically modified to correspond. This method updates the visible headers as follows: each visible header with a corresponding original header is set to the value of the original header, each visible header without a corresponding original header is removed, and any of *Date*, *From*, *Reply-To*, *To*, *CC*, and *Subject* that are present in the original headers but not the visible headers are added to the visible headers.

When a BabylMessage instance is created based upon a *MaildirMessage* instance, the following conversions take place:

Resulting state	*MaildirMessage* state
"unseen" label	no S flag
"deleted" label	T flag
"answered" label	R flag
"forwarded" label	P flag

When a BabylMessage instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

Resulting state	*mboxMessage* or *MMDFMessage* state
"unseen" label	no R flag
"deleted" label	D flag
"answered" label	A flag

When a BabylMessage instance is created based upon an *MHMessage* instance, the following conversions take place:

Resulting state	*MHMessage* state
"unseen" label	"unseen" sequence
"answered" label	"replied" sequence

MMDFMessage objects

class mailbox.**MMDFMessage**(*message=None*)

A message with MMDF-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

As with message in an mbox mailbox, MMDF messages are stored with the sender's address and the delivery date in an initial line beginning with "From ". Likewise, flags that indicate the state of the message are typically stored in *Status* and *X-Status* headers.

Conventional flags for MMDF messages are identical to those of mbox message and are as follows:

Flag	Meaning	Explanation
R	Read	Read
O	Old	Previously detected by MUA
D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

The "R" and "O" flags are stored in the *Status* header, and the "D", "F", and "A" flags are stored in the *X-Status* header. The flags and headers typically appear in the order mentioned.

MMDFMessage instances offer the following methods, which are identical to those offered by *mboxMessage*:

get_from()

Return a string representing the "From " line that marks the start of the message in an mbox mailbox. The leading "From " and the trailing newline are excluded.

set_from(*from_*, *time_*=*None*)

Set the "From " line to *from_*, which should be specified without a leading "From " or trailing newline. For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a *time.struct_time* instance, a tuple suitable for passing to *time.strftime()*, or True (to use *time.gmtime()*).

get_flags()

Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of 'R', 'O', 'D', 'F', and 'A'.

set_flags(*flags*)

Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of 'R', 'O', 'D', 'F', and 'A'.

add_flag(*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

remove_flag(*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an MMDFMessage instance is created based upon a *MaildirMessage* instance, a "From " line is generated based upon the *MaildirMessage* instance's delivery date, and the following conversions take place:

Resulting state	*MaildirMessage* state
R flag	S flag
O flag	"cur" subdirectory
D flag	T flag
F flag	F flag
A flag	R flag

When an MMDFMessage instance is created based upon an *MHMessage* instance, the following conversions take place:

Resulting state	*MHMessage* state
R flag and O flag	no “unseen” sequence
O flag	“unseen” sequence
F flag	“flagged” sequence
A flag	“replied” sequence

When an `MMDFMessage` instance is created based upon a *BabylMessage* instance, the following conversions take place:

Resulting state	*BabylMessage* state
R flag and O flag	no “unseen” label
O flag	“unseen” label
D flag	“deleted” label
A flag	“answered” label

When an `MMDFMessage` instance is created based upon an *mboxMessage* instance, the “From “ line is copied and all flags directly correspond:

Resulting state	*mboxMessage* state
R flag	R flag
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

19.3.3 Exceptions

The following exception classes are defined in the `mailbox` module:

exception `mailbox.`**Error**

The based class for all other module-specific exceptions.

exception `mailbox.`**NoSuchMailboxError**

Raised when a mailbox is expected but is not found, such as when instantiating a *Mailbox* subclass with a path that does not exist (and with the *create* parameter set to `False`), or when opening a folder that does not exist.

exception `mailbox.`**NotEmptyError**

Raised when a mailbox is not empty but is expected to be, such as when deleting a folder that contains messages.

exception `mailbox.`**ExternalClashError**

Raised when some mailbox-related condition beyond the control of the program causes it to be unable to proceed, such as when failing to acquire a lock that another program already holds a lock, or when a uniquely generated file name already exists.

exception `mailbox.`**FormatError**

Raised when the data in a file cannot be parsed, such as when an *MH* instance attempts to read a corrupted `.mh_sequences` file.

19.3.4 Examples

A simple example of printing the subjects of all messages in a mailbox that seem interesting:

```
import mailbox
for message in mailbox.mbox('~/mbox'):
    subject = message['subject']       # Could possibly be None.
    if subject and 'python' in subject.lower():
        print(subject)
```

To copy all mail from a Babyl mailbox to an MH mailbox, converting all of the format-specific information that can be converted:

```
import mailbox
destination = mailbox.MH('~/Mail')
destination.lock()
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

This example sorts mail from several mailing lists into different mailboxes, being careful to avoid mail corruption due to concurrent modification by other programs, mail loss due to interruption of the program, or premature termination due to malformed messages in the mailbox:

```
import mailbox
import email.errors

list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = {name: mailbox.mbox('~/email/%s' % name) for name in list_names}
inbox = mailbox.Maildir('~/Maildir', factory=None)

for key in inbox.iterkeys():
    try:
        message = inbox[key]
    except email.errors.MessageParseError:
        continue                # The message is malformed. Just leave it.

    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            # Get mailbox to use
            box = boxes[name]

            # Write copy to disk before removing original.
            # If there's a crash, you might duplicate a message, but
            # that's better than losing a message completely.
            box.lock()
            box.add(message)
            box.flush()
            box.unlock()

            # Remove original message
            inbox.lock()
            inbox.discard(key)
            inbox.flush()
```

(continues on next page)

(continued from previous page)

```
            inbox.unlock()
            break               # Found destination, so stop looking.

for box in boxes.itervalues():
    box.close()
```

19.4 mimetypes — Map filenames to MIME types

Source code: Lib/mimetypes.py

The *mimetypes* module converts between a filename or URL and the MIME type associated with the filename extension. Conversions are provided from filename to MIME type and from MIME type to filename extension; encodings are not supported for the latter conversion.

The module provides one class and a number of convenience functions. The functions are the normal interface to this module, but some applications may be interested in the class as well.

The functions described below provide the primary interface for this module. If the module has not been initialized, they will call *init()* if they rely on the information *init()* sets up.

mimetypes.**guess_type**(*url*, *strict=True*)

Guess the type of a file based on its filename, path or URL, given by *url*. URL can be a string or a *path-like object*.

The return value is a tuple `(type, encoding)` where *type* is `None` if the type can't be guessed (missing or unknown suffix) or a string of the form `'type/subtype'`, usable for a MIME *content-type* header.

encoding is `None` for no encoding or the name of the program used to encode (e.g. **compress** or **gzip**). The encoding is suitable for use as a *Content-Encoding* header, **not** as a *Content-Transfer-Encoding* header. The mappings are table driven. Encoding suffixes are case sensitive; type suffixes are first tried case sensitively, then case insensitively.

The optional *strict* argument is a flag specifying whether the list of known MIME types is limited to only the official types registered with IANA. When *strict* is `True` (the default), only the IANA types are supported; when *strict* is `False`, some additional non-standard but commonly used MIME types are also recognized.

Changed in version 3.8: Added support for *url* being a *path-like object*.

Deprecated since version 3.13: Passing a file path instead of URL is *soft deprecated*. Use *guess_file_type()* for this.

mimetypes.**guess_file_type**(*path*, ***, *strict=True*)

Guess the type of a file based on its path, given by *path*. Similar to the *guess_type()* function, but accepts a path instead of URL. Path can be a string, a bytes object or a *path-like object*.

Added in version 3.13.

mimetypes.**guess_all_extensions**(*type*, *strict=True*)

Guess the extensions for a file based on its MIME type, given by *type*. The return value is a list of strings giving all possible filename extensions, including the leading dot (`'.'`). The extensions are not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by *guess_type()* and *guess_file_type()*.

The optional *strict* argument has the same meaning as with the *guess_type()* function.

mimetypes.**guess_extension**(*type*, *strict=True*)

Guess the extension for a file based on its MIME type, given by *type*. The return value is a string giving a filename extension, including the leading dot (`'.'`). The extension is not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by *guess_type()* and *guess_file_type()*. If no extension can be guessed for *type*, `None` is returned.

The optional *strict* argument has the same meaning as with the *guess_type()* function.

Some additional functions and data items are available for controlling the behavior of the module.

mimetypes.**init**(*files=None*)

Initialize the internal data structures. If given, *files* must be a sequence of file names which should be used to augment the default type map. If omitted, the file names to use are taken from *knownfiles*; on Windows, the current registry settings are loaded. Each file named in *files* or *knownfiles* takes precedence over those named before it. Calling *init()* repeatedly is allowed.

Specifying an empty list for *files* will prevent the system defaults from being applied: only the well-known values will be present from a built-in list.

If *files* is `None` the internal data structure is completely rebuilt to its initial default value. This is a stable operation and will produce the same results when called multiple times.

Changed in version 3.2: Previously, Windows registry settings were ignored.

mimetypes.**read_mime_types**(*filename*)

Load the type map given in the file *filename*, if it exists. The type map is returned as a dictionary mapping filename extensions, including the leading dot (`'.'`), to strings of the form `'type/subtype'`. If the file *filename* does not exist or cannot be read, `None` is returned.

mimetypes.**add_type**(*type*, *ext*, *strict=True*)

Add a mapping from the MIME type *type* to the extension *ext*. When the extension is already known, the new type will replace the old one. When the type is already known the extension will be added to the list of known extensions.

When *strict* is `True` (the default), the mapping will be added to the official MIME types, otherwise to the non-standard ones.

mimetypes.**inited**

Flag indicating whether or not the global data structures have been initialized. This is set to `True` by *init()*.

mimetypes.**knownfiles**

List of type map file names commonly installed. These files are typically named `mime.types` and are installed in different locations by different packages.

mimetypes.**suffix_map**

Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately.

mimetypes.**encodings_map**

Dictionary mapping filename extensions to encoding types.

mimetypes.**types_map**

Dictionary mapping filename extensions to MIME types.

mimetypes.**common_types**

Dictionary mapping filename extensions to non-standard, but commonly found MIME types.

An example usage of the module:

```
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

19.4.1 MimeTypes Objects

The *MimeTypes* class may be useful for applications which may want more than one MIME-type database; it provides an interface similar to the one of the *mimetypes* module.

class mimetypes.**MimeTypes**(*filenames=()*, *strict=True*)

This class represents a MIME-types database. By default, it provides access to the same database as the rest of this module. The initial database is a copy of that provided by the module, and may be extended by loading additional `mime.types`-style files into the database using the *read()* or *readfp()* methods. The mapping dictionaries may also be cleared before loading additional data if the default data is not desired.

The optional *filenames* parameter can be used to cause additional files to be loaded "on top" of the default database.

suffix_map

Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately. This is initially a copy of the global *suffix_map* defined in the module.

encodings_map

Dictionary mapping filename extensions to encoding types. This is initially a copy of the global *encodings_map* defined in the module.

types_map

Tuple containing two dictionaries, mapping filename extensions to MIME types: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by *common_types* and *types_map*.

types_map_inv

Tuple containing two dictionaries, mapping MIME types to a list of filename extensions: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by *common_types* and *types_map*.

guess_extension(*type*, *strict=True*)

Similar to the *guess_extension()* function, using the tables stored as part of the object.

guess_type(*url*, *strict=True*)

Similar to the *guess_type()* function, using the tables stored as part of the object.

guess_file_type(*path*, ***, *strict=True*)

Similar to the *guess_file_type()* function, using the tables stored as part of the object.

Added in version 3.13.

guess_all_extensions(*type*, *strict=True*)

Similar to the *guess_all_extensions()* function, using the tables stored as part of the object.

read(*filename*, *strict=True*)

Load MIME information from a file named *filename*. This uses *readfp()* to parse the file.

If *strict* is `True`, information will be added to list of standard types, else to the list of non-standard types.

readfp(*fp*, *strict=True*)

Load MIME type information from an open file *fp*. The file must have the format of the standard `mime.types` files.

If *strict* is `True`, information will be added to the list of standard types, else to the list of non-standard types.

read_windows_registry(*strict=True*)

Load MIME type information from the Windows registry.

Availability: Windows.

If *strict* is `True`, information will be added to the list of standard types, else to the list of non-standard types.

Added in version 3.2.

add_type(*type*, *ext*, *strict=True*)

Add a mapping from the MIME type *type* to the extension *ext*. When the extension is already known, the new type will replace the old one. When the type is already known the extension will be added to the list of known extensions.

When *strict* is `True` (the default), the mapping will be added to the official MIME types, otherwise to the non-standard ones.

19.5 base64 — Base16, Base32, Base64, Base85 Data Encodings

Source code: Lib/base64.py

This module provides functions for encoding binary data to printable ASCII characters and decoding such encodings back to binary data. It provides encoding and decoding functions for the encodings specified in **RFC 4648**, which defines the Base16, Base32, and Base64 algorithms, and for the de-facto standard Ascii85 and Base85 encodings.

The **RFC 4648** encodings are suitable for encoding binary data so that it can be safely sent by email, used as parts of URLs, or included as part of an HTTP POST request. The encoding algorithm is not the same as the **uuencode** program.

There are two interfaces provided by this module. The modern interface supports encoding *bytes-like objects* to ASCII *bytes*, and decoding *bytes-like objects* or strings containing ASCII to *bytes*. Both base-64 alphabets defined in **RFC 4648** (normal, and URL- and filesystem-safe) are supported.

The legacy interface does not support decoding from strings, but it does provide functions for encoding and decoding to and from *file objects*. It only supports the Base64 standard alphabet, and it adds newlines every 76 characters as per **RFC 2045**. Note that if you are looking for **RFC 2045** support you probably want to be looking at the *email* package instead.

Changed in version 3.3: ASCII-only Unicode strings are now accepted by the decoding functions of the modern interface.

Changed in version 3.4: Any *bytes-like objects* are now accepted by all encoding and decoding functions in this module. Ascii85/Base85 support added.

The modern interface provides:

base64.**b64encode**(*s*, *altchars=None*)

Encode the *bytes-like object* *s* using Base64 and return the encoded *bytes*.

Optional *altchars* must be a *bytes-like object* of length 2 which specifies an alternative alphabet for the + and / characters. This allows an application to e.g. generate URL or filesystem safe Base64 strings. The default is None, for which the standard Base64 alphabet is used.

May assert or raise a *ValueError* if the length of *altchars* is not 2. Raises a *TypeError* if *altchars* is not a *bytes-like object*.

base64.**b64decode**(*s*, *altchars=None*, *validate=False*)

Decode the Base64 encoded *bytes-like object* or ASCII string *s* and return the decoded *bytes*.

Optional *altchars* must be a *bytes-like object* or ASCII string of length 2 which specifies the alternative alphabet used instead of the + and / characters.

A *binascii.Error* exception is raised if *s* is incorrectly padded.

If *validate* is False (the default), characters that are neither in the normal base-64 alphabet nor the alternative alphabet are discarded prior to the padding check. If *validate* is True, these non-alphabet characters in the input result in a *binascii.Error*.

For more information about the strict base64 check, see *binascii.a2b_base64()*

May assert or raise a *ValueError* if the length of *altchars* is not 2.

base64.**standard_b64encode**(*s*)

Encode *bytes-like object* *s* using the standard Base64 alphabet and return the encoded *bytes*.

base64.**standard_b64decode**(*s*)

Decode *bytes-like object* or ASCII string *s* using the standard Base64 alphabet and return the decoded *bytes*.

base64.**urlsafe_b64encode**(*s*)

Encode *bytes-like object* *s* using the URL- and filesystem-safe alphabet, which substitutes − instead of + and _ instead of / in the standard Base64 alphabet, and return the encoded *bytes*. The result can still contain =.

base64.**urlsafe_b64decode**(*s*)

Decode *bytes-like object* or ASCII string *s* using the URL- and filesystem-safe alphabet, which substitutes − instead of + and _ instead of / in the standard Base64 alphabet, and return the decoded *bytes*.

base64.**b32encode**(*s*)

Encode the *bytes-like object* *s* using Base32 and return the encoded *bytes*.

base64.**b32decode**(*s*, *casefold=False*, *map01=None*)

Decode the Base32 encoded *bytes-like object* or ASCII string *s* and return the decoded *bytes*.

Optional *casefold* is a flag specifying whether a lowercase alphabet is acceptable as input. For security purposes, the default is False.

RFC 4648 allows for optional mapping of the digit 0 (zero) to the letter O (oh), and for optional mapping of the digit 1 (one) to either the letter I (eye) or letter L (el). The optional argument *map01* when not None, specifies which letter the digit 1 should be mapped to (when *map01* is not None, the digit 0 is always mapped to the letter O). For security purposes the default is None, so that 0 and 1 are not allowed in the input.

A *binascii.Error* is raised if *s* is incorrectly padded or if there are non-alphabet characters present in the input.

base64.**b32hexencode**(*s*)

Similar to *b32encode()* but uses the Extended Hex Alphabet, as defined in **RFC 4648**.

Added in version 3.10.

base64.**b32hexdecode**(*s*, *casefold=False*)

Similar to *b32decode()* but uses the Extended Hex Alphabet, as defined in **RFC 4648**.

This version does not allow the digit 0 (zero) to the letter O (oh) and digit 1 (one) to either the letter I (eye) or letter L (el) mappings, all these characters are included in the Extended Hex Alphabet and are not interchangeable.

Added in version 3.10.

base64.**b16encode**(*s*)

Encode the *bytes-like object* *s* using Base16 and return the encoded *bytes*.

base64.**b16decode**(*s*, *casefold=False*)

Decode the Base16 encoded *bytes-like object* or ASCII string *s* and return the decoded *bytes*.

Optional *casefold* is a flag specifying whether a lowercase alphabet is acceptable as input. For security purposes, the default is `False`.

A *binascii.Error* is raised if *s* is incorrectly padded or if there are non-alphabet characters present in the input.

base64.**a85encode**(*b*, ***, *foldspaces=False*, *wrapcol=0*, *pad=False*, *adobe=False*)

Encode the *bytes-like object* *b* using Ascii85 and return the encoded *bytes*.

foldspaces is an optional flag that uses the special short sequence 'y' instead of 4 consecutive spaces (ASCII 0x20) as supported by 'btoa'. This feature is not supported by the "standard" Ascii85 encoding.

wrapcol controls whether the output should have newline (`b'\n'`) characters added to it. If this is non-zero, each output line will be at most this many characters long, excluding the trailing newline.

pad controls whether the input is padded to a multiple of 4 before encoding. Note that the `btoa` implementation always pads.

adobe controls whether the encoded byte sequence is framed with <~ and ~>, which is used by the Adobe implementation.

Added in version 3.4.

base64.**a85decode**(*b*, ***, *foldspaces=False*, *adobe=False*, *ignorechars=b' \t\n\r\x0b'*)

Decode the Ascii85 encoded *bytes-like object* or ASCII string *b* and return the decoded *bytes*.

foldspaces is a flag that specifies whether the 'y' short sequence should be accepted as shorthand for 4 consecutive spaces (ASCII 0x20). This feature is not supported by the "standard" Ascii85 encoding.

adobe controls whether the input sequence is in Adobe Ascii85 format (i.e. is framed with <~ and ~>).

ignorechars should be a *bytes-like object* or ASCII string containing characters to ignore from the input. This should only contain whitespace characters, and by default contains all whitespace characters in ASCII.

Added in version 3.4.

base64.**b85encode**(*b*, *pad=False*)

Encode the *bytes-like object* *b* using base85 (as used in e.g. git-style binary diffs) and return the encoded *bytes*.

If *pad* is true, the input is padded with `b'\0'` so its length is a multiple of 4 bytes before encoding.

Added in version 3.4.

base64.**b85decode**(*b*)

Decode the base85-encoded *bytes-like object* or ASCII string *b* and return the decoded *bytes*. Padding is implicitly removed, if necessary.

Added in version 3.4.

base64.**z85encode**(*s*)

Encode the *bytes-like object* *s* using Z85 (as used in ZeroMQ) and return the encoded *bytes*. See Z85 specification for more information.

Added in version 3.13.

base64.**z85decode**(*s*)

Decode the Z85-encoded *bytes-like object* or ASCII string *s* and return the decoded *bytes*. See Z85 specification for more information.

Added in version 3.13.

The legacy interface:

base64.**decode**(*input*, *output*)

Decode the contents of the binary *input* file and write the resulting binary data to the *output* file. *input* and *output* must be *file objects*. *input* will be read until `input.readline()` returns an empty bytes object.

base64.**decodebytes**(*s*)

Decode the *bytes-like object* *s*, which must contain one or more lines of base64 encoded data, and return the decoded *bytes*.

Added in version 3.1.

base64.**encode**(*input*, *output*)

Encode the contents of the binary *input* file and write the resulting base64 encoded data to the *output* file. *input* and *output* must be *file objects*. *input* will be read until `input.read()` returns an empty bytes object. *encode()* inserts a newline character (`b'\n'`) after every 76 bytes of the output, as well as ensuring that the output always ends with a newline, as per **RFC 2045** (MIME).

base64.**encodebytes**(*s*)

Encode the *bytes-like object* *s*, which can contain arbitrary binary data, and return *bytes* containing the base64-encoded data, with newlines (`b'\n'`) inserted after every 76 bytes of output, and ensuring that there is a trailing newline, as per **RFC 2045** (MIME).

Added in version 3.1.

An example usage of the module:

```
>>> import base64
>>> encoded = base64.b64encode(b'data to be encoded')
>>> encoded
b'ZGF0YSB0byBiZSBlbmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
b'data to be encoded'
```

19.5.1 Security Considerations

A new security considerations section was added to **RFC 4648** (section 12); it's recommended to review the security section for any code deployed to production.

See also

Module *binascii*
Support module containing ASCII-to-binary and binary-to-ASCII conversions.

RFC 1521 - MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies
Section 5.2, "Base64 Content-Transfer-Encoding," provides the definition of the base64 encoding.

19.6 `binascii` — Convert between binary and ASCII

The *binascii* module contains a number of methods to convert between binary and various ASCII-encoded binary representations. Normally, you will not use these functions directly but use wrapper modules like *base64* instead. The *binascii* module contains low-level functions written in C for greater speed that are used by the higher-level modules.

Note

`a2b_*` functions accept Unicode strings containing only ASCII characters. Other functions only accept *bytes-like objects* (such as *bytes*, *bytearray* and other objects that support the buffer protocol).

Changed in version 3.3: ASCII-only unicode strings are now accepted by the `a2b_*` functions.

The *binascii* module defines the following functions:

`binascii.a2b_uu`(*string*)

Convert a single line of uuencoded data back to binary and return the binary data. Lines normally contain 45 (binary) bytes, except for the last line. Line data may be followed by whitespace.

`binascii.b2a_uu`(*data*, *, *backtick=False*)

Convert binary data to a line of ASCII characters, the return value is the converted line, including a newline char. The length of *data* should be at most 45. If *backtick* is true, zeros are represented by `'`'` instead of spaces.

Changed in version 3.7: Added the *backtick* parameter.

`binascii.a2b_base64`(*string*, /, *, *strict_mode=False*)

Convert a block of base64 data back to binary and return the binary data. More than one line may be passed at a time.

If *strict_mode* is true, only valid base64 data will be converted. Invalid base64 data will raise *binascii.Error*.

Valid base64:

- Conforms to **RFC 3548**.
- Contains only characters from the base64 alphabet.
- Contains no excess data after padding (including excess padding, newlines, etc.).
- Does not start with a padding.

Changed in version 3.11: Added the *strict_mode* parameter.

`binascii.b2a_base64`(*data*, *, *newline=True*)

Convert binary data to a line of ASCII characters in base64 coding. The return value is the converted line, including a newline char if *newline* is true. The output of this function conforms to **RFC 3548**.

Changed in version 3.6: Added the *newline* parameter.

binascii.**a2b_qp**(*data*, *header=False*)

Convert a block of quoted-printable data back to binary and return the binary data. More than one line may be passed at a time. If the optional argument *header* is present and true, underscores will be decoded as spaces.

binascii.**b2a_qp**(*data*, *quotetabs=False*, *istext=True*, *header=False*)

Convert binary data to a line(s) of ASCII characters in quoted-printable encoding. The return value is the converted line(s). If the optional argument *quotetabs* is present and true, all tabs and spaces will be encoded. If the optional argument *istext* is present and true, newlines are not encoded but trailing whitespace will be encoded. If the optional argument *header* is present and true, spaces will be encoded as underscores per **RFC 1522**. If the optional argument *header* is present and false, newline characters will be encoded as well; otherwise linefeed conversion might corrupt the binary data stream.

binascii.**crc_hqx**(*data*, *value*)

Compute a 16-bit CRC value of *data*, starting with *value* as the initial CRC, and return the result. This uses the CRC-CCITT polynomial $x^{16} + x^{12} + x^5 + 1$, often represented as 0x1021. This CRC is used in the binhex4 format.

binascii.**crc32**(*data*[, *value*])

Compute CRC-32, the unsigned 32-bit checksum of *data*, starting with an initial CRC of *value*. The default initial CRC is zero. The algorithm is consistent with the ZIP file checksum. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm. Use as follows:

```
print(binascii.crc32(b"hello world"))
# Or, in two pieces:
crc = binascii.crc32(b"hello")
crc = binascii.crc32(b" world", crc)
print('crc32 = {:#010x}'.format(crc))
```

Changed in version 3.0: The result is always unsigned.

binascii.**b2a_hex**(*data*[, *sep*[, *bytes_per_sep=1*]])

binascii.**hexlify**(*data*[, *sep*[, *bytes_per_sep=1*]])

Return the hexadecimal representation of the binary *data*. Every byte of *data* is converted into the corresponding 2-digit hex representation. The returned bytes object is therefore twice as long as the length of *data*.

Similar functionality (but returning a text string) is also conveniently accessible using the *bytes.hex()* method.

If *sep* is specified, it must be a single character str or bytes object. It will be inserted in the output after every *bytes_per_sep* input bytes. Separator placement is counted from the right end of the output by default, if you wish to count from the left, supply a negative *bytes_per_sep* value.

```
>>> import binascii
>>> binascii.b2a_hex(b'\xb9\x01\xef')
b'b901ef'
>>> binascii.hexlify(b'\xb9\x01\xef', '-')
b'b9-01-ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b'_', 2)
b'b9_01ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b' ', -2)
b'b901 ef'
```

Changed in version 3.8: The *sep* and *bytes_per_sep* parameters were added.

binascii.**a2b_hex**(*hexstr*)

binascii.**unhexlify**(*hexstr*)

Return the binary data represented by the hexadecimal string *hexstr*. This function is the inverse of *b2a_hex()*. *hexstr* must contain an even number of hexadecimal digits (which can be upper or lower case), otherwise an *Error* exception is raised.

Similar functionality (accepting only text string arguments, but more liberal towards whitespace) is also accessible using the `bytes.fromhex()` class method.

exception binascii.**Error**

Exception raised on errors. These are usually programming errors.

exception binascii.**Incomplete**

Exception raised on incomplete data. These are usually not programming errors, but may be handled by reading a little more data and trying again.

See also

Module ***base64***
Support for RFC compliant base64-style encoding in base 16, 32, 64, and 85.

Module ***quopri***
Support for quoted-printable encoding used in MIME email messages.

19.7 quopri — Encode and decode MIME quoted-printable data

Source code: Lib/quopri.py

This module performs quoted-printable transport encoding and decoding, as defined in **RFC 1521**: "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies". The quoted-printable encoding is designed for data where there are relatively few nonprintable characters; the base64 encoding scheme available via the `base64` module is more compact if there are many such characters, as when sending a graphics file.

quopri.**decode**(*input*, *output*, *header=False*)

Decode the contents of the *input* file and write the resulting decoded binary data to the *output* file. *input* and *output* must be *binary file objects*. If the optional argument *header* is present and true, underscore will be decoded as space. This is used to decode "Q"-encoded headers as described in **RFC 1522**: "MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text".

quopri.**encode**(*input*, *output*, *quotetabs*, *header=False*)

Encode the contents of the *input* file and write the resulting quoted-printable data to the *output* file. *input* and *output* must be *binary file objects*. *quotetabs*, a non-optional flag which controls whether to encode embedded spaces and tabs; when true it encodes such embedded whitespace, and when false it leaves them unencoded. Note that spaces and tabs appearing at the end of lines are always encoded, as per **RFC 1521**. *header* is a flag which controls if spaces are encoded as underscores as per **RFC 1522**.

quopri.**decodestring**(*s*, *header=False*)

Like `decode()`, except that it accepts a source `bytes` and returns the corresponding decoded `bytes`.

quopri.**encodestring**(*s*, *quotetabs=False*, *header=False*)

Like `encode()`, except that it accepts a source `bytes` and returns the corresponding encoded `bytes`. By default, it sends a `False` value to *quotetabs* parameter of the `encode()` function.

See also

Module ***base64***
Encode and decode MIME base64 data

CHAPTER

TWENTY

STRUCTURED MARKUP PROCESSING TOOLS

Python supports a variety of modules to work with various forms of structured data markup. This includes modules to work with the Standard Generalized Markup Language (SGML) and the Hypertext Markup Language (HTML), and several interfaces for working with the Extensible Markup Language (XML).

20.1 `html` — HyperText Markup Language support

Source code: Lib/html/__init__.py

This module defines utilities to manipulate HTML.

`html.escape`(*s*, *quote=True*)

Convert the characters `&`, < and > in string *s* to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. If the optional flag *quote* is true, the characters (`"`) and (`'`) are also translated; this helps for inclusion in an HTML attribute value delimited by quotes, as in `<a href="...">`.

Added in version 3.2.

`html.unescape`(*s*)

Convert all named and numeric character references (e.g. `>`, `>`, `>`) in the string *s* to the corresponding Unicode characters. This function uses the rules defined by the HTML 5 standard for both valid and invalid character references, and the *list of HTML 5 named character references*.

Added in version 3.4.

Submodules in the `html` package are:

- *`html.parser`* – HTML/XHTML parser with lenient parsing mode
- *`html.entities`* – HTML entity definitions

20.2 `html.parser` — Simple HTML and XHTML parser

Source code: Lib/html/parser.py

This module defines a class *`HTMLParser`* which serves as the basis for parsing text files formatted in HTML (HyperText Mark-up Language) and XHTML.

class html.parser.**HTMLParser**(*, *convert_charrefs=True*)

Create a parser instance able to parse invalid markup.

If *convert_charrefs* is `True` (the default), all character references (except the ones in `script`/`style` elements) are automatically converted to the corresponding Unicode characters.

An *HTMLParser* instance is fed HTML data and calls handler methods when start tags, end tags, text, comments, and other markup elements are encountered. The user should subclass *HTMLParser* and override its methods to implement the desired behavior.

This parser does not check that end tags match start tags or call the end-tag handler for elements which are closed implicitly by closing an outer element.

Changed in version 3.4: *convert_charrefs* keyword argument added.

Changed in version 3.5: The default value for argument *convert_charrefs* is now `True`.

20.2.1 Example HTML Parser Application

As a basic example, below is a simple HTML parser that uses the *HTMLParser* class to print out start tags, end tags, and data as they are encountered:

```
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Encountered a start tag:", tag)

    def handle_endtag(self, tag):
        print("Encountered an end tag :", tag)

    def handle_data(self, data):
        print("Encountered some data  :", data)

parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>'
            '<body><h1>Parse me!</h1></body></html>')
```

The output will then be:

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data  : Test
Encountered an end tag : title
Encountered an end tag : head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data  : Parse me!
Encountered an end tag : h1
Encountered an end tag : body
Encountered an end tag : html
```

20.2.2 HTMLParser Methods

HTMLParser instances have the following methods:

HTMLParser.**feed**(*data*)

Feed some text to the parser. It is processed insofar as it consists of complete elements; incomplete data is buffered until more data is fed or *close()* is called. *data* must be *str*.

HTMLParser.**close**()

Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call the *HTMLParser* base class method *close()*.

HTMLParser.**reset**()

Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.

HTMLParser.**getpos**()

Return current line number and offset.

HTMLParser.**get_starttag_text**()

Return the text of the most recently opened start tag. This should not normally be needed for structured processing, but may be useful in dealing with HTML "as deployed" or for re-generating input with minimal changes (whitespace between attributes can be preserved, etc.).

The following methods are called when data or markup elements are encountered and they are meant to be overridden in a subclass. The base class implementations do nothing (except for *handle_startendtag()*):

HTMLParser.**handle_starttag**(*tag*, *attrs*)

This method is called to handle the start tag of an element (e.g. `<div id="main">`).

The *tag* argument is the name of the tag converted to lower case. The *attrs* argument is a list of `(name, value)` pairs containing the attributes found inside the tag's <> brackets. The *name* will be translated to lower case, and quotes in the *value* have been removed, and character and entity references have been replaced.

For instance, for the tag `<A HREF="https://www.cwi.nl/">`, this method would be called as `handle_starttag('a', [('href', 'https://www.cwi.nl/')])`.

All entity references from *html.entities* are replaced in the attribute values.

HTMLParser.**handle_endtag**(*tag*)

This method is called to handle the end tag of an element (e.g. `</div>`).

The *tag* argument is the name of the tag converted to lower case.

HTMLParser.**handle_startendtag**(*tag*, *attrs*)

Similar to *handle_starttag()*, but called when the parser encounters an XHTML-style empty tag (`<img ... />`). This method may be overridden by subclasses which require this particular lexical information; the default implementation simply calls *handle_starttag()* and *handle_endtag()*.

HTMLParser.**handle_data**(*data*)

This method is called to process arbitrary data (e.g. text nodes and the content of `<script>...</script>` and `<style>...</style>`).

HTMLParser.**handle_entityref**(*name*)

This method is called to process a named character reference of the form `&name;` (e.g. `>`), where *name* is a general entity reference (e.g. `'gt'`). This method is never called if *convert_charrefs* is `True`.

HTMLParser.**handle_charref**(*name*)

This method is called to process decimal and hexadecimal numeric character references of the form `&#NNN;` and `&#xNNN;`. For example, the decimal equivalent for `>` is `>`, whereas the hexadecimal is `>`; in this case the method will receive `'62'` or `'x3E'`. This method is never called if *convert_charrefs* is `True`.

HTMLParser.**handle_comment**(*data*)

This method is called when a comment is encountered (e.g. `<!--comment-->`).

For example, the comment `<!-- comment -->` will cause this method to be called with the argument `' comment '`.

The content of Internet Explorer conditional comments (condcoms) will also be sent to this method, so, for `<!--[if IE 9]>IE9-specific content<![endif]-->`, this method will receive `'[if IE 9]>IE9-specific content<![endif]'`.

HTMLParser.**handle_decl**(*decl*)

This method is called to handle an HTML doctype declaration (e.g. `<!DOCTYPE html>`).

The *decl* parameter will be the entire contents of the declaration inside the `<!...>` markup (e.g. `'DOCTYPE html'`).

HTMLParser.**handle_pi**(*data*)

Method called when a processing instruction is encountered. The *data* parameter will contain the entire processing instruction. For example, for the processing instruction `<?proc color='red'>`, this method would be called as `handle_pi("proc color='red'")`. It is intended to be overridden by a derived class; the base class implementation does nothing.

> **Note**
>
> The *HTMLParser* class uses the SGML syntactic rules for processing instructions. An XHTML processing instruction using the trailing `'?'` will cause the `'?'` to be included in *data*.

HTMLParser.**unknown_decl**(*data*)

This method is called when an unrecognized declaration is read by the parser.

The *data* parameter will be the entire contents of the declaration inside the `<![...]>` markup. It is sometimes useful to be overridden by a derived class. The base class implementation does nothing.

20.2.3 Examples

The following class implements a parser that will be used to illustrate more examples:

```
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Start tag:", tag)
        for attr in attrs:
            print("     attr:", attr)

    def handle_endtag(self, tag):
        print("End tag  :", tag)

    def handle_data(self, data):
```

(continues on next page)

(continued from previous page)

```
        print("Data     :", data)

    def handle_comment(self, data):
        print("Comment  :", data)

    def handle_entityref(self, name):
        c = chr(name2codepoint[name])
        print("Named ent:", c)

    def handle_charref(self, name):
        if name.startswith('x'):
            c = chr(int(name[1:], 16))
        else:
            c = chr(int(name))
        print("Num ent  :", c)

    def handle_decl(self, data):
        print("Decl     :", data)

parser = MyHTMLParser()
```

Parsing a doctype:

```
>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" '
...             '"http://www.w3.org/TR/html4/strict.dtd">')
Decl     : DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
↪html4/strict.dtd"
```

Parsing an element with a few attributes and a title:

```
>>> parser.feed('<img src="python-logo.png" alt="The Python logo">')
Start tag: img
     attr: ('src', 'python-logo.png')
     attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data     : Python
End tag  : h1
```

The content of `script` and `style` elements is returned as is, without further parsing:

```
>>> parser.feed('<style type="text/css">#python { color: green }</style>')
Start tag: style
     attr: ('type', 'text/css')
Data     : #python { color: green }
End tag  : style

>>> parser.feed('<script type="text/javascript">'
...             'alert("<strong>hello!</strong>");</script>')
Start tag: script
     attr: ('type', 'text/javascript')
Data     : alert("<strong>hello!</strong>");
End tag  : script
```

Parsing comments:

```
>>> parser.feed('<!-- a comment -->'
...             '<!--[if IE 9]>IE-specific content<![endif]-->')
Comment  :  a comment
Comment  : [if IE 9]>IE-specific content<![endif]
```

Parsing named and numeric character references and converting them to the correct char (note: these 3 references are all equivalent to `'>'`):

```
>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
Num ent  : >
Num ent  : >
```

Feeding incomplete chunks to *feed()* works, but *handle_data()* might be called more than once (unless *convert_charrefs* is set to `True`):

```
>>> for chunk in ['<sp', 'an>buff', 'ered ', 'text</s', 'pan>']:
...     parser.feed(chunk)
...
Start tag: span
Data     : buff
Data     : ered
Data     : text
End tag  : span
```

Parsing invalid HTML (e.g. unquoted attributes) also works:

```
>>> parser.feed('<p><a class=link href=#main>tag soup</p ></a>')
Start tag: p
Start tag: a
     attr: ('class', 'link')
     attr: ('href', '#main')
Data     : tag soup
End tag  : p
End tag  : a
```

20.3 `html.entities` — Definitions of HTML general entities

Source code: Lib/html/entities.py

This module defines four dictionaries, *html5*, *name2codepoint*, *codepoint2name*, and *entitydefs*.

`html.entities.`**`html5`**

A dictionary that maps HTML5 named character references[1] to the equivalent Unicode character(s), e.g. `html5['gt;'] == '>'`. Note that the trailing semicolon is included in the name (e.g. `'gt;'`), however some of the names are accepted by the standard even without the semicolon: in this case the name is present with and without the `';'`. See also *html.unescape()*.

Added in version 3.3.

`html.entities.`**`entitydefs`**

A dictionary mapping XHTML 1.0 entity definitions to their replacement text in ISO Latin-1.

[1] See https://html.spec.whatwg.org/multipage/named-characters.html#named-character-references

`html.entities.`**`name2codepoint`**
: A dictionary that maps HTML4 entity names to the Unicode code points.

`html.entities.`**`codepoint2name`**
: A dictionary that maps Unicode code points to HTML4 entity names.

20.4 XML Processing Modules

Source code: Lib/xml/

Python's interfaces for processing XML are grouped in the `xml` package.

> **Warning**
>
> The XML modules are not secure against erroneous or maliciously constructed data. If you need to parse untrusted or unauthenticated data see the *XML vulnerabilities* and *The defusedxml Package* sections.

It is important to note that modules in the *xml* package require that there be at least one SAX-compliant XML parser available. The Expat parser is included with Python, so the *xml.parsers.expat* module will always be available.

The documentation for the *xml.dom* and *xml.sax* packages are the definition of the Python bindings for the DOM and SAX interfaces.

The XML handling submodules are:

- *xml.etree.ElementTree*: the ElementTree API, a simple and lightweight XML processor
- *xml.dom*: the DOM API definition
- *xml.dom.minidom*: a minimal DOM implementation
- *xml.dom.pulldom*: support for building partial DOM trees
- *xml.sax*: SAX2 base classes and convenience functions
- *xml.parsers.expat*: the Expat parser binding

20.4.1 XML vulnerabilities

The XML processing modules are not secure against maliciously constructed data. An attacker can abuse XML features to carry out denial of service attacks, access local files, generate network connections to other machines, or circumvent firewalls.

The following table gives an overview of the known attacks and whether the various modules are vulnerable to them.

kind	sax	etree	minidom	pulldom	xmlrpc
billion laughs	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)
quadratic blowup	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)
external entity expansion	Safe (5)	Safe (2)	Safe (3)	Safe (5)	Safe (4)
DTD retrieval	Safe (5)	Safe	Safe	Safe (5)	Safe
decompression bomb	Safe	Safe	Safe	Safe	**Vulnerable**
large tokens	**Vulnerable** (6)	**Vulnerable** (6)	**Vulnerable** (6)	**Vulnerable** (6)	**Vulnerable** (6)

1. Expat 2.4.1 and newer is not vulnerable to the "billion laughs" and "quadratic blowup" vulnerabilities. Items still listed as vulnerable due to potential reliance on system-provided libraries. Check `pyexpat.EXPAT_VERSION`.
2. *xml.etree.ElementTree* doesn't expand external entities and raises a *ParseError* when an entity occurs.
3. *xml.dom.minidom* doesn't expand external entities and simply returns the unexpanded entity verbatim.
4. *xmlrpc.client* doesn't expand external entities and omits them.
5. Since Python 3.7.1, external general entities are no longer processed by default.
6. Expat 2.6.0 and newer is not vulnerable to denial of service through quadratic runtime caused by parsing large tokens. Items still listed as vulnerable due to potential reliance on system-provided libraries. Check `pyexpat.EXPAT_VERSION`.

billion laughs / exponential entity expansion
: The Billion Laughs attack – also known as exponential entity expansion – uses multiple levels of nested entities. Each entity refers to another entity several times, and the final entity definition contains a small string. The exponential expansion results in several gigabytes of text and consumes lots of memory and CPU time.

quadratic blowup entity expansion
: A quadratic blowup attack is similar to a Billion Laughs attack; it abuses entity expansion, too. Instead of nested entities it repeats one large entity with a couple of thousand chars over and over again. The attack isn't as efficient as the exponential case but it avoids triggering parser countermeasures that forbid deeply nested entities.

external entity expansion
: Entity declarations can contain more than just text for replacement. They can also point to external resources or local files. The XML parser accesses the resource and embeds the content into the XML document.

DTD retrieval
: Some XML libraries like Python's *xml.dom.pulldom* retrieve document type definitions from remote or local locations. The feature has similar implications as the external entity expansion issue.

decompression bomb
: Decompression bombs (aka ZIP bomb) apply to all XML libraries that can parse compressed XML streams such as gzipped HTTP streams or LZMA-compressed files. For an attacker it can reduce the amount of transmitted data by three magnitudes or more.

large tokens
: Expat needs to re-parse unfinished tokens; without the protection introduced in Expat 2.6.0, this can lead to quadratic runtime that can be used to cause denial of service in the application parsing XML. The issue is known as CVE-2023-52425.

The documentation for defusedxml on PyPI has further information about all known attack vectors with examples and references.

20.4.2 The `defusedxml` Package

defusedxml is a pure Python package with modified subclasses of all stdlib XML parsers that prevent any potentially malicious operation. Use of this package is recommended for any server code that parses untrusted XML data. The package also ships with example exploits and extended documentation on more XML exploits such as XPath injection.

20.5 xml.etree.ElementTree — The ElementTree XML API

Source code: Lib/xml/etree/ElementTree.py

The *xml.etree.ElementTree* module implements a simple and efficient API for parsing and creating XML data.

Changed in version 3.3: This module will use a fast implementation whenever available.

Deprecated since version 3.3: The `xml.etree.cElementTree` module is deprecated.

Warning

The *xml.etree.ElementTree* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML vulnerabilities*.

20.5.1 Tutorial

This is a short tutorial for using *xml.etree.ElementTree* (`ET` in short). The goal is to demonstrate some of the building blocks and basic concepts of the module.

XML tree and elements

XML is an inherently hierarchical data format, and the most natural way to represent it is with a tree. `ET` has two classes for this purpose - *ElementTree* represents the whole XML document as a tree, and *Element* represents a single node in this tree. Interactions with the whole document (reading and writing to/from files) are usually done on the *ElementTree* level. Interactions with a single XML element and its sub-elements are done on the *Element* level.

Parsing XML

We'll be using the fictive `country_data.xml` XML document as the sample data for this section:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank>68</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
```

(continues on next page)

(continued from previous page)

```
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

We can import this data by reading from a file:

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

Or directly from a string:

```
root = ET.fromstring(country_data_as_string)
```

fromstring() parses XML from a string directly into an *Element*, which is the root element of the parsed tree. Other parsing functions may create an *ElementTree*. Check the documentation to be sure.

As an *Element*, `root` has a tag and a dictionary of attributes:

```
>>> root.tag
'data'
>>> root.attrib
{}
```

It also has children nodes over which we can iterate:

```
>>> for child in root:
...     print(child.tag, child.attrib)
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

Children are nested, and we can access specific child nodes by index:

```
>>> root[0][1].text
'2008'
```

Note

Not all elements of the XML input will end up as elements of the parsed tree. Currently, this module skips over any XML comments, processing instructions, and document type declarations in the input. Nevertheless, trees built using this module's API rather than parsing from XML text can have comments and processing instructions in them; they will be included when generating XML output. A document type declaration may be accessed by passing a custom *TreeBuilder* instance to the *XMLParser* constructor.

Pull API for non-blocking parsing

Most parsing functions provided by this module require the whole document to be read at once before returning any result. It is possible to use an *XMLParser* and feed data into it incrementally, but it is a push API that calls methods on a callback target, which is too low-level and inconvenient for most needs. Sometimes what the user really wants is to be able to parse XML incrementally, without blocking operations, while enjoying the convenience of fully constructed *Element* objects.

The most powerful tool for doing this is *XMLPullParser*. It does not require a blocking read to obtain the XML data, and is instead fed with data incrementally with *XMLPullParser.feed()* calls. To get the parsed XML elements, call *XMLPullParser.read_events()*. Here is an example:

```
>>> parser = ET.XMLPullParser(['start', 'end'])
>>> parser.feed('<mytag>sometext')
>>> list(parser.read_events())
[('start', <Element 'mytag' at 0x7fa66db2be58>)]
>>> parser.feed(' more text</mytag>')
>>> for event, elem in parser.read_events():
...     print(event)
...     print(elem.tag, 'text=', elem.text)
...
end
mytag text= sometext more text
```

The obvious use case is applications that operate in a non-blocking fashion where the XML data is being received from a socket or read incrementally from some storage device. In such cases, blocking reads are unacceptable.

Because it's so flexible, *XMLPullParser* can be inconvenient to use for simpler use-cases. If you don't mind your application blocking on reading XML data but would still like to have incremental parsing capabilities, take a look at *iterparse()*. It can be useful when you're reading a large XML document and don't want to hold it wholly in memory.

Where *immediate* feedback through events is wanted, calling method *XMLPullParser.flush()* can help reduce delay; please make sure to study the related security notes.

Finding interesting elements

Element has some useful methods that help iterate recursively over all the sub-tree below it (its children, their children, and so on). For example, *Element.iter()*:

```
>>> for neighbor in root.iter('neighbor'):
...     print(neighbor.attrib)
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

Element.findall() finds only elements with a tag which are direct children of the current element. *Element.find()* finds the *first* child with a particular tag, and *Element.text* accesses the element's text content. *Element.get()* accesses the element's attributes:

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
```

(continues on next page)

(continued from previous page)

```
...     print(name, rank)
...
Liechtenstein 1
Singapore 4
Panama 68
```

More sophisticated specification of which elements to look for is possible by using *XPath*.

Modifying an XML File

ElementTree provides a simple way to build XML documents and write them to files. The *ElementTree.write()* method serves this purpose.

Once created, an *Element* object may be manipulated by directly changing its fields (such as *Element.text*), adding and modifying attributes (*Element.set()* method), as well as adding new children (for example with *Element.append()*).

Let's say we want to add one to each country's rank, and add an `updated` attribute to the rank element:

```
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
...
>>> tree.write('output.xml')
```

Our XML now looks like this:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank updated="yes">69</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

We can remove elements using *Element.remove()*. Let's say we want to remove all countries with a rank higher than 50:

```
>>> for country in root.findall('country'):
...     # using root.findall() to avoid removal during traversal
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')
```

Note that concurrent modification while iterating can lead to problems, just like when iterating and modifying Python lists or dicts. Therefore, the example first collects all matching elements with `root.findall()`, and only then iterates over the list of matches.

Our XML now looks like this:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
</data>
```

Building XML documents

The *SubElement()* function also provides a convenient way to create new sub-elements for a given element:

```
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /><c><d /></c></a>
```

Parsing XML with Namespaces

If the XML input has namespaces, tags and attributes with prefixes in the form `prefix:sometag` get expanded to `{uri}sometag` where the *prefix* is replaced by the full *URI*. Also, if there is a default namespace, that full URI gets prepended to all of the non-prefixed tags.

Here is an XML example that incorporates two namespaces, one with the prefix "fictional" and the other serving as the default namespace:

```
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com"
        xmlns="http://people.example.com">
```

(continues on next page)

(continued from previous page)

```
    <actor>
        <name>John Cleese</name>
        <fictional:character>Lancelot</fictional:character>
        <fictional:character>Archie Leach</fictional:character>
    </actor>
    <actor>
        <name>Eric Idle</name>
        <fictional:character>Sir Robin</fictional:character>
        <fictional:character>Gunther</fictional:character>
        <fictional:character>Commander Clement</fictional:character>
    </actor>
</actors>
```

One way to search and explore this XML example is to manually add the URI to every tag or attribute in the xpath of a *find()* or *findall()*:

```
root = fromstring(xml_text)
for actor in root.findall('{http://people.example.com}actor'):
    name = actor.find('{http://people.example.com}name')
    print(name.text)
    for char in actor.findall('{http://characters.example.com}character'):
        print(' |-->', char.text)
```

A better way to search the namespaced XML example is to create a dictionary with your own prefixes and use those in the search functions:

```
ns = {'real_person': 'http://people.example.com',
      'role': 'http://characters.example.com'}

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print(name.text)
    for char in actor.findall('role:character', ns):
        print(' |-->', char.text)
```

These two approaches both output:

```
John Cleese
 |--> Lancelot
 |--> Archie Leach
Eric Idle
 |--> Sir Robin
 |--> Gunther
 |--> Commander Clement
```

20.5.2 XPath support

This module provides limited support for XPath expressions for locating elements in a tree. The goal is to support a small subset of the abbreviated syntax; a full XPath engine is outside the scope of the module.

Example

Here's an example that demonstrates some of the XPath capabilities of the module. We'll be using the `countrydata` XML document from the *Parsing XML* section:

```
import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children of the top-level
# elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall(".//year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
root.findall(".//*[@name='Singapore']/year")

# All 'neighbor' nodes that are the second child of their parent
root.findall(".//neighbor[2]")
```

For XML with namespaces, use the usual qualified `{namespace}tag` notation:

```
# All dublin-core "title" tags in the document
root.findall(".//{http://purl.org/dc/elements/1.1/}title")
```

Supported XPath syntax

Syntax	Meaning
`tag`	Selects all child elements with the given tag. For example, `spam` selects all child elements named `spam`, and `spam/egg` selects all grandchildren named `egg` in all children named `spam`. `{namespace}*` selects all tags in the given namespace, `{*}spam` selects tags named `spam` in any (or no) namespace, and `{}*` only selects tags that are not in a namespace. Changed in version 3.8: Support for star-wildcards was added.
`*`	Selects all child elements, including comments and processing instructions. For example, `*/egg` selects all grandchildren named `egg`.
`.`	Selects the current node. This is mostly useful at the beginning of the path, to indicate that it's a relative path.
`//`	Selects all subelements, on all levels beneath the current element. For example, `.//egg` selects all `egg` elements in the entire tree.
`..`	Selects the parent element. Returns `None` if the path attempts to reach the ancestors of the start element (the element `find` was called on).
`[@attrib]`	Selects all elements that have the given attribute.
`[@attrib='value']`	Selects all elements for which the given attribute has the given value. The value cannot contain quotes.
`[@attrib!='value']`	Selects all elements for which the given attribute does not have the given value. The value cannot contain quotes. Added in version 3.10.
`[tag]`	Selects all elements that have a child named `tag`. Only immediate children are supported.
`[.='text']`	Selects all elements whose complete text content, including descendants, equals the given `text`. Added in version 3.7.
`[.!='text']`	Selects all elements whose complete text content, including descendants, does not equal the given `text`. Added in version 3.10.
`[tag='text']`	Selects all elements that have a child named `tag` whose complete text content, including descendants, equals the given `text`.
`[tag!='text']`	Selects all elements that have a child named `tag` whose complete text content, including descendants, does not equal the given `text`. Added in version 3.10.
`[position]`	Selects all elements that are located at the given position. The position can be either an integer (1 is the first position), the expression `last()` (for the last position), or a position relative to the last position (e.g. `last()-1`).

Predicates (expressions within square brackets) must be preceded by a tag name, an asterisk, or another predicate. `position` predicates must be preceded by a tag name.

20.5.3 Reference

Functions

xml.etree.ElementTree.**canonicalize**(*xml_data=None*, *, *out=None*, *from_file=None*, ***options*)

C14N 2.0 transformation function.

Canonicalization is a way to normalise XML output in a way that allows byte-by-byte comparisons and digital signatures. It reduces the freedom that XML serializers have and instead generates a more constrained XML representation. The main restrictions regard the placement of namespace declarations, the ordering of attributes, and ignorable whitespace.

This function takes an XML data string (*xml_data*) or a file path or file-like object (*from_file*) as input, converts it to the canonical form, and writes it out using the *out* file(-like) object, if provided, or returns it as a text string if not. The output file receives text, not bytes. It should therefore be opened in text mode with `utf-8` encoding.

Typical uses:

```
xml_data = "<root>...</root>"
print(canonicalize(xml_data))

with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:
    canonicalize(xml_data, out=out_file)

with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:
    canonicalize(from_file="inputfile.xml", out=out_file)
```

The configuration *options* are as follows:

- *with_comments*: set to true to include comments (default: false)
- ***strip_text*: set to true to strip whitespace before and after text content**
  (default: false)
- ***rewrite_prefixes*: set to true to replace namespace prefixes by "n{number}"**
  (default: false)
- ***qname_aware_tags*: a set of qname aware tag names in which prefixes**
  should be replaced in text content (default: empty)
- ***qname_aware_attrs*: a set of qname aware attribute names in which prefixes**
  should be replaced in text content (default: empty)
- *exclude_attrs*: a set of attribute names that should not be serialised
- *exclude_tags*: a set of tag names that should not be serialised

In the option list above, "a set" refers to any collection or iterable of strings, no ordering is expected.

Added in version 3.8.

xml.etree.ElementTree.**Comment**(*text=None*)

Comment element factory. This factory function creates a special element that will be serialized as an XML comment by the standard serializer. The comment string can be either a bytestring or a Unicode string. *text* is a string containing the comment string. Returns an element instance representing a comment.

Note that *XMLParser* skips over comments in the input instead of creating comment objects for them. An *ElementTree* will only contain comment nodes if they have been inserted into to the tree using one of the *Element* methods.

xml.etree.ElementTree.**dump**(*elem*)

Writes an element tree or element structure to sys.stdout. This function should be used for debugging only.

The exact output format is implementation dependent. In this version, it's written as an ordinary XML file.

elem is an element tree or an individual element.

Changed in version 3.8: The *dump()* function now preserves the attribute order specified by the user.

xml.etree.ElementTree.**fromstring**(*text*, *parser=None*)

Parses an XML section from a string constant. Same as *XML()*. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns an *Element* instance.

xml.etree.ElementTree.**fromstringlist**(*sequence*, *parser=None*)

Parses an XML document from a sequence of string fragments. *sequence* is a list or other sequence containing XML data fragments. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns an *Element* instance.

Added in version 3.2.

xml.etree.ElementTree.**indent**(*tree*, *space=' '*, *level=0*)

Appends whitespace to the subtree to indent the tree visually. This can be used to generate pretty-printed XML output. *tree* can be an Element or ElementTree. *space* is the whitespace string that will be inserted for each indentation level, two space characters by default. For indenting partial subtrees inside of an already indented tree, pass the initial indentation level as *level*.

Added in version 3.9.

xml.etree.ElementTree.**iselement**(*element*)

Check if an object appears to be a valid element object. *element* is an element instance. Return `True` if this is an element object.

xml.etree.ElementTree.**iterparse**(*source*, *events=None*, *parser=None*)

Parses an XML section into an element tree incrementally, and reports what's going on to the user. *source* is a filename or *file object* containing XML data. *events* is a sequence of events to report back. The supported events are the strings `"start"`, `"end"`, `"comment"`, `"pi"`, `"start-ns"` and `"end-ns"` (the "ns" events are used to get detailed namespace information). If *events* is omitted, only `"end"` events are reported. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. *parser* must be a subclass of *XMLParser* and can only use the default *TreeBuilder* as a target. Returns an *iterator* providing `(event, elem)` pairs; it has a `root` attribute that references the root element of the resulting XML tree once *source* is fully read. The iterator has the `close()` method that closes the internal file object if *source* is a filename.

Note that while *iterparse()* builds the tree incrementally, it issues blocking reads on *source* (or the file it names). As such, it's unsuitable for applications where blocking reads can't be made. For fully non-blocking parsing, see *XMLPullParser*.

> **Note**
>
> *iterparse()* only guarantees that it has seen the ">" character of a starting tag when it emits a "start" event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present.
>
> If you need a fully populated element, look for "end" events instead.

Deprecated since version 3.4: The *parser* argument.

Changed in version 3.8: The `comment` and `pi` events were added.

Changed in version 3.13: Added the `close()` method.

xml.etree.ElementTree.**parse**(*source*, *parser=None*)

Parses an XML section into an element tree. *source* is a filename or file object containing XML data. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `ElementTree` instance.

xml.etree.ElementTree.**ProcessingInstruction**(*target*, *text=None*)

PI element factory. This factory function creates a special element that will be serialized as an XML processing instruction. *target* is a string containing the PI target. *text* is a string containing the PI contents, if given. Returns an element instance, representing a processing instruction.

Note that `XMLParser` skips over processing instructions in the input instead of creating PI objects for them. An `ElementTree` will only contain processing instruction nodes if they have been inserted into to the tree using one of the `Element` methods.

xml.etree.ElementTree.**register_namespace**(*prefix*, *uri*)

Registers a namespace prefix. The registry is global, and any existing mapping for either the given prefix or the namespace URI will be removed. *prefix* is a namespace prefix. *uri* is a namespace uri. Tags and attributes in this namespace will be serialized with the given prefix, if at all possible.

Added in version 3.2.

xml.etree.ElementTree.**SubElement**(*parent*, *tag*, *attrib={}*, ***extra*)

Subelement factory. This function creates an element instance, and appends it to an existing element.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. *parent* is the parent element. *tag* is the subelement name. *attrib* is an optional dictionary, containing element attributes. *extra* contains additional attributes, given as keyword arguments. Returns an element instance.

xml.etree.ElementTree.**tostring**(*element*, *encoding='us-ascii'*, *method='xml'*, ***, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)

Generates a string representation of an XML element, including all subelements. *element* is an `Element` instance. *encoding*[1] is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). *xml_declaration*, *default_namespace* and *short_empty_elements* has the same meaning as in `ElementTree.write()`. Returns an (optionally) encoded string containing the XML data.

Changed in version 3.4: Added the *short_empty_elements* parameter.

Changed in version 3.8: Added the *xml_declaration* and *default_namespace* parameters.

Changed in version 3.8: The `tostring()` function now preserves the attribute order specified by the user.

xml.etree.ElementTree.**tostringlist**(*element*, *encoding='us-ascii'*, *method='xml'*, ***, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)

Generates a string representation of an XML element, including all subelements. *element* is an `Element` instance. *encoding*[1] is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). *xml_declaration*, *default_namespace* and *short_empty_elements* has the same meaning as in `ElementTree.write()`. Returns a list of (optionally) encoded strings containing the XML data. It does not guarantee any specific sequence, except that `b"".join(tostringlist(element)) == tostring(element)`.

Added in version 3.2.

Changed in version 3.4: Added the *short_empty_elements* parameter.

[1] The encoding string included in XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

Changed in version 3.8: Added the *xml_declaration* and *default_namespace* parameters.

Changed in version 3.8: The `tostringlist()` function now preserves the attribute order specified by the user.

`xml.etree.ElementTree.`**`XML`**(*text*, *parser=None*)

Parses an XML section from a string constant. This function can be used to embed "XML literals" in Python code. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `Element` instance.

`xml.etree.ElementTree.`**`XMLID`**(*text*, *parser=None*)

Parses an XML section from a string constant, and also returns a dictionary which maps from element id:s to elements. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns a tuple containing an `Element` instance and a dictionary.

20.5.4 XInclude support

This module provides limited support for XInclude directives, via the `xml.etree.ElementInclude` helper module. This module can be used to insert subtrees and text strings into element trees, based on information in the tree.

Example

Here's an example that demonstrates use of the XInclude module. To include an XML document in the current document, use the `{http://www.w3.org/2001/XInclude}include` element and set the **parse** attribute to `"xml"`, and use the **href** attribute to specify the document to include.

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="source.xml" parse="xml" />
</document>
```

By default, the **href** attribute is treated as a file name. You can use custom loaders to override this behaviour. Also note that the standard helper does not support XPointer syntax.

To process this file, load it as usual, and pass the root element to the `xml.etree.ElementTree` module:

```
from xml.etree import ElementTree, ElementInclude

tree = ElementTree.parse("document.xml")
root = tree.getroot()

ElementInclude.include(root)
```

The ElementInclude module replaces the `{http://www.w3.org/2001/XInclude}include` element with the root element from the **source.xml** document. The result might look something like this:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <para>This is a paragraph.</para>
</document>
```

If the **parse** attribute is omitted, it defaults to "xml". The href attribute is required.

To include a text document, use the `{http://www.w3.org/2001/XInclude}include` element, and set the **parse** attribute to "text":

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) <xi:include href="year.txt" parse="text" />.
</document>
```

The result might look something like:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) 2003.
</document>
```

20.5.5 Reference

Functions

xml.etree.ElementInclude.**default_loader**(*href*, *parse*, *encoding=None*)

Default loader. This default loader reads an included resource from disk. *href* is a URL. *parse* is for parse mode either "xml" or "text". *encoding* is an optional text encoding. If not given, encoding is utf-8. Returns the expanded resource. If the parse mode is "xml", this is an *Element* instance. If the parse mode is "text", this is a string. If the loader fails, it can return None or raise an exception.

xml.etree.ElementInclude.**include**(*elem*, *loader=None*, *base_url=None*, *max_depth=6*)

This function expands XInclude directives in-place in tree pointed by *elem*. *elem* is either the root *Element* or an *ElementTree* instance to find such element. *loader* is an optional resource loader. If omitted, it defaults to *default_loader()*. If given, it should be a callable that implements the same interface as *default_loader()*. *base_url* is base URL of the original file, to resolve relative include file references. *max_depth* is the maximum number of recursive inclusions. Limited to reduce the risk of malicious content explosion. Pass None to disable the limitation.

Changed in version 3.9: Added the *base_url* and *max_depth* parameters.

Element Objects

class xml.etree.ElementTree.**Element**(*tag*, *attrib={}*, ***extra*)

Element class. This class defines the Element interface, and provides a reference implementation of this interface.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. *tag* is the element name. *attrib* is an optional dictionary, containing element attributes. *extra* contains additional attributes, given as keyword arguments.

tag

A string identifying what kind of data this element represents (the element type, in other words).

text

tail

These attributes can be used to hold additional data associated with the element. Their values are usually strings but may be any application-specific object. If the element is created from an XML file, the *text* attribute holds either the text between the element's start tag and its first child or end tag, or None, and the *tail* attribute holds either the text between the element's end tag and the next tag, or None. For the XML data

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

the *a* element has `None` for both *text* and *tail* attributes, the *b* element has *text* `"1"` and *tail* `"4"`, the *c* element has *text* `"2"` and *tail* `None`, and the *d* element has *text* `None` and *tail* `"3"`.

To collect the inner text of an element, see *itertext()*, for example `"".join(element.itertext())`.

Applications may store arbitrary objects in these attributes.

attrib

A dictionary containing the element's attributes. Note that while the *attrib* value is always a real mutable Python dictionary, an ElementTree implementation may choose to use another internal representation, and create the dictionary only if someone asks for it. To take advantage of such implementations, use the dictionary methods below whenever possible.

The following dictionary-like methods work on the element attributes.

clear()

Resets an element. This function removes all subelements, clears all attributes, and sets the text and tail attributes to `None`.

get(*key*, *default=None*)

Gets the element attribute named *key*.

Returns the attribute value, or *default* if the attribute was not found.

items()

Returns the element attributes as a sequence of (name, value) pairs. The attributes are returned in an arbitrary order.

keys()

Returns the elements attribute names as a list. The names are returned in an arbitrary order.

set(*key*, *value*)

Set the attribute *key* on the element to *value*.

The following methods work on the element's children (subelements).

append(*subelement*)

Adds the element *subelement* to the end of this element's internal list of subelements. Raises *TypeError* if *subelement* is not an *Element*.

extend(*subelements*)

Appends *subelements* from a sequence object with zero or more elements. Raises *TypeError* if a subelement is not an *Element*.

Added in version 3.2.

find(*match*, *namespaces=None*)

Finds the first subelement matching *match*. *match* may be a tag name or a *path*. Returns an element instance or `None`. *namespaces* is an optional mapping from namespace prefix to full name. Pass `''` as prefix to move all unprefixed tag names in the expression into the given namespace.

findall(*match*, *namespaces=None*)

Finds all matching subelements, by tag name or *path*. Returns a list containing all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name. Pass `''` as prefix to move all unprefixed tag names in the expression into the given namespace.

findtext(*match*, *default=None*, *namespaces=None*)

Finds text for the first subelement matching *match*. *match* may be a tag name or a *path*. Returns the text content of the first matching element, or *default* if no element was found. Note that if the matching element has no text content an empty string is returned. *namespaces* is an optional mapping from namespace prefix to full name. Pass `''` as prefix to move all unprefixed tag names in the expression into the given namespace.

insert(*index*, *subelement*)

Inserts *subelement* at the given position in this element. Raises *TypeError* if *subelement* is not an *Element*.

iter(*tag=None*)

Creates a tree *iterator* with the current element as the root. The iterator iterates over this element and all elements below it, in document (depth first) order. If *tag* is not `None` or `'*'`, only elements whose tag equals *tag* are returned from the iterator. If the tree structure is modified during iteration, the result is undefined.

Added in version 3.2.

iterfind(*match*, *namespaces=None*)

Finds all matching subelements, by tag name or *path*. Returns an iterable yielding all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name.

Added in version 3.2.

itertext()

Creates a text iterator. The iterator loops over this element and all subelements, in document order, and returns all inner text.

Added in version 3.2.

makeelement(*tag*, *attrib*)

Creates a new element object of the same type as this element. Do not call this method, use the *SubElement()* factory function instead.

remove(*subelement*)

Removes *subelement* from the element. Unlike the find* methods this method compares elements based on the instance identity, not on tag value or contents.

Element objects also support the following sequence type methods for working with subelements: `__delitem__()`, `__getitem__()`, `__setitem__()`, `__len__()`.

Caution: Elements with no subelements will test as `False`. In a future release of Python, all elements will test as `True` regardless of whether subelements exist. Instead, prefer explicit `len(elem)` or `elem is not None` tests.:

```
element = root.find('foo')

if not element:  # careful!
    print("element not found, or element has no subelements")

if element is None:
    print("element not found")
```

Changed in version 3.12: Testing the truth value of an Element emits *DeprecationWarning*.

Prior to Python 3.8, the serialisation order of the XML attributes of elements was artificially made predictable by sorting the attributes by their name. Based on the now guaranteed ordering of dicts, this arbitrary reordering was removed in Python 3.8 to preserve the order in which attributes were originally parsed or created by user code.

In general, user code should try not to depend on a specific ordering of attributes, given that the XML Information Set explicitly excludes the attribute order from conveying information. Code should be prepared to deal with any

ordering on input. In cases where deterministic XML output is required, e.g. for cryptographic signing or test data sets, canonical serialisation is available with the *canonicalize()* function.

In cases where canonical output is not applicable but a specific attribute order is still desirable on output, code should aim for creating the attributes directly in the desired order, to avoid perceptual mismatches for readers of the code. In cases where this is difficult to achieve, a recipe like the following can be applied prior to serialisation to enforce an order independently from the Element creation:

```
def reorder_attributes(root):
    for el in root.iter():
        attrib = el.attrib
        if len(attrib) > 1:
            # adjust attribute order, e.g. by sorting
            attribs = sorted(attrib.items())
            attrib.clear()
            attrib.update(attribs)
```

ElementTree Objects

class xml.etree.ElementTree.**ElementTree**(*element=None*, *file=None*)

ElementTree wrapper class. This class represents an entire element hierarchy, and adds some extra support for serialization to and from standard XML.

element is the root element. The tree is initialized with the contents of the XML *file* if given.

_setroot(*element*)

Replaces the root element for this tree. This discards the current contents of the tree, and replaces it with the given element. Use with care. *element* is an element instance.

find(*match*, *namespaces=None*)

Same as *Element.find()*, starting at the root of the tree.

findall(*match*, *namespaces=None*)

Same as *Element.findall()*, starting at the root of the tree.

findtext(*match*, *default=None*, *namespaces=None*)

Same as *Element.findtext()*, starting at the root of the tree.

getroot()

Returns the root element for this tree.

iter(*tag=None*)

Creates and returns a tree iterator for the root element. The iterator loops over all elements in this tree, in section order. *tag* is the tag to look for (default is to return all elements).

iterfind(*match*, *namespaces=None*)

Same as *Element.iterfind()*, starting at the root of the tree.

Added in version 3.2.

parse(*source*, *parser=None*)

Loads an external XML section into this element tree. *source* is a file name or *file object*. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns the section root element.

write(*file*, *encoding='us-ascii'*, *xml_declaration=None*, *default_namespace=None*, *method='xml'*, ***, *short_empty_elements=True*)

Writes the element tree to a file, as XML. *file* is a file name, or a *file object* opened for writing. *encoding*[Page 1391, 1] is the output encoding (default is US-ASCII). *xml_declaration* controls if an XML declaration should be added to the file. Use `False` for never, `True` for always, `None` for only if not US-ASCII or UTF-8 or Unicode (default is `None`). *default_namespace* sets the default XML namespace (for "xmlns"). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). The keyword-only *short_empty_elements* parameter controls the formatting of elements that contain no content. If `True` (the default), they are emitted as a single self-closed tag, otherwise they are emitted as a pair of start/end tags.

The output is either a string (*str*) or binary (*bytes*). This is controlled by the *encoding* argument. If *encoding* is `"unicode"`, the output is a string; otherwise, it's binary. Note that this may conflict with the type of *file* if it's an open *file object*; make sure you do not try to write a string to a binary stream and vice versa.

Changed in version 3.4: Added the *short_empty_elements* parameter.

Changed in version 3.8: The *write()* method now preserves the attribute order specified by the user.

This is the XML file that is going to be manipulated:

```
<html>
    <head>
        <title>Example page</title>
    </head>
    <body>
        <p>Moved to <a href="http://example.org/">example.org</a>
        or <a href="http://example.com/">example.com</a>.</p>
    </body>
</html>
```

Example of changing the attribute "target" of every link in first paragraph:

```
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")      # Finds first occurrence of tag p in body
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))    # Returns list of all links
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:              # Iterates through all found links
...     i.attrib["target"] = "blank"
...
>>> tree.write("output.xhtml")
```

QName Objects

class xml.etree.ElementTree.**QName**(*text_or_uri*, *tag=None*)

QName wrapper. This can be used to wrap a QName attribute value, in order to get proper namespace handling on output. *text_or_uri* is a string containing the QName value, in the form {uri}local, or, if the tag argument is given, the URI part of a QName. If *tag* is given, the first argument is interpreted as a URI, and this argument is interpreted as a local name. *QName* instances are opaque.

TreeBuilder Objects

class xml.etree.ElementTree.**TreeBuilder**(*element_factory=None*, *, *comment_factory=None*, *pi_factory=None*, *insert_comments=False*, *insert_pis=False*)

Generic element structure builder. This builder converts a sequence of start, data, end, comment and pi method calls to a well-formed element structure. You can use this class to build an element structure using a custom XML parser, or a parser for some other XML-like format.

element_factory, when given, must be a callable accepting two positional arguments: a tag and a dict of attributes. It is expected to return a new element instance.

The *comment_factory* and *pi_factory* functions, when given, should behave like the `Comment()` and `ProcessingInstruction()` functions to create comments and processing instructions. When not given, the default factories will be used. When *insert_comments* and/or *insert_pis* is true, comments/pis will be inserted into the tree if they appear within the root element (but not outside of it).

close()

Flushes the builder buffers, and returns the toplevel document element. Returns an `Element` instance.

data(*data*)

Adds text to the current element. *data* is a string. This should be either a bytestring, or a Unicode string.

end(*tag*)

Closes the current element. *tag* is the element name. Returns the closed element.

start(*tag*, *attrs*)

Opens a new element. *tag* is the element name. *attrs* is a dictionary containing element attributes. Returns the opened element.

comment(*text*)

Creates a comment with the given *text*. If `insert_comments` is true, this will also add it to the tree.

Added in version 3.8.

pi(*target*, *text*)

Creates a process instruction with the given *target* name and *text*. If `insert_pis` is true, this will also add it to the tree.

Added in version 3.8.

In addition, a custom `TreeBuilder` object can provide the following methods:

doctype(*name*, *pubid*, *system*)

Handles a doctype declaration. *name* is the doctype name. *pubid* is the public identifier. *system* is the system identifier. This method does not exist on the default `TreeBuilder` class.

Added in version 3.2.

start_ns(*prefix*, *uri*)

Is called whenever the parser encounters a new namespace declaration, before the `start()` callback for the opening element that defines it. *prefix* is `''` for the default namespace and the declared namespace prefix name otherwise. *uri* is the namespace URI.

Added in version 3.8.

end_ns(*prefix*)

Is called after the `end()` callback of an element that declared a namespace prefix mapping, with the name of the *prefix* that went out of scope.

Added in version 3.8.

class xml.etree.ElementTree.**C14NWriterTarget**(*write*, *, *with_comments=False*, *strip_text=False*, *rewrite_prefixes=False*, *qname_aware_tags=None*, *qname_aware_attrs=None*, *exclude_attrs=None*, *exclude_tags=None*)

A C14N 2.0 writer. Arguments are the same as for the *canonicalize()* function. This class does not build a tree but translates the callback events directly into a serialised form using the *write* function.

Added in version 3.8.

XMLParser Objects

class xml.etree.ElementTree.**XMLParser**(*, *target=None*, *encoding=None*)

This class is the low-level building block of the module. It uses *xml.parsers.expat* for efficient, event-based parsing of XML. It can be fed XML data incrementally with the *feed()* method, and parsing events are translated to a push API - by invoking callbacks on the *target* object. If *target* is omitted, the standard *TreeBuilder* is used. If *encoding*[Page 1391, 1] is given, the value overrides the encoding specified in the XML file.

Changed in version 3.8: Parameters are now *keyword-only*. The *html* argument no longer supported.

close()

Finishes feeding data to the parser. Returns the result of calling the `close()` method of the *target* passed during construction; by default, this is the toplevel document element.

feed(*data*)

Feeds data to the parser. *data* is encoded data.

flush()

Triggers parsing of any previously fed unparsed data, which can be used to ensure more immediate feedback, in particular with Expat >=2.6.0. The implementation of *flush()* temporarily disables reparse deferral with Expat (if currently enabled) and triggers a reparse. Disabling reparse deferral has security consequences; please see *xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()* for details.

Note that *flush()* has been backported to some prior releases of CPython as a security fix. Check for availability of *flush()* using *hasattr()* if used in code running across a variety of Python versions.

Added in version 3.13.

XMLParser.feed() calls *target*'s `start(tag, attrs_dict)` method for each opening tag, its `end(tag)` method for each closing tag, and data is processed by method `data(data)`. For further supported callback methods, see the *TreeBuilder* class. *XMLParser.close()* calls *target*'s method `close()`. *XMLParser* can be used not only for building a tree structure. This is an example of counting the maximum depth of an XML file:

```
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:                     # The target object of the parser
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):   # Called for each opening tag.
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):             # Called for each closing tag.
...         self.depth -= 1
...     def data(self, data):
...         pass            # We do not need to do anything with data.
```

(continues on next page)

(continued from previous page)

```
...     def close(self):    # Called when all data has been parsed.
...         return self.maxDepth
...
>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """
... <a>
...   <b>
...   </b>
...   <b>
...     <c>
...       <d>
...       </d>
...     </c>
...   </b>
... </a>"""
>>> parser.feed(exampleXml)
>>> parser.close()
4
```

XMLPullParser Objects

class xml.etree.ElementTree.**XMLPullParser**(*events=None*)

A pull parser suitable for non-blocking applications. Its input-side API is similar to that of *XMLParser*, but instead of pushing calls to a callback target, *XMLPullParser* collects an internal list of parsing events and lets the user read from it. *events* is a sequence of events to report back. The supported events are the strings `"start"`, `"end"`, `"comment"`, `"pi"`, `"start-ns"` and `"end-ns"` (the "ns" events are used to get detailed namespace information). If *events* is omitted, only `"end"` events are reported.

feed(*data*)

Feed the given bytes data to the parser.

flush()

Triggers parsing of any previously fed unparsed data, which can be used to ensure more immediate feedback, in particular with Expat >=2.6.0. The implementation of *flush()* temporarily disables reparse deferral with Expat (if currently enabled) and triggers a reparse. Disabling reparse deferral has security consequences; please see *xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()* for details.

Note that *flush()* has been backported to some prior releases of CPython as a security fix. Check for availability of *flush()* using *hasattr()* if used in code running across a variety of Python versions.

Added in version 3.13.

close()

Signal the parser that the data stream is terminated. Unlike *XMLParser.close()*, this method always returns *None*. Any events not yet retrieved when the parser is closed can still be read with *read_events()*.

read_events()

Return an iterator over the events which have been encountered in the data fed to the parser. The iterator yields `(event, elem)` pairs, where *event* is a string representing the type of event (e.g. `"end"`) and *elem* is the encountered *Element* object, or other context value as follows.

- `start`, `end`: the current Element.
- `comment`, `pi`: the current comment / processing instruction

- `start-ns`: a tuple `(prefix, uri)` naming the declared namespace mapping.
- `end-ns`: *None* (this may change in a future version)

Events provided in a previous call to *read_events()* will not be yielded again. Events are consumed from the internal queue only when they are retrieved from the iterator, so multiple readers iterating in parallel over iterators obtained from *read_events()* will have unpredictable results.

> **Note**
>
> *XMLPullParser* only guarantees that it has seen the ">" character of a starting tag when it emits a "start" event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present.
>
> If you need a fully populated element, look for "end" events instead.

Added in version 3.4.

Changed in version 3.8: The `comment` and `pi` events were added.

Exceptions

class xml.etree.ElementTree.**ParseError**

XML parse error, raised by the various parsing methods in this module when parsing fails. The string representation of an instance of this exception will contain a user-friendly error message. In addition, it will have the following attributes available:

code

A numeric error code from the expat parser. See the documentation of *xml.parsers.expat* for the list of error codes and their meanings.

position

A tuple of *line*, *column* numbers, specifying where the error occurred.

20.6 xml.dom — The Document Object Model API

Source code: Lib/xml/dom/__init__.py

The Document Object Model, or "DOM," is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying XML documents. A DOM implementation presents an XML document as a tree structure, or allows client code to build such a structure from scratch. It then gives access to the structure through a set of objects which provided well-known interfaces.

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another. If you are looking at a text node, you have no access to a containing element. When you write a SAX application, you need to keep track of your program's position in the document somewhere in your own code. SAX does not do it for you. Also, if you need to look ahead in the XML document, you are just out of luck.

Some applications are simply impossible in an event driven model with no access to a tree. Of course you could build some sort of tree yourself in SAX events, but the DOM allows you to avoid writing that code. The DOM is a standard tree representation for XML data.

The Document Object Model is being defined by the W3C in stages, or "levels" in their terminology. The Python mapping of the API is substantially based on the DOM Level 2 recommendation.

DOM applications typically start by parsing some XML into a DOM. How this is accomplished is not covered at all by DOM Level 1, and Level 2 provides only limited improvements: There is a `DOMImplementation` object class which provides access to `Document` creation methods, but no way to access an XML reader/parser/Document builder in an implementation-independent way. There is also no well-defined way to access these methods without an existing `Document` object. In Python, each DOM implementation will provide a function *`getDOMImplementation()`*. DOM Level 3 adds a Load/Store specification, which defines an interface to the reader, but this is not yet available in the Python standard library.

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification; this portion of the reference manual describes the interpretation of the specification in Python.

The specification provided by the W3C defines the DOM API for Java, ECMAScript, and OMG IDL. The Python mapping defined here is based in large part on the IDL version of the specification, but strict compliance is not required (though implementations are free to support the strict mapping from IDL). See section *Conformance* for a detailed discussion of mapping requirements.

See also

Document Object Model (DOM) Level 2 Specification
The W3C recommendation upon which the Python DOM API is based.

Document Object Model (DOM) Level 1 Specification
The W3C recommendation for the DOM supported by *`xml.dom.minidom`*.

Python Language Mapping Specification
This specifies the mapping from OMG IDL to Python.

20.6.1 Module Contents

The *`xml.dom`* contains the following functions:

`xml.dom.`**`registerDOMImplementation`**(*name*, *factory*)

Register the *factory* function with the name *name*. The factory function should return an object which implements the `DOMImplementation` interface. The factory function can return the same object every time, or a new one for each call, as appropriate for the specific implementation (e.g. if that implementation supports some customization).

`xml.dom.`**`getDOMImplementation`**(*name=None*, *features=()*)

Return a suitable DOM implementation. The *name* is either well-known, the module name of a DOM implementation, or `None`. If it is not `None`, imports the corresponding module and returns a `DOMImplementation` object if the import succeeds. If no name is given, and if the environment variable `PYTHON_DOM` is set, this variable is used to find the implementation.

If name is not given, this examines the available implementations to find one with the required feature set. If no implementation can be found, raise an *`ImportError`*. The features list must be a sequence of `(feature, version)` pairs which are passed to the `hasFeature()` method on available `DOMImplementation` objects.

Some convenience constants are also provided:

`xml.dom.`**`EMPTY_NAMESPACE`**

The value used to indicate that no namespace is associated with a node in the DOM. This is typically found as the `namespaceURI` of a node, or used as the *namespaceURI* parameter to a namespaces-specific method.

xml.dom.**XML_NAMESPACE**

The namespace URI associated with the reserved prefix xml, as defined by Namespaces in XML (section 4).

xml.dom.**XMLNS_NAMESPACE**

The namespace URI for namespace declarations, as defined by Document Object Model (DOM) Level 2 Core Specification (section 1.1.8).

xml.dom.**XHTML_NAMESPACE**

The URI of the XHTML namespace as defined by XHTML 1.0: The Extensible HyperText Markup Language (section 3.1.1).

In addition, *xml.dom* contains a base Node class and the DOM exception classes. The Node class provided by this module does not implement any of the methods or attributes defined by the DOM specification; concrete DOM implementations must provide those. The Node class provided as part of this module does provide the constants used for the nodeType attribute on concrete Node objects; they are located within the class rather than at the module level to conform with the DOM specifications.

20.6.2 Objects in the DOM

The definitive documentation for the DOM is the DOM specification from the W3C.

Note that DOM attributes may also be manipulated as nodes instead of as simple strings. It is fairly rare that you must do this, however, so this usage is not yet documented.

Interface	Section	Purpose
DOMImplementation	*DOMImplementation Objects*	Interface to the underlying implementation.
Node	*Node Objects*	Base interface for most objects in a document.
NodeList	*NodeList Objects*	Interface for a sequence of nodes.
DocumentType	*DocumentType Objects*	Information about the declarations needed to process a document.
Document	*Document Objects*	Object which represents an entire document.
Element	*Element Objects*	Element nodes in the document hierarchy.
Attr	*Attr Objects*	Attribute value nodes on element nodes.
Comment	*Comment Objects*	Representation of comments in the source document.
Text	*Text and CDATASection Objects*	Nodes containing textual content from the document.
ProcessingInstruction	*ProcessingInstruction Objects*	Processing instruction representation.

An additional section describes the exceptions defined for working with the DOM in Python.

DOMImplementation Objects

The DOMImplementation interface provides a way for applications to determine the availability of particular features in the DOM they are using. DOM Level 2 added the ability to create new Document and DocumentType objects using the DOMImplementation as well.

DOMImplementation.**hasFeature**(*feature*, *version*)

Return True if the feature identified by the pair of strings *feature* and *version* is implemented.

DOMImplementation.**createDocument**(*namespaceUri*, *qualifiedName*, *doctype*)

Return a new Document object (the root of the DOM), with a child Element object having the given *namespaceUri* and *qualifiedName*. The *doctype* must be a DocumentType object created by *createDocumentType()*, or None. In the Python DOM API, the first two arguments can also be None in order to indicate that no Element child is to be created.

DOMImplementation.**createDocumentType**(*qualifiedName*, *publicId*, *systemId*)

Return a new DocumentType object that encapsulates the given *qualifiedName*, *publicId*, and *systemId* strings, representing the information contained in an XML document type declaration.

Node Objects

All of the components of an XML document are subclasses of Node.

Node.**nodeType**

An integer representing the node type. Symbolic constants for the types are on the Node object: ELEMENT_NODE, ATTRIBUTE_NODE, TEXT_NODE, CDATA_SECTION_NODE, ENTITY_NODE, PROCESSING_INSTRUCTION_NODE, COMMENT_NODE, DOCUMENT_NODE, DOCUMENT_TYPE_NODE, NOTATION_NODE. This is a read-only attribute.

Node.**parentNode**

The parent of the current node, or None for the document node. The value is always a Node object or None. For Element nodes, this will be the parent element, except for the root element, in which case it will be the Document object. For Attr nodes, this is always None. This is a read-only attribute.

Node.**attributes**

A NamedNodeMap of attribute objects. Only elements have actual values for this; others provide None for this attribute. This is a read-only attribute.

Node.**previousSibling**

The node that immediately precedes this one with the same parent. For instance the element with an end-tag that comes just before the *self* element's start-tag. Of course, XML documents are made up of more than just elements so the previous sibling could be text, a comment, or something else. If this node is the first child of the parent, this attribute will be None. This is a read-only attribute.

Node.**nextSibling**

The node that immediately follows this one with the same parent. See also *previousSibling*. If this is the last child of the parent, this attribute will be None. This is a read-only attribute.

Node.**childNodes**

A list of nodes contained within this node. This is a read-only attribute.

Node.**firstChild**

The first child of the node, if there are any, or None. This is a read-only attribute.

Node.**lastChild**

The last child of the node, if there are any, or None. This is a read-only attribute.

Node.**localName**

The part of the tagName following the colon if there is one, else the entire tagName. The value is a string.

Node.**prefix**

The part of the tagName preceding the colon if there is one, else the empty string. The value is a string, or None.

Node.**namespaceURI**

The namespace associated with the element name. This will be a string or None. This is a read-only attribute.

Node.**nodeName**

This has a different meaning for each node type; see the DOM specification for details. You can always get the information you would get here from another property such as the `tagName` property for elements or the `name` property for attributes. For all node types, the value of this attribute will be either a string or `None`. This is a read-only attribute.

Node.**nodeValue**

This has a different meaning for each node type; see the DOM specification for details. The situation is similar to that with *nodeName*. The value is a string or `None`.

Node.**hasAttributes**()

Return `True` if the node has any attributes.

Node.**hasChildNodes**()

Return `True` if the node has any child nodes.

Node.**isSameNode**(*other*)

Return `True` if *other* refers to the same node as this node. This is especially useful for DOM implementations which use any sort of proxy architecture (because more than one object can refer to the same node).

> **Note**
>
> This is based on a proposed DOM Level 3 API which is still in the "working draft" stage, but this particular interface appears uncontroversial. Changes from the W3C will not necessarily affect this method in the Python DOM interface (though any new W3C API for this would also be supported).

Node.**appendChild**(*newChild*)

Add a new child node to this node at the end of the list of children, returning *newChild*. If the node was already in the tree, it is removed first.

Node.**insertBefore**(*newChild*, *refChild*)

Insert a new child node before an existing child. It must be the case that *refChild* is a child of this node; if not, *ValueError* is raised. *newChild* is returned. If *refChild* is `None`, it inserts *newChild* at the end of the children's list.

Node.**removeChild**(*oldChild*)

Remove a child node. *oldChild* must be a child of this node; if not, *ValueError* is raised. *oldChild* is returned on success. If *oldChild* will not be used further, its `unlink()` method should be called.

Node.**replaceChild**(*newChild*, *oldChild*)

Replace an existing node with a new node. It must be the case that *oldChild* is a child of this node; if not, *ValueError* is raised.

Node.**normalize**()

Join adjacent text nodes so that all stretches of text are stored as single `Text` instances. This simplifies processing text from a DOM tree for many applications.

Node.**cloneNode**(*deep*)

Clone this node. Setting *deep* means to clone all child nodes as well. This returns the clone.

NodeList Objects

A NodeList represents a sequence of nodes. These objects are used in two ways in the DOM Core recommendation: an Element object provides one as its list of child nodes, and the getElementsByTagName() and getElementsByTagNameNS() methods of Node return objects with this interface to represent query results.

The DOM Level 2 recommendation defines one method and one attribute for these objects:

NodeList.**item**(*i*)

Return the *i*'th item from the sequence, if there is one, or None. The index *i* is not allowed to be less than zero or greater than or equal to the length of the sequence.

NodeList.**length**

The number of nodes in the sequence.

In addition, the Python DOM interface requires that some additional support is provided to allow NodeList objects to be used as Python sequences. All NodeList implementations must include support for __len__() and __getitem__(); this allows iteration over the NodeList in for statements and proper support for the *len()* built-in function.

If a DOM implementation supports modification of the document, the NodeList implementation must also support the __setitem__() and __delitem__() methods.

DocumentType Objects

Information about the notations and entities declared by a document (including the external subset if the parser uses it and can provide the information) is available from a DocumentType object. The DocumentType for a document is available from the Document object's doctype attribute; if there is no DOCTYPE declaration for the document, the document's doctype attribute will be set to None instead of an instance of this interface.

DocumentType is a specialization of Node, and adds the following attributes:

DocumentType.**publicId**

The public identifier for the external subset of the document type definition. This will be a string or None.

DocumentType.**systemId**

The system identifier for the external subset of the document type definition. This will be a URI as a string, or None.

DocumentType.**internalSubset**

A string giving the complete internal subset from the document. This does not include the brackets which enclose the subset. If the document has no internal subset, this should be None.

DocumentType.**name**

The name of the root element as given in the DOCTYPE declaration, if present.

DocumentType.**entities**

This is a NamedNodeMap giving the definitions of external entities. For entity names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be None if the information is not provided by the parser, or if no entities are defined.

DocumentType.**notations**

This is a NamedNodeMap giving the definitions of notations. For notation names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be None if the information is not provided by the parser, or if no notations are defined.

Document Objects

A `Document` represents an entire XML document, including its constituent elements, attributes, processing instructions, comments etc. Remember that it inherits properties from `Node`.

`Document.`**`documentElement`**

The one and only root element of the document.

`Document.`**`createElement`**(*tagName*)

Create and return a new element node. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as `insertBefore()` or `appendChild()`.

`Document.`**`createElementNS`**(*namespaceURI*, *tagName*)

Create and return a new element with a namespace. The *tagName* may have a prefix. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as `insertBefore()` or `appendChild()`.

`Document.`**`createTextNode`**(*data*)

Create and return a text node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

`Document.`**`createComment`**(*data*)

Create and return a comment node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

`Document.`**`createProcessingInstruction`**(*target*, *data*)

Create and return a processing instruction node containing the *target* and *data* passed as parameters. As with the other creation methods, this one does not insert the node into the tree.

`Document.`**`createAttribute`**(*name*)

Create and return an attribute node. This method does not associate the attribute node with any particular element. You must use `setAttributeNode()` on the appropriate `Element` object to use the newly created attribute instance.

`Document.`**`createAttributeNS`**(*namespaceURI*, *qualifiedName*)

Create and return an attribute node with a namespace. The *tagName* may have a prefix. This method does not associate the attribute node with any particular element. You must use `setAttributeNode()` on the appropriate `Element` object to use the newly created attribute instance.

`Document.`**`getElementsByTagName`**(*tagName*)

Search for all descendants (direct children, children's children, etc.) with a particular element type name.

`Document.`**`getElementsByTagNameNS`**(*namespaceURI*, *localName*)

Search for all descendants (direct children, children's children, etc.) with a particular namespace URI and localname. The localname is the part of the namespace after the prefix.

Element Objects

`Element` is a subclass of `Node`, so inherits all the attributes of that class.

`Element.`**`tagName`**

The element type name. In a namespace-using document it may have colons in it. The value is a string.

`Element.`**`getElementsByTagName`**(*tagName*)

Same as equivalent method in the `Document` class.

Element.**getElementsByTagNameNS**(*namespaceURI*, *localName*)

Same as equivalent method in the Document class.

Element.**hasAttribute**(*name*)

Return True if the element has an attribute named by *name*.

Element.**hasAttributeNS**(*namespaceURI*, *localName*)

Return True if the element has an attribute named by *namespaceURI* and *localName*.

Element.**getAttribute**(*name*)

Return the value of the attribute named by *name* as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

Element.**getAttributeNode**(*attrname*)

Return the Attr node for the attribute named by *attrname*.

Element.**getAttributeNS**(*namespaceURI*, *localName*)

Return the value of the attribute named by *namespaceURI* and *localName* as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

Element.**getAttributeNodeNS**(*namespaceURI*, *localName*)

Return an attribute value as a node, given a *namespaceURI* and *localName*.

Element.**removeAttribute**(*name*)

Remove an attribute by name. If there is no matching attribute, a *NotFoundErr* is raised.

Element.**removeAttributeNode**(*oldAttr*)

Remove and return *oldAttr* from the attribute list, if present. If *oldAttr* is not present, *NotFoundErr* is raised.

Element.**removeAttributeNS**(*namespaceURI*, *localName*)

Remove an attribute by name. Note that it uses a localName, not a qname. No exception is raised if there is no matching attribute.

Element.**setAttribute**(*name*, *value*)

Set an attribute value from a string.

Element.**setAttributeNode**(*newAttr*)

Add a new attribute node to the element, replacing an existing attribute if necessary if the name attribute matches. If a replacement occurs, the old attribute node will be returned. If *newAttr* is already in use, *InuseAttributeErr* will be raised.

Element.**setAttributeNodeNS**(*newAttr*)

Add a new attribute node to the element, replacing an existing attribute if necessary if the namespaceURI and localName attributes match. If a replacement occurs, the old attribute node will be returned. If *newAttr* is already in use, *InuseAttributeErr* will be raised.

Element.**setAttributeNS**(*namespaceURI*, *qname*, *value*)

Set an attribute value from a string, given a *namespaceURI* and a *qname*. Note that a qname is the whole attribute name. This is different than above.

Attr Objects

`Attr` inherits from `Node`, so inherits all its attributes.

`Attr.`**`name`**

The attribute name. In a namespace-using document it may include a colon.

`Attr.`**`localName`**

The part of the name following the colon if there is one, else the entire name. This is a read-only attribute.

`Attr.`**`prefix`**

The part of the name preceding the colon if there is one, else the empty string.

`Attr.`**`value`**

The text value of the attribute. This is a synonym for the `nodeValue` attribute.

NamedNodeMap Objects

`NamedNodeMap` does *not* inherit from `Node`.

`NamedNodeMap.`**`length`**

The length of the attribute list.

`NamedNodeMap.`**`item`**(*index*)

Return an attribute with a particular index. The order you get the attributes in is arbitrary but will be consistent for the life of a DOM. Each item is an attribute node. Get its value with the `value` attribute.

There are also experimental methods that give this class more mapping behavior. You can use them or you can use the standardized `getAttribute*()` family of methods on the `Element` objects.

Comment Objects

`Comment` represents a comment in the XML document. It is a subclass of `Node`, but cannot have child nodes.

`Comment.`**`data`**

The content of the comment as a string. The attribute contains all characters between the leading `<!--` and trailing `-->`, but does not include them.

Text and CDATASection Objects

The `Text` interface represents text in the XML document. If the parser and DOM implementation support the DOM's XML extension, portions of the text enclosed in CDATA marked sections are stored in `CDATASection` objects. These two interfaces are identical, but provide different values for the `nodeType` attribute.

These interfaces extend the `Node` interface. They cannot have child nodes.

`Text.`**`data`**

The content of the text node as a string.

Note

The use of a `CDATASection` node does not indicate that the node represents a complete CDATA marked section, only that the content of the node was part of a CDATA section. A single CDATA section may be represented by more

than one node in the document tree. There is no way to determine whether two adjacent `CDATASection` nodes represent different CDATA marked sections.

ProcessingInstruction Objects

Represents a processing instruction in the XML document; this inherits from the `Node` interface and cannot have child nodes.

`ProcessingInstruction.`**`target`**

The content of the processing instruction up to the first whitespace character. This is a read-only attribute.

`ProcessingInstruction.`**`data`**

The content of the processing instruction following the first whitespace character.

Exceptions

The DOM Level 2 recommendation defines a single exception, *`DOMException`*, and a number of constants that allow applications to determine what sort of error occurred. *`DOMException`* instances carry a *`code`* attribute that provides the appropriate value for the specific exception.

The Python DOM interface provides the constants, but also expands the set of exceptions so that a specific exception exists for each of the exception codes defined by the DOM. The implementations must raise the appropriate specific exception, each of which carries the appropriate value for the *`code`* attribute.

exception `xml.dom.`**`DOMException`**

Base exception class used for all specific DOM exceptions. This exception class cannot be directly instantiated.

exception `xml.dom.`**`DomstringSizeErr`**

Raised when a specified range of text does not fit into a string. This is not known to be used in the Python DOM implementations, but may be received from DOM implementations not written in Python.

exception `xml.dom.`**`HierarchyRequestErr`**

Raised when an attempt is made to insert a node where the node type is not allowed.

exception `xml.dom.`**`IndexSizeErr`**

Raised when an index or size parameter to a method is negative or exceeds the allowed values.

exception `xml.dom.`**`InuseAttributeErr`**

Raised when an attempt is made to insert an `Attr` node that is already present elsewhere in the document.

exception `xml.dom.`**`InvalidAccessErr`**

Raised if a parameter or an operation is not supported on the underlying object.

exception `xml.dom.`**`InvalidCharacterErr`**

This exception is raised when a string parameter contains a character that is not permitted in the context it's being used in by the XML 1.0 recommendation. For example, attempting to create an `Element` node with a space in the element type name will cause this error to be raised.

exception `xml.dom.`**`InvalidModificationErr`**

Raised when an attempt is made to modify the type of a node.

exception `xml.dom.`**`InvalidStateErr`**

Raised when an attempt is made to use an object that is not defined or is no longer usable.

exception xml.dom.**NamespaceErr**

If an attempt is made to change any object in a way that is not permitted with regard to the Namespaces in XML recommendation, this exception is raised.

exception xml.dom.**NotFoundErr**

Exception when a node does not exist in the referenced context. For example, `NamedNodeMap.removeNamedItem()` will raise this if the node passed in does not exist in the map.

exception xml.dom.**NotSupportedErr**

Raised when the implementation does not support the requested type of object or operation.

exception xml.dom.**NoDataAllowedErr**

This is raised if data is specified for a node which does not support data.

exception xml.dom.**NoModificationAllowedErr**

Raised on attempts to modify an object where modifications are not allowed (such as for read-only nodes).

exception xml.dom.**SyntaxErr**

Raised when an invalid or illegal string is specified.

exception xml.dom.**WrongDocumentErr**

Raised when a node is inserted in a different document than it currently belongs to, and the implementation does not support migrating the node from one document to the other.

The exception codes defined in the DOM recommendation map to the exceptions described above according to this table:

Constant	Exception
DOMSTRING_SIZE_ERR	*DomstringSizeErr*
HIERARCHY_REQUEST_ERR	*HierarchyRequestErr*
INDEX_SIZE_ERR	*IndexSizeErr*
INUSE_ATTRIBUTE_ERR	*InuseAttributeErr*
INVALID_ACCESS_ERR	*InvalidAccessErr*
INVALID_CHARACTER_ERR	*InvalidCharacterErr*
INVALID_MODIFICATION_ERR	*InvalidModificationErr*
INVALID_STATE_ERR	*InvalidStateErr*
NAMESPACE_ERR	*NamespaceErr*
NOT_FOUND_ERR	*NotFoundErr*
NOT_SUPPORTED_ERR	*NotSupportedErr*
NO_DATA_ALLOWED_ERR	*NoDataAllowedErr*
NO_MODIFICATION_ALLOWED_ERR	*NoModificationAllowedErr*
SYNTAX_ERR	*SyntaxErr*
WRONG_DOCUMENT_ERR	*WrongDocumentErr*

20.6.3 Conformance

This section describes the conformance requirements and relationships between the Python DOM API, the W3C DOM recommendations, and the OMG IDL mapping for Python.

Type Mapping

The IDL types used in the DOM specification are mapped to Python types according to the following table.

IDL Type	Python Type
`boolean`	`bool` or `int`
`int`	`int`
`long int`	`int`
`unsigned int`	`int`
`DOMString`	`str` or `bytes`
`null`	`None`

Accessor Methods

The mapping from OMG IDL to Python defines accessor functions for IDL `attribute` declarations in much the way the Java mapping does. Mapping the IDL declarations

```
readonly attribute string someValue;
         attribute string anotherValue;
```

yields three accessor functions: a "get" method for `someValue` (`_get_someValue()`), and "get" and "set" methods for `anotherValue` (`_get_anotherValue()` and `_set_anotherValue()`). The mapping, in particular, does not require that the IDL attributes are accessible as normal Python attributes: `object.someValue` is *not* required to work, and may raise an *AttributeError*.

The Python DOM API, however, *does* require that normal attribute access work. This means that the typical surrogates generated by Python IDL compilers are not likely to work, and wrapper objects may be needed on the client if the DOM objects are accessed via CORBA. While this does require some additional consideration for CORBA DOM clients, the implementers with experience using DOM over CORBA from Python do not consider this a problem. Attributes that are declared `readonly` may not restrict write access in all DOM implementations.

In the Python DOM API, accessor functions are not required. If provided, they should take the form defined by the Python IDL mapping, but these methods are considered unnecessary since the attributes are accessible directly from Python. "Set" accessors should never be provided for `readonly` attributes.

The IDL definitions do not fully embody the requirements of the W3C DOM API, such as the notion of certain objects, such as the return value of `getElementsByTagName()`, being "live". The Python DOM API does not require implementations to enforce such requirements.

20.7 `xml.dom.minidom` — Minimal DOM implementation

Source code: Lib/xml/dom/minidom.py

---

xml.dom.minidom is a minimal implementation of the Document Object Model interface, with an API similar to that in other languages. It is intended to be simpler than the full DOM and also significantly smaller. Users who are not already proficient with the DOM should consider using the *xml.etree.ElementTree* module for their XML processing instead.

Warning

The *xml.dom.minidom* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML vulnerabilities*.

DOM applications typically start by parsing some XML into a DOM. With *xml.dom.minidom*, this is done through the parse functions:

```
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\\temp\\mydata.xml')  # parse an XML file by name

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource)  # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')
```

The *parse()* function can take either a filename or an open file object.

xml.dom.minidom.**parse**(*filename_or_file*, *parser=None*, *bufsize=None*)

Return a Document from the given input. *filename_or_file* may be either a file name, or a file-like object. *parser*, if given, must be a SAX2 parser object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the *parseString()* function instead:

xml.dom.minidom.**parseString**(*string*, *parser=None*)

Return a Document that represents the *string*. This method creates an *io.StringIO* object for the string and passes that on to *parse()*.

Both functions return a Document object representing the content of the document.

What the *parse()* and *parseString()* functions do is connect an XML parser with a "DOM builder" that can accept parse events from any SAX parser and convert them into a DOM tree. The name of the functions are perhaps misleading, but are easy to grasp when learning the interfaces. The parsing of the document will be completed before these functions return; it's simply that these functions do not provide a parser implementation themselves.

You can also create a Document by calling a method on a "DOM Implementation" object. You can get this object either by calling the getDOMImplementation() function in the *xml.dom* package or the *xml.dom.minidom* module. Once you have a Document, you can add child nodes to it to populate the DOM:

```
from xml.dom.minidom import getDOMImplementation

impl = getDOMImplementation()

newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)
```

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification. The main property of the document object is the documentElement property. It gives you the main element in the XML document: the one that holds all others. Here is an example program:

```
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

When you are finished with a DOM tree, you may optionally call the `unlink()` method to encourage early cleanup of the now-unneeded objects. `unlink()` is an *xml.dom.minidom*-specific extension to the DOM API that renders the node and its descendants essentially useless. Otherwise, Python's garbage collector will eventually take care of the objects in the tree.

See also

Document Object Model (DOM) Level 1 Specification
The W3C recommendation for the DOM supported by *xml.dom.minidom*.

20.7.1 DOM Objects

The definition of the DOM API for Python is given as part of the *xml.dom* module documentation. This section lists the differences between the API and *xml.dom.minidom*.

`Node.`**`unlink`**`()`

Break internal references within the DOM so that it will be garbage collected on versions of Python without cyclic GC. Even when cyclic GC is available, using this can make large amounts of memory available sooner, so calling this on DOM objects as soon as they are no longer needed is good practice. This only needs to be called on the `Document` object, but may be called on child nodes to discard children of that node.

You can avoid calling this method explicitly by using the `with` statement. The following code will automatically unlink *dom* when the `with` block is exited:

```
with xml.dom.minidom.parse(datasource) as dom:
    ... # Work with dom.
```

`Node.`**`writexml`**`(`*writer*, *indent=''*, *addindent=''*, *newl=''*, *encoding=None*, *standalone=None*`)`

Write XML to the writer object. The writer receives texts but not bytes as input, it should have a `write()` method which matches that of the file object interface. The *indent* parameter is the indentation of the current node. The *addindent* parameter is the incremental indentation to use for subnodes of the current one. The *newl* parameter specifies the string to use to terminate newlines.

For the `Document` node, an additional keyword argument *encoding* can be used to specify the encoding field of the XML header.

Similarly, explicitly stating the *standalone* argument causes the standalone document declarations to be added to the prologue of the XML document. If the value is set to `True`, `standalone="yes"` is added, otherwise it is set to `"no"`. Not stating the argument will omit the declaration from the document.

Changed in version 3.8: The *writexml()* method now preserves the attribute order specified by the user.

Changed in version 3.9: The *standalone* parameter was added.

`Node.`**`toxml`**`(`*encoding=None*, *standalone=None*`)`

Return a string or byte string containing the XML represented by the DOM node.

With an explicit *encoding*[1] argument, the result is a byte string in the specified encoding. With no *encoding* argument, the result is a Unicode string, and the XML declaration in the resulting string does not specify an encoding. Encoding this string in an encoding other than UTF-8 is likely incorrect, since UTF-8 is the default encoding of XML.

The *standalone* argument behaves exactly as in *writexml()*.

[1] The encoding name included in the XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not valid in an XML document's declaration, even though Python accepts it as an encoding name. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

Changed in version 3.8: The *toxml()* method now preserves the attribute order specified by the user.

Changed in version 3.9: The *standalone* parameter was added.

Node.**toprettyxml**(*indent='\t'*, *newl='\n'*, *encoding=None*, *standalone=None*)

Return a pretty-printed version of the document. *indent* specifies the indentation string and defaults to a tabulator; *newl* specifies the string emitted at the end of each line and defaults to `\n`.

The *encoding* argument behaves like the corresponding argument of *toxml()*.

The *standalone* argument behaves exactly as in *writexml()*.

Changed in version 3.8: The *toprettyxml()* method now preserves the attribute order specified by the user.

Changed in version 3.9: The *standalone* parameter was added.

20.7.2 DOM Example

This example program is a fairly realistic example of a simple program. In this particular case, we do not take much advantage of the flexibility of the DOM.

```
import xml.dom.minidom

document = """\
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>

<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"""

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)

def handleSlideshow(slideshow):
    print("<html>")
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print("</html>")

def handleSlides(slides):
    for slide in slides:
```

(continues on next page)

(continued from previous page)

```
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print(f"<title>{getText(title.childNodes)}</title>")

def handleSlideTitle(title):
    print(f"<h2>{getText(title.childNodes)}</h2>")

def handlePoints(points):
    print("<ul>")
    for point in points:
        handlePoint(point)
    print("</ul>")

def handlePoint(point):
    print(f"<li>{getText(point.childNodes)}</li>")

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print(f"<p>{getText(title.childNodes)}</p>")

handleSlideshow(dom)
```

20.7.3 minidom and the DOM standard

The *xml.dom.minidom* module is essentially a DOM 1.0-compatible DOM with some DOM 2 features (primarily namespace features).

Usage of the DOM interface in Python is straight-forward. The following mapping rules apply:

- Interfaces are accessed through instance objects. Applications should not instantiate the classes themselves; they should use the creator functions available on the `Document` object. Derived interfaces support all operations (and attributes) from the base interfaces, plus any new operations.
- Operations are used as methods. Since the DOM uses only `in` parameters, the arguments are passed in normal order (from left to right). There are no optional arguments. `void` operations return `None`.
- IDL attributes map to instance attributes. For compatibility with the OMG IDL language mapping for Python, an attribute `foo` can also be accessed through accessor methods `_get_foo()` and `_set_foo()`. `readonly` attributes must not be changed; this is not enforced at runtime.
- The types `short int`, `unsigned int`, `unsigned long long`, and `boolean` all map to Python integer objects.
- The type `DOMString` maps to Python strings. *xml.dom.minidom* supports either bytes or strings, but will normally produce strings. Values of type `DOMString` may also be `None` where allowed to have the IDL `null` value by the DOM specification from the W3C.
- `const` declarations map to variables in their respective scope (e.g. `xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE`); they must not be changed.
- `DOMException` is currently not supported in *xml.dom.minidom*. Instead, *xml.dom.minidom* uses standard Python exceptions such as *TypeError* and *AttributeError*.

- `NodeList` objects are implemented using Python's built-in list type. These objects provide the interface defined in the DOM specification, but with earlier versions of Python they do not support the official API. They are, however, much more "Pythonic" than the interface defined in the W3C recommendations.

The following interfaces have no implementation in *xml.dom.minidom*:

- `DOMTimeStamp`
- `EntityReference`

Most of these reflect information in the XML document that is not of general utility to most DOM users.

20.8 xml.dom.pulldom — Support for building partial DOM trees

Source code: Lib/xml/dom/pulldom.py

The *xml.dom.pulldom* module provides a "pull parser" which can also be asked to produce DOM-accessible fragments of the document where necessary. The basic concept involves pulling "events" from a stream of incoming XML and processing them. In contrast to SAX which also employs an event-driven processing model together with callbacks, the user of a pull parser is responsible for explicitly pulling events from the stream, looping over those events until either processing is finished or an error condition occurs.

> **Warning**
>
> The *xml.dom.pulldom* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML vulnerabilities*.

Changed in version 3.7.1: The SAX parser no longer processes general external entities by default to increase security by default. To enable processing of external entities, pass a custom parser instance in:

```
from xml.dom.pulldom import parse
from xml.sax import make_parser
from xml.sax.handler import feature_external_ges

parser = make_parser()
parser.setFeature(feature_external_ges, True)
parse(filename, parser=parser)
```

Example:

```
from xml.dom import pulldom

doc = pulldom.parse('sales_items.xml')
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'item':
        if int(node.getAttribute('price')) > 50:
            doc.expandNode(node)
            print(node.toxml())
```

`event` is a constant and can be one of:

- `START_ELEMENT`
- `END_ELEMENT`

- COMMENT
- START_DOCUMENT
- END_DOCUMENT
- CHARACTERS
- PROCESSING_INSTRUCTION
- IGNORABLE_WHITESPACE

node is an object of type xml.dom.minidom.Document, xml.dom.minidom.Element or xml.dom.minidom.Text.

Since the document is treated as a "flat" stream of events, the document "tree" is implicitly traversed and the desired elements are found regardless of their depth in the tree. In other words, one does not need to consider hierarchical issues such as recursive searching of the document nodes, although if the context of elements were important, one would either need to maintain some context-related state (i.e. remembering where one is in the document at any given point) or to make use of the *DOMEventStream.expandNode()* method and switch to DOM-related processing.

class xml.dom.pulldom.**PullDom**(*documentFactory=None*)

Subclass of *xml.sax.handler.ContentHandler*.

class xml.dom.pulldom.**SAX2DOM**(*documentFactory=None*)

Subclass of *xml.sax.handler.ContentHandler*.

xml.dom.pulldom.**parse**(*stream_or_string*, *parser=None*, *bufsize=None*)

Return a *DOMEventStream* from the given input. *stream_or_string* may be either a file name, or a file-like object. *parser*, if given, must be an *XMLReader* object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the *parseString()* function instead:

xml.dom.pulldom.**parseString**(*string*, *parser=None*)

Return a *DOMEventStream* that represents the (Unicode) *string*.

xml.dom.pulldom.**default_bufsize**

Default value for the *bufsize* parameter to *parse()*.

The value of this variable can be changed before calling *parse()* and the new value will take effect.

20.8.1 DOMEventStream Objects

class xml.dom.pulldom.**DOMEventStream**(*stream*, *parser*, *bufsize*)

Changed in version 3.11: Support for __getitem__() method has been removed.

getEvent()

Return a tuple containing *event* and the current *node* as xml.dom.minidom.Document if event equals START_DOCUMENT, xml.dom.minidom.Element if event equals START_ELEMENT or END_ELEMENT or xml.dom.minidom.Text if event equals CHARACTERS. The current node does not contain information about its children, unless *expandNode()* is called.

expandNode(*node*)

Expands all children of *node* into *node*. Example:

```
from xml.dom import pulldom

xml = '<html><title>Foo</title> <p>Some text <div>and more</div></p> </html>'
doc = pulldom.parseString(xml)
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'p':
        # Following statement only prints '<p/>'
        print(node.toxml())
        doc.expandNode(node)
        # Following statement prints node with all its children '<p>Some text
↪<div>and more</div></p>'
        print(node.toxml())
```

reset()

20.9 xml.sax — Support for SAX2 parsers

Source code: Lib/xml/sax/__init__.py

The *xml.sax* package provides a number of modules which implement the Simple API for XML (SAX) interface for Python. The package itself provides the SAX exceptions and the convenience functions which will be most used by users of the SAX API.

> **Warning**
>
> The *xml.sax* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML vulnerabilities*.

Changed in version 3.7.1: The SAX parser no longer processes general external entities by default to increase security. Before, the parser created network connections to fetch remote files or loaded local files from the file system for DTD and entities. The feature can be enabled again with method *setFeature()* on the parser object and argument *feature_external_ges*.

The convenience functions are:

xml.sax.**make_parser**(*parser_list=[]*)

Create and return a SAX *XMLReader* object. The first parser found will be used. If *parser_list* is provided, it must be an iterable of strings which name modules that have a function named `create_parser()`. Modules listed in *parser_list* will be used before modules in the default list of parsers.

Changed in version 3.8: The *parser_list* argument can be any iterable, not just a list.

xml.sax.**parse**(*filename_or_stream*, *handler*, *error_handler=handler.ErrorHandler()*)

Create a SAX parser and use it to parse a document. The document, passed in as *filename_or_stream*, can be a filename or a file object. The *handler* parameter needs to be a SAX *ContentHandler* instance. If *error_handler* is given, it must be a SAX *ErrorHandler* instance; if omitted, *SAXParseException* will be raised on all errors. There is no return value; all work must be done by the *handler* passed in.

xml.sax.**parseString**(*string*, *handler*, *error_handler=handler.ErrorHandler()*)

Similar to *parse()*, but parses from a buffer *string* received as a parameter. *string* must be a *str* instance or a *bytes-like object*.

Changed in version 3.5: Added support of *str* instances.

A typical SAX application uses three kinds of objects: readers, handlers and input sources. "Reader" in this context is another term for parser, i.e. some piece of code that reads the bytes or characters from the input source, and produces a sequence of events. The events then get distributed to the handler objects, i.e. the reader invokes a method on the handler. A SAX application must therefore obtain a reader object, create or open the input sources, create the handlers, and connect these objects all together. As the final step of preparation, the reader is called to parse the input. During parsing, methods on the handler objects are called based on structural and syntactic events from the input data.

For these objects, only the interfaces are relevant; they are normally not instantiated by the application itself. Since Python does not have an explicit notion of interface, they are formally introduced as classes, but applications may use implementations which do not inherit from the provided classes. The *InputSource*, *Locator*, Attributes, AttributesNS, and *XMLReader* interfaces are defined in the module *xml.sax.xmlreader*. The handler interfaces are defined in *xml.sax.handler*. For convenience, *InputSource* (which is often instantiated directly) and the handler classes are also available from *xml.sax*. These interfaces are described below.

In addition to these classes, *xml.sax* provides the following exception classes.

exception xml.sax.**SAXException**(*msg*, *exception=None*)

Encapsulate an XML error or warning. This class can contain basic error or warning information from either the XML parser or the application: it can be subclassed to provide additional functionality or to add localization. Note that although the handlers defined in the *ErrorHandler* interface receive instances of this exception, it is not required to actually raise the exception — it is also useful as a container for information.

When instantiated, *msg* should be a human-readable description of the error. The optional *exception* parameter, if given, should be None or an exception that was caught by the parsing code and is being passed along as information.

This is the base class for the other SAX exception classes.

exception xml.sax.**SAXParseException**(*msg*, *exception*, *locator*)

Subclass of *SAXException* raised on parse errors. Instances of this class are passed to the methods of the SAX *ErrorHandler* interface to provide information about the parse error. This class supports the SAX *Locator* interface as well as the *SAXException* interface.

exception xml.sax.**SAXNotRecognizedException**(*msg*, *exception=None*)

Subclass of *SAXException* raised when a SAX *XMLReader* is confronted with an unrecognized feature or property. SAX applications and extensions may use this class for similar purposes.

exception xml.sax.**SAXNotSupportedException**(*msg*, *exception=None*)

Subclass of *SAXException* raised when a SAX *XMLReader* is asked to enable a feature that is not supported, or to set a property to a value that the implementation does not support. SAX applications and extensions may use this class for similar purposes.

See also

SAX: The Simple API for XML
This site is the focal point for the definition of the SAX API. It provides a Java implementation and online documentation. Links to implementations and historical information are also available.

Module *xml.sax.handler*
Definitions of the interfaces for application-provided objects.

Module *xml.sax.saxutils*
Convenience functions for use in SAX applications.

Module *xml.sax.xmlreader*
Definitions of the interfaces for parser-provided objects.

20.9.1 SAXException Objects

The *SAXException* exception class supports the following methods:

SAXException.**getMessage**()

Return a human-readable message describing the error condition.

SAXException.**getException**()

Return an encapsulated exception object, or None.

20.10 xml.sax.handler — Base classes for SAX handlers

Source code: Lib/xml/sax/handler.py

The SAX API defines five kinds of handlers: content handlers, DTD handlers, error handlers, entity resolvers and lexical handlers. Applications normally only need to implement those interfaces whose events they are interested in; they can implement the interfaces in a single object or in multiple objects. Handler implementations should inherit from the base classes provided in the module *xml.sax.handler*, so that all methods get default implementations.

class xml.sax.handler.**ContentHandler**

This is the main callback interface in SAX, and the one most important to applications. The order of events in this interface mirrors the order of the information in the document.

class xml.sax.handler.**DTDHandler**

Handle DTD events.

This interface specifies only those DTD events required for basic parsing (unparsed entities and attributes).

class xml.sax.handler.**EntityResolver**

Basic interface for resolving entities. If you create an object implementing this interface, then register the object with your Parser, the parser will call the method in your object to resolve all external entities.

class xml.sax.handler.**ErrorHandler**

Interface used by the parser to present error and warning messages to the application. The methods of this object control whether errors are immediately converted to exceptions or are handled in some other way.

class xml.sax.handler.**LexicalHandler**

Interface used by the parser to represent low frequency events which may not be of interest to many applications.

In addition to these classes, *xml.sax.handler* provides symbolic constants for the feature and property names.

xml.sax.handler.**feature_namespaces**

value: "http://xml.org/sax/features/namespaces"

true: Perform Namespace processing.

false: Optionally do not perform Namespace processing (implies namespace-prefixes; default).

access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_namespace_prefixes**

value: "http://xml.org/sax/features/namespace-prefixes"

true: Report the original prefixed names and attributes used for Namespace declarations.

false: Do not report attributes used for Namespace declarations, and optionally do not report original prefixed names (default).

access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_string_interning**

value: "http://xml.org/sax/features/string-interning"
true: All element names, prefixes, attribute names, Namespace URIs, and local names are interned using the built-in intern function.
false: Names are not necessarily interned, although they may be (default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_validation**

value: "http://xml.org/sax/features/validation"
true: Report all validation errors (implies external-general-entities and external-parameter-entities).
false: Do not report validation errors.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_external_ges**

value: "http://xml.org/sax/features/external-general-entities"
true: Include all external general (text) entities.
false: Do not include external general entities.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_external_pes**

value: "http://xml.org/sax/features/external-parameter-entities"
true: Include all external parameter entities, including the external DTD subset.
false: Do not include any external parameter entities, even the external DTD subset.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**all_features**

List of all features.

xml.sax.handler.**property_lexical_handler**

value: "http://xml.org/sax/properties/lexical-handler"
data type: xml.sax.handler.LexicalHandler (not supported in Python 2)
description: An optional extension handler for lexical events like comments.
access: read/write

xml.sax.handler.**property_declaration_handler**

value: "http://xml.org/sax/properties/declaration-handler"
data type: xml.sax.sax2lib.DeclHandler (not supported in Python 2)
description: An optional extension handler for DTD-related events other than notations and unparsed entities.
access: read/write

xml.sax.handler.**property_dom_node**

value: "http://xml.org/sax/properties/dom-node"
data type: org.w3c.dom.Node (not supported in Python 2)
description: When parsing, the current DOM node being visited if this is a DOM iterator; when not parsing, the root DOM node for iteration.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**property_xml_string**

value: "http://xml.org/sax/properties/xml-string"

data type: Bytes

description: The literal string of characters that was the source for the current event.

access: read-only

xml.sax.handler.**all_properties**

List of all known property names.

20.10.1 ContentHandler Objects

Users are expected to subclass *ContentHandler* to support their application. The following methods are called by the parser on the appropriate events in the input document:

ContentHandler.**setDocumentLocator**(*locator*)

Called by the parser to give the application a locator for locating the origin of document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the DocumentHandler interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application's business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.

ContentHandler.**startDocument**()

Receive notification of the beginning of a document.

The SAX parser will invoke this method only once, before any other methods in this interface or in DTDHandler (except for *setDocumentLocator()*).

ContentHandler.**endDocument**()

Receive notification of the end of a document.

The SAX parser will invoke this method only once, and it will be the last method invoked during the parse. The parser shall not invoke this method until it has either abandoned parsing (because of an unrecoverable error) or reached the end of input.

ContentHandler.**startPrefixMapping**(*prefix*, *uri*)

Begin the scope of a prefix-URI Namespace mapping.

The information from this event is not necessary for normal Namespace processing: the SAX XML reader will automatically replace prefixes for element and attribute names when the `feature_namespaces` feature is enabled (the default).

There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically; the *startPrefixMapping()* and *endPrefixMapping()* events supply the information to the application to expand prefixes in those contexts itself, if necessary.

Note that *startPrefixMapping()* and *endPrefixMapping()* events are not guaranteed to be properly nested relative to each-other: all *startPrefixMapping()* events will occur before the corresponding *startElement()* event, and all *endPrefixMapping()* events will occur after the corresponding *endElement()* event, but their order is not guaranteed.

ContentHandler.**endPrefixMapping**(*prefix*)

End the scope of a prefix-URI mapping.

See *startPrefixMapping()* for details. This event will always occur after the corresponding *endElement()* event, but the order of *endPrefixMapping()* events is not otherwise guaranteed.

ContentHandler.**startElement**(*name*, *attrs*)

Signals the start of an element in non-namespace mode.

The *name* parameter contains the raw XML 1.0 name of the element type as a string and the *attrs* parameter holds an object of the `Attributes` interface (see *The Attributes Interface*) containing the attributes of the element. The object passed as *attrs* may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the *copy()* method of the *attrs* object.

ContentHandler.**endElement**(*name*)

Signals the end of an element in non-namespace mode.

The *name* parameter contains the name of the element type, just as with the *startElement()* event.

ContentHandler.**startElementNS**(*name*, *qname*, *attrs*)

Signals the start of an element in namespace mode.

The *name* parameter contains the name of the element type as a `(uri, localname)` tuple, the *qname* parameter contains the raw XML 1.0 name used in the source document, and the *attrs* parameter holds an instance of the `AttributesNS` interface (see *The AttributesNS Interface*) containing the attributes of the element. If no namespace is associated with the element, the *uri* component of *name* will be `None`. The object passed as *attrs* may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the *copy()* method of the *attrs* object.

Parsers may set the *qname* parameter to `None`, unless the `feature_namespace_prefixes` feature is activated.

ContentHandler.**endElementNS**(*name*, *qname*)

Signals the end of an element in namespace mode.

The *name* parameter contains the name of the element type, just as with the *startElementNS()* method, likewise the *qname* parameter.

ContentHandler.**characters**(*content*)

Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the Locator provides useful information.

content may be a string or bytes instance; the `expat` reader module always produces strings.

> **Note**
>
> The earlier SAX 1 interface provided by the Python XML Special Interest Group used a more Java-like interface for this method. Since most parsers used from Python did not take advantage of the older interface, the simpler signature was chosen to replace it. To convert old code to the new interface, use *content* instead of slicing content with the old *offset* and *length* parameters.

ContentHandler.**ignorableWhitespace**(*whitespace*)

Receive notification of ignorable whitespace in element content.

Validating Parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 recommendation, section 2.10): non-validating parsers may also use this method if they are capable of parsing and using content models.

SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the Locator provides useful information.

`ContentHandler.`**`processingInstruction`**(*target*, *data*)

Receive notification of a processing instruction.

The Parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

`ContentHandler.`**`skippedEntity`**(*name*)

Receive notification of a skipped entity.

The Parser will invoke this method once for each entity skipped. Non-validating processors may skip entities if they have not seen the declarations (because, for example, the entity was declared in an external DTD subset). All processors may skip external entities, depending on the values of the `feature_external_ges` and the `feature_external_pes` properties.

20.10.2 DTDHandler Objects

`DTDHandler` instances provide the following methods:

`DTDHandler.`**`notationDecl`**(*name*, *publicId*, *systemId*)

Handle a notation declaration event.

`DTDHandler.`**`unparsedEntityDecl`**(*name*, *publicId*, *systemId*, *ndata*)

Handle an unparsed entity declaration event.

20.10.3 EntityResolver Objects

`EntityResolver.`**`resolveEntity`**(*publicId*, *systemId*)

Resolve the system identifier of an entity and return either the system identifier to read from as a string, or an InputSource to read from. The default implementation returns *systemId*.

20.10.4 ErrorHandler Objects

Objects with this interface are used to receive error and warning information from the *`XMLReader`*. If you create an object that implements this interface, then register the object with your *`XMLReader`*, the parser will call the methods in your object to report all warnings and errors. There are three levels of errors available: warnings, (possibly) recoverable errors, and unrecoverable errors. All methods take a *`SAXParseException`* as the only parameter. Errors and warnings may be converted to an exception by raising the passed-in exception object.

`ErrorHandler.`**`error`**(*exception*)

Called when the parser encounters a recoverable error. If this method does not raise an exception, parsing may continue, but further document information should not be expected by the application. Allowing the parser to continue may allow additional errors to be discovered in the input document.

ErrorHandler.**fatalError**(*exception*)

Called when the parser encounters an error it cannot recover from; parsing is expected to terminate when this method returns.

ErrorHandler.**warning**(*exception*)

Called when the parser presents minor warning information to the application. Parsing is expected to continue when this method returns, and document information will continue to be passed to the application. Raising an exception in this method will cause parsing to end.

20.10.5 LexicalHandler Objects

Optional SAX2 handler for lexical events.

This handler is used to obtain lexical information about an XML document. Lexical information includes information describing the document encoding used and XML comments embedded in the document, as well as section boundaries for the DTD and for any CDATA sections. The lexical handlers are used in the same manner as content handlers.

Set the LexicalHandler of an XMLReader by using the setProperty method with the property identifier `'http://xml.org/sax/properties/lexical-handler'`.

LexicalHandler.**comment**(*content*)

Reports a comment anywhere in the document (including the DTD and outside the document element).

LexicalHandler.**startDTD**(*name*, *public_id*, *system_id*)

Reports the start of the DTD declarations if the document has an associated DTD.

LexicalHandler.**endDTD**()

Reports the end of DTD declaration.

LexicalHandler.**startCDATA**()

Reports the start of a CDATA marked section.

The contents of the CDATA marked section will be reported through the characters handler.

LexicalHandler.**endCDATA**()

Reports the end of a CDATA marked section.

20.11 `xml.sax.saxutils` — SAX Utilities

Source code: Lib/xml/sax/saxutils.py

The module *`xml.sax.saxutils`* contains a number of classes and functions that are commonly useful when creating SAX applications, either in direct use, or as base classes.

xml.sax.saxutils.**escape**(*data*, *entities={}*)

Escape `'&'`, `'<'`, and `'>'` in a string of data.

You can escape other strings of data by passing a dictionary as the optional *entities* parameter. The keys and values must all be strings; each key will be replaced with its corresponding value. The characters `'&'`, `'<'` and `'>'` are always escaped, even if *entities* is provided.

> **Note**
>
> This function should only be used to escape characters that can't be used directly in XML. Do not use this function as a general string translation function.

xml.sax.saxutils.**unescape**(*data*, *entities={}*)

Unescape `'&'`, `'<'`, and `'>'` in a string of data.

You can unescape other strings of data by passing a dictionary as the optional *entities* parameter. The keys and values must all be strings; each key will be replaced with its corresponding value. `'&'`, `'<'`, and `'>'` are always unescaped, even if *entities* is provided.

xml.sax.saxutils.**quoteattr**(*data*, *entities={}*)

Similar to *escape()*, but also prepares *data* to be used as an attribute value. The return value is a quoted version of *data* with any additional required replacements. *quoteattr()* will select a quote character based on the content of *data*, attempting to avoid encoding any quote characters in the string. If both single- and double-quote characters are already in *data*, the double-quote characters will be encoded and *data* will be wrapped in double-quotes. The resulting string can be used directly as an attribute value:

```
>>> print("<element attr=%s>" % quoteattr("ab ' cd \" ef"))
<element attr="ab ' cd &quot; ef">
```

This function is useful when generating attribute values for HTML or any SGML using the reference concrete syntax.

class xml.sax.saxutils.**XMLGenerator**(*out=None*, *encoding='iso-8859-1'*, *short_empty_elements=False*)

This class implements the *ContentHandler* interface by writing SAX events back into an XML document. In other words, using an *XMLGenerator* as the content handler will reproduce the original document being parsed. *out* should be a file-like object which will default to *sys.stdout*. *encoding* is the encoding of the output stream which defaults to `'iso-8859-1'`. *short_empty_elements* controls the formatting of elements that contain no content: if `False` (the default) they are emitted as a pair of start/end tags, if set to `True` they are emitted as a single self-closed tag.

Changed in version 3.2: Added the *short_empty_elements* parameter.

class xml.sax.saxutils.**XMLFilterBase**(*base*)

This class is designed to sit between an *XMLReader* and the client application's event handlers. By default, it does nothing but pass requests up to the reader and events on to the handlers unmodified, but subclasses can override specific methods to modify the event stream or the configuration requests as they pass through.

xml.sax.saxutils.**prepare_input_source**(*source*, *base=''*)

This function takes an input source and an optional base URL and returns a fully resolved *InputSource* object ready for reading. The input source can be given as a string, a file-like object, or an *InputSource* object; parsers will use this function to implement the polymorphic *source* argument to their *parse()* method.

20.12 xml.sax.xmlreader — Interface for XML parsers

Source code: Lib/xml/sax/xmlreader.py

SAX parsers implement the *XMLReader* interface. They are implemented in a Python module, which must provide a function `create_parser()`. This function is invoked by *xml.sax.make_parser()* with no arguments to create a new parser object.

class xml.sax.xmlreader.**XMLReader**

Base class which can be inherited by SAX parsers.

class xml.sax.xmlreader.**IncrementalParser**

In some cases, it is desirable not to parse an input source at once, but to feed chunks of the document as they get available. Note that the reader will normally not read the entire file, but read it in chunks as well; still `parse()` won't return until the entire document is processed. So these interfaces should be used if the blocking behaviour of `parse()` is not desirable.

When the parser is instantiated it is ready to begin accepting data from the feed method immediately. After parsing has been finished with a call to close the reset method must be called to make the parser ready to accept new data, either from feed or using the parse method.

Note that these methods must *not* be called during parsing, that is, after parse has been called and before it returns.

By default, the class also implements the parse method of the XMLReader interface using the feed, close and reset methods of the IncrementalParser interface as a convenience to SAX 2.0 driver writers.

class xml.sax.xmlreader.**Locator**

Interface for associating a SAX event with a document location. A locator object will return valid results only during calls to DocumentHandler methods; at any other time, the results are unpredictable. If information is not available, methods may return `None`.

class xml.sax.xmlreader.**InputSource**(*system_id=None*)

Encapsulation of the information needed by the *XMLReader* to read entities.

This class may include information about the public identifier, system identifier, byte stream (possibly with character encoding information) and/or the character stream of an entity.

Applications will create objects of this class for use in the *XMLReader.parse()* method and for returning from EntityResolver.resolveEntity.

An *InputSource* belongs to the application, the *XMLReader* is not allowed to modify *InputSource* objects passed to it from the application, although it may make copies and modify those.

class xml.sax.xmlreader.**AttributesImpl**(*attrs*)

This is an implementation of the `Attributes` interface (see section *The Attributes Interface*). This is a dictionary-like object which represents the element attributes in a `startElement()` call. In addition to the most useful dictionary operations, it supports a number of other methods as described by the interface. Objects of this class should be instantiated by readers; *attrs* must be a dictionary-like object containing a mapping from attribute names to attribute values.

class xml.sax.xmlreader.**AttributesNSImpl**(*attrs*, *qnames*)

Namespace-aware variant of *AttributesImpl*, which will be passed to `startElementNS()`. It is derived from *AttributesImpl*, but understands attribute names as two-tuples of *namespaceURI* and *localname*. In addition, it provides a number of methods expecting qualified names as they appear in the original document. This class implements the `AttributesNS` interface (see section *The AttributesNS Interface*).

20.12.1 XMLReader Objects

The *XMLReader* interface supports the following methods:

XMLReader.**parse**(*source*)

Process an input source, producing SAX events. The *source* object can be a system identifier (a string identifying the input source – typically a file name or a URL), a *pathlib.Path* or *path-like* object, or an *InputSource* object. When *parse()* returns, the input is completely processed, and the parser object can be discarded or reset.

Changed in version 3.5: Added support of character streams.

Changed in version 3.8: Added support of path-like objects.

XMLReader.**getContentHandler**()

Return the current *ContentHandler*.

XMLReader.**setContentHandler**(*handler*)

Set the current *ContentHandler*. If no *ContentHandler* is set, content events will be discarded.

XMLReader.**getDTDHandler**()

Return the current *DTDHandler*.

XMLReader.**setDTDHandler**(*handler*)

Set the current *DTDHandler*. If no *DTDHandler* is set, DTD events will be discarded.

XMLReader.**getEntityResolver**()

Return the current *EntityResolver*.

XMLReader.**setEntityResolver**(*handler*)

Set the current *EntityResolver*. If no *EntityResolver* is set, attempts to resolve an external entity will result in opening the system identifier for the entity, and fail if it is not available.

XMLReader.**getErrorHandler**()

Return the current *ErrorHandler*.

XMLReader.**setErrorHandler**(*handler*)

Set the current error handler. If no *ErrorHandler* is set, errors will be raised as exceptions, and warnings will be printed.

XMLReader.**setLocale**(*locale*)

Allow an application to set the locale for errors and warnings.

SAX parsers are not required to provide localization for errors and warnings; if they cannot support the requested locale, however, they must raise a SAX exception. Applications may request a locale change in the middle of a parse.

XMLReader.**getFeature**(*featurename*)

Return the current setting for feature *featurename*. If the feature is not recognized, SAXNotRecognizedException is raised. The well-known featurenames are listed in the module *xml.sax.handler*.

XMLReader.**setFeature**(*featurename*, *value*)

Set the *featurename* to *value*. If the feature is not recognized, SAXNotRecognizedException is raised. If the feature or its setting is not supported by the parser, *SAXNotSupportedException* is raised.

XMLReader.**getProperty**(*propertyname*)

Return the current setting for property *propertyname*. If the property is not recognized, a SAXNotRecognizedException is raised. The well-known propertynames are listed in the module *xml.sax.handler*.

XMLReader.**setProperty**(*propertyname*, *value*)

Set the *propertyname* to *value*. If the property is not recognized, SAXNotRecognizedException is raised. If the property or its setting is not supported by the parser, *SAXNotSupportedException* is raised.

20.12.2 IncrementalParser Objects

Instances of *IncrementalParser* offer the following additional methods:

IncrementalParser.**feed**(*data*)

Process a chunk of *data*.

IncrementalParser.**close**()

Assume the end of the document. That will check well-formedness conditions that can be checked only at the end, invoke handlers, and may clean up resources allocated during parsing.

IncrementalParser.**reset**()

This method is called after close has been called to reset the parser so that it is ready to parse new documents. The results of calling parse or feed after close without calling reset are undefined.

20.12.3 Locator Objects

Instances of *Locator* provide these methods:

Locator.**getColumnNumber**()

Return the column number where the current event begins.

Locator.**getLineNumber**()

Return the line number where the current event begins.

Locator.**getPublicId**()

Return the public identifier for the current event.

Locator.**getSystemId**()

Return the system identifier for the current event.

20.12.4 InputSource Objects

InputSource.**setPublicId**(*id*)

Sets the public identifier of this *InputSource*.

InputSource.**getPublicId**()

Returns the public identifier of this *InputSource*.

InputSource.**setSystemId**(*id*)

Sets the system identifier of this *InputSource*.

InputSource.**getSystemId**()

Returns the system identifier of this *InputSource*.

InputSource.**setEncoding**(*encoding*)

Sets the character encoding of this *InputSource*.

The encoding must be a string acceptable for an XML encoding declaration (see section 4.3.3 of the XML recommendation).

The encoding attribute of the *InputSource* is ignored if the *InputSource* also contains a character stream.

InputSource.**getEncoding**()

Get the character encoding of this InputSource.

InputSource.**setByteStream**(*bytefile*)

Set the byte stream (a *binary file*) for this input source.

The SAX parser will ignore this if there is also a character stream specified, but it will use a byte stream in preference to opening a URI connection itself.

If the application knows the character encoding of the byte stream, it should set it with the setEncoding method.

InputSource.**getByteStream**()

Get the byte stream for this input source.

The getEncoding method will return the character encoding for this byte stream, or None if unknown.

InputSource.**setCharacterStream**(*charfile*)

Set the character stream (a *text file*) for this input source.

If there is a character stream specified, the SAX parser will ignore any byte stream and will not attempt to open a URI connection to the system identifier.

InputSource.**getCharacterStream**()

Get the character stream for this input source.

20.12.5 The Attributes Interface

Attributes objects implement a portion of the *mapping protocol*, including the methods copy(), get(), __contains__(), items(), keys(), and values(). The following methods are also provided:

Attributes.**getLength**()

Return the number of attributes.

Attributes.**getNames**()

Return the names of the attributes.

Attributes.**getType**(*name*)

Returns the type of the attribute *name*, which is normally 'CDATA'.

Attributes.**getValue**(*name*)

Return the value of attribute *name*.

20.12.6 The `AttributesNS` Interface

This interface is a subtype of the `Attributes` interface (see section *The Attributes Interface*). All methods supported by that interface are also available on `AttributesNS` objects.

The following methods are also available:

`AttributesNS.`**`getValueByQName`**`(`*name*`)`

Return the value for a qualified name.

`AttributesNS.`**`getNameByQName`**`(`*name*`)`

Return the `(namespace, localname)` pair for a qualified *name*.

`AttributesNS.`**`getQNameByName`**`(`*name*`)`

Return the qualified name for a `(namespace, localname)` pair.

`AttributesNS.`**`getQNames`**`()`

Return the qualified names of all attributes.

20.13 `xml.parsers.expat` — Fast XML parsing using Expat

Warning

The `pyexpat` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML vulnerabilities*.

The *`xml.parsers.expat`* module is a Python interface to the Expat non-validating XML parser. The module provides a single extension type, `xmlparser`, that represents the current state of an XML parser. After an `xmlparser` object has been created, various attributes of the object can be set to handler functions. When an XML document is then fed to the parser, the handler functions are called for the character data and markup in the XML document.

This module uses the `pyexpat` module to provide access to the Expat parser. Direct use of the `pyexpat` module is deprecated.

This module provides one exception and one type object:

`exception` `xml.parsers.expat.`**`ExpatError`**

The exception raised when Expat reports an error. See section *ExpatError Exceptions* for more information on interpreting Expat errors.

`exception` `xml.parsers.expat.`**`error`**

Alias for *`ExpatError`*.

`xml.parsers.expat.`**`XMLParserType`**

The type of the return values from the *`ParserCreate()`* function.

The *`xml.parsers.expat`* module contains two functions:

`xml.parsers.expat.`**`ErrorString`**`(`*errno*`)`

Returns an explanatory string for a given error number *errno*.

xml.parsers.expat.**ParserCreate**(*encoding=None*, *namespace_separator=None*)

Creates and returns a new xmlparser object. *encoding*, if specified, must be a string naming the encoding used by the XML data. Expat doesn't support as many encodings as Python does, and its repertoire of encodings can't be extended; it supports UTF-8, UTF-16, ISO-8859-1 (Latin1), and ASCII. If *encoding*[1] is given it will override the implicit or explicit encoding of the document.

Expat can optionally do XML namespace processing for you, enabled by providing a value for *namespace_separator*. The value must be a one-character string; a *ValueError* will be raised if the string has an illegal length (None is considered the same as omission). When namespace processing is enabled, element type names and attribute names that belong to a namespace will be expanded. The element name passed to the element handlers StartElementHandler and EndElementHandler will be the concatenation of the namespace URI, the namespace separator character, and the local part of the name. If the namespace separator is a zero byte (chr(0)) then the namespace URI and the local part will be concatenated without any separator.

For example, if *namespace_separator* is set to a space character (' ') and the following document is parsed:

```
<?xml version="1.0"?>
<root xmlns    = "http://default-namespace.org/"
      xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

StartElementHandler will receive the following strings for each element:

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

Due to limitations in the Expat library used by pyexpat, the xmlparser instance returned can only be used to parse a single XML document. Call ParserCreate for each document to provide unique parser instances.

See also

The Expat XML Parser
Home page of the Expat project.

20.13.1 XMLParser Objects

xmlparser objects have the following methods:

xmlparser.**Parse**(*data*[, *isfinal*])

Parses the contents of the string *data*, calling the appropriate handler functions to process the parsed data. *isfinal* must be true on the final call to this method; it allows the parsing of a single file in fragments, not the submission of multiple files. *data* can be the empty string at any time.

xmlparser.**ParseFile**(*file*)

Parse XML data reading from the object *file*. *file* only needs to provide the read(nbytes) method, returning the empty string when there's no more data.

[1] The encoding string included in XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

xmlparser.**SetBase**(*base*)

Sets the base to be used for resolving relative URIs in system identifiers in declarations. Resolving relative identifiers is left to the application: this value will be passed through as the *base* argument to the *ExternalEntityRefHandler()*, *NotationDeclHandler()*, and *UnparsedEntityDeclHandler()* functions.

xmlparser.**GetBase**()

Returns a string containing the base set by a previous call to *SetBase()*, or `None` if *SetBase()* hasn't been called.

xmlparser.**GetInputContext**()

Returns the input data that generated the current event as a string. The data is in the encoding of the entity which contains the text. When called while an event handler is not active, the return value is `None`.

xmlparser.**ExternalEntityParserCreate**(*context*[, *encoding*])

Create a "child" parser which can be used to parse an external parsed entity referred to by content parsed by the parent parser. The *context* parameter should be the string passed to the *ExternalEntityRefHandler()* handler function, described below. The child parser is created with the *ordered_attributes* and *specified_attributes* set to the values of this parser.

xmlparser.**SetParamEntityParsing**(*flag*)

Control parsing of parameter entities (including the external DTD subset). Possible *flag* values are `XML_PARAM_ENTITY_PARSING_NEVER`, `XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE` and `XML_PARAM_ENTITY_PARSING_ALWAYS`. Return true if setting the flag was successful.

xmlparser.**UseForeignDTD**([*flag*])

Calling this with a true value for *flag* (the default) will cause Expat to call the *ExternalEntityRefHandler* with *None* for all arguments to allow an alternate DTD to be loaded. If the document does not contain a document type declaration, the *ExternalEntityRefHandler* will still be called, but the *StartDoctypeDeclHandler* and *EndDoctypeDeclHandler* will not be called.

Passing a false value for *flag* will cancel a previous call that passed a true value, but otherwise has no effect.

This method can only be called before the *Parse()* or *ParseFile()* methods are called; calling it after either of those have been called causes *ExpatError* to be raised with the *code* attribute set to `errors.codes[errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING]`.

xmlparser.**SetReparseDeferralEnabled**(*enabled*)

> **Warning**
>
> Calling `SetReparseDeferralEnabled(False)` has security implications, as detailed below; please make sure to understand these consequences prior to using the `SetReparseDeferralEnabled` method.

Expat 2.6.0 introduced a security mechanism called "reparse deferral" where instead of causing denial of service through quadratic runtime from reparsing large tokens, reparsing of unfinished tokens is now delayed by default until a sufficient amount of input is reached. Due to this delay, registered handlers may — depending of the sizing of input chunks pushed to Expat — no longer be called right after pushing new input to the parser. Where immediate feedback and taking over responsibility of protecting against denial of service from large tokens are both wanted, calling `SetReparseDeferralEnabled(False)` disables reparse deferral for the current Expat parser instance, temporarily or altogether. Calling `SetReparseDeferralEnabled(True)` allows re-enabling reparse deferral.

Note that *SetReparseDeferralEnabled()* has been backported to some prior releases of CPython as a security fix. Check for availability of *SetReparseDeferralEnabled()* using *hasattr()* if used in code running across a variety of Python versions.

Added in version 3.13.

xmlparser.**GetReparseDeferralEnabled**()

Returns whether reparse deferral is currently enabled for the given Expat parser instance.

Added in version 3.13.

xmlparser objects have the following attributes:

xmlparser.**buffer_size**

The size of the buffer used when *buffer_text* is true. A new buffer size can be set by assigning a new integer value to this attribute. When the size is changed, the buffer will be flushed.

xmlparser.**buffer_text**

Setting this to true causes the xmlparser object to buffer textual content returned by Expat to avoid multiple calls to the *CharacterDataHandler()* callback whenever possible. This can improve performance substantially since Expat normally breaks character data into chunks at every line ending. This attribute is false by default, and may be changed at any time. Note that when it is false, data that does not contain newlines may be chunked too.

xmlparser.**buffer_used**

If *buffer_text* is enabled, the number of bytes stored in the buffer. These bytes represent UTF-8 encoded text. This attribute has no meaningful interpretation when *buffer_text* is false.

xmlparser.**ordered_attributes**

Setting this attribute to a non-zero integer causes the attributes to be reported as a list rather than a dictionary. The attributes are presented in the order found in the document text. For each attribute, two list entries are presented: the attribute name and the attribute value. (Older versions of this module also used this format.) By default, this attribute is false; it may be changed at any time.

xmlparser.**specified_attributes**

If set to a non-zero integer, the parser will report only those attributes which were specified in the document instance and not those which were derived from attribute declarations. Applications which set this need to be especially careful to use what additional information is available from the declarations as needed to comply with the standards for the behavior of XML processors. By default, this attribute is false; it may be changed at any time.

The following attributes contain values relating to the most recent error encountered by an xmlparser object, and will only have correct values once a call to Parse() or ParseFile() has raised an *xml.parsers.expat.ExpatError* exception.

xmlparser.**ErrorByteIndex**

Byte index at which an error occurred.

xmlparser.**ErrorCode**

Numeric code specifying the problem. This value can be passed to the *ErrorString()* function, or compared to one of the constants defined in the errors object.

xmlparser.**ErrorColumnNumber**

Column number at which an error occurred.

xmlparser.**ErrorLineNumber**

Line number at which an error occurred.

The following attributes contain values relating to the current parse location in an xmlparser object. During a callback reporting a parse event they indicate the location of the first of the sequence of characters that generated the event. When called outside of a callback, the position indicated will be just past the last parse event (regardless of whether there was an associated callback).

xmlparser.**CurrentByteIndex**

Current byte index in the parser input.

xmlparser.**CurrentColumnNumber**

Current column number in the parser input.

xmlparser.**CurrentLineNumber**

Current line number in the parser input.

Here is the list of handlers that can be set. To set a handler on an `xmlparser` object *o*, use `o.handlername = func`. *handlername* must be taken from the following list, and *func* must be a callable object accepting the correct number of arguments. The arguments are all strings, unless otherwise stated.

xmlparser.**XmlDeclHandler**(*version*, *encoding*, *standalone*)

Called when the XML declaration is parsed. The XML declaration is the (optional) declaration of the applicable version of the XML recommendation, the encoding of the document text, and an optional "standalone" declaration. *version* and *encoding* will be strings, and *standalone* will be `1` if the document is declared standalone, `0` if it is declared not to be standalone, or `-1` if the standalone clause was omitted. This is only available with Expat version 1.95.0 or newer.

xmlparser.**StartDoctypeDeclHandler**(*doctypeName*, *systemId*, *publicId*, *has_internal_subset*)

Called when Expat begins parsing the document type declaration (`<!DOCTYPE ...`). The *doctypeName* is provided exactly as presented. The *systemId* and *publicId* parameters give the system and public identifiers if specified, or `None` if omitted. *has_internal_subset* will be true if the document contains and internal document declaration subset. This requires Expat version 1.2 or newer.

xmlparser.**EndDoctypeDeclHandler**()

Called when Expat is done parsing the document type declaration. This requires Expat version 1.2 or newer.

xmlparser.**ElementDeclHandler**(*name*, *model*)

Called once for each element type declaration. *name* is the name of the element type, and *model* is a representation of the content model.

xmlparser.**AttlistDeclHandler**(*elname*, *attname*, *type*, *default*, *required*)

Called for each declared attribute for an element type. If an attribute list declaration declares three attributes, this handler is called three times, once for each attribute. *elname* is the name of the element to which the declaration applies and *attname* is the name of the attribute declared. The attribute type is a string passed as *type*; the possible values are `'CDATA'`, `'ID'`, `'IDREF'`, ... *default* gives the default value for the attribute used when the attribute is not specified by the document instance, or `None` if there is no default value (`#IMPLIED` values). If the attribute is required to be given in the document instance, *required* will be true. This requires Expat version 1.95.0 or newer.

xmlparser.**StartElementHandler**(*name*, *attributes*)

Called for the start of every element. *name* is a string containing the element name, and *attributes* is the element attributes. If *ordered_attributes* is true, this is a list (see *ordered_attributes* for a full description). Otherwise it's a dictionary mapping names to values.

xmlparser.**EndElementHandler**(*name*)

Called for the end of every element.

xmlparser.**ProcessingInstructionHandler**(*target*, *data*)

Called for every processing instruction.

xmlparser.**CharacterDataHandler**(*data*)

Called for character data. This will be called for normal character data, CDATA marked content, and ignorable whitespace. Applications which must distinguish these cases can use the *StartCdataSectionHandler*, *EndCdataSectionHandler*, and *ElementDeclHandler* callbacks to collect the required information. Note that the character data may be chunked even if it is short and so you may receive more than one call to *CharacterDataHandler()*. Set the *buffer_text* instance attribute to `True` to avoid that.

xmlparser.**UnparsedEntityDeclHandler**(*entityName*, *base*, *systemId*, *publicId*, *notationName*)

Called for unparsed (NDATA) entity declarations. This is only present for version 1.2 of the Expat library; for more recent versions, use *EntityDeclHandler* instead. (The underlying function in the Expat library has been declared obsolete.)

xmlparser.**EntityDeclHandler**(*entityName*, *is_parameter_entity*, *value*, *base*, *systemId*, *publicId*, *notationName*)

Called for all entity declarations. For parameter and internal entities, *value* will be a string giving the declared contents of the entity; this will be None for external entities. The *notationName* parameter will be None for parsed entities, and the name of the notation for unparsed entities. *is_parameter_entity* will be true if the entity is a parameter entity or false for general entities (most applications only need to be concerned with general entities). This is only available starting with version 1.95.0 of the Expat library.

xmlparser.**NotationDeclHandler**(*notationName*, *base*, *systemId*, *publicId*)

Called for notation declarations. *notationName*, *base*, and *systemId*, and *publicId* are strings if given. If the public identifier is omitted, *publicId* will be None.

xmlparser.**StartNamespaceDeclHandler**(*prefix*, *uri*)

Called when an element contains a namespace declaration. Namespace declarations are processed before the *StartElementHandler* is called for the element on which declarations are placed.

xmlparser.**EndNamespaceDeclHandler**(*prefix*)

Called when the closing tag is reached for an element that contained a namespace declaration. This is called once for each namespace declaration on the element in the reverse of the order for which the *StartNamespaceDeclHandler* was called to indicate the start of each namespace declaration's scope. Calls to this handler are made after the corresponding *EndElementHandler* for the end of the element.

xmlparser.**CommentHandler**(*data*)

Called for comments. *data* is the text of the comment, excluding the leading '<!--' and trailing '-->'.

xmlparser.**StartCdataSectionHandler**()

Called at the start of a CDATA section. This and *EndCdataSectionHandler* are needed to be able to identify the syntactical start and end for CDATA sections.

xmlparser.**EndCdataSectionHandler**()

Called at the end of a CDATA section.

xmlparser.**DefaultHandler**(*data*)

Called for any characters in the XML document for which no applicable handler has been specified. This means characters that are part of a construct which could be reported, but for which no handler has been supplied.

xmlparser.**DefaultHandlerExpand**(*data*)

This is the same as the *DefaultHandler()*, but doesn't inhibit expansion of internal entities. The entity reference will not be passed to the default handler.

xmlparser.**NotStandaloneHandler**()

Called if the XML document hasn't been declared as being a standalone document. This happens when there is an external subset or a reference to a parameter entity, but the XML declaration does not set standalone to yes in an XML declaration. If this handler returns 0, then the parser will raise an XML_ERROR_NOT_STANDALONE error. If this handler is not set, no exception is raised by the parser for this condition.

xmlparser.**ExternalEntityRefHandler**(*context*, *base*, *systemId*, *publicId*)

Called for references to external entities. *base* is the current base, as set by a previous call to *SetBase()*. The public and system identifiers, *systemId* and *publicId*, are strings if given; if the public identifier is not given, *publicId* will be None. The *context* value is opaque and should only be used as described below.

For external entities to be parsed, this handler must be implemented. It is responsible for creating the sub-parser using `ExternalEntityParserCreate(context)`, initializing it with the appropriate callbacks, and parsing the entity. This handler should return an integer; if it returns `0`, the parser will raise an `XML_ERROR_EXTERNAL_ENTITY_HANDLING` error, otherwise parsing will continue.

If this handler is not provided, external entities are reported by the *DefaultHandler* callback, if provided.

20.13.2 ExpatError Exceptions

ExpatError exceptions have a number of interesting attributes:

ExpatError.**code**

Expat's internal error number for the specific error. The *errors.messages* dictionary maps these error numbers to Expat's error messages. For example:

```
from xml.parsers.expat import ParserCreate, ExpatError, errors

p = ParserCreate()
try:
    p.Parse(some_xml_document)
except ExpatError as err:
    print("Error:", errors.messages[err.code])
```

The *errors* module also provides error message constants and a dictionary *codes* mapping these messages back to the error codes, see below.

ExpatError.**lineno**

Line number on which the error was detected. The first line is numbered `1`.

ExpatError.**offset**

Character offset into the line where the error occurred. The first column is numbered `0`.

20.13.3 Example

The following program defines three handlers that just print out their arguments.

```
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print('Start element:', name, attrs)
def end_element(name):
    print('End element:', name)
def char_data(data):
    print('Character data:', repr(data))

p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
```

The output from this program is:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

20.13.4 Content Model Descriptions

Content models are described using nested tuples. Each tuple contains four values: the type, the quantifier, the name, and a tuple of children. Children are simply additional content model descriptions.

The values of the first two fields are constants defined in the *xml.parsers.expat.model* module. These constants can be collected in two groups: the model type group and the quantifier group.

The constants in the model type group are:

xml.parsers.expat.model.**XML_CTYPE_ANY**

The element named by the model name was declared to have a content model of `ANY`.

xml.parsers.expat.model.**XML_CTYPE_CHOICE**

The named element allows a choice from a number of options; this is used for content models such as `(A | B | C)`.

xml.parsers.expat.model.**XML_CTYPE_EMPTY**

Elements which are declared to be `EMPTY` have this model type.

xml.parsers.expat.model.**XML_CTYPE_MIXED**

xml.parsers.expat.model.**XML_CTYPE_NAME**

xml.parsers.expat.model.**XML_CTYPE_SEQ**

Models which represent a series of models which follow one after the other are indicated with this model type. This is used for models such as `(A, B, C)`.

The constants in the quantifier group are:

xml.parsers.expat.model.**XML_CQUANT_NONE**

No modifier is given, so it can appear exactly once, as for `A`.

xml.parsers.expat.model.**XML_CQUANT_OPT**

The model is optional: it can appear once or not at all, as for `A?`.

xml.parsers.expat.model.**XML_CQUANT_PLUS**

The model must occur one or more times (like `A+`).

xml.parsers.expat.model.**XML_CQUANT_REP**

The model must occur zero or more times, as for `A*`.

20.13.5 Expat error constants

The following constants are provided in the *xml.parsers.expat.errors* module. These constants are useful in interpreting some of the attributes of the ExpatError exception objects raised when an error has occurred. Since for backwards compatibility reasons, the constants' value is the error *message* and not the numeric error *code*, you do this by comparing its *code* attribute with errors.codes[errors.XML_ERROR_*CONSTANT_NAME*].

The errors module has the following attributes:

xml.parsers.expat.errors.**codes**

A dictionary mapping string descriptions to their error codes.

Added in version 3.2.

xml.parsers.expat.errors.**messages**

A dictionary mapping numeric error codes to their string descriptions.

Added in version 3.2.

xml.parsers.expat.errors.**XML_ERROR_ASYNC_ENTITY**

xml.parsers.expat.errors.**XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF**

An entity reference in an attribute value referred to an external entity instead of an internal entity.

xml.parsers.expat.errors.**XML_ERROR_BAD_CHAR_REF**

A character reference referred to a character which is illegal in XML (for example, character 0, or '&#0;').

xml.parsers.expat.errors.**XML_ERROR_BINARY_ENTITY_REF**

An entity reference referred to an entity which was declared with a notation, so cannot be parsed.

xml.parsers.expat.errors.**XML_ERROR_DUPLICATE_ATTRIBUTE**

An attribute was used more than once in a start tag.

xml.parsers.expat.errors.**XML_ERROR_INCORRECT_ENCODING**

xml.parsers.expat.errors.**XML_ERROR_INVALID_TOKEN**

Raised when an input byte could not properly be assigned to a character; for example, a NUL byte (value 0) in a UTF-8 input stream.

xml.parsers.expat.errors.**XML_ERROR_JUNK_AFTER_DOC_ELEMENT**

Something other than whitespace occurred after the document element.

xml.parsers.expat.errors.**XML_ERROR_MISPLACED_XML_PI**

An XML declaration was found somewhere other than the start of the input data.

xml.parsers.expat.errors.**XML_ERROR_NO_ELEMENTS**

The document contains no elements (XML requires all documents to contain exactly one top-level element)..

xml.parsers.expat.errors.**XML_ERROR_NO_MEMORY**

Expat was not able to allocate memory internally.

xml.parsers.expat.errors.**XML_ERROR_PARAM_ENTITY_REF**

A parameter entity reference was found where it was not allowed.

xml.parsers.expat.errors.**XML_ERROR_PARTIAL_CHAR**

An incomplete character was found in the input.

xml.parsers.expat.errors.**XML_ERROR_RECURSIVE_ENTITY_REF**

An entity reference contained another reference to the same entity; possibly via a different name, and possibly indirectly.

xml.parsers.expat.errors.**XML_ERROR_SYNTAX**

Some unspecified syntax error was encountered.

xml.parsers.expat.errors.**XML_ERROR_TAG_MISMATCH**

An end tag did not match the innermost open start tag.

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_TOKEN**

Some token (such as a start tag) was not closed before the end of the stream or the next token was encountered.

xml.parsers.expat.errors.**XML_ERROR_UNDEFINED_ENTITY**

A reference was made to an entity which was not defined.

xml.parsers.expat.errors.**XML_ERROR_UNKNOWN_ENCODING**

The document encoding is not supported by Expat.

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_CDATA_SECTION**

A CDATA marked section was not closed.

xml.parsers.expat.errors.**XML_ERROR_EXTERNAL_ENTITY_HANDLING**

xml.parsers.expat.errors.**XML_ERROR_NOT_STANDALONE**

The parser determined that the document was not "standalone" though it declared itself to be in the XML declaration, and the `NotStandaloneHandler` was set and returned `0`.

xml.parsers.expat.errors.**XML_ERROR_UNEXPECTED_STATE**

xml.parsers.expat.errors.**XML_ERROR_ENTITY_DECLARED_IN_PE**

xml.parsers.expat.errors.**XML_ERROR_FEATURE_REQUIRES_XML_DTD**

An operation was requested that requires DTD support to be compiled in, but Expat was configured without DTD support. This should never be reported by a standard build of the *xml.parsers.expat* module.

xml.parsers.expat.errors.**XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING**

A behavioral change was requested after parsing started that can only be changed before parsing has started. This is (currently) only raised by `UseForeignDTD()`.

xml.parsers.expat.errors.**XML_ERROR_UNBOUND_PREFIX**

An undeclared prefix was found when namespace processing was enabled.

xml.parsers.expat.errors.**XML_ERROR_UNDECLARING_PREFIX**

The document attempted to remove the namespace declaration associated with a prefix.

xml.parsers.expat.errors.**XML_ERROR_INCOMPLETE_PE**

A parameter entity contained incomplete markup.

xml.parsers.expat.errors.**XML_ERROR_XML_DECL**

The document contained no document element at all.

xml.parsers.expat.errors.**XML_ERROR_TEXT_DECL**

There was an error parsing a text declaration in an external entity.

xml.parsers.expat.errors.**XML_ERROR_PUBLICID**

Characters were found in the public id that are not allowed.

xml.parsers.expat.errors.**XML_ERROR_SUSPENDED**

The requested operation was made on a suspended parser, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

xml.parsers.expat.errors.**XML_ERROR_NOT_SUSPENDED**

An attempt to resume the parser was made when the parser had not been suspended.

xml.parsers.expat.errors.**XML_ERROR_ABORTED**

This should not be reported to Python applications.

xml.parsers.expat.errors.**XML_ERROR_FINISHED**

The requested operation was made on a parser which was finished parsing input, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

xml.parsers.expat.errors.**XML_ERROR_SUSPEND_PE**

xml.parsers.expat.errors.**XML_ERROR_RESERVED_PREFIX_XML**

An attempt was made to undeclare reserved namespace prefix `xml` or to bind it to another namespace URI.

xml.parsers.expat.errors.**XML_ERROR_RESERVED_PREFIX_XMLNS**

An attempt was made to declare or undeclare reserved namespace prefix `xmlns`.

xml.parsers.expat.errors.**XML_ERROR_RESERVED_NAMESPACE_URI**

An attempt was made to bind the URI of one the reserved namespace prefixes `xml` and `xmlns` to another namespace prefix.

xml.parsers.expat.errors.**XML_ERROR_INVALID_ARGUMENT**

This should not be reported to Python applications.

xml.parsers.expat.errors.**XML_ERROR_NO_BUFFER**

This should not be reported to Python applications.

xml.parsers.expat.errors.**XML_ERROR_AMPLIFICATION_LIMIT_BREACH**

The limit on input amplification factor (from DTD and entities) has been breached.

CHAPTER

TWENTYONE

INTERNET PROTOCOLS AND SUPPORT

The modules described in this chapter implement internet protocols and support for related technology. They are all implemented in Python. Most of these modules require the presence of the system-dependent module *socket*, which is currently supported on most popular platforms. Here is an overview:

21.1 webbrowser — Convenient web-browser controller

Source code: Lib/webbrowser.py

The *webbrowser* module provides a high-level interface to allow displaying web-based documents to users. Under most circumstances, simply calling the *open()* function from this module will do the right thing.

Under Unix, graphical browsers are preferred under X11, but text-mode browsers will be used if graphical browsers are not available or an X11 display isn't available. If text-mode browsers are used, the calling process will block until the user exits the browser.

If the environment variable BROWSER exists, it is interpreted as the *os.pathsep*-separated list of browsers to try ahead of the platform defaults. When the value of a list part contains the string %s, then it is interpreted as a literal browser command line to be used with the argument URL substituted for %s; if the part does not contain %s, it is simply interpreted as the name of the browser to launch.[1]

For non-Unix platforms, or when a remote browser is available on Unix, the controlling process will not wait for the user to finish with the browser, but allow the remote browser to maintain its own windows on the display. If remote browsers are not available on Unix, the controlling process will launch a new browser and wait.

On iOS, the BROWSER environment variable, as well as any arguments controlling autoraise, browser preference, and new tab/window creation will be ignored. Web pages will *always* be opened in the user's preferred browser, in a new tab, with the browser being brought to the foreground. The use of the *webbrowser* module on iOS requires the *ctypes* module. If *ctypes* isn't available, calls to *open()* will fail.

The script **webbrowser** can be used as a command-line interface for the module. It accepts a URL as the argument. It accepts the following optional parameters:

- -n/--new-window opens the URL in a new browser window, if possible.
- -t/--new-tab opens the URL in a new browser page ("tab").

The options are, naturally, mutually exclusive. Usage example:

```
python -m webbrowser -t "https://www.python.org"
```

[1] Executables named here without a full path will be searched in the directories given in the PATH environment variable.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

The following exception is defined:

exception webbrowser.**Error**

Exception raised when a browser control error occurs.

The following functions are defined:

webbrowser.**open**(*url*, *new=0*, *autoraise=True*)

Display *url* using the default browser. If *new* is 0, the *url* is opened in the same browser window if possible. If *new* is 1, a new browser window is opened if possible. If *new* is 2, a new browser page ("tab") is opened if possible. If *autoraise* is `True`, the window is raised if possible (note that under many window managers this will occur regardless of the setting of this variable).

Note that on some platforms, trying to open a filename using this function, may work and start the operating system's associated program. However, this is neither supported nor portable.

Raises an *auditing event* `webbrowser.open` with argument `url`.

webbrowser.**open_new**(*url*)

Open *url* in a new window of the default browser, if possible, otherwise, open *url* in the only browser window.

webbrowser.**open_new_tab**(*url*)

Open *url* in a new page ("tab") of the default browser, if possible, otherwise equivalent to *open_new()*.

webbrowser.**get**(*using=None*)

Return a controller object for the browser type *using*. If *using* is `None`, return a controller for a default browser appropriate to the caller's environment.

webbrowser.**register**(*name*, *constructor*, *instance=None*, *, *preferred=False*)

Register the browser type *name*. Once a browser type is registered, the *get()* function can return a controller for that browser type. If *instance* is not provided, or is `None`, *constructor* will be called without parameters to create an instance when needed. If *instance* is provided, *constructor* will never be called, and may be `None`.

Setting *preferred* to `True` makes this browser a preferred result for a *get()* call with no argument. Otherwise, this entry point is only useful if you plan to either set the `BROWSER` variable or call *get()* with a nonempty argument matching the name of a handler you declare.

Changed in version 3.7: *preferred* keyword-only parameter was added.

A number of browser types are predefined. This table gives the type names that may be passed to the *get()* function and the corresponding instantiations for the controller classes, all defined in this module.

Type Name	Class Name	Notes
'mozilla'	Mozilla('mozilla')	
'firefox'	Mozilla('mozilla')	
'epiphany'	Epiphany('epiphany')	
'kfmclient'	Konqueror()	(1)
'konqueror'	Konqueror()	(1)
'kfm'	Konqueror()	(1)
'opera'	Opera()	
'links'	GenericBrowser('links')	
'elinks'	Elinks('elinks')	
'lynx'	GenericBrowser('lynx')	
'w3m'	GenericBrowser('w3m')	
'windows-default'	WindowsDefault	(2)
'macosx'	MacOSXOSAScript('default')	(3)
'safari'	MacOSXOSAScript('safari')	(3)
'google-chrome'	Chrome('google-chrome')	
'chrome'	Chrome('chrome')	
'chromium'	Chromium('chromium')	
'chromium-browser'	Chromium('chromium-browser')	
'iosbrowser'	IOSBrowser	(4)

Notes:

(1) "Konqueror" is the file manager for the KDE desktop environment for Unix, and only makes sense to use if KDE is running. Some way of reliably detecting KDE would be nice; the KDEDIR variable is not sufficient. Note also that the name "kfm" is used even when using the **konqueror** command with KDE 2 — the implementation selects the best strategy for running Konqueror.

(2) Only on Windows platforms.

(3) Only on macOS.

(4) Only on iOS.

Added in version 3.2: A new MacOSXOSAScript class has been added and is used on Mac instead of the previous MacOSX class. This adds support for opening browsers not currently set as the OS default.

Added in version 3.3: Support for Chrome/Chromium has been added.

Changed in version 3.12: Support for several obsolete browsers has been removed. Removed browsers include Grail, Mosaic, Netscape, Galeon, Skipstone, Iceape, and Firefox versions 35 and below.

Changed in version 3.13: Support for iOS has been added.

Here are some simple examples:

```
url = 'https://docs.python.org/'

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url)

# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```

21.1.1 Browser Controller Objects

Browser controllers provide these methods which parallel three of the module-level convenience functions:

`controller.`**`name`**

System-dependent name for the browser.

`controller.`**`open`**(*url*, *new=0*, *autoraise=True*)

Display *url* using the browser handled by this controller. If *new* is 1, a new browser window is opened if possible. If *new* is 2, a new browser page ("tab") is opened if possible.

`controller.`**`open_new`**(*url*)

Open *url* in a new window of the browser handled by this controller, if possible, otherwise, open *url* in the only browser window. Alias *open_new()*.

`controller.`**`open_new_tab`**(*url*)

Open *url* in a new page ("tab") of the browser handled by this controller, if possible, otherwise equivalent to *open_new()*.

21.2 `wsgiref` — WSGI Utilities and Reference Implementation

Source code: Lib/wsgiref

The Web Server Gateway Interface (WSGI) is a standard interface between web server software and web applications written in Python. Having a standard interface makes it easy to use an application that supports WSGI with a number of different web servers.

Only authors of web servers and programming frameworks need to know every detail and corner case of the WSGI design. You don't need to understand every detail of WSGI just to install a WSGI application or to write a web application using an existing framework.

wsgiref is a reference implementation of the WSGI specification that can be used to add WSGI support to a web server or framework. It provides utilities for manipulating WSGI environment variables and response headers, base classes for implementing WSGI servers, a demo HTTP server that serves WSGI applications, types for static type checking, and a validation tool that checks WSGI servers and applications for conformance to the WSGI specification (**PEP 3333**).

See wsgi.readthedocs.io for more information about WSGI, and links to tutorials and other resources.

21.2.1 `wsgiref.util` – WSGI environment utilities

This module provides a variety of utility functions for working with WSGI environments. A WSGI environment is a dictionary containing HTTP request variables as described in **PEP 3333**. All of the functions taking an *environ* parameter expect a WSGI-compliant dictionary to be supplied; please see **PEP 3333** for a detailed specification and *WSGIEnvironment* for a type alias that can be used in type annotations.

`wsgiref.util.`**`guess_scheme`**(*environ*)

Return a guess for whether `wsgi.url_scheme` should be "http" or "https", by checking for a `HTTPS` environment variable in the *environ* dictionary. The return value is a string.

This function is useful when creating a gateway that wraps CGI or a CGI-like protocol such as FastCGI. Typically, servers providing such protocols will include a `HTTPS` variable with a value of "1", "yes", or "on" when a request is received via SSL. So, this function returns "https" if such a value is found, and "http" otherwise.

wsgiref.util.**request_uri**(*environ*, *include_query=True*)

Return the full request URI, optionally including the query string, using the algorithm found in the "URL Reconstruction" section of **PEP 3333**. If *include_query* is false, the query string is not included in the resulting URI.

wsgiref.util.**application_uri**(*environ*)

Similar to *request_uri()*, except that the PATH_INFO and QUERY_STRING variables are ignored. The result is the base URI of the application object addressed by the request.

wsgiref.util.**shift_path_info**(*environ*)

Shift a single name from PATH_INFO to SCRIPT_NAME and return the name. The *environ* dictionary is *modified* in-place; use a copy if you need to keep the original PATH_INFO or SCRIPT_NAME intact.

If there are no remaining path segments in PATH_INFO, None is returned.

Typically, this routine is used to process each portion of a request URI path, for example to treat the path as a series of dictionary keys. This routine modifies the passed-in environment to make it suitable for invoking another WSGI application that is located at the target URI. For example, if there is a WSGI application at /foo, and the request URI path is /foo/bar/baz, and the WSGI application at /foo calls *shift_path_info()*, it will receive the string "bar", and the environment will be updated to be suitable for passing to a WSGI application at /foo/bar. That is, SCRIPT_NAME will change from /foo to /foo/bar, and PATH_INFO will change from /bar/baz to /baz.

When PATH_INFO is just a "/", this routine returns an empty string and appends a trailing slash to SCRIPT_NAME, even though empty path segments are normally ignored, and SCRIPT_NAME doesn't normally end in a slash. This is intentional behavior, to ensure that an application can tell the difference between URIs ending in /x from ones ending in /x/ when using this routine to do object traversal.

wsgiref.util.**setup_testing_defaults**(*environ*)

Update *environ* with trivial defaults for testing purposes.

This routine adds various parameters required for WSGI, including HTTP_HOST, SERVER_NAME, SERVER_PORT, REQUEST_METHOD, SCRIPT_NAME, PATH_INFO, and all of the **PEP 3333**-defined wsgi.* variables. It only supplies default values, and does not replace any existing settings for these variables.

This routine is intended to make it easier for unit tests of WSGI servers and applications to set up dummy environments. It should NOT be used by actual WSGI servers or applications, since the data is fake!

Example usage:

```
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# A relatively simple WSGI application. It's going to print out the
# environment dictionary after being updated by setup_testing_defaults
def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('Content-type', 'text/plain; charset=utf-8')]

    start_response(status, headers)

    ret = [("%s: %s\n" % (key, value)).encode("utf-8")
           for key, value in environ.items()]
    return ret

with make_server('', 8000, simple_app) as httpd:
    print("Serving on port 8000...")
    httpd.serve_forever()
```

In addition to the environment functions above, the *wsgiref.util* module also provides these miscellaneous utilities:

wsgiref.util.**is_hop_by_hop**(*header_name*)

Return `True` if 'header_name' is an HTTP/1.1 "Hop-by-Hop" header, as defined by **RFC 2616**.

class wsgiref.util.**FileWrapper**(*filelike*, *blksize=8192*)

A concrete implementation of the *wsgiref.types.FileWrapper* protocol used to convert a file-like object to an *iterator*. The resulting objects are *iterable*s. As the object is iterated over, the optional *blksize* parameter will be repeatedly passed to the *filelike* object's `read()` method to obtain bytestrings to yield. When `read()` returns an empty bytestring, iteration is ended and is not resumable.

If *filelike* has a `close()` method, the returned object will also have a `close()` method, and it will invoke the *filelike* object's `close()` method when called.

Example usage:

```
from io import StringIO
from wsgiref.util import FileWrapper

# We're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like object"*10)
wrapper = FileWrapper(filelike, blksize=5)

for chunk in wrapper:
    print(chunk)
```

Changed in version 3.11: Support for `__getitem__()` method has been removed.

21.2.2 wsgiref.headers – WSGI response header tools

This module provides a single class, *Headers*, for convenient manipulation of WSGI response headers using a mapping-like interface.

class wsgiref.headers.**Headers**([*headers*])

Create a mapping-like object wrapping *headers*, which must be a list of header name/value tuples as described in **PEP 3333**. The default value of *headers* is an empty list.

Headers objects support typical mapping operations including `__getitem__()`, *get()*, `__setitem__()`, *setdefault()*, `__delitem__()` and `__contains__()`. For each of these methods, the key is the header name (treated case-insensitively), and the value is the first value associated with that header name. Setting a header deletes any existing values for that header, then adds a new value at the end of the wrapped header list. Headers' existing order is generally maintained, with new headers added to the end of the wrapped list.

Unlike a dictionary, *Headers* objects do not raise an error when you try to get or delete a key that isn't in the wrapped header list. Getting a nonexistent header just returns `None`, and deleting a nonexistent header does nothing.

Headers objects also support `keys()`, `values()`, and `items()` methods. The lists returned by `keys()` and `items()` can include the same key more than once if there is a multi-valued header. The `len()` of a *Headers* object is the same as the length of its `items()`, which is the same as the length of the wrapped header list. In fact, the `items()` method just returns a copy of the wrapped header list.

Calling `bytes()` on a *Headers* object returns a formatted bytestring suitable for transmission as HTTP response headers. Each header is placed on a line with its value, separated by a colon and a space. Each line is terminated by a carriage return and line feed, and the bytestring is terminated with a blank line.

In addition to their mapping interface and formatting features, *Headers* objects also have the following methods for querying and adding multi-valued headers, and for adding headers with MIME parameters:

get_all(*name*)

Return a list of all the values for the named header.

The returned list will be sorted in the order they appeared in the original header list or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list. If no fields exist with the given name, returns an empty list.

add_header(*name*, *value*, ***_params*)

Add a (possibly multi-valued) header, with optional MIME parameters specified via keyword arguments.

name is the header field to add. Keyword arguments can be used to set MIME parameters for the header field. Each parameter must be a string or `None`. Underscores in parameter names are converted to dashes, since dashes are illegal in Python identifiers, but many MIME parameter names include dashes. If the parameter value is a string, it is added to the header value parameters in the form `name="value"`. If it is `None`, only the parameter name is added. (This is used for MIME parameters without a value.) Example usage:

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

The above will add a header that looks like this:

```
Content-Disposition: attachment; filename="bud.gif"
```

Changed in version 3.5: *headers* parameter is optional.

21.2.3 wsgiref.simple_server – a simple WSGI HTTP server

This module implements a simple HTTP server (based on *http.server*) that serves WSGI applications. Each server instance serves a single WSGI application on a given host and port. If you want to serve multiple applications on a single host and port, you should create a WSGI application that parses `PATH_INFO` to select which application to invoke for each request. (E.g., using the `shift_path_info()` function from *wsgiref.util*.)

wsgiref.simple_server.**make_server**(*host*, *port*, *app*, *server_class=WSGIServer*, *handler_class=WSGIRequestHandler*)

Create a new WSGI server listening on *host* and *port*, accepting connections for *app*. The return value is an instance of the supplied *server_class*, and will process requests using the specified *handler_class*. *app* must be a WSGI application object, as defined by **PEP 3333**.

Example usage:

```
from wsgiref.simple_server import make_server, demo_app

with make_server('', 8000, demo_app) as httpd:
    print("Serving HTTP on port 8000...")

    # Respond to requests until process is killed
    httpd.serve_forever()

    # Alternative: serve one request, then exit
    httpd.handle_request()
```

wsgiref.simple_server.**demo_app**(*environ*, *start_response*)

This function is a small but complete WSGI application that returns a text page containing the message "Hello world!" and a list of the key/value pairs provided in the *environ* parameter. It's useful for verifying that a WSGI server (such as *wsgiref.simple_server*) is able to run a simple WSGI application correctly.

class wsgiref.simple_server.**WSGIServer**(*server_address*, *RequestHandlerClass*)

Create a *WSGIServer* instance. *server_address* should be a `(host,port)` tuple, and *RequestHandlerClass* should be the subclass of *http.server.BaseHTTPRequestHandler* that will be used to process requests.

You do not normally need to call this constructor, as the *make_server()* function can handle all the details for you.

WSGIServer is a subclass of *http.server.HTTPServer*, so all of its methods (such as `serve_forever()` and `handle_request()`) are available. *WSGIServer* also provides these WSGI-specific methods:

set_app(*application*)

Sets the callable *application* as the WSGI application that will receive requests.

get_app()

Returns the currently set application callable.

Normally, however, you do not need to use these additional methods, as *set_app()* is normally called by *make_server()*, and the *get_app()* exists mainly for the benefit of request handler instances.

class wsgiref.simple_server.**WSGIRequestHandler**(*request*, *client_address*, *server*)

Create an HTTP handler for the given *request* (i.e. a socket), *client_address* (a `(host,port)` tuple), and *server* (*WSGIServer* instance).

You do not need to create instances of this class directly; they are automatically created as needed by *WSGIServer* objects. You can, however, subclass this class and supply it as a *handler_class* to the *make_server()* function. Some possibly relevant methods for overriding in subclasses:

get_environ()

Return a *WSGIEnvironment* dictionary for a request. The default implementation copies the contents of the *WSGIServer* object's `base_environ` dictionary attribute and then adds various headers derived from the HTTP request. Each call to this method should return a new dictionary containing all of the relevant CGI environment variables as specified in **PEP 3333**.

get_stderr()

Return the object that should be used as the `wsgi.errors` stream. The default implementation just returns `sys.stderr`.

handle()

Process the HTTP request. The default implementation creates a handler instance using a *wsgiref.handlers* class to implement the actual WSGI application interface.

21.2.4 wsgiref.validate — WSGI conformance checker

When creating new WSGI application objects, frameworks, servers, or middleware, it can be useful to validate the new code's conformance using *wsgiref.validate*. This module provides a function that creates WSGI application objects that validate communications between a WSGI server or gateway and a WSGI application object, to check both sides for protocol conformance.

Note that this utility does not guarantee complete **PEP 3333** compliance; an absence of errors from this module does not necessarily mean that errors do not exist. However, if this module does produce an error, then it is virtually certain that either the server or application is not 100% compliant.

This module is based on the `paste.lint` module from Ian Bicking's "Python Paste" library.

wsgiref.validate.**validator**(*application*)

Wrap *application* and return a new WSGI application object. The returned application will forward all requests to the original *application*, and will check that both the *application* and the server invoking it are conforming to the WSGI specification and to **RFC 2616**.

Any detected nonconformance results in an *AssertionError* being raised; note, however, that how these errors are handled is server-dependent. For example, *wsgiref.simple_server* and other servers based on *wsgiref.handlers* (that don't override the error handling methods to do something else) will simply output a message that an error has occurred, and dump the traceback to sys.stderr or some other error stream.

This wrapper may also generate output using the *warnings* module to indicate behaviors that are questionable but which may not actually be prohibited by **PEP 3333**. Unless they are suppressed using Python command-line options or the *warnings* API, any such warnings will be written to sys.stderr (*not* wsgi.errors, unless they happen to be the same object).

Example usage:

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# Our callable object which is intentionally not compliant to the
# standard, so the validator is going to break
def simple_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)

    # This is going to break because we need to return a list, and
    # the validator is going to inform us
    return b"Hello World"

# This is the application wrapped in a validator
validator_app = validator(simple_app)

with make_server('', 8000, validator_app) as httpd:
    print("Listening on port 8000....")
    httpd.serve_forever()
```

21.2.5 wsgiref.handlers – server/gateway base classes

This module provides base handler classes for implementing WSGI servers and gateways. These base classes handle most of the work of communicating with a WSGI application, as long as they are given a CGI-like environment, along with input, output, and error streams.

class wsgiref.handlers.**CGIHandler**

CGI-based invocation via sys.stdin, sys.stdout, sys.stderr and os.environ. This is useful when you have a WSGI application and want to run it as a CGI script. Simply invoke CGIHandler().run(app), where app is the WSGI application object you wish to invoke.

This class is a subclass of *BaseCGIHandler* that sets wsgi.run_once to true, wsgi.multithread to false, and wsgi.multiprocess to true, and always uses *sys* and *os* to obtain the necessary CGI streams and environment.

class wsgiref.handlers.**IISCGIHandler**

A specialized alternative to *CGIHandler*, for use when deploying on Microsoft's IIS web server, without having set the config allowPathInfo option (IIS>=7) or metabase allowPathInfoForScriptMappings (IIS<7).

By default, IIS gives a PATH_INFO that duplicates the SCRIPT_NAME at the front, causing problems for WSGI applications that wish to implement routing. This handler strips any such duplicated path.

IIS can be configured to pass the correct PATH_INFO, but this causes another bug where PATH_TRANSLATED is wrong. Luckily this variable is rarely used and is not guaranteed by WSGI. On IIS<7, though, the setting can only be made on a vhost level, affecting all other script mappings, many of which break when exposed to the PATH_TRANSLATED bug. For this reason IIS<7 is almost never deployed with the fix (Even IIS7 rarely uses it because there is still no UI for it.).

There is no way for CGI code to tell whether the option was set, so a separate handler class is provided. It is used in the same way as *CGIHandler*, i.e., by calling `IISCGIHandler().run(app)`, where `app` is the WSGI application object you wish to invoke.

Added in version 3.2.

class wsgiref.handlers.**BaseCGIHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

Similar to *CGIHandler*, but instead of using the *sys* and *os* modules, the CGI environment and I/O streams are specified explicitly. The *multithread* and *multiprocess* values are used to set the `wsgi.multithread` and `wsgi.multiprocess` flags for any applications run by the handler instance.

This class is a subclass of *SimpleHandler* intended for use with software other than HTTP "origin servers". If you are writing a gateway protocol implementation (such as CGI, FastCGI, SCGI, etc.) that uses a `Status:` header to send an HTTP status, you probably want to subclass this instead of *SimpleHandler*.

class wsgiref.handlers.**SimpleHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

Similar to *BaseCGIHandler*, but designed for use with HTTP origin servers. If you are writing an HTTP server implementation, you will probably want to subclass this instead of *BaseCGIHandler*.

This class is a subclass of *BaseHandler*. It overrides the `__init__()`, *get_stdin()*, *get_stderr()*, *add_cgi_vars()*, *_write()*, and *_flush()* methods to support explicitly setting the environment and streams via the constructor. The supplied environment and streams are stored in the `stdin`, `stdout`, `stderr`, and `environ` attributes.

The *write()* method of *stdout* should write each chunk in full, like *io.BufferedIOBase*.

class wsgiref.handlers.**BaseHandler**

This is an abstract base class for running WSGI applications. Each instance will handle a single HTTP request, although in principle you could create a subclass that was reusable for multiple requests.

BaseHandler instances have only one method intended for external use:

run(*app*)

Run the specified WSGI application, *app*.

All of the other *BaseHandler* methods are invoked by this method in the process of running the application, and thus exist primarily to allow customizing the process.

The following methods MUST be overridden in a subclass:

_write(*data*)

Buffer the bytes *data* for transmission to the client. It's okay if this method actually transmits the data; *BaseHandler* just separates write and flush operations for greater efficiency when the underlying system actually has such a distinction.

_flush()

Force buffered data to be transmitted to the client. It's okay if this method is a no-op (i.e., if *_write()* actually sends the data).

get_stdin()

Return an object compatible with *InputStream* suitable for use as the `wsgi.input` of the request currently being processed.

get_stderr()

Return an object compatible with *ErrorStream* suitable for use as the `wsgi.errors` of the request currently being processed.

add_cgi_vars()

Insert CGI variables for the current request into the `environ` attribute.

Here are some other methods and attributes you may wish to override. This list is only a summary, however, and does not include every method that can be overridden. You should consult the docstrings and source code for additional information before attempting to create a customized *BaseHandler* subclass.

Attributes and methods for customizing the WSGI environment:

wsgi_multithread

The value to be used for the `wsgi.multithread` environment variable. It defaults to true in *BaseHandler*, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_multiprocess

The value to be used for the `wsgi.multiprocess` environment variable. It defaults to true in *BaseHandler*, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_run_once

The value to be used for the `wsgi.run_once` environment variable. It defaults to false in *BaseHandler*, but *CGIHandler* sets it to true by default.

os_environ

The default environment variables to be included in every request's WSGI environment. By default, this is a copy of `os.environ` at the time that *wsgiref.handlers* was imported, but subclasses can either create their own at the class or instance level. Note that the dictionary should be considered read-only, since the default value is shared between multiple classes and instances.

server_software

If the *origin_server* attribute is set, this attribute's value is used to set the default `SERVER_SOFTWARE` WSGI environment variable, and also to set a default `Server:` header in HTTP responses. It is ignored for handlers (such as *BaseCGIHandler* and *CGIHandler*) that are not HTTP origin servers.

Changed in version 3.3: The term "Python" is replaced with implementation specific term like "CPython", "Jython" etc.

get_scheme()

Return the URL scheme being used for the current request. The default implementation uses the `guess_scheme()` function from *wsgiref.util* to guess whether the scheme should be "http" or "https", based on the current request's `environ` variables.

setup_environ()

Set the `environ` attribute to a fully populated WSGI environment. The default implementation uses all of the above methods and attributes, plus the *get_stdin()*, *get_stderr()*, and *add_cgi_vars()* methods and the *wsgi_file_wrapper* attribute. It also inserts a `SERVER_SOFTWARE` key if not present, as long as the *origin_server* attribute is a true value and the *server_software* attribute is set.

Methods and attributes for customizing exception handling:

log_exception(*exc_info*)

Log the *exc_info* tuple in the server log. *exc_info* is a `(type, value, traceback)` tuple. The default implementation simply writes the traceback to the request's `wsgi.errors` stream and flushes it. Subclasses can override this method to change the format or retarget the output, mail the traceback to an administrator, or whatever other action may be deemed suitable.

traceback_limit

The maximum number of frames to include in tracebacks output by the default *log_exception()* method. If `None`, all frames are included.

error_output(*environ*, *start_response*)

This method is a WSGI application to generate an error page for the user. It is only invoked if an error occurs before headers are sent to the client.

This method can access the current error using `sys.exception()`, and should pass that information to *start_response* when calling it (as described in the "Error Handling" section of **PEP 3333**).

The default implementation just uses the *error_status*, *error_headers*, and *error_body* attributes to generate an output page. Subclasses can override this to produce more dynamic error output.

Note, however, that it's not recommended from a security perspective to spit out diagnostics to any old user; ideally, you should have to do something special to enable diagnostic output, which is why the default implementation doesn't include any.

error_status

The HTTP status used for error responses. This should be a status string as defined in **PEP 3333**; it defaults to a 500 code and message.

error_headers

The HTTP headers used for error responses. This should be a list of WSGI response headers (`(name, value)` tuples), as described in **PEP 3333**. The default list just sets the content type to `text/plain`.

error_body

The error response body. This should be an HTTP response body bytestring. It defaults to the plain text, "A server error occurred. Please contact the administrator."

Methods and attributes for **PEP 3333**'s "Optional Platform-Specific File Handling" feature:

wsgi_file_wrapper

A `wsgi.file_wrapper` factory, compatible with *wsgiref.types.FileWrapper*, or `None`. The default value of this attribute is the *wsgiref.util.FileWrapper* class.

sendfile()

Override to implement platform-specific file transmission. This method is called only if the application's return value is an instance of the class specified by the *wsgi_file_wrapper* attribute. It should return a true value if it was able to successfully transmit the file, so that the default transmission code will not be executed. The default implementation of this method just returns a false value.

Miscellaneous methods and attributes:

origin_server

This attribute should be set to a true value if the handler's *_write()* and *_flush()* are being used to communicate directly to the client, rather than via a CGI-like gateway protocol that wants the HTTP status in a special `Status:` header.

This attribute's default value is true in *BaseHandler*, but false in *BaseCGIHandler* and *CGIHandler*.

http_version

If *origin_server* is true, this string attribute is used to set the HTTP version of the response set to the client. It defaults to `"1.0"`.

`wsgiref.handlers.`**read_environ**`()`

Transcode CGI variables from `os.environ` to **PEP 3333** "bytes in unicode" strings, returning a new dictionary. This function is used by *CGIHandler* and *IISCGIHandler* in place of directly using `os.environ`, which is not necessarily WSGI-compliant on all platforms and web servers using Python 3 – specifically, ones where the OS's actual environment is Unicode (i.e. Windows), or ones where the environment is bytes, but the system encoding used by Python to decode it is anything other than ISO-8859-1 (e.g. Unix systems using UTF-8).

If you are implementing a CGI-based handler of your own, you probably want to use this routine instead of just copying values out of `os.environ` directly.

Added in version 3.2.

21.2.6 `wsgiref.types` – WSGI types for static type checking

This module provides various types for static type checking as described in **PEP 3333**.

Added in version 3.11.

class wsgiref.types.**StartResponse**

A *typing.Protocol* describing start_response() callables (**PEP 3333**).

wsgiref.types.**WSGIEnvironment**

A type alias describing a WSGI environment dictionary.

wsgiref.types.**WSGIApplication**

A type alias describing a WSGI application callable.

class wsgiref.types.**InputStream**

A *typing.Protocol* describing a WSGI Input Stream.

class wsgiref.types.**ErrorStream**

A *typing.Protocol* describing a WSGI Error Stream.

class wsgiref.types.**FileWrapper**

A *typing.Protocol* describing a file wrapper. See *wsgiref.util.FileWrapper* for a concrete implementation of this protocol.

21.2.7 Examples

This is a working "Hello World" WSGI application:

```
"""
Every WSGI application must have an application object - a callable
object that accepts two arguments. For that purpose, we're going to
use a function (note that you're not limited to a function, you can
use a class for example). The first argument passed to the function
is a dictionary containing CGI-style environment variables and the
second variable is the callable object.
"""
from wsgiref.simple_server import make_server
```

(continues on next page)

(continued from previous page)

```
def hello_world_app(environ, start_response):
    status = "200 OK"  # HTTP Status
    headers = [("Content-type", "text/plain; charset=utf-8")]  # HTTP Headers
    start_response(status, headers)

    # The returned object is going to be printed
    return [b"Hello World"]

with make_server("", 8000, hello_world_app) as httpd:
    print("Serving on port 8000...")

    # Serve until process is killed
    httpd.serve_forever()
```

Example of a WSGI application serving the current directory, accept optional directory and port number (default: 8000) on the command line:

```
"""
Small wsgiref based web server. Takes a path to serve from and an
optional port number (defaults to 8000), then tries to serve files.
MIME types are guessed from the file names, 404 errors are raised
if the file is not found.
"""
import mimetypes
import os
import sys
from wsgiref import simple_server, util


def app(environ, respond):
    # Get the file name and MIME type
    fn = os.path.join(path, environ["PATH_INFO"][1:])
    if "." not in fn.split(os.path.sep)[-1]:
        fn = os.path.join(fn, "index.html")
    mime_type = mimetypes.guess_file_type(fn)[0]

    # Return 200 OK if file exists, otherwise 404 Not Found
    if os.path.exists(fn):
        respond("200 OK", [("Content-Type", mime_type)])
        return util.FileWrapper(open(fn, "rb"))
    else:
        respond("404 Not Found", [("Content-Type", "text/plain")])
        return [b"not found"]


if __name__ == "__main__":
    # Get the path and port from command-line arguments
    path = sys.argv[1] if len(sys.argv) > 1 else os.getcwd()
    port = int(sys.argv[2]) if len(sys.argv) > 2 else 8000

    # Make and start the server until control-c
    httpd = simple_server.make_server("", port, app)
    print(f"Serving {path} on port {port}, control-C to stop")
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
```

(continues on next page)

(continued from previous page)

```
        print("Shutting down.")
        httpd.server_close()
```

21.3 `urllib` — URL handling modules

Source code: Lib/urllib/

`urllib` is a package that collects several modules for working with URLs:

- *`urllib.request`* for opening and reading URLs
- *`urllib.error`* containing the exceptions raised by *`urllib.request`*
- *`urllib.parse`* for parsing URLs
- *`urllib.robotparser`* for parsing `robots.txt` files

21.4 `urllib.request` — Extensible library for opening URLs

Source code: Lib/urllib/request.py

The *`urllib.request`* module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more.

See also

The Requests package is recommended for a higher-level HTTP client interface.

Warning

On macOS it is unsafe to use this module in programs using *`os.fork()`* because the *`getproxies()`* implementation for macOS uses a higher-level system API. Set the environment variable `no_proxy` to `*` to avoid this problem (e.g. `os.environ["no_proxy"] = "*"`).

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

The *`urllib.request`* module defines the following functions:

`urllib.request.`**`urlopen`**(*url*, *data=None*, [*timeout*,]*, *context=None*)

Open *url*, which can be either a string containing a valid, properly encoded URL, or a *`Request`* object.

data must be an object specifying additional data to be sent to the server, or `None` if no such data is needed. See *`Request`* for details.

urllib.request module uses HTTP/1.1 and includes `Connection:close` header in its HTTP requests.

The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). This actually only works for HTTP, HTTPS and FTP connections.

If *context* is specified, it must be a *ssl.SSLContext* instance describing the various SSL options. See *HTTPSConnection* for more details.

This function always returns an object which can work as a *context manager* and has the properties *url*, *headers*, and *status*. See *urllib.response.addinfourl* for more detail on these properties.

For HTTP and HTTPS URLs, this function returns a *http.client.HTTPResponse* object slightly modified. In addition to the three new methods above, the msg attribute contains the same information as the *reason* attribute — the reason phrase returned by server — instead of the response headers as it is specified in the documentation for *HTTPResponse*.

For FTP, file, and data URLs and requests explicitly handled by legacy *URLopener* and *FancyURLopener* classes, this function returns a *urllib.response.addinfourl* object.

Raises *URLError* on protocol errors.

Note that `None` may be returned if no handler handles the request (though the default installed global *OpenerDirector* uses *UnknownHandler* to ensure this never happens).

In addition, if proxy settings are detected (for example, when a `*_proxy` environment variable like `http_proxy` is set), *ProxyHandler* is default installed and makes sure the requests are handled through the proxy.

The legacy `urllib.urlopen` function from Python 2.6 and earlier has been discontinued; *urllib.request.urlopen()* corresponds to the old `urllib2.urlopen`. Proxy handling, which was done by passing a dictionary parameter to `urllib.urlopen`, can be obtained by using *ProxyHandler* objects.

The default opener raises an *auditing event* `urllib.Request` with arguments `fullurl`, `data`, `headers`, `method` taken from the request object.

Changed in version 3.2: *cafile* and *capath* were added.

HTTPS virtual hosts are now supported if possible (that is, if *ssl.HAS_SNI* is true).

data can be an iterable object.

Changed in version 3.3: *cadefault* was added.

Changed in version 3.4.3: *context* was added.

Changed in version 3.10: HTTPS connection now send an ALPN extension with protocol indicator `http/1.1` when no *context* is given. Custom *context* should set ALPN protocols with *set_alpn_protocols()*.

Changed in version 3.13: Remove *cafile*, *capath* and *cadefault* parameters: use the *context* parameter instead.

`urllib.request.`**`install_opener`**(*opener*)

Install an *OpenerDirector* instance as the default global opener. Installing an opener is only necessary if you want urlopen to use that opener; otherwise, simply call *OpenerDirector.open()* instead of *urlopen()*. The code does not check for a real *OpenerDirector*, and any class with the appropriate interface will work.

`urllib.request.`**`build_opener`**([*handler*, ...])

Return an *OpenerDirector* instance, which chains the handlers in the order given. *handler*s can be either instances of *BaseHandler*, or subclasses of *BaseHandler* (in which case it must be possible to call the constructor without any parameters). Instances of the following classes will be in front of the *handler*s, unless the *handler*s contain them, instances of them or subclasses of them: *ProxyHandler* (if proxy settings are detected), *UnknownHandler*, *HTTPHandler*, *HTTPDefaultErrorHandler*, *HTTPRedirectHandler*, *FTPHandler*, *FileHandler*, *HTTPErrorProcessor*.

If the Python installation has SSL support (i.e., if the *ssl* module can be imported), *HTTPSHandler* will also be added.

A *BaseHandler* subclass may also change its `handler_order` attribute to modify its position in the handlers list.

`urllib.request.`**`pathname2url`**(*path*)

Convert the pathname *path* from the local syntax for a path to the form used in the path component of a URL. This does not produce a complete URL. The return value will already be quoted using the *quote()* function.

`urllib.request.`**`url2pathname`**(*path*)

Convert the path component *path* from a percent-encoded URL to the local syntax for a path. This does not accept a complete URL. This function uses *unquote()* to decode *path*.

`urllib.request.`**`getproxies`**()

This helper function returns a dictionary of scheme to proxy server URL mappings. It scans the environment for variables named `<scheme>_proxy`, in a case insensitive approach, for all operating systems first, and when it cannot find it, looks for proxy information from System Configuration for macOS and Windows Systems Registry for Windows. If both lowercase and uppercase environment variables exist (and disagree), lowercase is preferred.

> **Note**
>
> If the environment variable `REQUEST_METHOD` is set, which usually indicates your script is running in a CGI environment, the environment variable `HTTP_PROXY` (uppercase `_PROXY`) will be ignored. This is because that variable can be injected by a client using the "Proxy:" HTTP header. If you need to use an HTTP proxy in a CGI environment, either use `ProxyHandler` explicitly, or make sure the variable name is in lowercase (or at least the `_proxy` suffix).

The following classes are provided:

`class` `urllib.request.`**`Request`**(*url*, *data=None*, *headers={}*, *origin_req_host=None*, *unverifiable=False*, *method=None*)

This class is an abstraction of a URL request.

url should be a string containing a valid, properly encoded URL.

data must be an object specifying additional data to send to the server, or `None` if no such data is needed. Currently HTTP requests are the only ones that use *data*. The supported object types include bytes, file-like objects, and iterables of bytes-like objects. If no `Content-Length` nor `Transfer-Encoding` header field has been provided, *HTTPHandler* will set these headers according to the type of *data*. `Content-Length` will be used to send bytes objects, while `Transfer-Encoding: chunked` as specified in **RFC 7230**, Section 3.3.1 will be used to send files and other iterables.

For an HTTP POST request method, *data* should be a buffer in the standard *application/x-www-form-urlencoded* format. The *urllib.parse.urlencode()* function takes a mapping or sequence of 2-tuples and returns an ASCII string in this format. It should be encoded to bytes before being used as the *data* parameter.

headers should be a dictionary, and will be treated as if *add_header()* was called with each key and value as arguments. This is often used to "spoof" the `User-Agent` header value, which is used by a browser to identify itself – some HTTP servers only allow requests coming from common browsers as opposed to scripts. For example, Mozilla Firefox may identify itself as `"Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11"`, while *urllib*'s default user agent string is `"Python-urllib/2.6"` (on Python 2.6). All header keys are sent in camel case.

An appropriate `Content-Type` header should be included if the *data* argument is present. If this header has not been provided and *data* is not `None`, `Content-Type: application/x-www-form-urlencoded`

will be added as a default.

The next two arguments are only of interest for correct handling of third-party HTTP cookies:

origin_req_host should be the request-host of the origin transaction, as defined by **RFC 2965**. It defaults to `http.cookiejar.request_host(self)`. This is the host name or IP address of the original request that was initiated by the user. For example, if the request is for an image in an HTML document, this should be the request-host of the request for the page containing the image.

unverifiable should indicate whether the request is unverifiable, as defined by **RFC 2965**. It defaults to `False`. An unverifiable request is one whose URL the user did not have the option to approve. For example, if the request is for an image in an HTML document, and the user had no option to approve the automatic fetching of the image, this should be true.

method should be a string that indicates the HTTP request method that will be used (e.g. `'HEAD'`). If provided, its value is stored in the *method* attribute and is used by *get_method()*. The default is `'GET'` if *data* is `None` or `'POST'` otherwise. Subclasses may indicate a different default method by setting the *method* attribute in the class itself.

> **Note**
>
> The request will not work as expected if the data object is unable to deliver its content more than once (e.g. a file or an iterable that can produce the content only once) and the request is retried for HTTP redirects or authentication. The *data* is sent to the HTTP server right away after the headers. There is no support for a 100-continue expectation in the library.

Changed in version 3.3: *Request.method* argument is added to the Request class.

Changed in version 3.4: Default *Request.method* may be indicated at the class level.

Changed in version 3.6: Do not raise an error if the `Content-Length` has not been provided and *data* is neither `None` nor a bytes object. Fall back to use chunked transfer encoding instead.

class `urllib.request.`**OpenerDirector**

The *OpenerDirector* class opens URLs via *BaseHandler*s chained together. It manages the chaining of handlers, and recovery from errors.

class `urllib.request.`**BaseHandler**

This is the base class for all registered handlers — and handles only the simple mechanics of registration.

class `urllib.request.`**HTTPDefaultErrorHandler**

A class which defines a default handler for HTTP error responses; all responses are turned into *HTTPError* exceptions.

class `urllib.request.`**HTTPRedirectHandler**

A class to handle redirections.

class `urllib.request.`**HTTPCookieProcessor**(*cookiejar=None*)

A class to handle HTTP Cookies.

class `urllib.request.`**ProxyHandler**(*proxies=None*)

Cause requests to go through a proxy. If *proxies* is given, it must be a dictionary mapping protocol names to URLs of proxies. The default is to read the list of proxies from the environment variables `<protocol>_proxy`. If no proxy environment variables are set, then in a Windows environment proxy settings are obtained from the registry's Internet Settings section, and in a macOS environment proxy information is retrieved from the System Configuration Framework.

To disable autodetected proxy pass an empty dictionary.

The `no_proxy` environment variable can be used to specify hosts which shouldn't be reached via proxy; if set, it should be a comma-separated list of hostname suffixes, optionally with `:port` appended, for example `cern.ch,ncsa.uiuc.edu,some.host:8080`.

> **Note**
>
> `HTTP_PROXY` will be ignored if a variable `REQUEST_METHOD` is set; see the documentation on *getproxies()*.

class urllib.request.**HTTPPasswordMgr**

Keep a database of `(realm, uri) -> (user, password)` mappings.

class urllib.request.**HTTPPasswordMgrWithDefaultRealm**

Keep a database of `(realm, uri) -> (user, password)` mappings. A realm of `None` is considered a catch-all realm, which is searched if no other realm fits.

class urllib.request.**HTTPPasswordMgrWithPriorAuth**

A variant of *HTTPPasswordMgrWithDefaultRealm* that also has a database of `uri -> is_authenticated` mappings. Can be used by a BasicAuth handler to determine when to send authentication credentials immediately instead of waiting for a `401` response first.

Added in version 3.5.

class urllib.request.**AbstractBasicAuthHandler**(*password_mgr=None*)

This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported. If *passwd_mgr* also provides `is_authenticated` and `update_authenticated` methods (see *HTTPPasswordMgrWithPriorAuth Objects*), then the handler will use the `is_authenticated` result for a given URI to determine whether or not to send authentication credentials with the request. If `is_authenticated` returns `True` for the URI, credentials are sent. If `is_authenticated` is `False`, credentials are not sent, and then if a `401` response is received the request is re-sent with the authentication credentials. If authentication succeeds, `update_authenticated` is called to set `is_authenticated` `True` for the URI, so that subsequent requests to the URI or any of its super-URIs will automatically include the authentication credentials.

Added in version 3.5: Added `is_authenticated` support.

class urllib.request.**HTTPBasicAuthHandler**(*password_mgr=None*)

Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported. HTTPBasicAuthHandler will raise a *ValueError* when presented with a wrong Authentication scheme.

class urllib.request.**ProxyBasicAuthHandler**(*password_mgr=None*)

Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib.request.**AbstractDigestAuthHandler**(*password_mgr=None*)

This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib.request.**HTTPDigestAuthHandler**(*password_mgr=None*)

Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be

supported. When both Digest Authentication Handler and Basic Authentication Handler are both added, Digest Authentication is always tried first. If the Digest Authentication returns a 40x response again, it is sent to Basic Authentication handler to Handle. This Handler method will raise a *ValueError* when presented with an authentication scheme other than Digest or Basic.

Changed in version 3.3: Raise *ValueError* on unsupported Authentication Scheme.

class urllib.request.**ProxyDigestAuthHandler**(*password_mgr=None*)

Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib.request.**HTTPHandler**

A class to handle opening of HTTP URLs.

class urllib.request.**HTTPSHandler**(*debuglevel=0*, *context=None*, *check_hostname=None*)

A class to handle opening of HTTPS URLs. *context* and *check_hostname* have the same meaning as in *http.client.HTTPSConnection*.

Changed in version 3.2: *context* and *check_hostname* were added.

class urllib.request.**FileHandler**

Open local files.

class urllib.request.**DataHandler**

Open data URLs.

Added in version 3.4.

class urllib.request.**FTPHandler**

Open FTP URLs.

class urllib.request.**CacheFTPHandler**

Open FTP URLs, keeping a cache of open FTP connections to minimize delays.

class urllib.request.**UnknownHandler**

A catch-all class to handle unknown URLs.

class urllib.request.**HTTPErrorProcessor**

Process HTTP error responses.

21.4.1 Request Objects

The following methods describe *Request*'s public interface, and so all may be overridden in subclasses. It also defines several public attributes that can be used by clients to inspect the parsed request.

Request.**full_url**

The original URL passed to the constructor.

Changed in version 3.4.

Request.full_url is a property with setter, getter and a deleter. Getting *full_url* returns the original request URL with the fragment, if it was present.

Request.**type**

The URI scheme.

Request.**host**

The URI authority, typically a host, but may also contain a port separated by a colon.

Request.**origin_req_host**

The original host for the request, without port.

Request.**selector**

The URI path. If the *Request* uses a proxy, then selector will be the full URL that is passed to the proxy.

Request.**data**

The entity body for the request, or None if not specified.

Changed in version 3.4: Changing value of *Request.data* now deletes "Content-Length" header if it was previously set or calculated.

Request.**unverifiable**

boolean, indicates whether the request is unverifiable as defined by **RFC 2965**.

Request.**method**

The HTTP request method to use. By default its value is *None*, which means that *get_method()* will do its normal computation of the method to be used. Its value can be set (thus overriding the default computation in *get_method()*) either by providing a default value by setting it at the class level in a *Request* subclass, or by passing a value in to the *Request* constructor via the *method* argument.

Added in version 3.3.

Changed in version 3.4: A default value can now be set in subclasses; previously it could only be set via the constructor argument.

Request.**get_method**()

Return a string indicating the HTTP request method. If *Request.method* is not None, return its value, otherwise return 'GET' if *Request.data* is None, or 'POST' if it's not. This is only meaningful for HTTP requests.

Changed in version 3.3: get_method now looks at the value of *Request.method*.

Request.**add_header**(*key*, *val*)

Add another header to the request. Headers are currently ignored by all handlers except HTTP handlers, where they are added to the list of headers sent to the server. Note that there cannot be more than one header with the same name, and later calls will overwrite previous calls in case the *key* collides. Currently, this is no loss of HTTP functionality, since all headers which have meaning when used more than once have a (header-specific) way of gaining the same functionality using only one header. Note that headers added using this method are also added to redirected requests.

Request.**add_unredirected_header**(*key*, *header*)

Add a header that will not be added to a redirected request.

Request.**has_header**(*header*)

Return whether the instance has the named header (checks both regular and unredirected).

Request.**remove_header**(*header*)

Remove named header from the request instance (both from regular and unredirected headers).

Added in version 3.4.

Request.**get_full_url**()

Return the URL given in the constructor.

Changed in version 3.4.

Returns *Request.full_url*

Request.**set_proxy**(*host*, *type*)

Prepare the request by connecting to a proxy server. The *host* and *type* will replace those of the instance, and the instance's selector will be the original URL given in the constructor.

Request.**get_header**(*header_name*, *default=None*)

Return the value of the given header. If the header is not present, return the default value.

Request.**header_items**()

Return a list of tuples (header_name, header_value) of the Request headers.

Changed in version 3.4: The request methods add_data, has_data, get_data, get_type, get_host, get_selector, get_origin_req_host and is_unverifiable that were deprecated since 3.3 have been removed.

21.4.2 OpenerDirector Objects

OpenerDirector instances have the following methods:

OpenerDirector.**add_handler**(*handler*)

handler should be an instance of *BaseHandler*. The following methods are searched, and added to the possible chains (note that HTTP errors are a special case). Note that, in the following, *protocol* should be replaced with the actual protocol to handle, for example http_response() would be the HTTP protocol response handler. Also *type* should be replaced with the actual HTTP code, for example http_error_404() would handle HTTP 404 errors.

- <protocol>_open() — signal that the handler knows how to open *protocol* URLs.

  See *BaseHandler.<protocol>_open()* for more information.

- http_error_<type>() — signal that the handler knows how to handle HTTP errors with HTTP error code *type*.

  See *BaseHandler.http_error_<nnn>()* for more information.

- <protocol>_error() — signal that the handler knows how to handle errors from (non-http) *protocol*.

- <protocol>_request() — signal that the handler knows how to pre-process *protocol* requests.

  See *BaseHandler.<protocol>_request()* for more information.

- <protocol>_response() — signal that the handler knows how to post-process *protocol* responses.

  See *BaseHandler.<protocol>_response()* for more information.

OpenerDirector.**open**(*url*, *data=None*[, *timeout*])

Open the given *url* (which can be a request object or a string), optionally passing the given *data*. Arguments, return values and exceptions raised are the same as those of *urlopen()* (which simply calls the *open()* method on the currently installed global *OpenerDirector*). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). The timeout feature actually works only for HTTP, HTTPS and FTP connections.

OpenerDirector.**error**(*proto*, **args*)

Handle an error of the given protocol. This will call the registered error handlers for the given protocol with the given arguments (which are protocol specific). The HTTP protocol is a special case which uses the HTTP response code to determine the specific error handler; refer to the http_error_<type>() methods of the handler classes.

Return values and exceptions raised are the same as those of *urlopen()*.

OpenerDirector objects open URLs in three stages:

The order in which these methods are called within each stage is determined by sorting the handler instances.

1. Every handler with a method named like `<protocol>_request()` has that method called to pre-process the request.
2. Handlers with a method named like `<protocol>_open()` are called to handle the request. This stage ends when a handler either returns a non-*None* value (ie. a response), or raises an exception (usually *URLError*). Exceptions are allowed to propagate.

   In fact, the above algorithm is first tried for methods named *default_open()*. If all such methods return *None*, the algorithm is repeated for methods named like `<protocol>_open()`. If all such methods return *None*, the algorithm is repeated for methods named *unknown_open()*.

   Note that the implementation of these methods may involve calls of the parent *OpenerDirector* instance's *open()* and *error()* methods.
3. Every handler with a method named like `<protocol>_response()` has that method called to post-process the response.

21.4.3 BaseHandler Objects

BaseHandler objects provide a couple of methods that are directly useful, and others that are meant to be used by derived classes. These are intended for direct use:

`BaseHandler.`**`add_parent`**`(`*director*`)`

Add a director as parent.

`BaseHandler.`**`close`**`()`

Remove any parents.

The following attribute and methods should only be used by classes derived from *BaseHandler*.

Note

The convention has been adopted that subclasses defining `<protocol>_request()` or `<protocol>_response()` methods are named `*Processor`; all others are named `*Handler`.

`BaseHandler.`**`parent`**

A valid *OpenerDirector*, which can be used to open using a different protocol, or handle errors.

`BaseHandler.`**`default_open`**`(`*req*`)`

This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to catch all URLs.

This method, if implemented, will be called by the parent *OpenerDirector*. It should return a file-like object as described in the return value of the *open()* method of *OpenerDirector*, or `None`. It should raise *URLError*, unless a truly exceptional thing happens (for example, *MemoryError* should not be mapped to *URLError*).

This method will be called before any protocol-specific open method.

`BaseHandler.<protocol>_open(req)`

This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to handle URLs with the given protocol.

This method, if defined, will be called by the parent *OpenerDirector*. Return values should be the same as for *default_open()*.

BaseHandler.**unknown_open**(*req*)

This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to catch all URLs with no specific registered handler to open it.

This method, if implemented, will be called by the *parent OpenerDirector*. Return values should be the same as for *default_open()*.

BaseHandler.**http_error_default**(*req*, *fp*, *code*, *msg*, *hdrs*)

This method is *not* defined in *BaseHandler*, but subclasses should override it if they intend to provide a catch-all for otherwise unhandled HTTP errors. It will be called automatically by the *OpenerDirector* getting the error, and should not normally be called in other circumstances.

req will be a *Request* object, *fp* will be a file-like object with the HTTP error body, *code* will be the three-digit code of the error, *msg* will be the user-visible explanation of the code and *hdrs* will be a mapping object with the headers of the error.

Return values and exceptions raised should be the same as those of *urlopen()*.

BaseHandler.http_error_<nnn>(req, fp, code, msg, hdrs)

nnn should be a three-digit HTTP error code. This method is also not defined in *BaseHandler*, but will be called, if it exists, on an instance of a subclass, when an HTTP error with code *nnn* occurs.

Subclasses should override this method to handle specific HTTP errors.

Arguments, return values and exceptions raised should be the same as for *http_error_default()*.

BaseHandler.<protocol>_request(req)

This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to pre-process requests of the given protocol.

This method, if defined, will be called by the parent *OpenerDirector*. *req* will be a *Request* object. The return value should be a *Request* object.

BaseHandler.<protocol>_response(req, response)

This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to post-process responses of the given protocol.

This method, if defined, will be called by the parent *OpenerDirector*. *req* will be a *Request* object. *response* will be an object implementing the same interface as the return value of *urlopen()*. The return value should implement the same interface as the return value of *urlopen()*.

21.4.4 HTTPRedirectHandler Objects

Note

Some HTTP redirections require action from this module's client code. If this is the case, *HTTPError* is raised. See **RFC 2616** for details of the precise meanings of the various redirection codes.

An *HTTPError* exception raised as a security consideration if the HTTPRedirectHandler is presented with a redirected URL which is not an HTTP, HTTPS or FTP URL.

HTTPRedirectHandler.**redirect_request**(*req*, *fp*, *code*, *msg*, *hdrs*, *newurl*)

Return a *Request* or `None` in response to a redirect. This is called by the default implementations of the `http_error_30*()` methods when a redirection is received from the server. If a redirection should take place, return a new *Request* to allow `http_error_30*()` to perform the redirect to *newurl*. Otherwise,

raise *HTTPError* if no other handler should try to handle this URL, or return None if you can't but another handler might.

> **Note**
>
> The default implementation of this method does not strictly follow **RFC 2616**, which says that 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and the default implementation reproduces this behavior.

HTTPRedirectHandler.**http_error_301**(*req*, *fp*, *code*, *msg*, *hdrs*)

Redirect to the Location: or URI: URL. This method is called by the parent *OpenerDirector* when getting an HTTP 'moved permanently' response.

HTTPRedirectHandler.**http_error_302**(*req*, *fp*, *code*, *msg*, *hdrs*)

The same as *http_error_301()*, but called for the 'found' response.

HTTPRedirectHandler.**http_error_303**(*req*, *fp*, *code*, *msg*, *hdrs*)

The same as *http_error_301()*, but called for the 'see other' response.

HTTPRedirectHandler.**http_error_307**(*req*, *fp*, *code*, *msg*, *hdrs*)

The same as *http_error_301()*, but called for the 'temporary redirect' response. It does not allow changing the request method from POST to GET.

HTTPRedirectHandler.**http_error_308**(*req*, *fp*, *code*, *msg*, *hdrs*)

The same as *http_error_301()*, but called for the 'permanent redirect' response. It does not allow changing the request method from POST to GET.

Added in version 3.11.

21.4.5 HTTPCookieProcessor Objects

HTTPCookieProcessor instances have one attribute:

HTTPCookieProcessor.**cookiejar**

The *http.cookiejar.CookieJar* in which cookies are stored.

21.4.6 ProxyHandler Objects

ProxyHandler.<protocol>_open(request)

The *ProxyHandler* will have a method <protocol>_open() for every *protocol* which has a proxy in the *proxies* dictionary given in the constructor. The method will modify requests to go through the proxy, by calling request.set_proxy(), and call the next handler in the chain to actually execute the protocol.

21.4.7 HTTPPasswordMgr Objects

These methods are available on *HTTPPasswordMgr* and *HTTPPasswordMgrWithDefaultRealm* objects.

HTTPPasswordMgr.**add_password**(*realm*, *uri*, *user*, *passwd*)

uri can be either a single URI, or a sequence of URIs. *realm*, *user* and *passwd* must be strings. This causes `(user, passwd)` to be used as authentication tokens when authentication for *realm* and a super-URI of any of the given URIs is given.

HTTPPasswordMgr.**find_user_password**(*realm*, *authuri*)

Get user/password for given realm and URI, if any. This method will return `(None, None)` if there is no matching user/password.

For *HTTPPasswordMgrWithDefaultRealm* objects, the realm `None` will be searched if the given *realm* has no matching user/password.

21.4.8 HTTPPasswordMgrWithPriorAuth Objects

This password manager extends *HTTPPasswordMgrWithDefaultRealm* to support tracking URIs for which authentication credentials should always be sent.

HTTPPasswordMgrWithPriorAuth.**add_password**(*realm*, *uri*, *user*, *passwd*, *is_authenticated=False*)

realm, *uri*, *user*, *passwd* are as for *HTTPPasswordMgr.add_password()*. *is_authenticated* sets the initial value of the `is_authenticated` flag for the given URI or list of URIs. If *is_authenticated* is specified as `True`, *realm* is ignored.

HTTPPasswordMgrWithPriorAuth.**find_user_password**(*realm*, *authuri*)

Same as for *HTTPPasswordMgrWithDefaultRealm* objects

HTTPPasswordMgrWithPriorAuth.**update_authenticated**(*self*, *uri*, *is_authenticated=False*)

Update the `is_authenticated` flag for the given *uri* or list of URIs.

HTTPPasswordMgrWithPriorAuth.**is_authenticated**(*self*, *authuri*)

Returns the current state of the `is_authenticated` flag for the given URI.

21.4.9 AbstractBasicAuthHandler Objects

AbstractBasicAuthHandler.**http_error_auth_reqed**(*authreq*, *host*, *req*, *headers*)

Handle an authentication request by getting a user/password pair, and re-trying the request. *authreq* should be the name of the header where the information about the realm is included in the request, *host* specifies the URL and path to authenticate for, *req* should be the (failed) *Request* object, and *headers* should be the error headers.

host is either an authority (e.g. `"python.org"`) or a URL containing an authority component (e.g. `"http://python.org/"`). In either case, the authority must not contain a userinfo component (so, `"python.org"` and `"python.org:80"` are fine, `"joe:password@python.org"` is not).

21.4.10 HTTPBasicAuthHandler Objects

`HTTPBasicAuthHandler.`**`http_error_401`**(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

21.4.11 ProxyBasicAuthHandler Objects

`ProxyBasicAuthHandler.`**`http_error_407`**(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

21.4.12 AbstractDigestAuthHandler Objects

`AbstractDigestAuthHandler.`**`http_error_auth_reqed`**(*authreq*, *host*, *req*, *headers*)

authreq should be the name of the header where the information about the realm is included in the request, *host* should be the host to authenticate to, *req* should be the (failed) `Request` object, and *headers* should be the error headers.

21.4.13 HTTPDigestAuthHandler Objects

`HTTPDigestAuthHandler.`**`http_error_401`**(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

21.4.14 ProxyDigestAuthHandler Objects

`ProxyDigestAuthHandler.`**`http_error_407`**(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

21.4.15 HTTPHandler Objects

`HTTPHandler.`**`http_open`**(*req*)

Send an HTTP request, which can be either GET or POST, depending on `req.has_data()`.

21.4.16 HTTPSHandler Objects

`HTTPSHandler.`**`https_open`**(*req*)

Send an HTTPS request, which can be either GET or POST, depending on `req.has_data()`.

21.4.17 FileHandler Objects

`FileHandler.`**`file_open`**(*req*)

Open the file locally, if there is no host name, or the host name is `'localhost'`.

Changed in version 3.2: This method is applicable only for local hostnames. When a remote hostname is given, a *URLError* is raised.

21.4.18 DataHandler Objects

`DataHandler.`**`data_open`**(*req*)

Read a data URL. This kind of URL contains the content encoded in the URL itself. The data URL syntax is specified in **RFC 2397**. This implementation ignores white spaces in base64 encoded data URLs so the URL may be wrapped in whatever source file it comes from. But even though some browsers don't mind about a missing padding at the end of a base64 encoded data URL, this implementation will raise a *ValueError* in that case.

21.4.19 FTPHandler Objects

`FTPHandler.`**`ftp_open`**(*req*)

Open the FTP file indicated by *req*. The login is always done with empty username and password.

21.4.20 CacheFTPHandler Objects

CacheFTPHandler objects are *FTPHandler* objects with the following additional methods:

`CacheFTPHandler.`**`setTimeout`**(*t*)

Set timeout of connections to *t* seconds.

`CacheFTPHandler.`**`setMaxConns`**(*m*)

Set maximum number of cached connections to *m*.

21.4.21 UnknownHandler Objects

`UnknownHandler.`**`unknown_open`**()

Raise a *URLError* exception.

21.4.22 HTTPErrorProcessor Objects

`HTTPErrorProcessor.`**`http_response`**(*request*, *response*)

Process HTTP error responses.

For 200 error codes, the response object is returned immediately.

For non-200 error codes, this simply passes the job on to the `http_error_<type>()` handler methods, via *OpenerDirector.error()*. Eventually, *HTTPDefaultErrorHandler* will raise an *HTTPError* if no other handler handles the error.

`HTTPErrorProcessor.`**`https_response`**(*request*, *response*)

Process HTTPS error responses.

The behavior is same as *http_response()*.

21.4.23 Examples

In addition to the examples below, more examples are given in urllib-howto.

This example gets the python.org main page and displays the first 300 bytes of it.

```
>>> import urllib.request
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(300))
...
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\n\n\n<html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">\n\n<head>\n
<meta http-equiv="content-type" content="text/html; charset=utf-8" />\n
<title>Python Programming '
```

Note that urlopen returns a bytes object. This is because there is no way for urlopen to automatically determine the encoding of the byte stream it receives from the HTTP server. In general, a program will decode the returned bytes object to string once it determines or guesses the appropriate encoding.

The following W3C document, https://www.w3.org/International/O-charset, lists the various ways in which an (X)HTML or an XML document could have specified its encoding information.

As the python.org website uses *utf-8* encoding as specified in its meta tag, we will use the same for decoding the bytes object.

```
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(100).decode('utf-8'))
...
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

It is also possible to achieve the same result without using the *context manager* approach.

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/')
>>> print(f.read(100).decode('utf-8'))
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

In the following example, we are sending a data-stream to the stdin of a CGI and reading the data it returns to us. Note that this example will only work when the Python installation supports SSL.

```
>>> import urllib.request
>>> req = urllib.request.Request(url='https://localhost/cgi-bin/test.cgi',
...                       data=b'This data is passed to stdin of the CGI')
>>> with urllib.request.urlopen(req) as f:
...     print(f.read().decode('utf-8'))
...
Got Data: "This data is passed to stdin of the CGI"
```

The code for the sample CGI used in the above example is:

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print('Content-type: text/plain\n\nGot Data: "%s"' % data)
```

Here is an example of doing a `PUT` request using *Request*:

```
import urllib.request
DATA = b'some data'
req = urllib.request.Request(url='http://localhost:8080', data=DATA, method='PUT')
with urllib.request.urlopen(req) as f:
    pass
print(f.status)
print(f.reason)
```

Use of Basic HTTP Authentication:

```
import urllib.request
# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
                          passwd='kadidd!ehopper')
opener = urllib.request.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib.request.install_opener(opener)
urllib.request.urlopen('http://www.example.com/login.html')
```

build_opener() provides many handlers by default, including a *ProxyHandler*. By default, *ProxyHandler* uses the environment variables named <scheme>_proxy, where <scheme> is the URL scheme involved. For example, the http_proxy environment variable is read to obtain the HTTP proxy's URL.

This example replaces the default *ProxyHandler* with one that uses programmatically supplied proxy URLs, and adds proxy authorization support with *ProxyBasicAuthHandler*.

```
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib.request.build_opener(proxy_handler, proxy_auth_handler)
# This time, rather than install the OpenerDirector, we use it directly:
opener.open('http://www.example.com/login.html')
```

Adding HTTP headers:

Use the *headers* argument to the *Request* constructor, or:

```
import urllib.request
req = urllib.request.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
# Customize the default User-Agent header value:
req.add_header('User-Agent', 'urllib-example/0.1 (Contact: . . .)')
r = urllib.request.urlopen(req)
```

OpenerDirector automatically adds a *User-Agent* header to every *Request*. To change this:

```
import urllib.request
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')
```

Also, remember that a few standard headers (*Content-Length*, *Content-Type* and *Host*) are added when the *Request* is passed to *urlopen()* (or *OpenerDirector.open()*).

Here is an example session that uses the GET method to retrieve a URL containing parameters:

```
>>> import urllib.request
>>> import urllib.parse
>>> params = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> url = "http://www.musi-cal.com/cgi-bin/query?%s" % params
>>> with urllib.request.urlopen(url) as f:
...     print(f.read().decode('utf-8'))
...
```

The following example uses the `POST` method instead. Note that params output from urlencode is encoded to bytes before it is sent to urlopen as data:

```
>>> import urllib.request
>>> import urllib.parse
>>> data = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> data = data.encode('ascii')
>>> with urllib.request.urlopen("http://requestb.in/xrbl82xr", data) as f:
...     print(f.read().decode('utf-8'))
...
```

The following example uses an explicitly specified HTTP proxy, overriding environment settings:

```
>>> import urllib.request
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.request.FancyURLopener(proxies)
>>> with opener.open("http://www.python.org") as f:
...     f.read().decode('utf-8')
...
```

The following example uses no proxies at all, overriding environment settings:

```
>>> import urllib.request
>>> opener = urllib.request.FancyURLopener({})
>>> with opener.open("http://www.python.org/") as f:
...     f.read().decode('utf-8')
...
```

21.4.24 Legacy interface

The following functions and classes are ported from the Python 2 module `urllib` (as opposed to `urllib2`). They might become deprecated at some point in the future.

`urllib.request.`**`urlretrieve`**(*url*, *filename=None*, *reporthook=None*, *data=None*)

Copy a network object denoted by a URL to a local file. If the URL points to a local file, the object will not be copied unless filename is supplied. Return a tuple `(filename, headers)` where *filename* is the local file name under which the object can be found, and *headers* is whatever the `info()` method of the object returned by *`urlopen()`* returned (for a remote object). Exceptions are the same as for *`urlopen()`*.

The second argument, if present, specifies the file location to copy to (if absent, the location will be a tempfile with a generated name). The third argument, if present, is a callable that will be called once on establishment of the network connection and once after each block read thereafter. The callable will be passed three arguments; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be `-1` on older FTP servers which do not return a file size in response to a retrieval request.

The following example illustrates the most common usage scenario:

```
>>> import urllib.request
>>> local_filename, headers = urllib.request.urlretrieve('http://python.org/')
>>> html = open(local_filename)
>>> html.close()
```

If the *url* uses the `http:` scheme identifier, the optional *data* argument may be given to specify a `POST` request (normally the request type is `GET`). The *data* argument must be a bytes object in standard *application/x-www-form-urlencoded* format; see the *urllib.parse.urlencode()* function.

urlretrieve() will raise *ContentTooShortError* when it detects that the amount of data available was less than the expected amount (which is the size reported by a *Content-Length* header). This can occur, for example, when the download is interrupted.

The *Content-Length* is treated as a lower bound: if there's more data to read, urlretrieve reads more data, but if less data is available, it raises the exception.

You can still retrieve the downloaded data in this case, it is stored in the `content` attribute of the exception instance.

If no *Content-Length* header was supplied, urlretrieve can not check the size of the data it has downloaded, and just returns it. In this case you just have to assume that the download was successful.

`urllib.request.`**`urlcleanup`**`()`

Cleans up temporary files that may have been left behind by previous calls to *urlretrieve()*.

class `urllib.request.`**`URLopener`**(*proxies=None*, ***x509*)

Deprecated since version 3.3.

Base class for opening and reading URLs. Unless you need to support opening objects using schemes other than `http:`, `ftp:`, or `file:`, you probably want to use *FancyURLopener*.

By default, the *URLopener* class sends a *User-Agent* header of `urllib/VVV`, where *VVV* is the *urllib* version number. Applications can define their own *User-Agent* header by subclassing *URLopener* or *FancyURLopener* and setting the class attribute *version* to an appropriate string value in the subclass definition.

The optional *proxies* parameter should be a dictionary mapping scheme names to proxy URLs, where an empty dictionary turns proxies off completely. Its default value is `None`, in which case environmental proxy settings will be used if present, as discussed in the definition of *urlopen()*, above.

Additional keyword parameters, collected in *x509*, may be used for authentication of the client when using the `https:` scheme. The keywords *key_file* and *cert_file* are supported to provide an SSL key and certificate; both are needed to support client authentication.

URLopener objects will raise an *OSError* exception if the server returns an error code.

open(*fullurl*, *data=None*)

Open *fullurl* using the appropriate protocol. This method sets up cache and proxy information, then calls the appropriate open method with its input arguments. If the scheme is not recognized, *open_unknown()* is called. The *data* argument has the same meaning as the *data* argument of *urlopen()*.

This method always quotes *fullurl* using *quote()*.

open_unknown(*fullurl*, *data=None*)

Overridable interface to open unknown URL types.

retrieve(*url*, *filename=None*, *reporthook=None*, *data=None*)

Retrieves the contents of *url* and places it in *filename*. The return value is a tuple consisting of a local filename and either an *email.message.Message* object containing the response headers (for remote URLs) or `None` (for local URLs). The caller must then open and read the contents of *filename*. If *filename* is not given and the URL refers to a local file, the input filename is returned. If the URL is non-local and *filename* is

not given, the filename is the output of *tempfile.mktemp()* with a suffix that matches the suffix of the last path component of the input URL. If *reporthook* is given, it must be a function accepting three numeric parameters: A chunk number, the maximum size chunks are read in and the total size of the download (-1 if unknown). It will be called once at the start and after each chunk of data is read from the network. *reporthook* is ignored for local URLs.

If the *url* uses the `http:` scheme identifier, the optional *data* argument may be given to specify a `POST` request (normally the request type is `GET`). The *data* argument must in standard *application/x-www-form-urlencoded* format; see the *urllib.parse.urlencode()* function.

version

Variable that specifies the user agent of the opener object. To get *urllib* to tell servers that it is a particular user agent, set this in a subclass as a class variable or in the constructor before calling the base constructor.

class urllib.request.**FancyURLopener**(...)

Deprecated since version 3.3.

FancyURLopener subclasses *URLopener* providing default handling for the following HTTP response codes: 301, 302, 303, 307 and 401. For the 30x response codes listed above, the *Location* header is used to fetch the actual URL. For 401 response codes (authentication required), basic HTTP authentication is performed. For the 30x response codes, recursion is bounded by the value of the *maxtries* attribute, which defaults to 10.

For all other response codes, the method *http_error_default()* is called which you can override in subclasses to handle the error appropriately.

Note

According to the letter of **RFC 2616**, 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and *urllib* reproduces this behaviour.

The parameters to the constructor are the same as those for *URLopener*.

Note

When performing basic authentication, a *FancyURLopener* instance calls its *prompt_user_passwd()* method. The default implementation asks the users for the required information on the controlling terminal. A subclass may override this method to support more appropriate behavior if needed.

The *FancyURLopener* class offers one additional method that should be overloaded to provide the appropriate behavior:

prompt_user_passwd(*host*, *realm*)

Return information needed to authenticate the user at the given host in the specified security realm. The return value should be a tuple, `(user, password)`, which can be used for basic authentication.

The implementation prompts for this information on the terminal; an application should override this method to use an appropriate interaction model in the local environment.

21.4.25 `urllib.request` Restrictions

- Currently, only the following protocols are supported: HTTP (versions 0.9 and 1.0), FTP, local files, and data URLs.

  Changed in version 3.4: Added support for data URLs.

- The caching feature of *`urlretrieve()`* has been disabled until someone finds the time to hack proper processing of Expiration time headers.
- There should be a function to query whether a particular URL is in the cache.
- For backward compatibility, if a URL appears to point to a local file but the file can't be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.
- The *`urlopen()`* and *`urlretrieve()`* functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive web client using these functions without using threads.
- The data returned by *`urlopen()`* or *`urlretrieve()`* is the raw data returned by the server. This may be binary data (such as an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the *`Content-Type`* header. If the returned data is HTML, you can use the module *`html.parser`* to parse it.
- The code handling the FTP protocol cannot differentiate between a file and a directory. This can lead to unexpected behavior when attempting to read a URL that points to a file that is not accessible. If the URL ends in a `/`, it is assumed to refer to a directory and will be handled accordingly. But if an attempt to read a file leads to a 550 error (meaning the URL cannot be found or is not accessible, often for permission reasons), then the path is treated as a directory in order to handle the case when a directory is specified by a URL but the trailing `/` has been left off. This can cause misleading results when you try to fetch a file whose read permissions make it inaccessible; the FTP code will try to read it, fail with a 550 error, and then perform a directory listing for the unreadable file. If fine-grained control is needed, consider using the *`ftplib`* module, subclassing *`FancyURLopener`*, or changing *_urlopener* to meet your needs.

21.5 `urllib.response` — Response classes used by urllib

The *`urllib.response`* module defines functions and classes which define a minimal file-like interface, including `read()` and `readline()`. Functions defined by this module are used internally by the *`urllib.request`* module. The typical response object is a *`urllib.response.addinfourl`* instance:

class urllib.response.**addinfourl**

url

URL of the resource retrieved, commonly used to determine if a redirect was followed.

headers

Returns the headers of the response in the form of an *`EmailMessage`* instance.

status

Added in version 3.9.

Status code returned by server.

geturl()

Deprecated since version 3.9: Deprecated in favor of *`url`*.

info()

Deprecated since version 3.9: Deprecated in favor of *headers*.

code

Deprecated since version 3.9: Deprecated in favor of *status*.

getcode()

Deprecated since version 3.9: Deprecated in favor of *status*.

21.6 urllib.parse — Parse URLs into components

Source code: Lib/urllib/parse.py

This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a "relative URL" to an absolute URL given a "base URL."

The module has been designed to match the internet RFC on Relative Uniform Resource Locators. It supports the following URL schemes: `file`, `ftp`, `gopher`, `hdl`, `http`, `https`, `imap`, `itms-services`, `mailto`, `mms`, `news`, `nntp`, `prospero`, `rsync`, `rtsp`, `rtsps`, `rtspu`, `sftp`, `shttp`, `sip`, `sips`, `snews`, `svn`, `svn+ssh`, `telnet`, `wais`, `ws`, `wss`.

CPython implementation detail: The inclusion of the `itms-services` URL scheme can prevent an app from passing Apple's App Store review process for the macOS and iOS App Stores. Handling for the `itms-services` scheme is always removed on iOS; on macOS, it *may* be removed if CPython has been built with the `--with-app-store-compliance` option.

The *urllib.parse* module defines functions that fall into two broad categories: URL parsing and URL quoting. These are covered in detail in the following sections.

This module's functions use the deprecated term `netloc` (or `net_loc`), which was introduced in **RFC 1808**. However, this term has been obsoleted by **RFC 3986**, which introduced the term `authority` as its replacement. The use of `netloc` is continued for backward compatibility.

21.6.1 URL Parsing

The URL parsing functions focus on splitting a URL string into its components, or on combining URL components into a URL string.

`urllib.parse.`**urlparse**(*urlstring*, *scheme=''*, *allow_fragments=True*)

Parse a URL into six components, returning a 6-item *named tuple*. This corresponds to the general structure of a URL: `scheme://netloc/path;parameters?query#fragment`. Each tuple item is a string, possibly empty. The components are not broken up into smaller parts (for example, the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the *path* component, which is retained if present. For example:

```
>>> from urllib.parse import urlparse
>>> urlparse("scheme://netloc/path;parameters?query#fragment")
ParseResult(scheme='scheme', netloc='netloc', path='/path;parameters', params='',
            query='query', fragment='fragment')
>>> o = urlparse("http://docs.python.org:80/3/library/urllib.parse.html?"
...              "highlight=params#url-parsing")
>>> o
```

(continues on next page)

(continued from previous page)

```
ParseResult(scheme='http', netloc='docs.python.org:80',
            path='/3/library/urllib.parse.html', params='',
            query='highlight=params', fragment='url-parsing')
>>> o.scheme
'http'
>>> o.netloc
'docs.python.org:80'
>>> o.hostname
'docs.python.org'
>>> o.port
80
>>> o._replace(fragment="").geturl()
'http://docs.python.org:80/3/library/urllib.parse.html?highlight=params'
```

Following the syntax specifications in **RFC 1808**, urlparse recognizes a netloc only if it is properly introduced by '//'. Otherwise the input is presumed to be a relative URL and thus to start with a path component.

```
>>> from urllib.parse import urlparse
>>> urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
ParseResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html', params='',
            query='', fragment='')
```

The *scheme* argument gives the default addressing scheme, to be used only if the URL does not specify one. It should be the same type (text or bytes) as *urlstring*, except that the default value `''` is always allowed, and is automatically converted to `b''` if appropriate.

If the *allow_fragments* argument is false, fragment identifiers are not recognized. Instead, they are parsed as part of the path, parameters or query component, and `fragment` is set to the empty string in the return value.

The return value is a *named tuple*, which means that its items can be accessed by index or as named attributes, which are:

Attribute	Index	Value	Value if not present
`scheme`	0	URL scheme specifier	*scheme* parameter
`netloc`	1	Network location part	empty string
`path`	2	Hierarchical path	empty string
`params`	3	Parameters for last path element	empty string
`query`	4	Query component	empty string
`fragment`	5	Fragment identifier	empty string
`username`		User name	*None*
`password`		Password	*None*
`hostname`		Host name (lower case)	*None*
`port`		Port number as integer, if present	*None*

Reading the `port` attribute will raise a *ValueError* if an invalid port is specified in the URL. See section *Structured Parse Results* for more information on the result object.

Unmatched square brackets in the `netloc` attribute will raise a *ValueError*.

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a *ValueError*. If the URL is decomposed before parsing, no error will be raised.

As is the case with all named tuples, the subclass has a few additional methods and attributes that are particularly useful. One such method is `_replace()`. The `_replace()` method will return a new ParseResult object replacing specified fields with new values.

```
>>> from urllib.parse import urlparse
>>> u = urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
>>> u
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> u._replace(scheme='http')
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
```

Warning

urlparse() does not perform validation. See *URL parsing security* for details.

Changed in version 3.2: Added IPv6 URL parsing capabilities.

Changed in version 3.3: The fragment is now parsed for all URL schemes (unless *allow_fragments* is false), in accordance with **RFC 3986**. Previously, an allowlist of schemes that support fragments existed.

Changed in version 3.6: Out-of-range port numbers now raise *ValueError*, instead of returning *None*.

Changed in version 3.8: Characters that affect netloc parsing under NFKC normalization will now raise *ValueError*.

urllib.parse.**parse_qs**(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

Parse a query string given as a string argument (data of type *application/x-www-form-urlencoded*). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a *ValueError* exception.

The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the *bytes.decode()* method.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a *ValueError* if there are more than *max_num_fields* fields read.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to `&`.

Use the *urllib.parse.urlencode()* function (with the `doseq` parameter set to `True`) to convert such dictionaries into query strings.

Changed in version 3.2: Add *encoding* and *errors* parameters.

Changed in version 3.8: Added *max_num_fields* parameter.

Changed in version 3.10: Added *separator* parameter with the default value of `&`. Python versions earlier than Python 3.10 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

`urllib.parse.`**`parse_qsl`**(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

Parse a query string given as a string argument (data of type *application/x-www-form-urlencoded*). Data are returned as a list of name, value pairs.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a *ValueError* exception.

The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the *bytes.decode()* method.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a *ValueError* if there are more than *max_num_fields* fields read.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to `&`.

Use the *urllib.parse.urlencode()* function to convert such lists of pairs into query strings.

Changed in version 3.2: Add *encoding* and *errors* parameters.

Changed in version 3.8: Added *max_num_fields* parameter.

Changed in version 3.10: Added *separator* parameter with the default value of `&`. Python versions earlier than Python 3.10 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

`urllib.parse.`**`urlunparse`**(*parts*)

Construct a URL from a tuple as returned by `urlparse()`. The *parts* argument can be any six-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a `?` with an empty query; the RFC states that these are equivalent).

`urllib.parse.`**`urlsplit`**(*urlstring*, *scheme=''*, *allow_fragments=True*)

This is similar to *urlparse()*, but does not split the params from the URL. This should generally be used instead of *urlparse()* if the more recent URL syntax allowing parameters to be applied to each segment of the *path* portion of the URL (see **RFC 2396**) is wanted. A separate function is needed to separate the path segments and parameters. This function returns a 5-item *named tuple*:

```
(addressing scheme, network location, path, query, fragment identifier).
```

The return value is a *named tuple*, its items can be accessed by index or as named attributes:

Attribute	Index	Value	Value if not present
`scheme`	0	URL scheme specifier	*scheme* parameter
`netloc`	1	Network location part	empty string
`path`	2	Hierarchical path	empty string
`query`	3	Query component	empty string
`fragment`	4	Fragment identifier	empty string
`username`		User name	*None*
`password`		Password	*None*
`hostname`		Host name (lower case)	*None*
`port`		Port number as integer, if present	*None*

Reading the `port` attribute will raise a *ValueError* if an invalid port is specified in the URL. See section *Structured Parse Results* for more information on the result object.

Unmatched square brackets in the `netloc` attribute will raise a *ValueError*.

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a *ValueError*. If the URL is decomposed before parsing, no error will be raised.

Following some of the WHATWG spec that updates RFC 3986, leading C0 control and space characters are stripped from the URL. `\n`, `\r` and tab `\t` characters are removed from the URL at any position.

Warning

urlsplit() does not perform validation. See *URL parsing security* for details.

Changed in version 3.6: Out-of-range port numbers now raise *ValueError*, instead of returning *None*.

Changed in version 3.8: Characters that affect netloc parsing under NFKC normalization will now raise *ValueError*.

Changed in version 3.10: ASCII newline and tab characters are stripped from the URL.

Changed in version 3.12: Leading WHATWG C0 control and space characters are stripped from the URL.

`urllib.parse.`**`urlunsplit`**(*parts*)

Combine the elements of a tuple as returned by *urlsplit()* into a complete URL as a string. The *parts* argument can be any five-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent).

`urllib.parse.`**`urljoin`**(*base*, *url*, *allow_fragments=True*)

Construct a full ("absolute") URL by combining a "base URL" (*base*) with another URL (*url*). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL. For example:

```
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

The *allow_fragments* argument has the same meaning and default as for *urlparse()*.

Note

If *url* is an absolute URL (that is, it starts with `//` or `scheme://`), the *url*'s hostname and/or scheme will be present in the result. For example:

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

If you do not want that behavior, preprocess the *url* with *urlsplit()* and *urlunsplit()*, removing possible *scheme* and *netloc* parts.

Changed in version 3.5: Behavior updated to match the semantics defined in **RFC 3986**.

urllib.parse.**urldefrag**(*url*)

If *url* contains a fragment identifier, return a modified version of *url* with no fragment identifier, and the fragment identifier as a separate string. If there is no fragment identifier in *url*, return *url* unmodified and an empty string.

The return value is a *named tuple*, its items can be accessed by index or as named attributes:

Attribute	Index	Value	Value if not present
`url`	0	URL with no fragment	empty string
`fragment`	1	Fragment identifier	empty string

See section *Structured Parse Results* for more information on the result object.

Changed in version 3.2: Result is a structured object rather than a simple 2-tuple.

urllib.parse.**unwrap**(*url*)

Extract the url from a wrapped URL (that is, a string formatted as `<URL:scheme://host/path>`, `<scheme://host/path>`, `URL:scheme://host/path` or `scheme://host/path`). If *url* is not a wrapped URL, it is returned without changes.

21.6.2 URL parsing security

The *urlsplit()* and *urlparse()* APIs do not perform **validation** of inputs. They may not raise errors on inputs that other applications consider invalid. They may also succeed on some inputs that might not be considered URLs elsewhere. Their purpose is for practical functionality rather than purity.

Instead of raising an exception on unusual input, they may instead return some component parts as empty strings. Or components may contain more than perhaps they should.

We recommend that users of these APIs where the values may be used anywhere with security implications code defensively. Do some verification within your code before trusting a returned component part. Does that `scheme` make sense? Is that a sensible `path`? Is there anything strange about that `hostname`? etc.

What constitutes a URL is not universally well defined. Different applications have different needs and desired constraints. For instance the living WHATWG spec describes what user facing web clients such as a web browser require. While **RFC 3986** is more general. These functions incorporate some aspects of both, but cannot be claimed compliant with either. The APIs and existing user code with expectations on specific behaviors predate both standards leading us to be very cautious about making API behavior changes.

21.6.3 Parsing ASCII Encoded Bytes

The URL parsing functions were originally designed to operate on character strings only. In practice, it is useful to be able to manipulate properly quoted and encoded URLs as sequences of ASCII bytes. Accordingly, the URL parsing functions in this module all operate on *bytes* and *bytearray* objects in addition to *str* objects.

If *str* data is passed in, the result will also contain only *str* data. If *bytes* or *bytearray* data is passed in, the result will contain only *bytes* data.

Attempting to mix *str* data with *bytes* or *bytearray* in a single function call will result in a *TypeError* being raised, while attempting to pass in non-ASCII byte values will trigger *UnicodeDecodeError*.

To support easier conversion of result objects between *str* and *bytes*, all return values from URL parsing functions provide either an `encode()` method (when the result contains *str* data) or a `decode()` method (when the result contains *bytes* data). The signatures of these methods match those of the corresponding *str* and *bytes* methods (except that the default encoding is `'ascii'` rather than `'utf-8'`). Each produces a value of a corresponding type that contains either *bytes* data (for `encode()` methods) or *str* data (for `decode()` methods).

Applications that need to operate on potentially improperly quoted URLs that may contain non-ASCII data will need to do their own decoding from bytes to characters before invoking the URL parsing methods.

The behaviour described in this section applies only to the URL parsing functions. The URL quoting functions use their own rules when producing or consuming byte sequences as detailed in the documentation of the individual URL quoting functions.

Changed in version 3.2: URL parsing functions now accept ASCII encoded byte sequences

21.6.4 Structured Parse Results

The result objects from the *urlparse()*, *urlsplit()* and *urldefrag()* functions are subclasses of the *tuple* type. These subclasses add the attributes listed in the documentation for those functions, the encoding and decoding support described in the previous section, as well as an additional method:

urllib.parse.SplitResult.**geturl**()

Return the re-combined version of the original URL as a string. This may differ from the original URL in that the scheme may be normalized to lower case and empty components may be dropped. Specifically, empty parameters, queries, and fragment identifiers will be removed.

For *urldefrag()* results, only empty fragment identifiers will be removed. For *urlsplit()* and *urlparse()* results, all noted changes will be made to the URL returned by this method.

The result of this method remains unchanged if passed back through the original parsing function:

```
>>> from urllib.parse import urlsplit
>>> url = 'HTTP://www.Python.org/doc/#'
>>> r1 = urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
>>> r2 = urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

The following classes provide the implementations of the structured parse results when operating on *str* objects:

class urllib.parse.**DefragResult**(*url*, *fragment*)

Concrete class for *urldefrag()* results containing *str* data. The `encode()` method returns a *DefragResultBytes* instance.

Added in version 3.2.

class urllib.parse.**ParseResult**(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)

Concrete class for *urlparse()* results containing *str* data. The `encode()` method returns a *ParseResultBytes* instance.

class urllib.parse.**SplitResult**(*scheme*, *netloc*, *path*, *query*, *fragment*)

Concrete class for *urlsplit()* results containing *str* data. The `encode()` method returns a *SplitResultBytes* instance.

The following classes provide the implementations of the parse results when operating on *bytes* or *bytearray* objects:

class urllib.parse.**DefragResultBytes**(*url*, *fragment*)

Concrete class for *urldefrag()* results containing *bytes* data. The `decode()` method returns a *DefragResult* instance.

Added in version 3.2.

class urllib.parse.**ParseResultBytes**(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)

Concrete class for *urlparse()* results containing *bytes* data. The `decode()` method returns a *ParseResult* instance.

Added in version 3.2.

class urllib.parse.**SplitResultBytes**(*scheme*, *netloc*, *path*, *query*, *fragment*)

Concrete class for *urlsplit()* results containing *bytes* data. The `decode()` method returns a *SplitResult* instance.

Added in version 3.2.

21.6.5 URL Quoting

The URL quoting functions focus on taking program data and making it safe for use as URL components by quoting special characters and appropriately encoding non-ASCII text. They also support reversing these operations to recreate the original data from the contents of a URL component if that task isn't already covered by the URL parsing functions above.

urllib.parse.**quote**(*string*, *safe='/'*, *encoding=None*, *errors=None*)

Replace special characters in *string* using the `%xx` escape. Letters, digits, and the characters `'_.-~'` are never quoted. By default, this function is intended for quoting the path section of a URL. The optional *safe* parameter specifies additional ASCII characters that should not be quoted — its default value is `'/'`.

string may be either a *str* or a *bytes* object.

Changed in version 3.7: Moved from **RFC 2396** to **RFC 3986** for quoting URL strings. "~" is now included in the set of unreserved characters.

The optional *encoding* and *errors* parameters specify how to deal with non-ASCII characters, as accepted by the *str.encode()* method. *encoding* defaults to `'utf-8'`. *errors* defaults to `'strict'`, meaning unsupported characters raise a *UnicodeEncodeError*. *encoding* and *errors* must not be supplied if *string* is a *bytes*, or a *TypeError* is raised.

Note that `quote(string, safe, encoding, errors)` is equivalent to `quote_from_bytes(string.encode(encoding, errors), safe)`.

Example: `quote('/El Niño/')` yields `'/El%20Ni%C3%B1o/'`.

urllib.parse.**quote_plus**(*string*, *safe=''*, *encoding=None*, *errors=None*)

Like *quote()*, but also replace spaces with plus signs, as required for quoting HTML form values when building up a query string to go into a URL. Plus signs in the original string are escaped unless they are included in *safe*. It also does not have *safe* default to `'/'`.

Example: `quote_plus('/El Niño/')` yields `'%2FEl+Ni%C3%B1o%2F'`.

urllib.parse.**quote_from_bytes**(*bytes*, *safe='/'*)

Like *quote()*, but accepts a *bytes* object rather than a *str*, and does not perform string-to-bytes encoding.

Example: `quote_from_bytes(b'a&\xef')` yields `'a%26%EF'`.

urllib.parse.**unquote**(*string*, *encoding='utf-8'*, *errors='replace'*)

Replace `%xx` escapes with their single-character equivalent. The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the *bytes.decode()* method.

string may be either a *str* or a *bytes* object.

encoding defaults to `'utf-8'`. *errors* defaults to `'replace'`, meaning invalid sequences are replaced by a placeholder character.

Example: `unquote('/El%20Ni%C3%B1o/')` yields `'/El Niño/'`.

Changed in version 3.9: *string* parameter supports bytes and str objects (previously only str).

`urllib.parse.`**`unquote_plus`**(*string*, *encoding='utf-8'*, *errors='replace'*)

Like *unquote()*, but also replace plus signs with spaces, as required for unquoting HTML form values.

string must be a *str*.

Example: `unquote_plus('/El+Ni%C3%B1o/')` yields `'/El Niño/'`.

`urllib.parse.`**`unquote_to_bytes`**(*string*)

Replace `%xx` escapes with their single-octet equivalent, and return a *bytes* object.

string may be either a *str* or a *bytes* object.

If it is a *str*, unescaped non-ASCII characters in *string* are encoded into UTF-8 bytes.

Example: `unquote_to_bytes('a%26%EF')` yields `b'a&\xef'`.

`urllib.parse.`**`urlencode`**(*query*, *doseq=False*, *safe=''*, *encoding=None*, *errors=None*, *quote_via=quote_plus*)

Convert a mapping object or a sequence of two-element tuples, which may contain *str* or *bytes* objects, to a percent-encoded ASCII text string. If the resultant string is to be used as a *data* for POST operation with the *urlopen()* function, then it should be encoded to bytes, otherwise it would result in a *TypeError*.

The resulting string is a series of `key=value` pairs separated by `'&'` characters, where both *key* and *value* are quoted using the *quote_via* function. By default, *quote_plus()* is used to quote the values, which means spaces are quoted as a `'+'` character and '/' characters are encoded as `%2F`, which follows the standard for GET requests (`application/x-www-form-urlencoded`). An alternate function that can be passed as *quote_via* is *quote()*, which will encode spaces as `%20` and not encode '/' characters. For maximum control of what is quoted, use `quote` and specify a value for *safe*.

When a sequence of two-element tuples is used as the *query* argument, the first element of each tuple is a key and the second is a value. The value element in itself can be a sequence and in that case, if the optional parameter *doseq* evaluates to `True`, individual `key=value` pairs separated by `'&'` are generated for each element of the value sequence for the key. The order of parameters in the encoded string will match the order of parameter tuples in the sequence.

The *safe*, *encoding*, and *errors* parameters are passed down to *quote_via* (the *encoding* and *errors* parameters are only passed when a query element is a *str*).

To reverse this encoding process, *parse_qs()* and *parse_qsl()* are provided in this module to parse query strings into Python data structures.

Refer to *urllib examples* to find out how the *urllib.parse.urlencode()* method can be used for generating the query string of a URL or data for a POST request.

Changed in version 3.2: *query* supports bytes and string objects.

Changed in version 3.5: Added the *quote_via* parameter.

See also

WHATWG - URL Living standard
Working Group for the URL Standard that defines URLs, domains, IP addresses, the application/x-www-form-urlencoded format, and their API.

RFC 3986 - Uniform Resource Identifiers
This is the current standard (STD66). Any changes to urllib.parse module should conform to this. Certain deviations could be observed, which are mostly for backward compatibility purposes and for certain de-facto parsing requirements as commonly observed in major browsers.

RFC 2732 - Format for Literal IPv6 Addresses in URL's.
This specifies the parsing requirements of IPv6 URLs.

RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax
Document describing the generic syntactic requirements for both Uniform Resource Names (URNs) and Uniform Resource Locators (URLs).

RFC 2368 - The mailto URL scheme.
Parsing requirements for mailto URL schemes.

RFC 1808 - Relative Uniform Resource Locators
This Request For Comments includes the rules for joining an absolute and a relative URL, including a fair number of "Abnormal Examples" which govern the treatment of border cases.

RFC 1738 - Uniform Resource Locators (URL)
This specifies the formal syntax and semantics of absolute URLs.

21.7 `urllib.error` — Exception classes raised by urllib.request

Source code: Lib/urllib/error.py

The *urllib.error* module defines the exception classes for exceptions raised by *urllib.request*. The base exception class is *URLError*.

The following exceptions are raised by *urllib.error* as appropriate:

exception urllib.error.**URLError**

The handlers raise this exception (or derived exceptions) when they run into a problem. It is a subclass of *OSError*.

reason

The reason for this error. It can be a message string or another exception instance.

Changed in version 3.3: *URLError* used to be a subtype of *IOError*, which is now an alias of *OSError*.

exception urllib.error.**HTTPError**(*url*, *code*, *msg*, *hdrs*, *fp*)

Though being an exception (a subclass of *URLError*), an *HTTPError* can also function as a non-exceptional file-like return value (the same thing that *urlopen()* returns). This is useful when handling exotic HTTP errors, such as requests for authentication.

url

Contains the request URL. An alias for *filename* attribute.

code

An HTTP status code as defined in **RFC 2616**. This numeric value corresponds to a value found in the dictionary of codes as found in *http.server.BaseHTTPRequestHandler.responses*.

reason

This is usually a string explaining the reason for this error. An alias for *msg* attribute.

headers

The HTTP response headers for the HTTP request that caused the *HTTPError*. An alias for *hdrs* attribute.

Added in version 3.4.

fp

A file-like object where the HTTP error body can be read from.

exception urllib.error.**ContentTooShortError**(*msg*, *content*)

This exception is raised when the *urlretrieve()* function detects that the amount of the downloaded data is less than the expected amount (given by the *Content-Length* header).

content

The downloaded (and supposedly truncated) data.

21.8 urllib.robotparser — Parser for robots.txt

Source code: Lib/urllib/robotparser.py

This module provides a single class, *RobotFileParser*, which answers questions about whether or not a particular user agent can fetch a URL on the web site that published the robots.txt file. For more details on the structure of robots.txt files, see http://www.robotstxt.org/orig.html.

class urllib.robotparser.**RobotFileParser**(*url=''*)

This class provides methods to read, parse and answer questions about the robots.txt file at *url*.

set_url(*url*)

Sets the URL referring to a robots.txt file.

read()

Reads the robots.txt URL and feeds it to the parser.

parse(*lines*)

Parses the lines argument.

can_fetch(*useragent*, *url*)

Returns True if the *useragent* is allowed to fetch the *url* according to the rules contained in the parsed robots.txt file.

mtime()

Returns the time the robots.txt file was last fetched. This is useful for long-running web spiders that need to check for new robots.txt files periodically.

modified()

Sets the time the robots.txt file was last fetched to the current time.

crawl_delay(*useragent*)

Returns the value of the Crawl-delay parameter from robots.txt for the *useragent* in question. If there is no such parameter or it doesn't apply to the *useragent* specified or the robots.txt entry for this parameter has invalid syntax, return None.

Added in version 3.6.

request_rate(*useragent*)

Returns the contents of the Request-rate parameter from robots.txt as a *named tuple* RequestRate(requests, seconds). If there is no such parameter or it doesn't apply to the *useragent* specified or the robots.txt entry for this parameter has invalid syntax, return None.

Added in version 3.6.

site_maps()

Returns the contents of the `Sitemap` parameter from `robots.txt` in the form of a *list()*. If there is no such parameter or the `robots.txt` entry for this parameter has invalid syntax, return `None`.

Added in version 3.8.

The following example demonstrates basic use of the *RobotFileParser* class:

```
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rrate = rp.request_rate("*")
>>> rrate.requests
3
>>> rrate.seconds
20
>>> rp.crawl_delay("*")
6
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```

21.9 http — HTTP modules

Source code: Lib/http/__init__.py

---

http is a package that collects several modules for working with the HyperText Transfer Protocol:

- *http.client* is a low-level HTTP protocol client; for high-level URL opening use *urllib.request*
- *http.server* contains basic HTTP server classes based on *socketserver*
- *http.cookies* has utilities for implementing state management with cookies
- *http.cookiejar* provides persistence of cookies

The *http* module also defines the following enums that help you work with http related code:

class http.**HTTPStatus**

Added in version 3.5.

A subclass of *enum.IntEnum* that defines a set of HTTP status codes, reason phrases and long descriptions written in English.

Usage:

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK
HTTPStatus.OK
>>> HTTPStatus.OK == 200
True
>>> HTTPStatus.OK.value
200
>>> HTTPStatus.OK.phrase
'OK'
```

(continues on next page)

(continued from previous page)

```
>>> HTTPStatus.OK.description
'Request fulfilled, document follows'
>>> list(HTTPStatus)
[HTTPStatus.CONTINUE, HTTPStatus.SWITCHING_PROTOCOLS, ...]
```

21.9.1 HTTP status codes

Supported, IANA-registered status codes available in *http.HTTPStatus* are:

Code	Enum Name	Details
100	CONTINUE	HTTP Semantics **RFC 9110**, Section 15.2.1
101	SWITCHING_PROTOCOLS	HTTP Semantics **RFC 9110**, Section 15.2.2
102	PROCESSING	WebDAV **RFC 2518**, Section 10.1
103	EARLY_HINTS	An HTTP Status Code for Indicating Hints **RFC 8297**
200	OK	HTTP Semantics **RFC 9110**, Section 15.3.1
201	CREATED	HTTP Semantics **RFC 9110**, Section 15.3.2
202	ACCEPTED	HTTP Semantics **RFC 9110**, Section 15.3.3
203	NON_AUTHORITATIVE_INFORMATION	HTTP Semantics **RFC 9110**, Section 15.3.4
204	NO_CONTENT	HTTP Semantics **RFC 9110**, Section 15.3.5
205	RESET_CONTENT	HTTP Semantics **RFC 9110**, Section 15.3.6
206	PARTIAL_CONTENT	HTTP Semantics **RFC 9110**, Section 15.3.7
207	MULTI_STATUS	WebDAV **RFC 4918**, Section 11.1
208	ALREADY_REPORTED	WebDAV Binding Extensions **RFC 5842**, Section 7.1 (Experimental)
226	IM_USED	Delta Encoding in HTTP **RFC 3229**, Section 10.4.1
300	MULTIPLE_CHOICES	HTTP Semantics **RFC 9110**, Section 15.4.1
301	MOVED_PERMANENTLY	HTTP Semantics **RFC 9110**, Section 15.4.2
302	FOUND	HTTP Semantics **RFC 9110**, Section 15.4.3
303	SEE_OTHER	HTTP Semantics **RFC 9110**, Section 15.4.4
304	NOT_MODIFIED	HTTP Semantics **RFC 9110**, Section 15.4.5
305	USE_PROXY	HTTP Semantics **RFC 9110**, Section 15.4.6
307	TEMPORARY_REDIRECT	HTTP Semantics **RFC 9110**, Section 15.4.8
308	PERMANENT_REDIRECT	HTTP Semantics **RFC 9110**, Section 15.4.9
400	BAD_REQUEST	HTTP Semantics **RFC 9110**, Section 15.5.1
401	UNAUTHORIZED	HTTP Semantics **RFC 9110**, Section 15.5.2
402	PAYMENT_REQUIRED	HTTP Semantics **RFC 9110**, Section 15.5.3
403	FORBIDDEN	HTTP Semantics **RFC 9110**, Section 15.5.4
404	NOT_FOUND	HTTP Semantics **RFC 9110**, Section 15.5.5
405	METHOD_NOT_ALLOWED	HTTP Semantics **RFC 9110**, Section 15.5.6
406	NOT_ACCEPTABLE	HTTP Semantics **RFC 9110**, Section 15.5.7
407	PROXY_AUTHENTICATION_REQUIRED	HTTP Semantics **RFC 9110**, Section 15.5.8
408	REQUEST_TIMEOUT	HTTP Semantics **RFC 9110**, Section 15.5.9
409	CONFLICT	HTTP Semantics **RFC 9110**, Section 15.5.10
410	GONE	HTTP Semantics **RFC 9110**, Section 15.5.11
411	LENGTH_REQUIRED	HTTP Semantics **RFC 9110**, Section 15.5.12
412	PRECONDITION_FAILED	HTTP Semantics **RFC 9110**, Section 15.5.13
413	CONTENT_TOO_LARGE	HTTP Semantics **RFC 9110**, Section 15.5.14
414	URI_TOO_LONG	HTTP Semantics **RFC 9110**, Section 15.5.15
415	UNSUPPORTED_MEDIA_TYPE	HTTP Semantics **RFC 9110**, Section 15.5.16
416	RANGE_NOT_SATISFIABLE	HTTP Semantics **RFC 9110**, Section 15.5.17
417	EXPECTATION_FAILED	HTTP Semantics **RFC 9110**, Section 15.5.18

continues on next p

Table 1 – continued from previous page

Code	Enum Name	Details
418	IM_A_TEAPOT	HTCPCP/1.0 RFC 2324, Section 2.3.2
421	MISDIRECTED_REQUEST	HTTP Semantics RFC 9110, Section 15.5.20
422	UNPROCESSABLE_CONTENT	HTTP Semantics RFC 9110, Section 15.5.21
423	LOCKED	WebDAV RFC 4918, Section 11.3
424	FAILED_DEPENDENCY	WebDAV RFC 4918, Section 11.4
425	TOO_EARLY	Using Early Data in HTTP RFC 8470
426	UPGRADE_REQUIRED	HTTP Semantics RFC 9110, Section 15.5.22
428	PRECONDITION_REQUIRED	Additional HTTP Status Codes RFC 6585
429	TOO_MANY_REQUESTS	Additional HTTP Status Codes RFC 6585
431	REQUEST_HEADER_FIELDS_TOO_LARGE	Additional HTTP Status Codes RFC 6585
451	UNAVAILABLE_FOR_LEGAL_REASONS	An HTTP Status Code to Report Legal Obstacles RFC 7725
500	INTERNAL_SERVER_ERROR	HTTP Semantics RFC 9110, Section 15.6.1
501	NOT_IMPLEMENTED	HTTP Semantics RFC 9110, Section 15.6.2
502	BAD_GATEWAY	HTTP Semantics RFC 9110, Section 15.6.3
503	SERVICE_UNAVAILABLE	HTTP Semantics RFC 9110, Section 15.6.4
504	GATEWAY_TIMEOUT	HTTP Semantics RFC 9110, Section 15.6.5
505	HTTP_VERSION_NOT_SUPPORTED	HTTP Semantics RFC 9110, Section 15.6.6
506	VARIANT_ALSO_NEGOTIATES	Transparent Content Negotiation in HTTP RFC 2295, Section 8.1 (Experime
507	INSUFFICIENT_STORAGE	WebDAV RFC 4918, Section 11.5
508	LOOP_DETECTED	WebDAV Binding Extensions RFC 5842, Section 7.2 (Experimental)
510	NOT_EXTENDED	An HTTP Extension Framework RFC 2774, Section 7 (Experimental)
511	NETWORK_AUTHENTICATION_REQUIRED	Additional HTTP Status Codes RFC 6585, Section 6

In order to preserve backwards compatibility, enum values are also present in the *http.client* module in the form of constants. The enum name is equal to the constant name (i.e. `http.HTTPStatus.OK` is also available as `http.client.OK`).

Changed in version 3.7: Added `421 MISDIRECTED_REQUEST` status code.

Added in version 3.8: Added `451 UNAVAILABLE_FOR_LEGAL_REASONS` status code.

Added in version 3.9: Added `103 EARLY_HINTS`, `418 IM_A_TEAPOT` and `425 TOO_EARLY` status codes.

Changed in version 3.13: Implemented RFC9110 naming for status constants. Old constant names are preserved for backwards compatibility.

21.9.2 HTTP status category

Added in version 3.12.

The enum values have several properties to indicate the HTTP status category:

Property	Indicates that	Details
is_informational	100 <= status <= 199	HTTP Semantics RFC 9110, Section 15
is_success	200 <= status <= 299	HTTP Semantics RFC 9110, Section 15
is_redirection	300 <= status <= 399	HTTP Semantics RFC 9110, Section 15
is_client_error	400 <= status <= 499	HTTP Semantics RFC 9110, Section 15
is_server_error	500 <= status <= 599	HTTP Semantics RFC 9110, Section 15

Usage:

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK.is_success
True
>>> HTTPStatus.OK.is_client_error
False
```

class http.**HTTPMethod**

Added in version 3.11.

A subclass of *enum.StrEnum* that defines a set of HTTP methods and descriptions written in English.

Usage:

```
>>> from http import HTTPMethod
>>>
>>> HTTPMethod.GET
<HTTPMethod.GET>
>>> HTTPMethod.GET == 'GET'
True
>>> HTTPMethod.GET.value
'GET'
>>> HTTPMethod.GET.description
'Retrieve the target.'
>>> list(HTTPMethod)
[<HTTPMethod.CONNECT>,
 <HTTPMethod.DELETE>,
 <HTTPMethod.GET>,
 <HTTPMethod.HEAD>,
 <HTTPMethod.OPTIONS>,
 <HTTPMethod.PATCH>,
 <HTTPMethod.POST>,
 <HTTPMethod.PUT>,
 <HTTPMethod.TRACE>]
```

21.9.3 HTTP methods

Supported, IANA-registered methods available in *http.HTTPMethod* are:

Method	Enum Name	Details
GET	GET	HTTP Semantics **RFC 9110**, Section 9.3.1
HEAD	HEAD	HTTP Semantics **RFC 9110**, Section 9.3.2
POST	POST	HTTP Semantics **RFC 9110**, Section 9.3.3
PUT	PUT	HTTP Semantics **RFC 9110**, Section 9.3.4
DELETE	DELETE	HTTP Semantics **RFC 9110**, Section 9.3.5
CONNECT	CONNECT	HTTP Semantics **RFC 9110**, Section 9.3.6
OPTIONS	OPTIONS	HTTP Semantics **RFC 9110**, Section 9.3.7
TRACE	TRACE	HTTP Semantics **RFC 9110**, Section 9.3.8
PATCH	PATCH	HTTP/1.1 **RFC 5789**

21.10 `http.client` — HTTP protocol client

Source code: Lib/http/client.py

This module defines classes that implement the client side of the HTTP and HTTPS protocols. It is normally not used directly — the module `urllib.request` uses it to handle URLs that use HTTP and HTTPS.

See also

The Requests package is recommended for a higher-level HTTP client interface.

Note

HTTPS support is only available if Python was compiled with SSL support (through the `ssl` module).

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

The module provides the following classes:

class `http.client.`**HTTPConnection**(*host*, *port=None*, [*timeout*,]*source_address=None*, *blocksize=8192*)

An `HTTPConnection` instance represents one transaction with an HTTP server. It should be instantiated by passing it a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form `host:port`, else the default HTTP port (80) is used. If the optional *timeout* parameter is given, blocking operations (like connection attempts) will timeout after that many seconds (if it is not given, the global default timeout setting is used). The optional *source_address* parameter may be a tuple of a (host, port) to use as the source address the HTTP connection is made from. The optional *blocksize* parameter sets the buffer size in bytes for sending a file-like message body.

For example, the following calls all create instances that connect to the server at the same host and port:

```
>>> h1 = http.client.HTTPConnection('www.python.org')
>>> h2 = http.client.HTTPConnection('www.python.org:80')
>>> h3 = http.client.HTTPConnection('www.python.org', 80)
>>> h4 = http.client.HTTPConnection('www.python.org', 80, timeout=10)
```

Changed in version 3.2: *source_address* was added.

Changed in version 3.4: The *strict* parameter was removed. HTTP 0.9-style "Simple Responses" are no longer supported.

Changed in version 3.7: *blocksize* parameter was added.

class `http.client.`**HTTPSConnection**(*host*, *port=None*, *, [*timeout*,]*source_address=None*, *context=None*, *blocksize=8192*)

A subclass of `HTTPConnection` that uses SSL for communication with secure servers. Default port is `443`. If *context* is specified, it must be a `ssl.SSLContext` instance describing the various SSL options.

Please read *Security considerations* for more information on best practices.

Changed in version 3.2: *source_address*, *context* and *check_hostname* were added.

Changed in version 3.2: This class now supports HTTPS virtual hosts if possible (that is, if `ssl.HAS_SNI` is true).

Changed in version 3.4: The *strict* parameter was removed. HTTP 0.9-style "Simple Responses" are no longer supported.

Changed in version 3.4.3: This class now performs all the necessary certificate and hostname checks by default. To revert to the previous, unverified, behavior `ssl._create_unverified_context()` can be passed to the *context* parameter.

Changed in version 3.8: This class now enables TLS 1.3 *ssl.SSLContext.post_handshake_auth* for the default *context* or when *cert_file* is passed with a custom *context*.

Changed in version 3.10: This class now sends an ALPN extension with protocol indicator `http/1.1` when no *context* is given. Custom *context* should set ALPN protocols with *set_alpn_protocols()*.

Changed in version 3.12: The deprecated *key_file*, *cert_file* and *check_hostname* parameters have been removed.

class `http.client.`**HTTPResponse**(*sock*, *debuglevel=0*, *method=None*, *url=None*)

Class whose instances are returned upon successful connection. Not instantiated directly by user.

Changed in version 3.4: The *strict* parameter was removed. HTTP 0.9 style "Simple Responses" are no longer supported.

This module provides the following function:

`http.client.`**parse_headers**(*fp*)

Parse the headers from a file pointer *fp* representing a HTTP request/response. The file has to be a *BufferedIOBase* reader (i.e. not text) and must provide a valid **RFC 2822** style header.

This function returns an instance of *http.client.HTTPMessage* that holds the header fields, but no payload (the same as *HTTPResponse.msg* and *http.server.BaseHTTPRequestHandler.headers*). After returning, the file pointer *fp* is ready to read the HTTP body.

> **Note**
>
> *parse_headers()* does not parse the start-line of a HTTP message; it only parses the `Name: value` lines. The file has to be ready to read these field lines, so the first line should already be consumed before calling the function.

The following exceptions are raised as appropriate:

exception `http.client.`**HTTPException**

The base class of the other exceptions in this module. It is a subclass of *Exception*.

exception `http.client.`**NotConnected**

A subclass of *HTTPException*.

exception `http.client.`**InvalidURL**

A subclass of *HTTPException*, raised if a port is given and is either non-numeric or empty.

exception `http.client.`**UnknownProtocol**

A subclass of *HTTPException*.

exception `http.client.`**UnknownTransferEncoding**

A subclass of *HTTPException*.

exception `http.client.`**UnimplementedFileMode**

A subclass of *HTTPException*.

exception `http.client.`**IncompleteRead**

A subclass of *HTTPException*.

exception http.client.**ImproperConnectionState**

A subclass of *HTTPException*.

exception http.client.**CannotSendRequest**

A subclass of *ImproperConnectionState*.

exception http.client.**CannotSendHeader**

A subclass of *ImproperConnectionState*.

exception http.client.**ResponseNotReady**

A subclass of *ImproperConnectionState*.

exception http.client.**BadStatusLine**

A subclass of *HTTPException*. Raised if a server responds with a HTTP status code that we don't understand.

exception http.client.**LineTooLong**

A subclass of *HTTPException*. Raised if an excessively long line is received in the HTTP protocol from the server.

exception http.client.**RemoteDisconnected**

A subclass of *ConnectionResetError* and *BadStatusLine*. Raised by *HTTPConnection.getresponse()* when the attempt to read the response results in no data read from the connection, indicating that the remote end has closed the connection.

Added in version 3.5: Previously, *BadStatusLine*`('')` was raised.

The constants defined in this module are:

http.client.**HTTP_PORT**

The default port for the HTTP protocol (always `80`).

http.client.**HTTPS_PORT**

The default port for the HTTPS protocol (always `443`).

http.client.**responses**

This dictionary maps the HTTP 1.1 status codes to the W3C names.

Example: `http.client.responses[http.client.NOT_FOUND]` is `'Not Found'`.

See *HTTP status codes* for a list of HTTP status codes that are available in this module as constants.

21.10.1 HTTPConnection Objects

HTTPConnection instances have the following methods:

HTTPConnection.**request**(*method*, *url*, *body=None*, *headers={}*, ***, *encode_chunked=False*)

This will send a request to the server using the HTTP request method *method* and the request URI *url*. The provided *url* must be an absolute path to conform with **RFC 2616 §5.1.2** (unless connecting to an HTTP proxy server or using the `OPTIONS` or `CONNECT` methods).

If *body* is specified, the specified data is sent after the headers are finished. It may be a *str*, a *bytes-like object*, an open *file object*, or an iterable of *bytes*. If *body* is a string, it is encoded as ISO-8859-1, the default for HTTP. If it is a bytes-like object, the bytes are sent as is. If it is a *file object*, the contents of the file is sent; this file object should support at least the `read()` method. If the file object is an instance of *io.TextIOBase*, the data returned by the `read()` method will be encoded as ISO-8859-1, otherwise the data returned by `read()` is sent as is. If *body* is an iterable, the elements of the iterable are sent as is until the iterable is exhausted.

The *headers* argument should be a mapping of extra HTTP headers to send with the request. A **Host header** must be provided to conform with **RFC 2616 §5.1.2** (unless connecting to an HTTP proxy server or using the `OPTIONS` or `CONNECT` methods).

If *headers* contains neither Content-Length nor Transfer-Encoding, but there is a request body, one of those header fields will be added automatically. If *body* is `None`, the Content-Length header is set to `0` for methods that expect a body (`PUT`, `POST`, and `PATCH`). If *body* is a string or a bytes-like object that is not also a *file*, the Content-Length header is set to its length. Any other type of *body* (files and iterables in general) will be chunk-encoded, and the Transfer-Encoding header will automatically be set instead of Content-Length.

The *encode_chunked* argument is only relevant if Transfer-Encoding is specified in *headers*. If *encode_chunked* is `False`, the HTTPConnection object assumes that all encoding is handled by the calling code. If it is `True`, the body will be chunk-encoded.

For example, to perform a `GET` request to `https://docs.python.org/3/`:

```
>>> import http.client
>>> host = "docs.python.org"
>>> conn = http.client.HTTPSConnection(host)
>>> conn.request("GET", "/3/", headers={"Host": host})
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200 OK
```

Note

Chunked transfer encoding has been added to the HTTP protocol version 1.1. Unless the HTTP server is known to handle HTTP 1.1, the caller must either specify the Content-Length, or must pass a *str* or bytes-like object that is not also a file as the body representation.

Changed in version 3.2: *body* can now be an iterable.

Changed in version 3.6: If neither Content-Length nor Transfer-Encoding are set in *headers*, file and iterable *body* objects are now chunk-encoded. The *encode_chunked* argument was added. No attempt is made to determine the Content-Length for file objects.

`HTTPConnection.`**`getresponse`**`()`

Should be called after a request is sent to get the response from the server. Returns an *HTTPResponse* instance.

Note

Note that you must have read the whole response before you can send a new request to the server.

Changed in version 3.5: If a *ConnectionError* or subclass is raised, the *HTTPConnection* object will be ready to reconnect when a new request is sent.

`HTTPConnection.`**`set_debuglevel`**`(`*level*`)`

Set the debugging level. The default debug level is `0`, meaning no debugging output is printed. Any value greater than `0` will cause all currently defined debug output to be printed to stdout. The `debuglevel` is passed to any new *HTTPResponse* objects that are created.

Added in version 3.1.

`HTTPConnection.`**`set_tunnel`**`(`*host*, *port=None*, *headers=None*`)`

Set the host and the port for HTTP Connect Tunnelling. This allows running the connection through a proxy server.

The *host* and *port* arguments specify the endpoint of the tunneled connection (i.e. the address included in the CONNECT request, *not* the address of the proxy server).

The *headers* argument should be a mapping of extra HTTP headers to send with the CONNECT request.

As HTTP/1.1 is used for HTTP CONNECT tunnelling request, as per the RFC, a HTTP `Host:` header must be provided, matching the authority-form of the request target provided as the destination for the CONNECT request. If a HTTP `Host:` header is not provided via the headers argument, one is generated and transmitted automatically.

For example, to tunnel through a HTTPS proxy server running locally on port 8080, we would pass the address of the proxy to the *`HTTPSConnection`* constructor, and the address of the host that we eventually want to reach to the *`set_tunnel()`* method:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("localhost", 8080)
>>> conn.set_tunnel("www.python.org")
>>> conn.request("HEAD","/index.html")
```

Added in version 3.2.

Changed in version 3.12: HTTP CONNECT tunnelling requests use protocol HTTP/1.1, upgraded from protocol HTTP/1.0. `Host:` HTTP headers are mandatory for HTTP/1.1, so one will be automatically generated and transmitted if not provided in the headers argument.

`HTTPConnection.`**`get_proxy_response_headers`**`()`

Returns a dictionary with the headers of the response received from the proxy server to the CONNECT request.

If the CONNECT request was not sent, the method returns `None`.

Added in version 3.12.

`HTTPConnection.`**`connect`**`()`

Connect to the server specified when the object was created. By default, this is called automatically when making a request if the client does not already have a connection.

Raises an *auditing event* `http.client.connect` with arguments `self`, `host`, `port`.

`HTTPConnection.`**`close`**`()`

Close the connection to the server.

`HTTPConnection.`**`blocksize`**

Buffer size in bytes for sending a file-like message body.

Added in version 3.7.

As an alternative to using the *`request()`* method described above, you can also send your request step by step, by using the four functions below.

`HTTPConnection.`**`putrequest`**`(`*method*, *url*, *skip_host=False*, *skip_accept_encoding=False*`)`

This should be the first call after the connection to the server has been made. It sends a line to the server consisting of the *method* string, the *url* string, and the HTTP version (`HTTP/1.1`). To disable automatic sending of `Host:` or `Accept-Encoding:` headers (for example to accept additional content encodings), specify *skip_host* or *skip_accept_encoding* with non-False values.

`HTTPConnection.`**`putheader`**`(`*header*, *argument*`[`, ...`]`)

Send an **RFC 822**-style header to the server. It sends a line to the server consisting of the header, a colon and a space, and the first argument. If more arguments are given, continuation lines are sent, each consisting of a tab and an argument.

HTTPConnection.**endheaders**(*message_body=None*, *, *encode_chunked=False*)

Send a blank line to the server, signalling the end of the headers. The optional *message_body* argument can be used to pass a message body associated with the request.

If *encode_chunked* is `True`, the result of each iteration of *message_body* will be chunk-encoded as specified in **RFC 7230**, Section 3.3.1. How the data is encoded is dependent on the type of *message_body*. If *message_body* implements the buffer interface the encoding will result in a single chunk. If *message_body* is a *collections.abc.Iterable*, each iteration of *message_body* will result in a chunk. If *message_body* is a *file object*, each call to `.read()` will result in a chunk. The method automatically signals the end of the chunk-encoded data immediately after *message_body*.

> **Note**
>
> Due to the chunked encoding specification, empty chunks yielded by an iterator body will be ignored by the chunk-encoder. This is to avoid premature termination of the read of the request by the target server due to malformed encoding.

Changed in version 3.6: Added chunked encoding support and the *encode_chunked* parameter.

HTTPConnection.**send**(*data*)

Send data to the server. This should be used directly only after the *endheaders()* method has been called and before *getresponse()* is called.

Raises an *auditing event* `http.client.send` with arguments `self`, `data`.

21.10.2 HTTPResponse Objects

An *HTTPResponse* instance wraps the HTTP response from the server. It provides access to the request headers and the entity body. The response is an iterable object and can be used in a with statement.

Changed in version 3.5: The *io.BufferedIOBase* interface is now implemented and all of its reader operations are supported.

HTTPResponse.**read**([*amt*])

Reads and returns the response body, or up to the next *amt* bytes.

HTTPResponse.**readinto**(*b*)

Reads up to the next len(b) bytes of the response body into the buffer *b*. Returns the number of bytes read.

Added in version 3.3.

HTTPResponse.**getheader**(*name*, *default=None*)

Return the value of the header *name*, or *default* if there is no header matching *name*. If there is more than one header with the name *name*, return all of the values joined by ', '. If *default* is any iterable other than a single string, its elements are similarly returned joined by commas.

HTTPResponse.**getheaders**()

Return a list of (header, value) tuples.

HTTPResponse.**fileno**()

Return the `fileno` of the underlying socket.

HTTPResponse.**msg**

A *http.client.HTTPMessage* instance containing the response headers. *http.client.HTTPMessage* is a subclass of *email.message.Message*.

HTTPResponse.**version**

HTTP protocol version used by server. 10 for HTTP/1.0, 11 for HTTP/1.1.

HTTPResponse.**url**

URL of the resource retrieved, commonly used to determine if a redirect was followed.

HTTPResponse.**headers**

Headers of the response in the form of an *email.message.EmailMessage* instance.

HTTPResponse.**status**

Status code returned by server.

HTTPResponse.**reason**

Reason phrase returned by server.

HTTPResponse.**debuglevel**

A debugging hook. If *debuglevel* is greater than zero, messages will be printed to stdout as the response is read and parsed.

HTTPResponse.**closed**

Is `True` if the stream is closed.

HTTPResponse.**geturl**()

Deprecated since version 3.9: Deprecated in favor of *url*.

HTTPResponse.**info**()

Deprecated since version 3.9: Deprecated in favor of *headers*.

HTTPResponse.**getcode**()

Deprecated since version 3.9: Deprecated in favor of *status*.

21.10.3 Examples

Here is an example session that uses the `GET` method:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read()  # This will return entire content.
>>> # The following example demonstrates reading data in chunks.
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> while chunk := r1.read(200):
...     print(repr(chunk))
b'<!doctype html>\n<!--[if"...
...
>>> # Example of an invalid request
>>> conn = http.client.HTTPSConnection("docs.python.org")
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print(r2.status, r2.reason)
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

Here is an example session that uses the `HEAD` method. Note that the `HEAD` method never returns any data.

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("HEAD", "/")
>>> res = conn.getresponse()
>>> print(res.status, res.reason)
200 OK
>>> data = res.read()
>>> print(len(data))
0
>>> data == b''
True
```

Here is an example session that uses the `POST` method:

```
>>> import http.client, urllib.parse
>>> params = urllib.parse.urlencode({'@number': 12524, '@type': 'issue', '@action':
↪'show'})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...            "Accept": "text/plain"}
>>> conn = http.client.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> data
b'Redirecting to <a href="https://bugs.python.org/issue12524">https://bugs.python.org/
↪issue12524</a>'
>>> conn.close()
```

Client side HTTP `PUT` requests are very similar to `POST` requests. The difference lies only on the server side where HTTP servers will allow resources to be created via `PUT` requests. It should be noted that custom HTTP methods are also handled in *`urllib.request.Request`* by setting the appropriate method attribute. Here is an example session that uses the `PUT` method:

```
>>> # This creates an HTTP request
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/file
...
>>> import http.client
>>> BODY = "***filecontents***"
>>> conn = http.client.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200, OK
```

21.10.4 HTTPMessage Objects

class http.client.**HTTPMessage**(*email.message.Message*)

An *http.client.HTTPMessage* instance holds the headers from an HTTP response. It is implemented using the *email.message.Message* class.

21.11 ftplib — FTP protocol client

Source code: Lib/ftplib.py

This module defines the class *FTP* and a few related items. The *FTP* class implements the client side of the FTP protocol. You can use this to write Python programs that perform a variety of automated FTP jobs, such as mirroring other FTP servers. It is also used by the module *urllib.request* to handle URLs that use FTP. For more information on FTP (File Transfer Protocol), see internet **RFC 959**.

The default encoding is UTF-8, following **RFC 2640**.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

Here's a sample session using the *ftplib* module:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.us.debian.org')  # connect to host, default port
>>> ftp.login()                     # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')               # change into "debian" directory
'250 Directory successfully changed.'
>>> ftp.retrlines('LIST')           # list directory contents
-rw-rw-r--    1 1176     1176         1063 Jun 15 10:18 README
...
drwxr-sr-x    5 1176     1176         4096 Dec 19  2000 pool
drwxr-sr-x    4 1176     1176         4096 Nov 17  2008 project
drwxr-xr-x    3 1176     1176         4096 Oct 10  2012 tools
'226 Directory send OK.'
>>> with open('README', 'wb') as fp:
>>>     ftp.retrbinary('RETR README', fp.write)
'226 Transfer complete.'
>>> ftp.quit()
'221 Goodbye.'
```

21.11.1 Reference

FTP objects

class ftplib.**FTP**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *timeout=None*, *source_address=None*, ***, *encoding='utf-8'*)

Return a new instance of the *FTP* class.

Parameters

- **host** (str) – The hostname to connect to. If given, `connect(host)` is implicitly called by the constructor.
- **user** (str) – The username to log in with (default: `'anonymous'`). If given, `login(host, passwd, acct)` is implicitly called by the constructor.
- **passwd** (str) – The password to use when logging in. If not given, and if *passwd* is the empty string or `"-"`, a password will be automatically generated.
- **acct** (str) – Account information to be used for the `ACCT` FTP command. Few systems implement this. See RFC-959 for more details.
- **timeout** (float | *None*) – A timeout in seconds for blocking operations like *connect()* (default: the global default timeout setting).
- **source_address** (tuple | *None*) – A 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.
- **encoding** (str) – The encoding for directories and filenames (default: `'utf-8'`).

The *FTP* class supports the `with` statement, e.g.:

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()
...
'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 .
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp          4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp            18 Jul 10  2008 Fedora
>>>
```

Changed in version 3.2: Support for the `with` statement was added.

Changed in version 3.3: *source_address* parameter was added.

Changed in version 3.9: If the *timeout* parameter is set to be zero, it will raise a *ValueError* to prevent the creation of a non-blocking socket. The *encoding* parameter was added, and the default was changed from Latin-1 to UTF-8 to follow **RFC 2640**.

Several `FTP` methods are available in two flavors: one for handling text files and another for binary files. The methods are named for the command which is used followed by `lines` for the text version or `binary` for the binary version.

FTP instances have the following methods:

set_debuglevel(*level*)

Set the instance's debugging level as an *int*. This controls the amount of debugging output printed. The debug levels are:

- `0` (default): No debug output.
- `1`: Produce a moderate amount of debug output, generally a single line per request.
- `2` or higher: Produce the maximum amount of debugging output, logging each line sent and received on the control connection.

connect(*host=''*, *port=0*, *timeout=None*, *source_address=None*)

Connect to the given host and port. This function should be called only once for each instance; it should not be called if a *host* argument was given when the *FTP* instance was created. All other `FTP` methods can only be called after a connection has successfully been made.

Parameters

- **host** (str) – The host to connect to.
- **port** (int) – The TCP port to connect to (default: 21, as specified by the FTP protocol specification). It is rarely needed to specify a different port number.
- **timeout** (float | *None*) – A timeout in seconds for the connection attempt (default: the global default timeout setting).
- **source_address** (tuple | *None*) – A 2-tuple (host, port) for the socket to bind to as its source address before connecting.

Raises an *auditing event* ftplib.connect with arguments self, host, port.

Changed in version 3.3: *source_address* parameter was added.

getwelcome()

Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

login(*user='anonymous'*, *passwd=''*, *acct=''*)

Log on to the connected FTP server. This function should be called only once for each instance, after a connection has been established; it should not be called if the *host* and *user* arguments were given when the *FTP* instance was created. Most FTP commands are only allowed after the client has logged in.

Parameters

- **user** (str) – The username to log in with (default: 'anonymous').
- **passwd** (str) – The password to use when logging in. If not given, and if *passwd* is the empty string or "-", a password will be automatically generated.
- **acct** (str) – Account information to be used for the ACCT FTP command. Few systems implement this. See RFC-959 for more details.

abort()

Abort a file transfer that is in progress. Using this does not always work, but it's worth a try.

sendcmd(*cmd*)

Send a simple command string to the server and return the response string.

Raises an *auditing event* ftplib.sendcmd with arguments self, cmd.

voidcmd(*cmd*)

Send a simple command string to the server and handle the response. Return the response string if the response code corresponds to success (codes in the range 200–299). Raise *error_reply* otherwise.

Raises an *auditing event* ftplib.sendcmd with arguments self, cmd.

retrbinary(*cmd*, *callback*, *blocksize=8192*, *rest=None*)

Retrieve a file in binary transfer mode.

Parameters

- **cmd** (str) – An appropriate RETR command: "RETR *filename*".
- **callback** (*callable*) – A single parameter callable that is called for each block of data received, with its single argument being the data as *bytes*.
- **blocksize** (int) – The maximum chunk size to read on the low-level *socket* object created to do the actual transfer. This also corresponds to the largest size of data that will be passed to *callback*. Defaults to 8192.

- **rest** (int) – A REST command to be sent to the server. See the documentation for the *rest* parameter of the *transfercmd()* method.

retrlines(*cmd*, *callback=None*)

Retrieve a file or directory listing in the encoding specified by the *encoding* parameter at initialization. *cmd* should be an appropriate RETR command (see *retrbinary()*) or a command such as LIST or NLST (usually just the string 'LIST'). LIST retrieves a list of files and information about those files. NLST retrieves a list of file names. The *callback* function is called for each line with a string argument containing the line with the trailing CRLF stripped. The default *callback* prints the line to *sys.stdout*.

set_pasv(*val*)

Enable "passive" mode if *val* is true, otherwise disable passive mode. Passive mode is on by default.

storbinary(*cmd*, *fp*, *blocksize=8192*, *callback=None*, *rest=None*)

Store a file in binary transfer mode.

Parameters

- **cmd** (str) – An appropriate STOR command: "STOR *filename*".
- **fp** (*file object*) – A file object (opened in binary mode) which is read until EOF, using its *read()* method in blocks of size *blocksize* to provide the data to be stored.
- **blocksize** (int) – The read block size. Defaults to 8192.
- **callback** (*callable*) – A single parameter callable that is called for each block of data sent, with its single argument being the data as *bytes*.
- **rest** (int) – A REST command to be sent to the server. See the documentation for the *rest* parameter of the *transfercmd()* method.

Changed in version 3.2: The *rest* parameter was added.

storlines(*cmd*, *fp*, *callback=None*)

Store a file in line mode. *cmd* should be an appropriate STOR command (see *storbinary()*). Lines are read until EOF from the *file object* *fp* (opened in binary mode) using its *readline()* method to provide the data to be stored. *callback* is an optional single parameter callable that is called on each line after it is sent.

transfercmd(*cmd*, *rest=None*)

Initiate a transfer over the data connection. If the transfer is active, send an EPRT or PORT command and the transfer command specified by *cmd*, and accept the connection. If the server is passive, send an EPSV or PASV command, connect to it, and start the transfer command. Either way, return the socket for the connection.

If optional *rest* is given, a REST command is sent to the server, passing *rest* as an argument. *rest* is usually a byte offset into the requested file, telling the server to restart sending the file's bytes at the requested offset, skipping over the initial bytes. Note however that the *transfercmd()* method converts *rest* to a string with the *encoding* parameter specified at initialization, but no check is performed on the string's contents. If the server does not recognize the REST command, an *error_reply* exception will be raised. If this happens, simply call *transfercmd()* without a *rest* argument.

ntransfercmd(*cmd*, *rest=None*)

Like *transfercmd()*, but returns a tuple of the data connection and the expected size of the data. If the expected size could not be computed, None will be returned as the expected size. *cmd* and *rest* means the same thing as in *transfercmd()*.

mlsd(*path=''*, *facts=[]*)

List a directory in a standardized format by using MLSD command (**RFC 3659**). If *path* is omitted the current directory is assumed. *facts* is a list of strings representing the type of information desired (e.g. ["type",

`"size", "perm"]`). Return a generator object yielding a tuple of two elements for every file found in path. First element is the file name, the second one is a dictionary containing facts about the file name. Content of this dictionary might be limited by the *facts* argument but server is not guaranteed to return all requested facts.

Added in version 3.3.

nlst(*argument*[, ...])

Return a list of file names as returned by the `NLST` command. The optional *argument* is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `NLST` command.

> **Note**
>
> If your server supports the command, *mlsd()* offers a better API.

dir(*argument*[, ...])

Produce a directory listing as returned by the `LIST` command, printing it to standard output. The optional *argument* is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `LIST` command. If the last argument is a function, it is used as a *callback* function as for *retrlines()*; the default prints to *sys.stdout*. This method returns `None`.

> **Note**
>
> If your server supports the command, *mlsd()* offers a better API.

rename(*fromname*, *toname*)

Rename file *fromname* on the server to *toname*.

delete(*filename*)

Remove the file named *filename* from the server. If successful, returns the text of the response, otherwise raises *error_perm* on permission errors or *error_reply* on other errors.

cwd(*pathname*)

Set the current directory on the server.

mkd(*pathname*)

Create a new directory on the server.

pwd()

Return the pathname of the current directory on the server.

rmd(*dirname*)

Remove the directory named *dirname* on the server.

size(*filename*)

Request the size of the file named *filename* on the server. On success, the size of the file is returned as an integer, otherwise `None` is returned. Note that the `SIZE` command is not standardized, but is supported by many common server implementations.

quit()

Send a `QUIT` command to the server and close the connection. This is the "polite" way to close a connection, but it may raise an exception if the server responds with an error to the `QUIT` command. This implies a call to the *close()* method which renders the *FTP* instance useless for subsequent calls (see below).

close()

Close the connection unilaterally. This should not be applied to an already closed connection such as after a successful call to *quit()*. After this call the *FTP* instance should not be used any more (after a call to *close()* or *quit()* you cannot reopen the connection by issuing another *login()* method).

FTP_TLS objects

class ftplib.**FTP_TLS**(*host=''*, *user=''*, *passwd=''*, *acct=''*, ***, *context=None*, *timeout=None*, *source_address=None*, *encoding='utf-8'*)

An *FTP* subclass which adds TLS support to FTP as described in **RFC 4217**. Connect to port 21 implicitly securing the FTP control connection before authenticating.

> **Note**
>
> The user must explicitly secure the data connection by calling the *prot_p()* method.

Parameters

- **host** (str) – The hostname to connect to. If given, `connect(host)` is implicitly called by the constructor.
- **user** (str) – The username to log in with (default: `'anonymous'`). If given, `login(host, passwd, acct)` is implicitly called by the constructor.
- **passwd** (str) – The password to use when logging in. If not given, and if *passwd* is the empty string or `"-"`, a password will be automatically generated.
- **acct** (str) – Account information to be used for the `ACCT` FTP command. Few systems implement this. See RFC-959 for more details.
- **context** (*ssl.SSLContext*) – An SSL context object which allows bundling SSL configuration options, certificates and private keys into a single, potentially long-lived, structure. Please read *Security considerations* for best practices.
- **timeout** (*float | None*) – A timeout in seconds for blocking operations like *connect()* (default: the global default timeout setting).
- **source_address** (*tuple | None*) – A 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.
- **encoding** (str) – The encoding for directories and filenames (default: `'utf-8'`).

Added in version 3.2.

Changed in version 3.3: Added the *source_address* parameter.

Changed in version 3.4: The class now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

Changed in version 3.9: If the *timeout* parameter is set to be zero, it will raise a *ValueError* to prevent the creation of a non-blocking socket. The *encoding* parameter was added, and the default was changed from Latin-1 to UTF-8 to follow **RFC 2640**.

Changed in version 3.12: The deprecated *keyfile* and *certfile* parameters have been removed.

Here's a sample session using the *FTP_TLS* class:

```
>>> ftps = FTP_TLS('ftp.pureftpd.org')
>>> ftps.login()
'230 Anonymous user logged in'
>>> ftps.prot_p()
'200 Data protection level set to "private"'
>>> ftps.nlst()
['6jack', 'OpenBSD', 'antilink', 'blogbench', 'bsdcam', 'clockspeed', 'djbdns-jedi
↪', 'docs', 'eaccelerator-jedi', 'favicon.ico', 'francotone', 'fugu', 'ignore',
↪'libpuzzle', 'metalog', 'minidentd', 'misc', 'mysql-udf-global-user-variables',
↪'php-jenkins-hash', 'php-skein-hash', 'php-webdav', 'phpaudit', 'phpbench',
↪'pincaster', 'ping', 'posto', 'pub', 'public', 'public_keys', 'pure-ftpd',
↪'qscan', 'qtc', 'sharedance', 'skycache', 'sound', 'tmp', 'ucarp']
```

`FTP_TLS` class inherits from *FTP*, defining these additional methods and attributes:

ssl_version

The SSL version to use (defaults to *ssl.PROTOCOL_SSLv23*).

auth()

Set up a secure control connection by using TLS or SSL, depending on what is specified in the *ssl_version* attribute.

Changed in version 3.4: The method now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

ccc()

Revert control channel back to plaintext. This can be useful to take advantage of firewalls that know how to handle NAT with non-secure FTP without opening fixed ports.

Added in version 3.3.

prot_p()

Set up secure data connection.

prot_c()

Set up clear text data connection.

Module variables

exception ftplib.**error_reply**

Exception raised when an unexpected reply is received from the server.

exception ftplib.**error_temp**

Exception raised when an error code signifying a temporary error (response codes in the range 400–499) is received.

exception ftplib.**error_perm**

Exception raised when an error code signifying a permanent error (response codes in the range 500–599) is received.

exception ftplib.**error_proto**

Exception raised when a reply is received from the server that does not fit the response specifications of the File Transfer Protocol, i.e. begin with a digit in the range 1–5.

ftplib.**all_errors**

The set of all exceptions (as a tuple) that methods of *FTP* instances may raise as a result of problems with the FTP connection (as opposed to programming errors made by the caller). This set includes the four exceptions listed above as well as *OSError* and *EOFError*.

See also

Module *netrc*
Parser for the `.netrc` file format. The file `.netrc` is typically used by FTP clients to load user authentication information before prompting the user.

21.12 `poplib` — POP3 protocol client

Source code: Lib/poplib.py

This module defines a class, *POP3*, which encapsulates a connection to a POP3 server and implements the protocol as defined in **RFC 1939**. The *POP3* class supports both the minimal and optional command sets from **RFC 1939**. The *POP3* class also supports the `STLS` command introduced in **RFC 2595** to enable encrypted communication on an already established connection.

Additionally, this module provides a class *POP3_SSL*, which provides support for connecting to POP3 servers that use SSL as an underlying protocol layer.

Note that POP3, though widely supported, is obsolescent. The implementation quality of POP3 servers varies widely, and too many are quite poor. If your mailserver supports IMAP, you would be better off using the *imaplib.IMAP4* class, as IMAP servers tend to be better implemented.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

The *poplib* module provides two classes:

class poplib.**POP3**(*host*, *port=POP3_PORT*[, *timeout*])

This class implements the actual POP3 protocol. The connection is created when the instance is initialized. If *port* is omitted, the standard POP3 port (110) is used. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt (if not specified, the global default timeout setting will be used).

Raises an *auditing event* `poplib.connect` with arguments `self`, `host`, `port`.

All commands will raise an *auditing event* `poplib.putline` with arguments `self` and `line`, where `line` is the bytes about to be sent to the remote host.

Changed in version 3.9: If the *timeout* parameter is set to be zero, it will raise a *ValueError* to prevent the creation of a non-blocking socket.

class poplib.**POP3_SSL**(*host*, *port=POP3_SSL_PORT*, *, *timeout=None*, *context=None*)

This is a subclass of *POP3* that connects to the server over an SSL encrypted socket. If *port* is not specified, 995, the standard POP3-over-SSL port is used. *timeout* works as in the *POP3* constructor. *context* is an optional *ssl.SSLContext* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read *Security considerations* for best practices.

Raises an *auditing event* `poplib.connect` with arguments `self`, `host`, `port`.

All commands will raise an *auditing event* `poplib.putline` with arguments `self` and `line`, where `line` is the bytes about to be sent to the remote host.

Changed in version 3.2: *context* parameter added.

Changed in version 3.4: The class now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

Changed in version 3.9: If the *timeout* parameter is set to be zero, it will raise a *ValueError* to prevent the creation of a non-blocking socket.

Changed in version 3.12: The deprecated *keyfile* and *certfile* parameters have been removed.

One exception is defined as an attribute of the *poplib* module:

exception poplib.**error_proto**

Exception raised on any errors from this module (errors from *socket* module are not caught). The reason for the exception is passed to the constructor as a string.

See also

Module *imaplib*
The standard Python IMAP module.

Frequently Asked Questions About Fetchmail
The FAQ for the **fetchmail** POP/IMAP client collects information on POP3 server variations and RFC noncompliance that may be useful if you need to write an application based on the POP protocol.

21.12.1 POP3 Objects

All POP3 commands are represented by methods of the same name, in lowercase; most return the response text sent by the server.

A *POP3* instance has the following methods:

POP3.**set_debuglevel**(*level*)

Set the instance's debugging level. This controls the amount of debugging output printed. The default, `0`, produces no debugging output. A value of `1` produces a moderate amount of debugging output, generally a single line per request. A value of `2` or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

POP3.**getwelcome**()

Returns the greeting string sent by the POP3 server.

POP3.**capa**()

Query the server's capabilities as specified in **RFC 2449**. Returns a dictionary in the form `{'name': ['param'...]}`.

Added in version 3.4.

POP3.**user**(*username*)

Send user command, response should indicate that a password is required.

POP3.**pass_**(*password*)

Send password, response includes message count and mailbox size. Note: the mailbox on the server is locked until *quit()* is called.

POP3.**apop**(*user*, *secret*)

Use the more secure APOP authentication to log into the POP3 server.

POP3.**rpop**(*user*)

Use RPOP authentication (similar to UNIX r-commands) to log into POP3 server.

POP3.**stat**()

Get mailbox status. The result is a tuple of 2 integers: `(message count, mailbox size)`.

POP3.**list**([*which*])

Request message list, result is in the form `(response, ['mesg_num octets', ...], octets)`. If *which* is set, it is the message to list.

POP3.**retr**(*which*)

Retrieve whole message number *which*, and set its seen flag. Result is in form `(response, ['line', ...], octets)`.

POP3.**dele**(*which*)

Flag message number *which* for deletion. On most servers deletions are not actually performed until QUIT (the major exception is Eudora QPOP, which deliberately violates the RFCs by doing pending deletes on any disconnect).

POP3.**rset**()

Remove any deletion marks for the mailbox.

POP3.**noop**()

Do nothing. Might be used as a keep-alive.

POP3.**quit**()

Signoff: commit changes, unlock mailbox, drop connection.

POP3.**top**(*which*, *howmuch*)

Retrieves the message header plus *howmuch* lines of the message after the header of message number *which*. Result is in form `(response, ['line', ...], octets)`.

The POP3 TOP command this method uses, unlike the RETR command, doesn't set the message's seen flag; unfortunately, TOP is poorly specified in the RFCs and is frequently broken in off-brand servers. Test this method by hand against the POP3 servers you will use before trusting it.

POP3.**uidl**(*which=None*)

Return message digest (unique id) list. If *which* is specified, result contains the unique id for that message in the form `'response mesgnum uid`, otherwise result is list `(response, ['mesgnum uid', ...], octets)`.

POP3.**utf8**()

Try to switch to UTF-8 mode. Returns the server response if successful, raises *`error_proto`* if not. Specified in **RFC 6856**.

Added in version 3.5.

POP3.**stls**(*context=None*)

Start a TLS session on the active connection as specified in **RFC 2595**. This is only allowed before user authentication

context parameter is a *`ssl.SSLContext`* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read *Security considerations* for best practices.

This method supports hostname checking via *`ssl.SSLContext.check_hostname`* and *Server Name Indication* (see *`ssl.HAS_SNI`*).

Added in version 3.4.

Instances of *`POP3_SSL`* have no additional methods. The interface of this subclass is identical to its parent.

21.12.2 POP3 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print(j)
```

At the end of the module, there is a test section that contains a more extensive example of usage.

21.13 `imaplib` — IMAP4 protocol client

Source code: Lib/imaplib.py

This module defines three classes, *IMAP4*, *IMAP4_SSL* and *IMAP4_stream*, which encapsulate a connection to an IMAP4 server and implement a large subset of the IMAP4rev1 client protocol as defined in **RFC 2060**. It is backward compatible with IMAP4 (**RFC 1730**) servers, but note that the `STATUS` command is not supported in IMAP4.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

Three classes are provided by the *imaplib* module, *IMAP4* is the base class:

class imaplib.**IMAP4**(*host=''*, *port=IMAP4_PORT*, *timeout=None*)

This class implements the actual IMAP4 protocol. The connection is created and protocol version (IMAP4 or IMAP4rev1) is determined when the instance is initialized. If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4 port (143) is used. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If timeout is not given or is `None`, the global default socket timeout is used.

The *IMAP4* class supports the `with` statement. When used like this, the IMAP4 `LOGOUT` command is issued automatically when the `with` statement exits. E.g.:

```
>>> from imaplib import IMAP4
>>> with IMAP4("domain.org") as M:
...     M.noop()
...
('OK', [b'Nothing Accomplished. d25if65hy903weo.87'])
```

Changed in version 3.5: Support for the `with` statement was added.

Changed in version 3.9: The optional *timeout* parameter was added.

Three exceptions are defined as attributes of the *IMAP4* class:

exception IMAP4.**error**

Exception raised on any errors. The reason for the exception is passed to the constructor as a string.

exception IMAP4.**abort**

IMAP4 server errors cause this exception to be raised. This is a sub-class of *IMAP4.error*. Note that closing the instance and instantiating a new one will usually allow recovery from this exception.

exception IMAP4.**readonly**

This exception is raised when a writable mailbox has its status changed by the server. This is a sub-class of *IMAP4.error*. Some other client now has write permission, and the mailbox will need to be re-opened to re-obtain write permission.

There's also a subclass for secure connections:

class imaplib.**IMAP4_SSL**(*host=''*, *port=IMAP4_SSL_PORT*, ***, *ssl_context=None*, *timeout=None*)

This is a subclass derived from *IMAP4* that connects over an SSL encrypted socket (to use this class you need a socket module that was compiled with SSL support). If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4-over-SSL port (993) is used. *ssl_context* is a *ssl.SSLContext* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read *Security considerations* for best practices.

The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If timeout is not given or is `None`, the global default socket timeout is used.

Changed in version 3.3: *ssl_context* parameter was added.

Changed in version 3.4: The class now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

Changed in version 3.9: The optional *timeout* parameter was added.

Changed in version 3.12: The deprecated *keyfile* and *certfile* parameters have been removed.

The second subclass allows for connections created by a child process:

class imaplib.**IMAP4_stream**(*command*)

This is a subclass derived from *IMAP4* that connects to the `stdin/stdout` file descriptors created by passing *command* to `subprocess.Popen()`.

The following utility functions are defined:

imaplib.**Internaldate2tuple**(*datestr*)

Parse an IMAP4 `INTERNALDATE` string and return corresponding local time. The return value is a *time.struct_time* tuple or `None` if the string has wrong format.

imaplib.**Int2AP**(*num*)

Converts an integer into a bytes representation using characters from the set [`A` .. `P`].

imaplib.**ParseFlags**(*flagstr*)

Converts an IMAP4 `FLAGS` response to a tuple of individual flags.

imaplib.**Time2Internaldate**(*date_time*)

Convert *date_time* to an IMAP4 `INTERNALDATE` representation. The return value is a string in the form: `"DD-Mmm-YYYY HH:MM:SS +HHMM"` (including double-quotes). The *date_time* argument can be a number (int or float) representing seconds since epoch (as returned by *time.time()*), a 9-tuple representing local time an instance of *time.struct_time* (as returned by *time.localtime()*), an aware instance of *datetime.datetime*, or a double-quoted string. In the last case, it is assumed to already be in the correct format.

Note that IMAP4 message numbers change as the mailbox changes; in particular, after an `EXPUNGE` command performs deletions the remaining messages are renumbered. So it is highly advisable to use UIDs instead, with the UID command.

At the end of the module, there is a test section that contains a more extensive example of usage.

See also

Documents describing the protocol, sources for servers implementing it, by the University of Washington's IMAP Information Center can all be found at (**Source Code**) https://github.com/uw-imap/imap (**Not Maintained**).

21.13.1 IMAP4 Objects

All IMAP4rev1 commands are represented by methods of the same name, either uppercase or lowercase.

All arguments to commands are converted to strings, except for `AUTHENTICATE`, and the last argument to `APPEND` which is passed as an IMAP4 literal. If necessary (the string contains IMAP4 protocol-sensitive characters and isn't enclosed with either parentheses or double quotes) each string is quoted. However, the *password* argument to the `LOGIN` command is always quoted. If you want to avoid having an argument string quoted (eg: the *flags* argument to `STORE`) then enclose the string in parentheses (eg: `r'(\Deleted)'`).

Each command returns a tuple: `(type, [data, ...])` where *type* is usually `'OK'` or `'NO'`, and *data* is either the text from the command response, or mandated results from the command. Each *data* is either a `bytes`, or a tuple. If a tuple, then the first part is the header of the response, and the second part contains the data (ie: 'literal' value).

The *message_set* options to commands below is a string specifying one or more messages to be acted upon. It may be a simple message number (`'1'`), a range of message numbers (`'2:4'`), or a group of non-contiguous ranges separated by commas (`'1:3,6:9'`). A range can contain an asterisk to indicate an infinite upper bound (`'3:*'`).

An *IMAP4* instance has the following methods:

`IMAP4.`**`append`**(*mailbox*, *flags*, *date_time*, *message*)

Append *message* to named mailbox.

`IMAP4.`**`authenticate`**(*mechanism*, *authobject*)

Authenticate command — requires response processing.

mechanism specifies which authentication mechanism is to be used - it should appear in the instance variable `capabilities` in the form `AUTH=mechanism`.

authobject must be a callable object:

```
data = authobject(response)
```

It will be called to process server continuation responses; the *response* argument it is passed will be `bytes`. It should return `bytes` *data* that will be base64 encoded and sent to the server. It should return `None` if the client abort response `*` should be sent instead.

Changed in version 3.5: string usernames and passwords are now encoded to `utf-8` instead of being limited to ASCII.

`IMAP4.`**`check`**()

Checkpoint mailbox on server.

`IMAP4.`**`close`**()

Close currently selected mailbox. Deleted messages are removed from writable mailbox. This is the recommended command before `LOGOUT`.

`IMAP4.`**`copy`**(*message_set*, *new_mailbox*)

Copy *message_set* messages onto end of *new_mailbox*.

`IMAP4.`**`create`**(*mailbox*)

Create new mailbox named *mailbox*.

IMAP4.**delete**(*mailbox*)

Delete old mailbox named *mailbox*.

IMAP4.**deleteacl**(*mailbox*, *who*)

Delete the ACLs (remove any rights) set for who on mailbox.

IMAP4.**enable**(*capability*)

Enable *capability* (see **RFC 5161**). Most capabilities do not need to be enabled. Currently only the UTF8=ACCEPT capability is supported (see **RFC 6855**).

Added in version 3.5: The *enable()* method itself, and **RFC 6855** support.

IMAP4.**expunge**()

Permanently remove deleted items from selected mailbox. Generates an EXPUNGE response for each deleted message. Returned data contains a list of EXPUNGE message numbers in order received.

IMAP4.**fetch**(*message_set*, *message_parts*)

Fetch (parts of) messages. *message_parts* should be a string of message part names enclosed within parentheses, eg: "(UID BODY[TEXT])". Returned data are tuples of message part envelope and data.

IMAP4.**getacl**(*mailbox*)

Get the ACLs for *mailbox*. The method is non-standard, but is supported by the Cyrus server.

IMAP4.**getannotation**(*mailbox*, *entry*, *attribute*)

Retrieve the specified ANNOTATIONs for *mailbox*. The method is non-standard, but is supported by the Cyrus server.

IMAP4.**getquota**(*root*)

Get the quota *root*'s resource usage and limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**getquotaroot**(*mailbox*)

Get the list of quota roots for the named *mailbox*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**list**([*directory*[, *pattern*]])

List mailbox names in *directory* matching *pattern*. *directory* defaults to the top-level mail folder, and *pattern* defaults to match anything. Returned data contains a list of LIST responses.

IMAP4.**login**(*user*, *password*)

Identify the client using a plaintext password. The *password* will be quoted.

IMAP4.**login_cram_md5**(*user*, *password*)

Force use of CRAM-MD5 authentication when identifying the client to protect the password. Will only work if the server CAPABILITY response includes the phrase AUTH=CRAM-MD5.

IMAP4.**logout**()

Shutdown connection to server. Returns server BYE response.

Changed in version 3.8: The method no longer ignores silently arbitrary exceptions.

IMAP4.**lsub**(*directory=""*, *pattern='*'*)

List subscribed mailbox names in directory matching pattern. *directory* defaults to the top level directory and *pattern* defaults to match any mailbox. Returned data are tuples of message part envelope and data.

IMAP4.**myrights**(*mailbox*)

Show my ACLs for a mailbox (i.e. the rights that I have on mailbox).

IMAP4.**namespace**()

Returns IMAP namespaces as defined in **RFC 2342**.

IMAP4.**noop**()

Send NOOP to server.

IMAP4.**open**(*host*, *port*, *timeout=None*)

Opens socket to *port* at *host*. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If timeout is not given or is None, the global default socket timeout is used. Also note that if the *timeout* parameter is set to be zero, it will raise a *ValueError* to reject creating a non-blocking socket. This method is implicitly called by the *IMAP4* constructor. The connection objects established by this method will be used in the *IMAP4.read()*, *IMAP4.readline()*, *IMAP4.send()*, and *IMAP4.shutdown()* methods. You may override this method.

Raises an *auditing event* imaplib.open with arguments self, host, port.

Changed in version 3.9: The *timeout* parameter was added.

IMAP4.**partial**(*message_num*, *message_part*, *start*, *length*)

Fetch truncated part of a message. Returned data is a tuple of message part envelope and data.

IMAP4.**proxyauth**(*user*)

Assume authentication as *user*. Allows an authorised administrator to proxy into any user's mailbox.

IMAP4.**read**(*size*)

Reads *size* bytes from the remote server. You may override this method.

IMAP4.**readline**()

Reads one line from the remote server. You may override this method.

IMAP4.**recent**()

Prompt server for an update. Returned data is None if no new messages, else value of RECENT response.

IMAP4.**rename**(*oldmailbox*, *newmailbox*)

Rename mailbox named *oldmailbox* to *newmailbox*.

IMAP4.**response**(*code*)

Return data for response *code* if received, or None. Returns the given code, instead of the usual type.

IMAP4.**search**(*charset*, *criterion*[, ...])

Search mailbox for matching messages. *charset* may be None, in which case no CHARSET will be specified in the request to the server. The IMAP protocol requires that at least one criterion be specified; an exception will be raised when the server returns an error. *charset* must be None if the UTF8=ACCEPT capability was enabled using the *enable()* command.

Example:

```
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

IMAP4.**select**(*mailbox='INBOX'*, *readonly=False*)

Select a mailbox. Returned data is the count of messages in *mailbox* (EXISTS response). The default *mailbox* is 'INBOX'. If the *readonly* flag is set, modifications to the mailbox are not allowed.

IMAP4.**send**(*data*)

Sends `data` to the remote server. You may override this method.

Raises an *auditing event* `imaplib.send` with arguments `self`, `data`.

IMAP4.**setacl**(*mailbox*, *who*, *what*)

Set an `ACL` for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

IMAP4.**setannotation**(*mailbox*, *entry*, *attribute*[, ...])

Set `ANNOTATION`s for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

IMAP4.**setquota**(*root*, *limits*)

Set the `quota` *root*'s resource *limits*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**shutdown**()

Close connection established in `open`. This method is implicitly called by *`IMAP4.logout()`*. You may override this method.

IMAP4.**socket**()

Returns socket instance used to connect to server.

IMAP4.**sort**(*sort_criteria*, *charset*, *search_criterion*[, ...])

The `sort` command is a variant of `search` with sorting semantics for the results. Returned data contains a space separated list of matching message numbers.

Sort has two arguments before the *search_criterion* argument(s); a parenthesized list of *sort_criteria*, and the searching *charset*. Note that unlike `search`, the searching *charset* argument is mandatory. There is also a `uid sort` command which corresponds to `sort` the way that `uid search` corresponds to `search`. The `sort` command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the numbers of matching messages.

This is an `IMAP4rev1` extension command.

IMAP4.**starttls**(*ssl_context=None*)

Send a `STARTTLS` command. The *ssl_context* argument is optional and should be a *`ssl.SSLContext`* object. This will enable encryption on the IMAP connection. Please read *Security considerations* for best practices.

Added in version 3.2.

Changed in version 3.4: The method now supports hostname check with *`ssl.SSLContext.check_hostname`* and *Server Name Indication* (see *`ssl.HAS_SNI`*).

IMAP4.**status**(*mailbox*, *names*)

Request named status conditions for *mailbox*.

IMAP4.**store**(*message_set*, *command*, *flag_list*)

Alters flag dispositions for messages in mailbox. *command* is specified by section 6.4.6 of **RFC 2060** as being one of "FLAGS", "+FLAGS", or "-FLAGS", optionally with a suffix of ".SILENT".

For example, to set the delete flag on all messages:

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
   M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

> **Note**
>
> Creating flags containing ']' (for example: "[test]") violates **RFC 3501** (the IMAP protocol). However, imaplib has historically allowed creation of such tags, and popular IMAP servers, such as Gmail, accept and produce such flags. There are non-Python programs which also create such tags. Although it is an RFC violation and IMAP clients and servers are supposed to be strict, imaplib still continues to allow such tags to be created for backward compatibility reasons, and as of Python 3.6, handles them if they are sent from the server, since this improves real-world compatibility.

`IMAP4.`**`subscribe`**(*mailbox*)

Subscribe to new mailbox.

`IMAP4.`**`thread`**(*threading_algorithm*, *charset*, *search_criterion*[, ...])

The `thread` command is a variant of `search` with threading semantics for the results. Returned data contains a space separated list of thread members.

Thread members consist of zero or more messages numbers, delimited by spaces, indicating successive parent and child.

Thread has two arguments before the *search_criterion* argument(s); a *threading_algorithm*, and the searching *charset*. Note that unlike `search`, the searching *charset* argument is mandatory. There is also a `uid thread` command which corresponds to `thread` the way that `uid search` corresponds to `search`. The `thread` command first searches the mailbox for messages that match the given searching criteria using the *charset* argument for the interpretation of strings in the searching criteria. It then returns the matching messages threaded according to the specified threading algorithm.

This is an `IMAP4rev1` extension command.

`IMAP4.`**`uid`**(*command*, *arg*[, ...])

Execute command args with messages identified by UID, rather than message number. Returns response appropriate to command. At least one argument must be supplied; if none are provided, the server will return an error and an exception will be raised.

`IMAP4.`**`unsubscribe`**(*mailbox*)

Unsubscribe from old mailbox.

`IMAP4.`**`unselect`**()

`imaplib.IMAP4.unselect()` frees server's resources associated with the selected mailbox and returns the server to the authenticated state. This command performs the same actions as *`imaplib.IMAP4.close()`*, except that no messages are permanently removed from the currently selected mailbox.

Added in version 3.9.

`IMAP4.`**`xatom`**(*name*[, ...])

Allow simple extension commands notified by server in `CAPABILITY` response.

The following attributes are defined on instances of *`IMAP4`*:

`IMAP4.`**`PROTOCOL_VERSION`**

The most recent supported protocol in the `CAPABILITY` response from the server.

`IMAP4.`**`debug`**

Integer value to control debugging output. The initialize value is taken from the module variable `Debug`. Values greater than three trace each command.

IMAP4.**utf8_enabled**

Boolean value that is normally `False`, but is set to `True` if an *enable()* command is successfully issued for the `UTF8=ACCEPT` capability.

Added in version 3.5.

21.13.2 IMAP4 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```
import getpass, imaplib

M = imaplib.IMAP4(host='example.org')
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print('Message %s\n%s\n' % (num, data[0][1]))
M.close()
M.logout()
```

21.14 smtplib — SMTP protocol client

Source code: Lib/smtplib.py

The *smtplib* module defines an SMTP client session object that can be used to send mail to any internet machine with an SMTP or ESMTP listener daemon. For details of SMTP and ESMTP operation, consult **RFC 821** (Simple Mail Transfer Protocol) and **RFC 1869** (SMTP Service Extensions).

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

class smtplib.**SMTP**(*host=''*, *port=0*, *local_hostname=None*, [*timeout*,]*source_address=None*)

An *SMTP* instance encapsulates an SMTP connection. It has methods that support a full repertoire of SMTP and ESMTP operations. If the optional *host* and *port* parameters are given, the SMTP *connect()* method is called with those parameters during initialization. If specified, *local_hostname* is used as the FQDN of the local host in the HELO/EHLO command. Otherwise, the local hostname is found using *socket.getfqdn()*. If the *connect()* call returns anything other than a success code, an *SMTPConnectError* is raised. The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). If the timeout expires, *TimeoutError* is raised. The optional *source_address* parameter allows binding to some specific source address in a machine with multiple network interfaces, and/or to some specific source TCP port. It takes a 2-tuple `(host, port)`, for the socket to bind to as its source address before connecting. If omitted (or if *host* or *port* are `''` and/or `0` respectively) the OS default behavior will be used.

For normal use, you should only require the initialization/connect, *sendmail()*, and *SMTP.quit()* methods. An example is included below.

The *SMTP* class supports the `with` statement. When used like this, the SMTP `QUIT` command is issued automatically when the `with` statement exits. E.g.:

```
>>> from smtplib import SMTP
>>> with SMTP("domain.org") as smtp:
...     smtp.noop()
...
(250, b'Ok')
>>>
```

All commands will raise an *auditing event* `smtplib.SMTP.send` with arguments `self` and `data`, where `data` is the bytes about to be sent to the remote host.

Changed in version 3.3: Support for the `with` statement was added.

Changed in version 3.3: *source_address* argument was added.

Added in version 3.5: The SMTPUTF8 extension (**RFC 6531**) is now supported.

Changed in version 3.9: If the *timeout* parameter is set to be zero, it will raise a `ValueError` to prevent the creation of a non-blocking socket.

class smtplib.**SMTP_SSL**(*host=''*, *port=0*, *local_hostname=None*, *, [*timeout*,]*context=None*, *source_address=None*)

An `SMTP_SSL` instance behaves exactly the same as instances of `SMTP`. `SMTP_SSL` should be used for situations where SSL is required from the beginning of the connection and using `starttls()` is not appropriate. If *host* is not specified, the local host is used. If *port* is zero, the standard SMTP-over-SSL port (465) is used. The optional arguments *local_hostname*, *timeout* and *source_address* have the same meaning as they do in the `SMTP` class. *context*, also optional, can contain a `SSLContext` and allows configuring various aspects of the secure connection. Please read *Security considerations* for best practices.

Changed in version 3.3: *context* was added.

Changed in version 3.3: The *source_address* argument was added.

Changed in version 3.4: The class now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

Changed in version 3.9: If the *timeout* parameter is set to be zero, it will raise a `ValueError` to prevent the creation of a non-blocking socket

Changed in version 3.12: The deprecated *keyfile* and *certfile* parameters have been removed.

class smtplib.**LMTP**(*host=''*, *port=LMTP_PORT*, *local_hostname=None*, *source_address=None*[, *timeout*])

The LMTP protocol, which is very similar to ESMTP, is heavily based on the standard SMTP client. It's common to use Unix sockets for LMTP, so our `connect()` method must support that as well as a regular host:port server. The optional arguments *local_hostname* and *source_address* have the same meaning as they do in the `SMTP` class. To specify a Unix socket, you must use an absolute path for *host*, starting with a '/'.

Authentication is supported, using the regular SMTP mechanism. When using a Unix socket, LMTP generally don't support or require any authentication, but your mileage might vary.

Changed in version 3.9: The optional *timeout* parameter was added.

A nice selection of exceptions is defined as well:

exception smtplib.**SMTPException**

Subclass of `OSError` that is the base exception class for all the other exceptions provided by this module.

Changed in version 3.4: SMTPException became subclass of `OSError`

exception smtplib.**SMTPServerDisconnected**

This exception is raised when the server unexpectedly disconnects, or when an attempt is made to use the *SMTP* instance before connecting it to a server.

exception smtplib.**SMTPResponseException**

Base class for all exceptions that include an SMTP error code. These exceptions are generated in some instances when the SMTP server returns an error code. The error code is stored in the `smtp_code` attribute of the error, and the `smtp_error` attribute is set to the error message.

exception smtplib.**SMTPSenderRefused**

Sender address refused. In addition to the attributes set by on all *SMTPResponseException* exceptions, this sets 'sender' to the string that the SMTP server refused.

exception smtplib.**SMTPRecipientsRefused**

All recipient addresses refused. The errors for each recipient are accessible through the attribute `recipients`, which is a dictionary of exactly the same sort as *SMTP.sendmail()* returns.

exception smtplib.**SMTPDataError**

The SMTP server refused to accept the message data.

exception smtplib.**SMTPConnectError**

Error occurred during establishment of a connection with the server.

exception smtplib.**SMTPHeloError**

The server refused our `HELO` message.

exception smtplib.**SMTPNotSupportedError**

The command or option attempted is not supported by the server.

Added in version 3.5.

exception smtplib.**SMTPAuthenticationError**

SMTP authentication went wrong. Most probably the server didn't accept the username/password combination provided.

See also

RFC 821 - Simple Mail Transfer Protocol
Protocol definition for SMTP. This document covers the model, operating procedure, and protocol details for SMTP.

RFC 1869 - SMTP Service Extensions
Definition of the ESMTP extensions for SMTP. This describes a framework for extending SMTP with new commands, supporting dynamic discovery of the commands provided by the server, and defines a few additional commands.

21.14.1 SMTP Objects

An *SMTP* instance has the following methods:

SMTP.**set_debuglevel**(*level*)

Set the debug output level. A value of 1 or `True` for *level* results in debug messages for connection and for all messages sent to and received from the server. A value of 2 for *level* results in these messages being timestamped.

Changed in version 3.5: Added debuglevel 2.

SMTP.**docmd**(*cmd*, *args=''*)

Send a command *cmd* to the server. The optional argument *args* is simply concatenated to the command, separated by a space.

This returns a 2-tuple composed of a numeric response code and the actual response line (multiline responses are joined into one long line.)

In normal operation it should not be necessary to call this method explicitly. It is used to implement other methods and may be useful for testing private extensions.

If the connection to the server is lost while waiting for the reply, *SMTPServerDisconnected* will be raised.

SMTP.**connect**(*host='localhost'*, *port=0*)

Connect to a host on a given port. The defaults are to connect to the local host at the standard SMTP port (25). If the hostname ends with a colon (`':'`) followed by a number, that suffix will be stripped off and the number interpreted as the port number to use. This method is automatically invoked by the constructor if a host is specified during instantiation. Returns a 2-tuple of the response code and message sent by the server in its connection response.

Raises an *auditing event* `smtplib.connect` with arguments `self`, `host`, `port`.

SMTP.**helo**(*name=''*)

Identify yourself to the SMTP server using `HELO`. The hostname argument defaults to the fully qualified domain name of the local host. The message returned by the server is stored as the `helo_resp` attribute of the object.

In normal operation it should not be necessary to call this method explicitly. It will be implicitly called by the *sendmail()* when necessary.

SMTP.**ehlo**(*name=''*)

Identify yourself to an ESMTP server using `EHLO`. The hostname argument defaults to the fully qualified domain name of the local host. Examine the response for ESMTP option and store them for use by *has_extn()*. Also sets several informational attributes: the message returned by the server is stored as the `ehlo_resp` attribute, `does_esmtp` is set to `True` or `False` depending on whether the server supports ESMTP, and `esmtp_features` will be a dictionary containing the names of the SMTP service extensions this server supports, and their parameters (if any).

Unless you wish to use *has_extn()* before sending mail, it should not be necessary to call this method explicitly. It will be implicitly called by *sendmail()* when necessary.

SMTP.**ehlo_or_helo_if_needed**()

This method calls *ehlo()* and/or *helo()* if there has been no previous `EHLO` or `HELO` command this session. It tries ESMTP `EHLO` first.

SMTPHeloError
The server didn't reply properly to the `HELO` greeting.

SMTP.**has_extn**(*name*)

Return *True* if *name* is in the set of SMTP service extensions returned by the server, *False* otherwise. Case is ignored.

SMTP.**verify**(*address*)

Check the validity of an address on this server using SMTP VRFY. Returns a tuple consisting of code 250 and a full **RFC 822** address (including human name) if the user address is valid. Otherwise returns an SMTP error code of 400 or greater and an error string.

> **Note**
>
> Many sites disable SMTP VRFY in order to foil spammers.

SMTP.**login**(*user*, *password*, *, *initial_response_ok=True*)

Log in on an SMTP server that requires authentication. The arguments are the username and the password to authenticate with. If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first. This method will return normally if the authentication was successful, or may raise the following exceptions:

SMTPHeloError
The server didn't reply properly to the HELO greeting.

SMTPAuthenticationError
The server didn't accept the username/password combination.

SMTPNotSupportedError
The AUTH command is not supported by the server.

SMTPException
No suitable authentication method was found.

Each of the authentication methods supported by *smtplib* are tried in turn if they are advertised as supported by the server. See *auth()* for a list of supported authentication methods. *initial_response_ok* is passed through to *auth()*.

Optional keyword argument *initial_response_ok* specifies whether, for authentication methods that support it, an "initial response" as specified in **RFC 4954** can be sent along with the AUTH command, rather than requiring a challenge/response.

Changed in version 3.5: *SMTPNotSupportedError* may be raised, and the *initial_response_ok* parameter was added.

SMTP.**auth**(*mechanism*, *authobject*, *, *initial_response_ok=True*)

Issue an SMTP AUTH command for the specified authentication *mechanism*, and handle the challenge response via *authobject*.

mechanism specifies which authentication mechanism is to be used as argument to the AUTH command; the valid values are those listed in the auth element of esmtp_features.

authobject must be a callable object taking an optional single argument:

```
data = authobject(challenge=None)
```

If optional keyword argument *initial_response_ok* is true, authobject() will be called first with no argument. It can return the **RFC 4954** "initial response" ASCII str which will be encoded and sent with the AUTH command as below. If the authobject() does not support an initial response (e.g. because it requires a challenge), it should return None when called with challenge=None. If *initial_response_ok* is false, then authobject() will not be called first with None.

If the initial response check returns None, or if *initial_response_ok* is false, authobject() will be called to process the server's challenge response; the *challenge* argument it is passed will be a bytes. It should return ASCII str *data* that will be base64 encoded and sent to the server.

The SMTP class provides authobjects for the CRAM-MD5, PLAIN, and LOGIN mechanisms; they are named SMTP.auth_cram_md5, SMTP.auth_plain, and SMTP.auth_login respectively. They all require that the user and password properties of the SMTP instance are set to appropriate values.

User code does not normally need to call auth directly, but can instead call the *login()* method, which will try each of the above mechanisms in turn, in the order listed. auth is exposed to facilitate the implementation of authentication methods not (or not yet) supported directly by *smtplib*.

Added in version 3.5.

SMTP.**starttls**(*, *context=None*)

Put the SMTP connection in TLS (Transport Layer Security) mode. All SMTP commands that follow will be encrypted. You should then call *ehlo()* again.

If *keyfile* and *certfile* are provided, they are used to create an *ssl.SSLContext*.

Optional *context* parameter is an *ssl.SSLContext* object; This is an alternative to using a keyfile and a certfile and if specified both *keyfile* and *certfile* should be None.

If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first.

Changed in version 3.12: The deprecated *keyfile* and *certfile* parameters have been removed.

SMTPHeloError
The server didn't reply properly to the HELO greeting.

SMTPNotSupportedError
The server does not support the STARTTLS extension.

RuntimeError
SSL/TLS support is not available to your Python interpreter.

Changed in version 3.3: *context* was added.

Changed in version 3.4: The method now supports hostname check with SSLContext.check_hostname and *Server Name Indicator* (see *HAS_SNI*).

Changed in version 3.5: The error raised for lack of STARTTLS support is now the *SMTPNotSupportedError* subclass instead of the base *SMTPException*.

SMTP.**sendmail**(*from_addr*, *to_addrs*, *msg*, *mail_options=()*, *rcpt_options=()*)

Send mail. The required arguments are an **RFC 822** from-address string, a list of **RFC 822** to-address strings (a bare string will be treated as a list with 1 address), and a message string. The caller may pass a list of ESMTP options (such as 8bitmime) to be used in MAIL FROM commands as *mail_options*. ESMTP options (such as DSN commands) that should be used with all RCPT commands can be passed as *rcpt_options*. (If you need to use different ESMTP options to different recipients you have to use the low-level methods such as mail(), rcpt() and data() to send the message.)

> **Note**
>
> The *from_addr* and *to_addrs* parameters are used to construct the message envelope used by the transport agents. sendmail does not modify the message headers in any way.

msg may be a string containing characters in the ASCII range, or a byte string. A string is encoded to bytes using the ascii codec, and lone \r and \n characters are converted to \r\n characters. A byte string is not modified.

If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first. If the server does ESMTP, message size and each of the specified options will be passed to it (if the option is in the feature set the server advertises). If EHLO fails, HELO will be tried and ESMTP options suppressed.

This method will return normally if the mail is accepted for at least one recipient. Otherwise it will raise an exception. That is, if this method does not raise an exception, then someone should get your mail. If this method does not raise an exception, it returns a dictionary, with one entry for each recipient that was refused. Each entry contains a tuple of the SMTP error code and the accompanying error message sent by the server.

If `SMTPUTF8` is included in *mail_options*, and the server supports it, *from_addr* and *to_addrs* may contain non-ASCII characters.

This method may raise the following exceptions:

SMTPRecipientsRefused
All recipients were refused. Nobody got the mail. The `recipients` attribute of the exception object is a dictionary with information about the refused recipients (like the one returned when at least one recipient was accepted).

SMTPHeloError
The server didn't reply properly to the `HELO` greeting.

SMTPSenderRefused
The server didn't accept the *from_addr*.

SMTPDataError
The server replied with an unexpected error code (other than a refusal of a recipient).

SMTPNotSupportedError
`SMTPUTF8` was given in the *mail_options* but is not supported by the server.

Unless otherwise noted, the connection will be open even after an exception is raised.

Changed in version 3.2: *msg* may be a byte string.

Changed in version 3.5: `SMTPUTF8` support added, and *SMTPNotSupportedError* may be raised if `SMTPUTF8` is specified but the server does not support it.

`SMTP.`**`send_message`**(*msg*, *from_addr=None*, *to_addrs=None*, *mail_options=()*, *rcpt_options=()*)

This is a convenience method for calling *sendmail()* with the message represented by an *email.message.Message* object. The arguments have the same meaning as for *sendmail()*, except that *msg* is a `Message` object.

If *from_addr* is `None` or *to_addrs* is `None`, `send_message` fills those arguments with addresses extracted from the headers of *msg* as specified in **RFC 5322**: *from_addr* is set to the *Sender* field if it is present, and otherwise to the *From* field. *to_addrs* combines the values (if any) of the *To*, *Cc*, and *Bcc* fields from *msg*. If exactly one set of *Resent-** headers appear in the message, the regular headers are ignored and the *Resent-** headers are used instead. If the message contains more than one set of *Resent-** headers, a *ValueError* is raised, since there is no way to unambiguously detect the most recent set of *Resent-* headers.

`send_message` serializes *msg* using *BytesGenerator* with `\r\n` as the *linesep*, and calls *sendmail()* to transmit the resulting message. Regardless of the values of *from_addr* and *to_addrs*, `send_message` does not transmit any *Bcc* or *Resent-Bcc* headers that may appear in *msg*. If any of the addresses in *from_addr* and *to_addrs* contain non-ASCII characters and the server does not advertise `SMTPUTF8` support, an `SMTPNotSupported` error is raised. Otherwise the `Message` is serialized with a clone of its *policy* with the *utf8* attribute set to `True`, and `SMTPUTF8` and `BODY=8BITMIME` are added to *mail_options*.

Added in version 3.2.

Added in version 3.5: Support for internationalized addresses (`SMTPUTF8`).

`SMTP.`**`quit`**`()`

Terminate the SMTP session and close the connection. Return the result of the SMTP `QUIT` command.

Low-level methods corresponding to the standard SMTP/ESMTP commands `HELP`, `RSET`, `NOOP`, `MAIL`, `RCPT`, and `DATA` are also supported. Normally these do not need to be called directly, so they are not documented here. For details, consult the module code.

21.14.2 SMTP Example

This example prompts the user for addresses needed in the message envelope ('To' and 'From' addresses), and the message to be delivered. Note that the headers to be included with the message must be included in the message as entered; this example doesn't do any processing of the **RFC 822** headers. In particular, the 'To' and 'From' addresses must be included in the message headers explicitly:

```
import smtplib

def prompt(title):
    return input(title).strip()

from_addr = prompt("From: ")
to_addrs  = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")

# Add the From: and To: headers at the start!
lines = [f"From: {from_addr}", f"To: {', '.join(to_addrs)}", ""]
while True:
    try:
        line = input()
    except EOFError:
        break
    else:
        lines.append(line)

msg = "\r\n".join(lines)
print("Message length is", len(msg))

server = smtplib.SMTP("localhost")
server.set_debuglevel(1)
server.sendmail(from_addr, to_addrs, msg)
server.quit()
```

Note

In general, you will want to use the *email* package's features to construct an email message, which you can then send via *send_message()*; see *email: Examples*.

21.15 uuid — UUID objects according to RFC 4122

Source code: Lib/uuid.py

This module provides immutable *UUID* objects (the *UUID* class) and the functions *uuid1()*, *uuid3()*, *uuid4()*, *uuid5()* for generating version 1, 3, 4, and 5 UUIDs as specified in **RFC 4122**.

If all you want is a unique ID, you should probably call *uuid1()* or *uuid4()*. Note that *uuid1()* may compromise privacy since it creates a UUID containing the computer's network address. *uuid4()* creates a random UUID.

Depending on support from the underlying platform, *uuid1()* may or may not return a "safe" UUID. A safe UUID is one which is generated using synchronization methods that ensure no two processes can obtain the same UUID. All instances of *UUID* have an *is_safe* attribute which relays any information about the UUID's safety, using this enumeration:

class uuid.**SafeUUID**

Added in version 3.7.

safe

The UUID was generated by the platform in a multiprocessing-safe way.

unsafe

The UUID was not generated in a multiprocessing-safe way.

unknown

The platform does not provide information on whether the UUID was generated safely or not.

class uuid.**UUID**(*hex=None*, *bytes=None*, *bytes_le=None*, *fields=None*, *int=None*, *version=None*, ***, *is_safe=SafeUUID.unknown*)

Create a UUID from either a string of 32 hexadecimal digits, a string of 16 bytes in big-endian order as the *bytes* argument, a string of 16 bytes in little-endian order as the *bytes_le* argument, a tuple of six integers (32-bit *time_low*, 16-bit *time_mid*, 16-bit *time_hi_version*, 8-bit *clock_seq_hi_variant*, 8-bit *clock_seq_low*, 48-bit *node*) as the *fields* argument, or a single 128-bit integer as the *int* argument. When a string of hex digits is given, curly braces, hyphens, and a URN prefix are all optional. For example, these expressions all yield the same UUID:

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes=b'\x12\x34\x56\x78'*4)
UUID(bytes_le=b'\x78\x56\x34\x12\x34\x12\x78\x56' +
              b'\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

Exactly one of *hex*, *bytes*, *bytes_le*, *fields*, or *int* must be given. The *version* argument is optional; if given, the resulting UUID will have its variant and version number set according to **RFC 4122**, overriding bits in the given *hex*, *bytes*, *bytes_le*, *fields*, or *int*.

Comparison of UUID objects are made by way of comparing their *UUID.int* attributes. Comparison with a non-UUID object raises a *TypeError*.

`str(uuid)` returns a string in the form `12345678-1234-5678-1234-567812345678` where the 32 hexadecimal digits represent the UUID.

UUID instances have these read-only attributes:

UUID.**bytes**

The UUID as a 16-byte string (containing the six integer fields in big-endian byte order).

UUID.**bytes_le**

The UUID as a 16-byte string (with *time_low*, *time_mid*, and *time_hi_version* in little-endian byte order).

UUID.**fields**

A tuple of the six integer fields of the UUID, which are also available as six individual attributes and two derived attributes:

Field	Meaning
UUID.**time_low**	The first 32 bits of the UUID.
UUID.**time_mid**	The next 16 bits of the UUID.
UUID.**time_hi_version**	The next 16 bits of the UUID.
UUID.**clock_seq_hi_variant**	The next 8 bits of the UUID.
UUID.**clock_seq_low**	The next 8 bits of the UUID.
UUID.**node**	The last 48 bits of the UUID.
UUID.**time**	The 60-bit timestamp.
UUID.**clock_seq**	The 14-bit sequence number.

UUID.**hex**

The UUID as a 32-character lowercase hexadecimal string.

UUID.**int**

The UUID as a 128-bit integer.

UUID.**urn**

The UUID as a URN as specified in **RFC 4122**.

UUID.**variant**

The UUID variant, which determines the internal layout of the UUID. This will be one of the constants *RESERVED_NCS*, *RFC_4122*, *RESERVED_MICROSOFT*, or *RESERVED_FUTURE*.

UUID.**version**

The UUID version number (1 through 5, meaningful only when the variant is *RFC_4122*).

UUID.**is_safe**

An enumeration of *SafeUUID* which indicates whether the platform generated the UUID in a multiprocessing-safe way.

Added in version 3.7.

The *uuid* module defines the following functions:

uuid.**getnode**()

Get the hardware address as a 48-bit positive integer. The first time this runs, it may launch a separate program, which could be quite slow. If all attempts to obtain the hardware address fail, we choose a random 48-bit number with the multicast bit (least significant bit of the first octet) set to 1 as recommended in **RFC 4122**. "Hardware address" means the MAC address of a network interface. On a machine with multiple network interfaces, universally

administered MAC addresses (i.e. where the second least significant bit of the first octet is *unset*) will be preferred over locally administered MAC addresses, but with no other ordering guarantees.

Changed in version 3.7: Universally administered MAC addresses are preferred over locally administered MAC addresses, since the former are guaranteed to be globally unique, while the latter are not.

uuid.**uuid1**(*node=None*, *clock_seq=None*)

Generate a UUID from a host ID, sequence number, and the current time. If *node* is not given, `getnode()` is used to obtain the hardware address. If *clock_seq* is given, it is used as the sequence number; otherwise a random 14-bit sequence number is chosen.

uuid.**uuid3**(*namespace*, *name*)

Generate a UUID based on the MD5 hash of a namespace identifier (which is a UUID) and a name (which is a `bytes` object or a string that will be encoded using UTF-8).

uuid.**uuid4**()

Generate a random UUID.

uuid.**uuid5**(*namespace*, *name*)

Generate a UUID based on the SHA-1 hash of a namespace identifier (which is a UUID) and a name (which is a `bytes` object or a string that will be encoded using UTF-8).

The `uuid` module defines the following namespace identifiers for use with `uuid3()` or `uuid5()`.

uuid.**NAMESPACE_DNS**

When this namespace is specified, the *name* string is a fully qualified domain name.

uuid.**NAMESPACE_URL**

When this namespace is specified, the *name* string is a URL.

uuid.**NAMESPACE_OID**

When this namespace is specified, the *name* string is an ISO OID.

uuid.**NAMESPACE_X500**

When this namespace is specified, the *name* string is an X.500 DN in DER or a text output format.

The `uuid` module defines the following constants for the possible values of the `variant` attribute:

uuid.**RESERVED_NCS**

Reserved for NCS compatibility.

uuid.**RFC_4122**

Specifies the UUID layout given in **RFC 4122**.

uuid.**RESERVED_MICROSOFT**

Reserved for Microsoft compatibility.

uuid.**RESERVED_FUTURE**

Reserved for future definition.

See also

RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace
This specification defines a Uniform Resource Name namespace for UUIDs, the internal format of UUIDs, and methods of generating UUIDs.

21.15.1 Command-Line Usage

Added in version 3.12.

The *uuid* module can be executed as a script from the command line.

```
python -m uuid [-h] [-u {uuid1,uuid3,uuid4,uuid5}] [-n NAMESPACE] [-N NAME]
```

The following options are accepted:

-h, --help
Show the help message and exit.

-u <uuid>

--uuid <uuid>
Specify the function name to use to generate the uuid. By default *uuid4()* is used.

-n <namespace>

--namespace <namespace>
The namespace is a `UUID`, or `@ns` where `ns` is a well-known predefined UUID addressed by namespace name. Such as `@dns`, `@url`, `@oid`, and `@x500`. Only required for *uuid3()* / *uuid5()* functions.

-N <name>

--name <name>
The name used as part of generating the uuid. Only required for *uuid3()* / *uuid5()* functions.

21.15.2 Example

Here are some examples of typical usage of the *uuid* module:

```
>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')

>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
```

(continues on next page)

(continued from previous page)

```
>>> x.bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
```

21.15.3 Command-Line Example

Here are some examples of typical usage of the *uuid* command line interface:

```
# generate a random uuid - by default uuid4() is used
$ python -m uuid

# generate a uuid using uuid1()
$ python -m uuid -u uuid1

# generate a uuid using uuid5
$ python -m uuid -u uuid5 -n @url -N example.com
```

21.16 socketserver — A framework for network servers

Source code: Lib/socketserver.py

The *socketserver* module simplifies the task of writing network servers.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

There are four basic concrete server classes:

class socketserver.**TCPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

This uses the internet TCP protocol, which provides for continuous streams of data between the client and server. If *bind_and_activate* is true, the constructor automatically attempts to invoke *server_bind()* and *server_activate()*. The other parameters are passed to the *BaseServer* base class.

class socketserver.**UDPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

This uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The parameters are the same as for *TCPServer*.

class socketserver.**UnixStreamServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

class socketserver.**UnixDatagramServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

These more infrequently used classes are similar to the TCP and UDP classes, but use Unix domain sockets; they're not available on non-Unix platforms. The parameters are the same as for *TCPServer*.

These four classes process requests *synchronously*; each request must be completed before the next request can be started. This isn't suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process. The solution is to create a separate process or thread to handle each request; the *ForkingMixIn* and *ThreadingMixIn* mix-in classes can be used to support asynchronous behaviour.

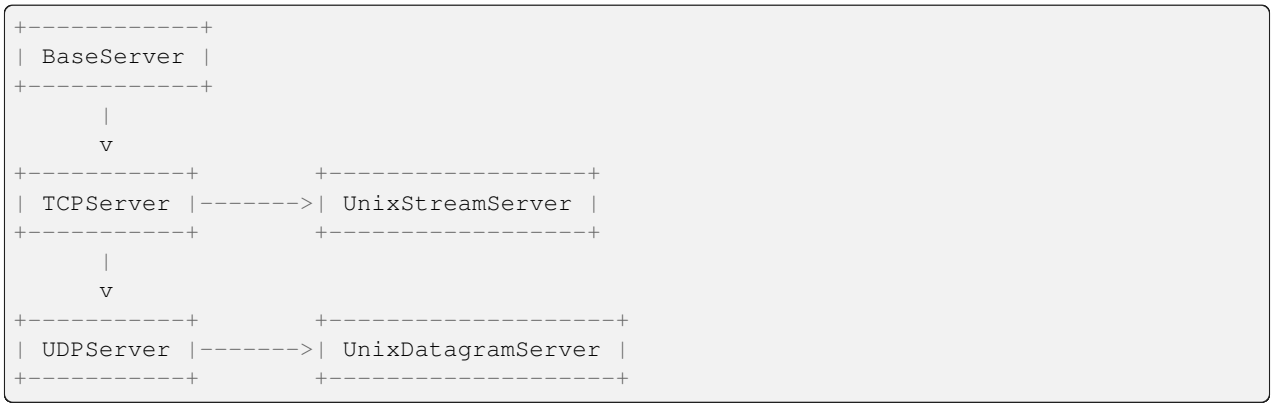
Creating a server requires several steps. First, you must create a request handler class by subclassing the *BaseRequestHandler* class and overriding its *handle()* method; this method will process incoming requests. Second, you must instantiate one of the server classes, passing it the server's address and the request handler class. It is recommended to use the server in a `with` statement. Then call the *handle_request()* or *serve_forever()* method of the server object to process one or many requests. Finally, call *server_close()* to close the socket (unless you used a `with` statement).

When inheriting from *ThreadingMixIn* for threaded connection behavior, you should explicitly declare how you want your threads to behave on an abrupt shutdown. The *ThreadingMixIn* class defines an attribute *daemon_threads*, which indicates whether or not the server should wait for thread termination. You should set the flag explicitly if you would like threads to behave autonomously; the default is *False*, meaning that Python will not exit until all threads created by *ThreadingMixIn* have exited.

Server classes have the same external methods and attributes, no matter what network protocol they use.

21.16.1 Server Creation Notes

There are five classes in an inheritance diagram, four of which represent synchronous servers of four types:



Note that *UnixDatagramServer* derives from *UDPServer*, not from *UnixStreamServer* — the only difference between an IP and a Unix server is the address family.

class socketserver.**ForkingMixIn**

class socketserver.**ThreadingMixIn**

Forking and threading versions of each type of server can be created using these mix-in classes. For instance, *ThreadingUDPServer* is created as follows:

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

The mix-in class comes first, since it overrides a method defined in *UDPServer*. Setting the various attributes also changes the behavior of the underlying server mechanism.

ForkingMixIn and the Forking classes mentioned below are only available on POSIX platforms that support *fork()*.

block_on_close

ForkingMixIn.server_close waits until all child processes complete, except if *block_on_close* attribute is `False`.

ThreadingMixIn.server_close waits until all non-daemon threads complete, except if *block_on_close* attribute is `False`.

daemon_threads

For *ThreadingMixIn* use daemonic threads by setting *ThreadingMixIn.daemon_threads* to `True` to not wait until threads complete.

Changed in version 3.7: *ForkingMixIn.server_close* and *ThreadingMixIn.server_close* now waits until all child processes and non-daemonic threads complete. Add a new *ForkingMixIn.block_on_close* class attribute to opt-in for the pre-3.7 behaviour.

class socketserver.**ForkingTCPServer**

class socketserver.**ForkingUDPServer**

class socketserver.**ThreadingTCPServer**

class socketserver.**ThreadingUDPServer**

class socketserver.**ForkingUnixStreamServer**

class socketserver.**ForkingUnixDatagramServer**

class socketserver.**ThreadingUnixStreamServer**

class socketserver.**ThreadingUnixDatagramServer**

These classes are pre-defined using the mix-in classes.

Added in version 3.12: The `ForkingUnixStreamServer` and `ForkingUnixDatagramServer` classes were added.

To implement a service, you must derive a class from *BaseRequestHandler* and redefine its *handle()* method. You can then run various versions of the service by combining one of the server classes with your request handler class. The request handler class must be different for datagram or stream services. This can be hidden by using the handler subclasses *StreamRequestHandler* or *DatagramRequestHandler*.

Of course, you still have to use your head! For instance, it makes no sense to use a forking server if the service contains state in memory that can be modified by different requests, since the modifications in the child process would never reach the initial state kept in the parent process and passed to each child. In this case, you can use a threading server, but you will probably have to use locks to protect the integrity of the shared data.

On the other hand, if you are building an HTTP server where all data is stored externally (for instance, in the file system), a synchronous class will essentially render the service "deaf" while one request is being handled – which may be for a very long time if a client is slow to receive all the data it has requested. Here a threading or forking server is appropriate.

In some cases, it may be appropriate to process part of a request synchronously, but to finish processing in a forked child depending on the request data. This can be implemented by using a synchronous server and doing an explicit fork in the request handler class *handle()* method.

Another approach to handling multiple simultaneous requests in an environment that supports neither threads nor *fork()* (or where these are too expensive or inappropriate for the service) is to maintain an explicit table of partially finished requests and to use *selectors* to decide which request to work on next (or whether to handle a new incoming request). This is particularly important for stream services where each client can potentially be connected for a long time (if threads or subprocesses cannot be used).

21.16.2 Server Objects

class socketserver.**BaseServer**(*server_address*, *RequestHandlerClass*)

This is the superclass of all Server objects in the module. It defines the interface, given below, but does not implement most of the methods, which is done in subclasses. The two parameters are stored in the respective *server_address* and *RequestHandlerClass* attributes.

fileno()

Return an integer file descriptor for the socket on which the server is listening. This function is most commonly passed to *selectors*, to allow monitoring multiple servers in the same process.

handle_request()

Process a single request. This function calls the following methods in order: *get_request()*, *verify_request()*, and *process_request()*. If the user-provided *handle()* method of the handler class raises an exception, the server's *handle_error()* method will be called. If no request is received within *timeout* seconds, *handle_timeout()* will be called and *handle_request()* will return.

serve_forever(*poll_interval=0.5*)

Handle requests until an explicit *shutdown()* request. Poll for shutdown every *poll_interval* seconds. Ignores the *timeout* attribute. It also calls *service_actions()*, which may be used by a subclass or mixin to provide actions specific to a given service. For example, the *ForkingMixIn* class uses *service_actions()* to clean up zombie child processes.

Changed in version 3.3: Added `service_actions` call to the `serve_forever` method.

service_actions()

This is called in the *serve_forever()* loop. This method can be overridden by subclasses or mixin classes to perform actions specific to a given service, such as cleanup actions.

Added in version 3.3.

shutdown()

Tell the *serve_forever()* loop to stop and wait until it does. *shutdown()* must be called while *serve_forever()* is running in a different thread otherwise it will deadlock.

server_close()

Clean up the server. May be overridden.

address_family

The family of protocols to which the server's socket belongs. Common examples are *socket.AF_INET* and *socket.AF_UNIX*.

RequestHandlerClass

The user-provided request handler class; an instance of this class is created for each request.

server_address

The address on which the server is listening. The format of addresses varies depending on the protocol family; see the documentation for the *socket* module for details. For internet protocols, this is a tuple containing a string giving the address, and an integer port number: `('127.0.0.1', 80)`, for example.

socket

The socket object on which the server will listen for incoming requests.

The server classes support the following class variables:

allow_reuse_address

Whether the server will allow the reuse of an address. This defaults to *False*, and can be set in subclasses to change the policy.

request_queue_size

The size of the request queue. If it takes a long time to process a single request, any requests that arrive while the server is busy are placed into a queue, up to *request_queue_size* requests. Once the queue is full, further requests from clients will get a "Connection denied" error. The default value is usually 5, but this can be overridden by subclasses.

socket_type

The type of socket used by the server; *socket.SOCK_STREAM* and *socket.SOCK_DGRAM* are two common values.

timeout

Timeout duration, measured in seconds, or *None* if no timeout is desired. If *handle_request()* receives no incoming requests within the timeout period, the *handle_timeout()* method is called.

There are various server methods that can be overridden by subclasses of base server classes like *TCPServer*; these methods aren't useful to external users of the server object.

finish_request(*request*, *client_address*)

Actually processes the request by instantiating *RequestHandlerClass* and calling its *handle()* method.

get_request()

Must accept a request from the socket, and return a 2-tuple containing the *new* socket object to be used to communicate with the client, and the client's address.

handle_error(*request*, *client_address*)

This function is called if the *handle()* method of a *RequestHandlerClass* instance raises an exception. The default action is to print the traceback to standard error and continue handling further requests.

Changed in version 3.6: Now only called for exceptions derived from the *Exception* class.

handle_timeout()

This function is called when the *timeout* attribute has been set to a value other than *None* and the timeout period has passed with no requests being received. The default action for forking servers is to collect the status of any child processes that have exited, while in threading servers this method does nothing.

process_request(*request*, *client_address*)

Calls *finish_request()* to create an instance of the *RequestHandlerClass*. If desired, this function can create a new process or thread to handle the request; the *ForkingMixIn* and *ThreadingMixIn* classes do this.

server_activate()

Called by the server's constructor to activate the server. The default behavior for a TCP server just invokes *listen()* on the server's socket. May be overridden.

server_bind()

Called by the server's constructor to bind the socket to the desired address. May be overridden.

verify_request(*request*, *client_address*)

Must return a Boolean value; if the value is *True*, the request will be processed, and if it's *False*, the request will be denied. This function can be overridden to implement access controls for a server. The default implementation always returns *True*.

Changed in version 3.6: Support for the *context manager* protocol was added. Exiting the context manager is equivalent to calling *server_close()*.

21.16.3 Request Handler Objects

class socketserver.**BaseRequestHandler**

This is the superclass of all request handler objects. It defines the interface, given below. A concrete request handler subclass must define a new *handle()* method, and can override any of the other methods. A new instance of the subclass is created for each request.

setup()

Called before the *handle()* method to perform any initialization actions required. The default implementation does nothing.

handle()

This function must do all the work required to service a request. The default implementation does nothing. Several instance attributes are available to it; the request is available as *request*; the client address as *client_address*; and the server instance as *server*, in case it needs access to per-server information.

The type of *request* is different for datagram or stream services. For stream services, *request* is a socket object; for datagram services, *request* is a pair of string and socket.

finish()

Called after the *handle()* method to perform any clean-up actions required. The default implementation does nothing. If *setup()* raises an exception, this function will not be called.

request

The *new* *socket.socket* object to be used to communicate with the client.

client_address

Client address returned by *BaseServer.get_request()*.

server

BaseServer object used for handling the request.

class socketserver.**StreamRequestHandler**

class socketserver.**DatagramRequestHandler**

These *BaseRequestHandler* subclasses override the *setup()* and *finish()* methods, and provide *rfile* and *wfile* attributes.

rfile

A file object from which receives the request is read. Support the *io.BufferedIOBase* readable interface.

wfile

A file object to which the reply is written. Support the *io.BufferedIOBase* writable interface

Changed in version 3.6: *wfile* also supports the *io.BufferedIOBase* writable interface.

21.16.4 Examples

socketserver.TCPServer Example

This is the server side:

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print("Received from {}:".format(self.client_address[0]))
```

(continues on next page)

(continued from previous page)

```
        print(self.data)
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        # Activate the server; this will keep running until you
        # interrupt the program with Ctrl-C
        server.serve_forever()
```

An alternative request handler class that makes use of streams (file-like objects that simplify communication by providing the standard file interface):

```
class MyTCPHandler(socketserver.StreamRequestHandler):

    def handle(self):
        # self.rfile is a file-like object created by the handler;
        # we can now use e.g. readline() instead of raw recv() calls
        self.data = self.rfile.readline().strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # Likewise, self.wfile is a file-like object used to write back
        # to the client
        self.wfile.write(self.data.upper())
```

The difference is that the `readline()` call in the second handler will call `recv()` multiple times until it encounters a newline character, while the single `recv()` call in the first handler will just return what has been received so far from the client's `sendall()` call (typically all of it, but this is not guaranteed by the TCP protocol).

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data + "\n", "utf-8"))

    # Receive data from the server and shut down
    received = str(sock.recv(1024), "utf-8")

print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

The output of the example should look something like this:

Server:

```
$ python TCPServer.py
127.0.0.1 wrote:
b'hello world with TCP'
127.0.0.1 wrote:
b'python is nice'
```

Client:

```
$ python TCPClient.py hello world with TCP
Sent:     hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent:     python is nice
Received: PYTHON IS NICE
```

`socketserver.UDPServer` Example

This is the server side:

```
import socketserver

class MyUDPHandler(socketserver.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and since
    there is no connection the client address must be given explicitly
    when sending data back via sendto().
    """

    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print("{} wrote:".format(self.client_address[0]))
        print(data)
        socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.UDPServer((HOST, PORT), MyUDPHandler) as server:
        server.serve_forever()
```

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connections.
# Instead, data is directly sent to the recipient via sendto().
sock.sendto(bytes(data + "\n", "utf-8"), (HOST, PORT))
received = str(sock.recv(1024), "utf-8")
```

(continues on next page)

(continued from previous page)

```
print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

The output of the example should look exactly like for the TCP server example.

Asynchronous Mixins

To build asynchronous handlers, use the *ThreadingMixIn* and *ForkingMixIn* classes.

An example for the *ThreadingMixIn* class:

```
import socket
import threading
import socketserver

class ThreadedTCPRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024), 'ascii')
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data), 'ascii')
        self.request.sendall(response)

class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    pass

def client(ip, port, message):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((ip, port))
        sock.sendall(bytes(message, 'ascii'))
        response = str(sock.recv(1024), 'ascii')
        print("Received: {}".format(response))

if __name__ == "__main__":
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    with server:
        ip, port = server.server_address

        # Start a thread with the server -- that thread will then start one
        # more thread for each request
        server_thread = threading.Thread(target=server.serve_forever)
        # Exit the server thread when the main thread terminates
        server_thread.daemon = True
        server_thread.start()
        print("Server loop running in thread:", server_thread.name)

        client(ip, port, "Hello World 1")
        client(ip, port, "Hello World 2")
        client(ip, port, "Hello World 3")

        server.shutdown()
```

The output of the example should look something like this:

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

The *ForkingMixIn* class is used in the same way, except that the server will spawn a new process for each request. Available only on POSIX platforms that support *fork()*.

21.17 http.server — HTTP servers

Source code: Lib/http/server.py

This module defines classes for implementing HTTP servers.

> **Warning**
>
> *http.server* is not recommended for production. It only implements *basic security checks*.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

One class, *HTTPServer*, is a *socketserver.TCPServer* subclass. It creates and listens at the HTTP socket, dispatching the requests to a handler. Code to create and run the server looks like this:

```
def run(server_class=HTTPServer, handler_class=BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

class http.server.**HTTPServer**(*server_address*, *RequestHandlerClass*)

This class builds on the *TCPServer* class by storing the server address as instance variables named `server_name` and `server_port`. The server is accessible by the handler, typically through the handler's `server` instance variable.

class http.server.**ThreadingHTTPServer**(*server_address*, *RequestHandlerClass*)

This class is identical to HTTPServer but uses threads to handle requests by using the *ThreadingMixIn*. This is useful to handle web browsers pre-opening sockets, on which *HTTPServer* would wait indefinitely.

Added in version 3.7.

The *HTTPServer* and *ThreadingHTTPServer* must be given a *RequestHandlerClass* on instantiation, of which this module provides three different variants:

class http.server.**BaseHTTPRequestHandler**(*request*, *client_address*, *server*)

This class is used to handle the HTTP requests that arrive at the server. By itself, it cannot respond to any actual HTTP requests; it must be subclassed to handle each request method (e.g. GET or POST). *BaseHTTPRequestHandler* provides a number of class and instance variables, and methods for use by subclasses.

The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request. For example, for the request method `SPAM`, the `do_SPAM()` method will be

called with no arguments. All of the relevant information is stored in instance variables of the handler. Subclasses should not need to override or extend the `__init__()` method.

BaseHTTPRequestHandler has the following instance variables:

client_address

Contains a tuple of the form `(host, port)` referring to the client's address.

server

Contains the server instance.

close_connection

Boolean that should be set before *handle_one_request()* returns, indicating if another request may be expected, or if the connection should be shut down.

requestline

Contains the string representation of the HTTP request line. The terminating CRLF is stripped. This attribute should be set by *handle_one_request()*. If no valid request line was processed, it should be set to the empty string.

command

Contains the command (request type). For example, `'GET'`.

path

Contains the request path. If query component of the URL is present, then `path` includes the query. Using the terminology of **RFC 3986**, `path` here includes `hier-part` and the `query`.

request_version

Contains the version string from the request. For example, `'HTTP/1.0'`.

headers

Holds an instance of the class specified by the *MessageClass* class variable. This instance parses and manages the headers in the HTTP request. The *parse_headers()* function from *http.client* is used to parse the headers and it requires that the HTTP request provide a valid **RFC 2822** style header.

rfile

An *io.BufferedIOBase* input stream, ready to read from the start of the optional input data.

wfile

Contains the output stream for writing a response back to the client. Proper adherence to the HTTP protocol must be used when writing to this stream in order to achieve successful interoperation with HTTP clients.

Changed in version 3.6: This is an *io.BufferedIOBase* stream.

BaseHTTPRequestHandler has the following attributes:

server_version

Specifies the server software version. You may want to override this. The format is multiple whitespace-separated strings, where each string is of the form name[/version]. For example, `'BaseHTTP/0.2'`.

sys_version

Contains the Python system version, in a form usable by the *version_string* method and the *server_version* class variable. For example, `'Python/1.4'`.

error_message_format

Specifies a format string that should be used by *send_error()* method for building an error response to the client. The string is filled by default with variables from *responses* based on the status code that passed to *send_error()*.

error_content_type

Specifies the Content-Type HTTP header of error responses sent to the client. The default value is `'text/html'`.

protocol_version

Specifies the HTTP version to which the server is conformant. It is sent in responses to let the client know the server's communication capabilities for future requests. If set to `'HTTP/1.1'`, the server will permit HTTP persistent connections; however, your server *must* then include an accurate `Content-Length` header (using *send_header()*) in all of its responses to clients. For backwards compatibility, the setting defaults to `'HTTP/1.0'`.

MessageClass

Specifies an *email.message.Message*-like class to parse HTTP headers. Typically, this is not overridden, and it defaults to *http.client.HTTPMessage*.

responses

This attribute contains a mapping of error code integers to two-element tuples containing a short and long message. For example, `{code: (shortmessage, longmessage)}`. The *shortmessage* is usually used as the *message* key in an error response, and *longmessage* as the *explain* key. It is used by *send_response_only()* and *send_error()* methods.

A *BaseHTTPRequestHandler* instance has the following methods:

handle()

Calls *handle_one_request()* once (or, if persistent connections are enabled, multiple times) to handle incoming HTTP requests. You should never need to override it; instead, implement appropriate `do_*()` methods.

handle_one_request()

This method will parse and dispatch the request to the appropriate `do_*()` method. You should never need to override it.

handle_expect_100()

When an HTTP/1.1 conformant server receives an `Expect: 100-continue` request header it responds back with a `100 Continue` followed by `200 OK` headers. This method can be overridden to raise an error if the server does not want the client to continue. For e.g. server can choose to send `417 Expectation Failed` as a response header and `return False`.

Added in version 3.2.

send_error(*code*, *message=None*, *explain=None*)

Sends and logs a complete error reply to the client. The numeric *code* specifies the HTTP error code, with *message* as an optional, short, human readable description of the error. The *explain* argument can be used to provide more detailed information about the error; it will be formatted using the *error_message_format* attribute and emitted, after a complete set of headers, as the response body. The *responses* attribute holds the default values for *message* and *explain* that will be used if no value is provided; for unknown codes the default value for both is the string `???`. The body will be empty if the method is HEAD or the response code is one of the following: `1xx`, `204 No Content`, `205 Reset Content`, `304 Not Modified`.

Changed in version 3.4: The error response includes a Content-Length header. Added the *explain* argument.

send_response(*code*, *message=None*)

Adds a response header to the headers buffer and logs the accepted request. The HTTP response line is written to the internal buffer, followed by *Server* and *Date* headers. The values for these two headers are picked up from the *version_string()* and *date_time_string()* methods, respectively. If the server does not intend to send any other headers using the *send_header()* method, then *send_response()* should be followed by an *end_headers()* call.

Changed in version 3.3: Headers are stored to an internal buffer and *end_headers()* needs to be called explicitly.

send_header(*keyword*, *value*)

Adds the HTTP header to an internal buffer which will be written to the output stream when either *end_headers()* or *flush_headers()* is invoked. *keyword* should specify the header keyword, with *value* specifying its value. Note that, after the send_header calls are done, *end_headers()* MUST BE called in order to complete the operation.

Changed in version 3.2: Headers are stored in an internal buffer.

send_response_only(*code*, *message=None*)

Sends the response header only, used for the purposes when `100 Continue` response is sent by the server to the client. The headers not buffered and sent directly the output stream.If the *message* is not specified, the HTTP message corresponding the response *code* is sent.

Added in version 3.2.

end_headers()

Adds a blank line (indicating the end of the HTTP headers in the response) to the headers buffer and calls *flush_headers()*.

Changed in version 3.2: The buffered headers are written to the output stream.

flush_headers()

Finally send the headers to the output stream and flush the internal headers buffer.

Added in version 3.3.

log_request(*code='-'*, *size='-'*)

Logs an accepted (successful) request. *code* should specify the numeric HTTP code associated with the response. If a size of the response is available, then it should be passed as the *size* parameter.

log_error(...)

Logs an error when a request cannot be fulfilled. By default, it passes the message to *log_message()*, so it takes the same arguments (*format* and additional values).

log_message(*format*, ...)

Logs an arbitrary message to `sys.stderr`. This is typically overridden to create custom error logging mechanisms. The *format* argument is a standard printf-style format string, where the additional arguments to *log_message()* are applied as inputs to the formatting. The client ip address and current date and time are prefixed to every message logged.

version_string()

Returns the server software's version string. This is a combination of the *server_version* and *sys_version* attributes.

date_time_string(*timestamp=None*)

Returns the date and time given by *timestamp* (which must be `None` or in the format returned by *time.time()*), formatted for a message header. If *timestamp* is omitted, it uses the current date and time.

The result looks like `'Sun, 06 Nov 1994 08:49:37 GMT'`.

log_date_time_string()

Returns the current date and time, formatted for logging.

address_string()

Returns the client address.

Changed in version 3.3: Previously, a name lookup was performed. To avoid name resolution delays, it now always returns the IP address.

class http.server.**SimpleHTTPRequestHandler**(*request*, *client_address*, *server*, *directory=None*)

This class serves files from the directory *directory* and below, or the current directory if *directory* is not provided, directly mapping the directory structure to HTTP requests.

Changed in version 3.7: Added the *directory* parameter.

Changed in version 3.9: The *directory* parameter accepts a *path-like object*.

A lot of the work, such as parsing the request, is done by the base class *BaseHTTPRequestHandler*. This class implements the *do_GET()* and *do_HEAD()* functions.

The following are defined as class-level attributes of *SimpleHTTPRequestHandler*:

server_version

This will be "SimpleHTTP/" + __version__, where __version__ is defined at the module level.

extensions_map

A dictionary mapping suffixes into MIME types, contains custom overrides for the default system mappings. The mapping is used case-insensitively, and so should contain only lower-cased keys.

Changed in version 3.9: This dictionary is no longer filled with the default system mappings, but only contains overrides.

The *SimpleHTTPRequestHandler* class defines the following methods:

do_HEAD()

This method serves the 'HEAD' request type: it sends the headers it would send for the equivalent GET request. See the *do_GET()* method for a more complete explanation of the possible headers.

do_GET()

The request is mapped to a local file by interpreting the request as a path relative to the current working directory.

If the request was mapped to a directory, the directory is checked for a file named index.html or index.htm (in that order). If found, the file's contents are returned; otherwise a directory listing is generated by calling the list_directory() method. This method uses *os.listdir()* to scan the directory, and returns a 404 error response if the *listdir()* fails.

If the request was mapped to a file, it is opened. Any *OSError* exception in opening the requested file is mapped to a 404, 'File not found' error. If there was an 'If-Modified-Since' header in the request, and the file was not modified after this time, a 304, 'Not Modified' response is sent. Otherwise, the content type is guessed by calling the guess_type() method, which in turn uses the *extensions_map* variable, and the file contents are returned.

A 'Content-type:' header with the guessed content type is output, followed by a 'Content-Length:' header with the file's size and a 'Last-Modified:' header with the file's modification time.

Then follows a blank line signifying the end of the headers, and then the contents of the file are output. If the file's MIME type starts with text/ the file is opened in text mode; otherwise binary mode is used.

For example usage, see the implementation of the test function in Lib/http/server.py.

Changed in version 3.7: Support of the 'If-Modified-Since' header.

The *SimpleHTTPRequestHandler* class can be used in the following manner in order to create a very basic web-server serving files relative to the current directory:

```
import http.server
import socketserver

PORT = 8000

Handler = http.server.SimpleHTTPRequestHandler

with socketserver.TCPServer(("", PORT), Handler) as httpd:
    print("serving at port", PORT)
    httpd.serve_forever()
```

SimpleHTTPRequestHandler can also be subclassed to enhance behavior, such as using different index file names by overriding the class attribute `index_pages`.

http.server can also be invoked directly using the `-m` switch of the interpreter. Similar to the previous example, this serves files relative to the current directory:

```
python -m http.server
```

The server listens to port 8000 by default. The default can be overridden by passing the desired port number as an argument:

```
python -m http.server 9000
```

By default, the server binds itself to all interfaces. The option `-b/--bind` specifies a specific address to which it should bind. Both IPv4 and IPv6 addresses are supported. For example, the following command causes the server to bind to localhost only:

```
python -m http.server --bind 127.0.0.1
```

Changed in version 3.4: Added the `--bind` option.

Changed in version 3.8: Support IPv6 in the `--bind` option.

By default, the server uses the current directory. The option `-d/--directory` specifies a directory to which it should serve the files. For example, the following command uses a specific directory:

```
python -m http.server --directory /tmp/
```

Changed in version 3.7: Added the `--directory` option.

By default, the server is conformant to HTTP/1.0. The option `-p/--protocol` specifies the HTTP version to which the server is conformant. For example, the following command runs an HTTP/1.1 conformant server:

```
python -m http.server --protocol HTTP/1.1
```

Changed in version 3.11: Added the `--protocol` option.

class `http.server.`**CGIHTTPRequestHandler**(*request*, *client_address*, *server*)

This class is used to serve either files or output of CGI scripts from the current directory and below. Note that mapping HTTP hierarchic structure to local directory structure is exactly as in *SimpleHTTPRequestHandler*.

> **Note**
>
> CGI scripts run by the *CGIHTTPRequestHandler* class cannot execute redirects (HTTP code 302), because code 200 (script output follows) is sent prior to execution of the CGI script. This pre-empts the status code.

The class will however, run the CGI script, instead of serving it as a file, if it guesses it to be a CGI script. Only directory-based CGI are used — the other common server configuration is to treat special extensions as denoting CGI scripts.

The `do_GET()` and `do_HEAD()` functions are modified to run CGI scripts and serve the output, instead of serving files, if the request leads to somewhere below the `cgi_directories` path.

The *CGIHTTPRequestHandler* defines the following data member:

cgi_directories

This defaults to `['/cgi-bin', '/htbin']` and describes directories to treat as containing CGI scripts.

The *CGIHTTPRequestHandler* defines the following method:

do_POST()

This method serves the `'POST'` request type, only allowed for CGI scripts. Error 501, "Can only POST to CGI scripts", is output when trying to POST to a non-CGI url.

Note that CGI scripts will be run with UID of user nobody, for security reasons. Problems with the CGI script will be translated to error 403.

Deprecated since version 3.13, will be removed in version 3.15: *CGIHTTPRequestHandler* is being removed in 3.15. CGI has not been considered a good way to do things for well over a decade. This code has been unmaintained for a while now and sees very little practical use. Retaining it could lead to further *security considerations*.

CGIHTTPRequestHandler can be enabled in the command line by passing the `--cgi` option:

```
python -m http.server --cgi
```

Deprecated since version 3.13, will be removed in version 3.15: *http.server* command line `--cgi` support is being removed because *CGIHTTPRequestHandler* is being removed.

Warning

CGIHTTPRequestHandler and the `--cgi` command line option are not intended for use by untrusted clients and may be vulnerable to exploitation. Always use within a secure environment.

21.17.1 Security Considerations

SimpleHTTPRequestHandler will follow symbolic links when handling requests, this makes it possible for files outside of the specified directory to be served.

Earlier versions of Python did not scrub control characters from the log messages emitted to stderr from `python -m http.server` or the default *BaseHTTPRequestHandler* `.log_message` implementation. This could allow remote clients connecting to your server to send nefarious control codes to your terminal.

Changed in version 3.12: Control characters are scrubbed in stderr logs.

21.18 http.cookies — HTTP state management

Source code: Lib/http/cookies.py

The *http.cookies* module defines classes for abstracting the concept of cookies, an HTTP state management mechanism. It supports both simple string-only cookies, and provides an abstraction for having any serializable data-type as cookie value.

The module formerly strictly applied the parsing rules described in the **RFC 2109** and **RFC 2068** specifications. It has since been discovered that MSIE 3.0x didn't follow the character rules outlined in those specs; many current-day browsers and servers have also relaxed parsing rules when it comes to cookie handling. As a result, this module now uses parsing rules that are a bit less strict than they once were.

The character set, *string.ascii_letters*, *string.digits* and `!#$%&'*+-.^_`|~:` denote the set of valid characters allowed by this module in a cookie name (as *key*).

Changed in version 3.3: Allowed ':' as a valid cookie name character.

Note

On encountering an invalid cookie, *CookieError* is raised, so if your cookie data comes from a browser you should always prepare for invalid data and catch *CookieError* on parsing.

exception http.cookies.**CookieError**

Exception failing because of **RFC 2109** invalidity: incorrect attributes, incorrect *Set-Cookie* header, etc.

class http.cookies.**BaseCookie**([*input*])

This class is a dictionary-like object whose keys are strings and whose values are *Morsel* instances. Note that upon setting a key to a value, the value is first converted to a *Morsel* containing the key and the value.

If *input* is given, it is passed to the *load()* method.

class http.cookies.**SimpleCookie**([*input*])

This class derives from *BaseCookie* and overrides *value_decode()* and *value_encode()*. SimpleCookie supports strings as cookie values. When setting the value, SimpleCookie calls the builtin *str()* to convert the value to a string. Values received from HTTP are kept as strings.

See also

Module *http.cookiejar*
HTTP cookie handling for web *clients*. The *http.cookiejar* and *http.cookies* modules do not depend on each other.

RFC 2109 - HTTP State Management Mechanism
This is the state management specification implemented by this module.

21.18.1 Cookie Objects

BaseCookie.**value_decode**(*val*)

Return a tuple `(real_value, coded_value)` from a string representation. `real_value` can be any type. This method does no decoding in *BaseCookie* — it exists so it can be overridden.

BaseCookie.**value_encode**(*val*)

Return a tuple `(real_value, coded_value)`. *val* can be any type, but `coded_value` will always be converted to a string. This method does no encoding in *BaseCookie* — it exists so it can be overridden.

In general, it should be the case that *value_encode()* and *value_decode()* are inverses on the range of *value_decode*.

BaseCookie.**output**(*attrs=None*, *header='Set-Cookie:'*, *sep='\r\n'*)

Return a string representation suitable to be sent as HTTP headers. *attrs* and *header* are sent to each *Morsel*'s *output()* method. *sep* is used to join the headers together, and is by default the combination `'\r\n'` (CRLF).

BaseCookie.**js_output**(*attrs=None*)

Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP headers was sent.

The meaning for *attrs* is the same as in *output()*.

BaseCookie.**load**(*rawdata*)

If *rawdata* is a string, parse it as an `HTTP_COOKIE` and add the values found there as *Morsel*s. If it is a dictionary, it is equivalent to:

```
for k, v in rawdata.items():
    cookie[k] = v
```

21.18.2 Morsel Objects

class http.cookies.**Morsel**

Abstract a key/value pair, which has some **RFC 2109** attributes.

Morsels are dictionary-like objects, whose set of keys is constant — the valid **RFC 2109** attributes, which are:

- **expires**
- **path**
- **comment**
- **domain**
- **max-age**
- **secure**
- **version**
- **httponly**
- **samesite**

The attribute *httponly* specifies that the cookie is only transferred in HTTP requests, and is not accessible through JavaScript. This is intended to mitigate some forms of cross-site scripting.

The attribute *samesite* specifies that the browser is not allowed to send the cookie along with cross-site requests. This helps to mitigate CSRF attacks. Valid values for this attribute are "Strict" and "Lax".

The keys are case-insensitive and their default value is `''`.

Changed in version 3.5: `__eq__()` now takes *key* and *value* into account.

Changed in version 3.7: Attributes *key*, *value* and *coded_value* are read-only. Use *set()* for setting them.

Changed in version 3.8: Added support for the *samesite* attribute.

`Morsel.`**value**

The value of the cookie.

`Morsel.`**coded_value**

The encoded value of the cookie — this is what should be sent.

`Morsel.`**key**

The name of the cookie.

`Morsel.`**set**(*key*, *value*, *coded_value*)

Set the *key*, *value* and *coded_value* attributes.

`Morsel.`**isReservedKey**(*K*)

Whether *K* is a member of the set of keys of a *Morsel*.

`Morsel.`**output**(*attrs=None*, *header='Set-Cookie:'*)

Return a string representation of the Morsel, suitable to be sent as an HTTP header. By default, all the attributes are included, unless *attrs* is given, in which case it should be a list of attributes to use. *header* is by default `"Set-Cookie:"`.

`Morsel.`**js_output**(*attrs=None*)

Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP header was sent.

The meaning for *attrs* is the same as in *output()*.

`Morsel.`**OutputString**(*attrs=None*)

Return a string representing the Morsel, without any surrounding HTTP or JavaScript.

The meaning for *attrs* is the same as in *output()*.

`Morsel.`**update**(*values*)

Update the values in the Morsel dictionary with the values in the dictionary *values*. Raise an error if any of the keys in the *values* dict is not a valid **RFC 2109** attribute.

Changed in version 3.5: an error is raised for invalid keys.

`Morsel.`**copy**(*value*)

Return a shallow copy of the Morsel object.

Changed in version 3.5: return a Morsel object instead of a dict.

`Morsel.`**setdefault**(*key*, *value=None*)

Raise an error if key is not a valid **RFC 2109** attribute, otherwise behave the same as *dict.setdefault()*.

21.18.3 Example

The following example demonstrates how to use the *http.cookies* module.

```
>>> from http import cookies
>>> C = cookies.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print(C) # generate HTTP headers
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print(C.output()) # same thing
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> C = cookies.SimpleCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print(C.output(header="Cookie:"))
Cookie: rocky=road; Path=/cookie
>>> print(C.output(attrs=[], header="Cookie:"))
Cookie: rocky=road
>>> C = cookies.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger") # load from a string (HTTP header)
>>> print(C)
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = cookies.SimpleCookie()
>>> C.load('keebler="E=everybody; L=\\"Loves\\"; fudge=\\012;";')
>>> print(C)
Set-Cookie: keebler="E=everybody; L=\"Loves\"; fudge=\012;"
>>> C = cookies.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print(C)
Set-Cookie: oreo=doublestuff; Path=/
>>> C = cookies.SimpleCookie()
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = cookies.SimpleCookie()
>>> C["number"] = 7 # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print(C)
Set-Cookie: number=7
Set-Cookie: string=seven
```

21.19 http.cookiejar — Cookie handling for HTTP clients

Source code: Lib/http/cookiejar.py

The *http.cookiejar* module defines classes for automatic handling of HTTP cookies. It is useful for accessing web sites that require small pieces of data – *cookies* – to be set on the client machine by an HTTP response from a web server, and then returned to the server in later HTTP requests.

Both the regular Netscape cookie protocol and the protocol defined by **RFC 2965** are handled. RFC 2965 handling is switched off by default. **RFC 2109** cookies are parsed as Netscape cookies and subsequently treated either as Netscape or RFC 2965 cookies according to the 'policy' in effect. Note that the great majority of cookies on the internet are Netscape cookies. *http.cookiejar* attempts to follow the de-facto Netscape cookie protocol (which differs substantially from that set out in the original Netscape specification), including taking note of the `max-age` and `port` cookie-attributes introduced with RFC 2965.

> **Note**
>
> The various named parameters found in *Set-Cookie* and *Set-Cookie2* headers (eg. `domain` and `expires`) are conventionally referred to as *attributes*. To distinguish them from Python attributes, the documentation for this module uses the term *cookie-attribute* instead.

The module defines the following exception:

exception http.cookiejar.**LoadError**

Instances of *FileCookieJar* raise this exception on failure to load cookies from a file. *LoadError* is a subclass of *OSError*.

Changed in version 3.3: *LoadError* used to be a subtype of *IOError*, which is now an alias of *OSError*.

The following classes are provided:

class http.cookiejar.**CookieJar**(*policy=None*)

policy is an object implementing the *CookiePolicy* interface.

The *CookieJar* class stores HTTP cookies. It extracts cookies from HTTP requests, and returns them in HTTP responses. *CookieJar* instances automatically expire contained cookies when necessary. Subclasses are also responsible for storing and retrieving cookies from a file or database.

class http.cookiejar.**FileCookieJar**(*filename=None*, *delayload=None*, *policy=None*)

policy is an object implementing the *CookiePolicy* interface. For the other arguments, see the documentation for the corresponding attributes.

A *CookieJar* which can load cookies from, and perhaps save cookies to, a file on disk. Cookies are **NOT** loaded from the named file until either the *load()* or *revert()* method is called. Subclasses of this class are documented in section *FileCookieJar subclasses and co-operation with web browsers*.

This should not be initialized directly – use its subclasses below instead.

Changed in version 3.8: The filename parameter supports a *path-like object*.

class http.cookiejar.**CookiePolicy**

This class is responsible for deciding whether each cookie should be accepted from / returned to the server.

class http.cookiejar.**DefaultCookiePolicy**(*blocked_domains=None*, *allowed_domains=None*, *netscape=True*, *rfc2965=False*, *rfc2109_as_netscape=None*, *hide_cookie2=False*, *strict_domain=False*, *strict_rfc2965_unverifiable=True*, *strict_ns_unverifiable=False*, *strict_ns_domain=DefaultCookiePolicy.DomainLiberal*, *strict_ns_set_initial_dollar=False*, *strict_ns_set_path=False*, *secure_protocols=('https', 'wss')*)

Constructor arguments should be passed as keyword arguments only. *blocked_domains* is a sequence of domain names that we never accept cookies from, nor return cookies to. *allowed_domains* if not `None`, this is a sequence of the only domains for which we accept and return cookies. *secure_protocols* is a sequence of protocols for which secure cookies can be added to. By default *https* and *wss* (secure websocket) are considered secure protocols. For all other arguments, see the documentation for `CookiePolicy` and `DefaultCookiePolicy` objects.

`DefaultCookiePolicy` implements the standard accept / reject rules for Netscape and **RFC 2965** cookies. By default, **RFC 2109** cookies (ie. cookies received in a `Set-Cookie` header with a version cookie-attribute of 1) are treated according to the RFC 2965 rules. However, if RFC 2965 handling is turned off or `rfc2109_as_netscape` is `True`, RFC 2109 cookies are 'downgraded' by the `CookieJar` instance to Netscape cookies, by setting the `version` attribute of the `Cookie` instance to 0. `DefaultCookiePolicy` also provides some parameters to allow some fine-tuning of policy.

class http.cookiejar.**Cookie**

This class represents Netscape, **RFC 2109** and **RFC 2965** cookies. It is not expected that users of `http.cookiejar` construct their own `Cookie` instances. Instead, if necessary, call `make_cookies()` on a `CookieJar` instance.

See also

Module `urllib.request`
URL opening with automatic cookie handling.

Module `http.cookies`
HTTP cookie classes, principally useful for server-side code. The `http.cookiejar` and `http.cookies` modules do not depend on each other.

https://curl.se/rfc/cookie_spec.html
The specification of the original Netscape cookie protocol. Though this is still the dominant protocol, the 'Netscape cookie protocol' implemented by all the major browsers (and `http.cookiejar`) only bears a passing resemblance to the one sketched out in `cookie_spec.html`.

RFC 2109 - HTTP State Management Mechanism
Obsoleted by **RFC 2965**. Uses `Set-Cookie` with version=1.

RFC 2965 - HTTP State Management Mechanism
The Netscape protocol with the bugs fixed. Uses `Set-Cookie2` in place of `Set-Cookie`. Not widely used.

http://kristol.org/cookie/errata.html
Unfinished errata to **RFC 2965**.

RFC 2964 - Use of HTTP State Management

21.19.1 CookieJar and FileCookieJar Objects

CookieJar objects support the *iterator* protocol for iterating over contained *Cookie* objects.

CookieJar has the following methods:

CookieJar.**add_cookie_header**(*request*)

Add correct *Cookie* header to *request*.

If policy allows (ie. the `rfc2965` and `hide_cookie2` attributes of the *CookieJar*'s *CookiePolicy* instance are true and false respectively), the *Cookie2* header is also added when appropriate.

The *request* object (usually a *urllib.request.Request* instance) must support the methods `get_full_url()`, `has_header()`, `get_header()`, `header_items()`, `add_unredirected_header()` and the attributes `host`, `type`, `unverifiable` and `origin_req_host` as documented by *urllib.request*.

Changed in version 3.3: *request* object needs `origin_req_host` attribute. Dependency on a deprecated method `get_origin_req_host()` has been removed.

CookieJar.**extract_cookies**(*response*, *request*)

Extract cookies from HTTP *response* and store them in the *CookieJar*, where allowed by policy.

The *CookieJar* will look for allowable *Set-Cookie* and *Set-Cookie2* headers in the *response* argument, and store cookies as appropriate (subject to the *CookiePolicy.set_ok()* method's approval).

The *response* object (usually the result of a call to *urllib.request.urlopen()*, or similar) should support an `info()` method, which returns an *email.message.Message* instance.

The *request* object (usually a *urllib.request.Request* instance) must support the method `get_full_url()` and the attributes `host`, `unverifiable` and `origin_req_host`, as documented by *urllib.request*. The request is used to set default values for cookie-attributes as well as for checking that the cookie is allowed to be set.

Changed in version 3.3: *request* object needs `origin_req_host` attribute. Dependency on a deprecated method `get_origin_req_host()` has been removed.

CookieJar.**set_policy**(*policy*)

Set the *CookiePolicy* instance to be used.

CookieJar.**make_cookies**(*response*, *request*)

Return sequence of *Cookie* objects extracted from *response* object.

See the documentation for *extract_cookies()* for the interfaces required of the *response* and *request* arguments.

CookieJar.**set_cookie_if_ok**(*cookie*, *request*)

Set a *Cookie* if policy says it's OK to do so.

CookieJar.**set_cookie**(*cookie*)

Set a *Cookie*, without checking with policy to see whether or not it should be set.

CookieJar.**clear**([*domain*[, *path*[, *name*]]])

Clear some cookies.

If invoked without arguments, clear all cookies. If given a single argument, only cookies belonging to that *domain* will be removed. If given two arguments, cookies belonging to the specified *domain* and URL *path* are removed. If given three arguments, then the cookie with the specified *domain*, *path* and *name* is removed.

Raises *KeyError* if no matching cookie exists.

CookieJar.**clear_session_cookies**()

Discard all session cookies.

Discards all contained cookies that have a true `discard` attribute (usually because they had either no `max-age` or `expires` cookie-attribute, or an explicit `discard` cookie-attribute). For interactive browsers, the end of a session usually corresponds to closing the browser window.

Note that the `save()` method won't save session cookies anyway, unless you ask otherwise by passing a true *ignore_discard* argument.

FileCookieJar implements the following additional methods:

FileCookieJar.**save**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)

Save cookies to a file.

This base class raises *NotImplementedError*. Subclasses may leave this method unimplemented.

filename is the name of file in which to save cookies. If *filename* is not specified, `self.filename` is used (whose default is the value passed to the constructor, if any); if `self.filename` is *None*, *ValueError* is raised.

ignore_discard: save even cookies set to be discarded. *ignore_expires*: save even cookies that have expired

The file is overwritten if it already exists, thus wiping all the cookies it contains. Saved cookies can be restored later using the *load()* or *revert()* methods.

FileCookieJar.**load**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)

Load cookies from a file.

Old cookies are kept unless overwritten by newly loaded ones.

Arguments are as for *save()*.

The named file must be in the format understood by the class, or *LoadError* will be raised. Also, *OSError* may be raised, for example if the file does not exist.

Changed in version 3.3: *IOError* used to be raised, it is now an alias of *OSError*.

FileCookieJar.**revert**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)

Clear all cookies and reload cookies from a saved file.

revert() can raise the same exceptions as *load()*. If there is a failure, the object's state will not be altered.

FileCookieJar instances have the following public attributes:

FileCookieJar.**filename**

Filename of default file in which to keep cookies. This attribute may be assigned to.

FileCookieJar.**delayload**

If true, load cookies lazily from disk. This attribute should not be assigned to. This is only a hint, since this only affects performance, not behaviour (unless the cookies on disk are changing). A *CookieJar* object may ignore it. None of the *FileCookieJar* classes included in the standard library lazily loads cookies.

21.19.2 FileCookieJar subclasses and co-operation with web browsers

The following *CookieJar* subclasses are provided for reading and writing.

class http.cookiejar.**MozillaCookieJar**(*filename=None*, *delayload=None*, *policy=None*)

A *FileCookieJar* that can load from and save cookies to disk in the Mozilla `cookies.txt` file format (which is also used by curl and the Lynx and Netscape browsers).

> **Note**
>
> This loses information about **RFC 2965** cookies, and also about newer or non-standard cookie-attributes such as `port`.

> **Warning**
>
> Back up your cookies before saving if you have cookies whose loss / corruption would be inconvenient (there are some subtleties which may lead to slight changes in the file over a load / save round-trip).

Also note that cookies saved while Mozilla is running will get clobbered by Mozilla.

class http.cookiejar.**LWPCookieJar**(*filename=None*, *delayload=None*, *policy=None*)

A *FileCookieJar* that can load from and save cookies to disk in format compatible with the libwww-perl library's `Set-Cookie3` file format. This is convenient if you want to store cookies in a human-readable file.

Changed in version 3.8: The filename parameter supports a *path-like object*.

21.19.3 CookiePolicy Objects

Objects implementing the *CookiePolicy* interface have the following methods:

CookiePolicy.**set_ok**(*cookie*, *request*)

Return boolean value indicating whether cookie should be accepted from server.

cookie is a *Cookie* instance. *request* is an object implementing the interface defined by the documentation for *CookieJar.extract_cookies()*.

CookiePolicy.**return_ok**(*cookie*, *request*)

Return boolean value indicating whether cookie should be returned to server.

cookie is a *Cookie* instance. *request* is an object implementing the interface defined by the documentation for *CookieJar.add_cookie_header()*.

CookiePolicy.**domain_return_ok**(*domain*, *request*)

Return `False` if cookies should not be returned, given cookie domain.

This method is an optimization. It removes the need for checking every cookie with a particular domain (which might involve reading many files). Returning true from *domain_return_ok()* and *path_return_ok()* leaves all the work to *return_ok()*.

If *domain_return_ok()* returns true for the cookie domain, *path_return_ok()* is called for the cookie path. Otherwise, *path_return_ok()* and *return_ok()* are never called for that cookie domain. If *path_return_ok()* returns true, *return_ok()* is called with the *Cookie* object itself for a full check. Otherwise, *return_ok()* is never called for that cookie path.

Note that *domain_return_ok()* is called for every *cookie* domain, not just for the *request* domain. For example, the function might be called with both ".example.com" and "www.example.com" if the request domain is "www.example.com". The same goes for *path_return_ok()*.

The *request* argument is as documented for *return_ok()*.

CookiePolicy.**path_return_ok**(*path*, *request*)

Return False if cookies should not be returned, given cookie path.

See the documentation for *domain_return_ok()*.

In addition to implementing the methods above, implementations of the *CookiePolicy* interface must also supply the following attributes, indicating which protocols should be used, and how. All of these attributes may be assigned to.

CookiePolicy.**netscape**

Implement Netscape protocol.

CookiePolicy.**rfc2965**

Implement **RFC 2965** protocol.

CookiePolicy.**hide_cookie2**

Don't add *Cookie2* header to requests (the presence of this header indicates to the server that we understand **RFC 2965** cookies).

The most useful way to define a *CookiePolicy* class is by subclassing from *DefaultCookiePolicy* and overriding some or all of the methods above. *CookiePolicy* itself may be used as a 'null policy' to allow setting and receiving any and all cookies (this is unlikely to be useful).

21.19.4 DefaultCookiePolicy Objects

Implements the standard rules for accepting and returning cookies.

Both **RFC 2965** and Netscape cookies are covered. RFC 2965 handling is switched off by default.

The easiest way to provide your own policy is to override this class and call its methods in your overridden implementations before adding your own additional checks:

```
import http.cookiejar
class MyCookiePolicy(http.cookiejar.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not http.cookiejar.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

In addition to the features required to implement the *CookiePolicy* interface, this class allows you to block and allow domains from setting and receiving cookies. There are also some strictness switches that allow you to tighten up the rather loose Netscape protocol rules a little bit (at the cost of blocking some benign cookies).

A domain blocklist and allowlist is provided (both off by default). Only domains not in the blocklist and present in the allowlist (if the allowlist is active) participate in cookie setting and returning. Use the *blocked_domains* constructor argument, and blocked_domains() and set_blocked_domains() methods (and the corresponding argument and methods for *allowed_domains*). If you set an allowlist, you can turn it off again by setting it to *None*.

Domains in block or allow lists that do not start with a dot must equal the cookie domain to be matched. For example, "example.com" matches a blocklist entry of "example.com", but "www.example.com" does not. Domains that do start with a dot are matched by more specific domains too. For example, both "www.example.com" and "www.coyote.example.com" match ".example.com" (but "example.com" itself does not). IP addresses

are an exception, and must match exactly. For example, if blocked_domains contains `".192.168.1.2"` and `".168.1.2"`, 192.168.1.2 is blocked, but 193.168.1.2 is not.

DefaultCookiePolicy implements the following additional methods:

DefaultCookiePolicy.**blocked_domains**()

Return the sequence of blocked domains (as a tuple).

DefaultCookiePolicy.**set_blocked_domains**(*blocked_domains*)

Set the sequence of blocked domains.

DefaultCookiePolicy.**is_blocked**(*domain*)

Return `True` if *domain* is on the blocklist for setting or receiving cookies.

DefaultCookiePolicy.**allowed_domains**()

Return *None*, or the sequence of allowed domains (as a tuple).

DefaultCookiePolicy.**set_allowed_domains**(*allowed_domains*)

Set the sequence of allowed domains, or *None*.

DefaultCookiePolicy.**is_not_allowed**(*domain*)

Return `True` if *domain* is not on the allowlist for setting or receiving cookies.

DefaultCookiePolicy instances have the following attributes, which are all initialised from the constructor arguments of the same name, and which may all be assigned to.

DefaultCookiePolicy.**rfc2109_as_netscape**

If true, request that the *CookieJar* instance downgrade **RFC 2109** cookies (ie. cookies received in a *Set-Cookie* header with a version cookie-attribute of 1) to Netscape cookies by setting the version attribute of the *Cookie* instance to 0. The default value is *None*, in which case RFC 2109 cookies are downgraded if and only if **RFC 2965** handling is turned off. Therefore, RFC 2109 cookies are downgraded by default.

General strictness switches:

DefaultCookiePolicy.**strict_domain**

Don't allow sites to set two-component domains with country-code top-level domains like `.co.uk`, `.gov.uk`, `.co.nz`.etc. This is far from perfect and isn't guaranteed to work!

RFC 2965 protocol strictness switches:

DefaultCookiePolicy.**strict_rfc2965_unverifiable**

Follow **RFC 2965** rules on unverifiable transactions (usually, an unverifiable transaction is one resulting from a redirect or a request for an image hosted on another site). If this is false, cookies are *never* blocked on the basis of verifiability

Netscape protocol strictness switches:

DefaultCookiePolicy.**strict_ns_unverifiable**

Apply **RFC 2965** rules on unverifiable transactions even to Netscape cookies.

DefaultCookiePolicy.**strict_ns_domain**

Flags indicating how strict to be with domain-matching rules for Netscape cookies. See below for acceptable values.

DefaultCookiePolicy.**strict_ns_set_initial_dollar**

Ignore cookies in Set-Cookie: headers that have names starting with `'$'`.

DefaultCookiePolicy.**strict_ns_set_path**

Don't allow setting cookies whose path doesn't path-match request URI.

`strict_ns_domain` is a collection of flags. Its value is constructed by or-ing together (for example, `DomainStrictNoDots|DomainStrictNonDomain` means both flags are set).

DefaultCookiePolicy.**DomainStrictNoDots**

When setting cookies, the 'host prefix' must not contain a dot (eg. `www.foo.bar.com` can't set a cookie for `.bar.com`, because `www.foo` contains a dot).

DefaultCookiePolicy.**DomainStrictNonDomain**

Cookies that did not explicitly specify a `domain` cookie-attribute can only be returned to a domain equal to the domain that set the cookie (eg. `spam.example.com` won't be returned cookies from `example.com` that had no `domain` cookie-attribute).

DefaultCookiePolicy.**DomainRFC2965Match**

When setting cookies, require a full **RFC 2965** domain-match.

The following attributes are provided for convenience, and are the most useful combinations of the above flags:

DefaultCookiePolicy.**DomainLiberal**

Equivalent to 0 (ie. all of the above Netscape domain strictness flags switched off).

DefaultCookiePolicy.**DomainStrict**

Equivalent to `DomainStrictNoDots|DomainStrictNonDomain`.

21.19.5 Cookie Objects

Cookie instances have Python attributes roughly corresponding to the standard cookie-attributes specified in the various cookie standards. The correspondence is not one-to-one, because there are complicated rules for assigning default values, because the `max-age` and `expires` cookie-attributes contain equivalent information, and because **RFC 2109** cookies may be 'downgraded' by *http.cookiejar* from version 1 to version 0 (Netscape) cookies.

Assignment to these attributes should not be necessary other than in rare circumstances in a *CookiePolicy* method. The class does not enforce internal consistency, so you should know what you're doing if you do that.

Cookie.**version**

Integer or *None*. Netscape cookies have *version* 0. **RFC 2965** and **RFC 2109** cookies have a `version` cookie-attribute of 1. However, note that *http.cookiejar* may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case *version* is 0.

Cookie.**name**

Cookie name (a string).

Cookie.**value**

Cookie value (a string), or *None*.

Cookie.**port**

String representing a port or a set of ports (eg. '80', or '80,8080'), or *None*.

Cookie.**domain**

Cookie domain (a string).

Cookie.**path**

Cookie path (a string, eg. `'/acme/rocket_launchers'`).

Cookie.**secure**

`True` if cookie should only be returned over a secure connection.

Cookie.**expires**

Integer expiry date in seconds since epoch, or *None*. See also the *is_expired()* method.

Cookie.**discard**
: `True` if this is a session cookie.

Cookie.**comment**
: String comment from the server explaining the function of this cookie, or *None*.

Cookie.**comment_url**
: URL linking to a comment from the server explaining the function of this cookie, or *None*.

Cookie.**rfc2109**
: `True` if this cookie was received as an **RFC 2109** cookie (ie. the cookie arrived in a *Set-Cookie* header, and the value of the Version cookie-attribute in that header was 1). This attribute is provided because *http.cookiejar* may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case *version* is 0.

Cookie.**port_specified**
: `True` if a port or set of ports was explicitly specified by the server (in the *Set-Cookie* / *Set-Cookie2* header).

Cookie.**domain_specified**
: `True` if a domain was explicitly specified by the server.

Cookie.**domain_initial_dot**
: `True` if the domain explicitly specified by the server began with a dot (`'.'`).

Cookies may have additional non-standard cookie-attributes. These may be accessed using the following methods:

Cookie.**has_nonstandard_attr**(*name*)
: Return `True` if cookie has the named cookie-attribute.

Cookie.**get_nonstandard_attr**(*name*, *default=None*)
: If cookie has the named cookie-attribute, return its value. Otherwise, return *default*.

Cookie.**set_nonstandard_attr**(*name*, *value*)
: Set the value of the named cookie-attribute.

The *Cookie* class also defines the following method:

Cookie.**is_expired**(*now=None*)
: `True` if cookie has passed the time at which the server requested it should expire. If *now* is given (in seconds since the epoch), return whether the cookie has expired at the specified time.

21.19.6 Examples

The first example shows the most common usage of *http.cookiejar*:

```
import http.cookiejar, urllib.request
cj = http.cookiejar.CookieJar()
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

This example illustrates how to open a URL using your Netscape, Mozilla, or Lynx cookies (assumes Unix/Netscape convention for location of the cookies file):

```
import os, http.cookiejar, urllib.request
cj = http.cookiejar.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "cookies.txt"))
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

The next example illustrates the use of *DefaultCookiePolicy*. Turn on **RFC 2965** cookies, be more strict about domains when setting and returning Netscape cookies, and block some domains from setting cookies or having them returned:

```
import urllib.request
from http.cookiejar import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=Policy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

21.20 xmlrpc — XMLRPC server and client modules

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data.

`xmlrpc` is a package that collects server and client modules implementing XML-RPC. The modules are:

- *xmlrpc.client*
- *xmlrpc.server*

21.21 xmlrpc.client — XML-RPC client access

Source code: Lib/xmlrpc/client.py

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP(S) as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. This module supports writing XML-RPC client code; it handles all the details of translating between conformable Python objects and XML on the wire.

> **Warning**
>
> The *xmlrpc.client* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML vulnerabilities*.

Changed in version 3.5: For HTTPS URIs, *xmlrpc.client* now performs all the necessary certificate and hostname checks by default.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

class xmlrpc.client.**ServerProxy**(*uri*, *transport=None*, *encoding=None*, *verbose=False*, *allow_none=False*, *use_datetime=False*, *use_builtin_types=False*, ***, *headers=()*, *context=None*)

A *ServerProxy* instance is an object that manages communication with a remote XML-RPC server. The required first argument is a URI (Uniform Resource Indicator), and will normally be the URL of the server. The optional second argument is a transport factory instance; by default it is an internal `SafeTransport` instance for

https: URLs and an internal HTTP `Transport` instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag.

The following parameters govern the use of the returned proxy instance. If *allow_none* is true, the Python constant `None` will be translated into XML; the default behaviour is for `None` to raise a *TypeError*. This is a commonly used extension to the XML-RPC specification, but isn't supported by all clients and servers; see http://ontosys.com/xml-rpc/extensions.php for a description. The *use_builtin_types* flag can be used to cause date/time values to be presented as *datetime.datetime* objects and binary data to be presented as *bytes* objects; this flag is false by default. *datetime.datetime*, *bytes* and *bytearray* objects may be passed to calls. The *headers* parameter is an optional sequence of HTTP headers to send with each request, expressed as a sequence of 2-tuples representing the header name and value. (e.g. `[('Header-Name', 'value')]`). The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

Changed in version 3.3: The *use_builtin_types* flag was added.

Changed in version 3.8: The *headers* parameter was added.

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication: `http://user:pass@host:port/path`. The `user:pass` portion will be base64-encoded as an HTTP 'Authorization' header, and sent to the remote server as part of the connection process when invoking an XML-RPC method. You only need to use this if the remote server requires a Basic Authentication user and password. If an HTTPS URL is provided, *context* may be *ssl.SSLContext* and configures the SSL settings of the underlying HTTPS connection.

The returned instance is a proxy object with methods that can be used to invoke corresponding RPC calls on the remote server. If the remote server supports the introspection API, the proxy can also be used to query the remote server for the methods it supports (service discovery) and fetch other server-associated metadata.

Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type):

XML-RPC type	Python type
`boolean`	*bool*
`int`, `i1`, `i2`, `i4`, `i8` or `biginteger`	*int* in range from -2147483648 to 2147483647. Values get the `<int>` tag.
`double` or `float`	*float*. Values get the `<double>` tag.
`string`	*str*
`array`	*list* or *tuple* containing conformable elements. Arrays are returned as *lists*.
`struct`	*dict*. Keys must be strings, values may be any conformable type. Objects of user-defined classes can be passed in; only their *__dict__* attribute is transmitted.
`dateTime.iso8601`	*DateTime* or *datetime.datetime*. Returned type depends on values of *use_builtin_types* and *use_datetime* flags.
`base64`	*Binary*, *bytes* or *bytearray*. Returned type depends on the value of the *use_builtin_types* flag.
`nil`	The `None` constant. Passing is allowed only if *allow_none* is true.
`bigdecimal`	*decimal.Decimal*. Returned type only.

This is the full set of data types supported by XML-RPC. Method calls may also raise a special *Fault* instance, used to signal XML-RPC server errors, or *ProtocolError* used to signal an error in the HTTP/HTTPS transport layer. Both *Fault* and *ProtocolError* derive from a base class called `Error`. Note that the xmlrpc client module currently does not marshal instances of subclasses of built-in types.

When passing strings, characters special to XML such as `<`, `>`, and `&` will be automatically escaped. However, it's the caller's responsibility to ensure that the string is free of characters that aren't allowed in XML, such as the control characters with ASCII values between 0 and 31 (except, of course, tab, newline and carriage return); failing to do this will result

in an XML-RPC request that isn't well-formed XML. If you have to pass arbitrary bytes via XML-RPC, use *bytes* or *bytearray* classes or the *Binary* wrapper class described below.

Server is retained as an alias for *ServerProxy* for backwards compatibility. New code should use *ServerProxy*.

Changed in version 3.5: Added the *context* argument.

Changed in version 3.6: Added support of type tags with prefixes (e.g. ex:nil). Added support of unmarshalling additional types used by Apache XML-RPC implementation for numerics: i1, i2, i8, biginteger, float and bigdecimal. See https://ws.apache.org/xmlrpc/types.html for a description.

See also

XML-RPC HOWTO
A good description of XML-RPC operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.

XML-RPC Introspection
Describes the XML-RPC protocol extension for introspection.

XML-RPC Specification
The official specification.

21.21.1 ServerProxy Objects

A *ServerProxy* instance has a method corresponding to each remote procedure call accepted by the XML-RPC server. Calling the method performs an RPC, dispatched by both name and argument signature (e.g. the same method name can be overloaded with multiple argument signatures). The RPC finishes by returning a value, which may be either returned data in a conformant type or a *Fault* or *ProtocolError* object indicating an error.

Servers that support the XML introspection API support some common methods grouped under the reserved system attribute:

ServerProxy.system.**listMethods**()
This method returns a list of strings, one for each (non-system) method supported by the XML-RPC server.

ServerProxy.system.**methodSignature**(*name*)
This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

Because multiple signatures (ie. overloading) is permitted, this method returns a list of signatures rather than a singleton.

Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply "string, array". If it expects three integers and returns a string, its signature is "string, int, int, int".

If no signature is defined for the method, a non-array value is returned. In Python this means that the type of the returned value will be something other than list.

ServerProxy.system.**methodHelp**(*name*)
This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned. The documentation string may contain HTML markup.

Changed in version 3.5: Instances of *ServerProxy* support the *context manager* protocol for closing the underlying transport.

A working example follows. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(is_even, "is_even")
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:
    print("3 is even: %s" % str(proxy.is_even(3)))
    print("100 is even: %s" % str(proxy.is_even(100)))
```

21.21.2 DateTime Objects

class xmlrpc.client.**DateTime**

This class may be initialized with seconds since the epoch, a time tuple, an ISO 8601 time/date string, or a *datetime.datetime* instance. It has the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(*string*)

Accept a string as the instance's new time value.

encode(*out*)

Write the XML-RPC encoding of this *DateTime* item to the *out* stream object.

It also supports certain of Python's built-in operators through rich comparison and __repr__() methods.

A working example follows. The server code:

```
import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
    return xmlrpc.client.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(today, "today")
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client
import datetime

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
```

(continues on next page)

(continued from previous page)

```
today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print("Today: %s" % converted.strftime("%d.%m.%Y, %H:%M"))
```

21.21.3 Binary Objects

class xmlrpc.client.**Binary**

This class may be initialized from bytes data (which may include NULs). The primary access to the content of a *Binary* object is provided by an attribute:

data

The binary data encapsulated by the *Binary* instance. The data is provided as a *bytes* object.

Binary objects have the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(*bytes*)

Accept a base64 *bytes* object and decode it as the instance's new data.

encode(*out*)

Write the XML-RPC base 64 encoding of this binary item to the *out* stream object.

The encoded data will have newlines every 76 characters as per **RFC 2045 section 6.8**, which was the de facto standard base64 specification when the XML-RPC spec was written.

It also supports certain of Python's built-in operators through `__eq__()` and `__ne__()` methods.

Example usage of the binary objects. We're going to transfer an image over XMLRPC:

```
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpc.client.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(python_logo, 'python_logo')

server.serve_forever()
```

The client gets the image and saves it to a file:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

21.21.4 Fault Objects

class xmlrpc.client.**Fault**

A *Fault* object encapsulates the content of an XML-RPC fault tag. Fault objects have the following attributes:

faultCode

An int indicating the fault type.

faultString

A string containing a diagnostic message associated with the fault.

In the following example we're going to intentionally cause a *Fault* by returning a complex type object. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

# A marshalling error is going to occur because we're returning a
# complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(add, 'add')

server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)
```

21.21.5 ProtocolError Objects

class xmlrpc.client.**ProtocolError**

A *ProtocolError* object describes a protocol error in the underlying transport layer (such as a 404 'not found' error if the server named by the URI does not exist). It has the following attributes:

url

The URI or URL that triggered the error.

errcode

The error code.

errmsg

The error message or diagnostic string.

headers

A dict containing the headers of the HTTP/HTTPS request that triggered the error.

In the following example we're going to intentionally cause a *ProtocolError* by providing an invalid URI:

```
import xmlrpc.client

# create a ServerProxy with a URI that doesn't respond to XMLRPC requests
proxy = xmlrpc.client.ServerProxy("http://google.com/")

try:
    proxy.some_method()
except xmlrpc.client.ProtocolError as err:
    print("A protocol error occurred")
    print("URL: %s" % err.url)
    print("HTTP/HTTPS headers: %s" % err.headers)
    print("Error code: %d" % err.errcode)
    print("Error message: %s" % err.errmsg)
```

21.21.6 MultiCall Objects

The *MultiCall* object provides a way to encapsulate multiple calls to a remote server into a single request[1].

class xmlrpc.client.**MultiCall**(*server*)

Create an object used to boxcar method calls. *server* is the eventual target of the call. Calls can be made to the result object, but they will immediately return `None`, and only store the call name and parameters in the *MultiCall* object. Calling the object itself causes all stored calls to be transmitted as a single `system.multicall` request. The result of this call is a *generator*; iterating over this generator yields the individual results.

A usage example of this class follows. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x // y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

[1] This approach has been first presented in a discussion on xmlrpc.com.

The client code for the preceding server:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
multicall = xmlrpc.client.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
multicall.divide(7, 3)
result = multicall()

print("7+3=%d, 7-3=%d, 7*3=%d, 7//3=%d" % tuple(result))
```

21.21.7 Convenience Functions

xmlrpc.client.**dumps**(*params*, *methodname=None*, *methodresponse=None*, *encoding=None*, *allow_none=False*)

Convert *params* into an XML-RPC request. or into a response if *methodresponse* is true. *params* can be either a tuple of arguments or an instance of the `Fault` exception class. If *methodresponse* is true, only a single value can be returned, meaning that *params* must be of length 1. *encoding*, if supplied, is the encoding to use in the generated XML; the default is UTF-8. Python's `None` value cannot be used in standard XML-RPC; to allow using it via an extension, provide a true value for *allow_none*.

xmlrpc.client.**loads**(*data*, *use_datetime=False*, *use_builtin_types=False*)

Convert an XML-RPC request or response into Python objects, a `(params, methodname)`. *params* is a tuple of argument; *methodname* is a string, or `None` if no method name is present in the packet. If the XML-RPC packet represents a fault condition, this function will raise a `Fault` exception. The *use_builtin_types* flag can be used to cause date/time values to be presented as `datetime.datetime` objects and binary data to be presented as `bytes` objects; this flag is false by default.

The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

Changed in version 3.3: The *use_builtin_types* flag was added.

21.21.8 Example of Client Usage

```
# simple test program (from the XML-RPC specification)
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # local server
with ServerProxy("http://betty.userland.com") as proxy:

    print(proxy)

    try:
        print(proxy.examples.getStateName(41))
    except Error as v:
        print("ERROR", v)
```

To access an XML-RPC server through a HTTP proxy, you need to define a custom transport. The following example shows how:

```
import http.client
import xmlrpc.client

class ProxiedTransport(xmlrpc.client.Transport):

    def set_proxy(self, host, port=None, headers=None):
        self.proxy = host, port
        self.proxy_headers = headers

    def make_connection(self, host):
        connection = http.client.HTTPConnection(*self.proxy)
        connection.set_tunnel(host, headers=self.proxy_headers)
        self._connection = host, connection
        return connection

transport = ProxiedTransport()
transport.set_proxy('proxy-server', 8080)
server = xmlrpc.client.ServerProxy('http://betty.userland.com', transport=transport)
print(server.examples.getStateName(41))
```

21.21.9 Example of Client and Server Usage

See *SimpleXMLRPCServer Example*.

21.22 `xmlrpc.server` — Basic XML-RPC servers

Source code: Lib/xmlrpc/server.py

The `xmlrpc.server` module provides a basic server framework for XML-RPC servers written in Python. Servers can either be free standing, using `SimpleXMLRPCServer`, or embedded in a CGI environment, using `CGIXMLRPCRequestHandler`.

> **Warning**
>
> The `xmlrpc.server` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML vulnerabilities*.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

class xmlrpc.server.**SimpleXMLRPCServer**(*addr*, *requestHandler=SimpleXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=False*)

Create a new server instance. This class provides methods for registration of functions that can be called by the XML-RPC protocol. The *requestHandler* parameter should be a factory for request handler instances; it defaults to `SimpleXMLRPCRequestHandler`. The *addr* and *requestHandler* parameters are passed to the `socketserver.TCPServer` constructor. If *logRequests* is true (the default), requests will be logged; setting this parameter to false will turn off logging. The *allow_none* and *encoding* parameters are passed on to `xmlrpc.client` and control the XML-RPC responses that will be returned from the server. The *bind_and_activate* parameter controls whether `server_bind()` and `server_activate()` are called immediately by the constructor;

it defaults to true. Setting it to false allows code to manipulate the *allow_reuse_address* class variable before the address is bound. The *use_builtin_types* parameter is passed to the `loads()` function and controls which types are processed when date/times values or binary data are received; it defaults to false.

Changed in version 3.3: The *use_builtin_types* flag was added.

class xmlrpc.server.**CGIXMLRPCRequestHandler**(*allow_none=False*, *encoding=None*, *use_builtin_types=False*)

Create a new instance to handle XML-RPC requests in a CGI environment. The *allow_none* and *encoding* parameters are passed on to `xmlrpc.client` and control the XML-RPC responses that will be returned from the server. The *use_builtin_types* parameter is passed to the `loads()` function and controls which types are processed when date/times values or binary data are received; it defaults to false.

Changed in version 3.3: The *use_builtin_types* flag was added.

class xmlrpc.server.**SimpleXMLRPCRequestHandler**

Create a new request handler instance. This request handler supports `POST` requests and modifies logging so that the *logRequests* parameter to the `SimpleXMLRPCServer` constructor parameter is honored.

21.22.1 SimpleXMLRPCServer Objects

The `SimpleXMLRPCServer` class is based on `socketserver.TCPServer` and provides a means of creating simple, stand alone XML-RPC servers.

SimpleXMLRPCServer.**register_function**(*function=None*, *name=None*)

Register a function that can respond to XML-RPC requests. If *name* is given, it will be the method name associated with *function*, otherwise `function.__name__` will be used. *name* is a string, and may contain characters not legal in Python identifiers, including the period character.

This method can also be used as a decorator. When used as a decorator, *name* can only be given as a keyword argument to register *function* under *name*. If no *name* is given, `function.__name__` will be used.

Changed in version 3.7: `register_function()` can be used as a decorator.

SimpleXMLRPCServer.**register_instance**(*instance*, *allow_dotted_names=False*)

Register an object which is used to expose method names which have not been registered using `register_function()`. If *instance* contains a `_dispatch()` method, it is called with the requested method name and the parameters from the request. Its API is `def _dispatch(self, method, params)` (note that *params* does not represent a variable argument list). If it calls an underlying function to perform its task, that function is called as `func(*params)`, expanding the parameter list. The return value from `_dispatch()` is returned to the client as the result. If *instance* does not have a `_dispatch()` method, it is searched for an attribute matching the name of the requested method.

If the optional *allow_dotted_names* argument is true and the instance does not have a `_dispatch()` method, then if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

> **Warning**
>
> Enabling the *allow_dotted_names* option allows intruders to access your module's global variables and may allow intruders to execute arbitrary code on your machine. Only use this option on a secure, closed network.

SimpleXMLRPCServer.**register_introspection_functions**()

Registers the XML-RPC introspection functions `system.listMethods`, `system.methodHelp` and `system.methodSignature`.

SimpleXMLRPCServer.**register_multicall_functions**()

Registers the XML-RPC multicall function system.multicall.

SimpleXMLRPCRequestHandler.**rpc_paths**

An attribute value that must be a tuple listing valid path portions of the URL for receiving XML-RPC requests. Requests posted to other paths will result in a 404 "no such page" HTTP error. If this tuple is empty, all paths will be considered valid. The default value is `('/', '/RPC2')`.

SimpleXMLRPCServer Example

Server code:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name
    def adder_function(x, y):
        return x + y
    server.register_function(adder_function, 'add')

    # Register an instance; all the methods of the instance are
    # published as XML-RPC methods (in this case, just 'mul').
    class MyFuncs:
        def mul(self, x, y):
            return x * y

    server.register_instance(MyFuncs())

    # Run the server's main loop
    server.serve_forever()
```

The following client code will call the methods made available by the preceding server:

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2,3))  # Returns 2**3 = 8
print(s.add(2,3))  # Returns 5
print(s.mul(5,2))  # Returns 5*2 = 10
```

(continues on next page)

(continued from previous page)

```
# Print list of available methods
print(s.system.listMethods())
```

`register_function()` can also be used as a decorator. The previous server example can register functions in a decorator way:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name, using
    # register_function as a decorator. *name* can only be given
    # as a keyword argument.
    @server.register_function(name='add')
    def adder_function(x, y):
        return x + y

    # Register a function under function.__name__.
    @server.register_function
    def mul(x, y):
        return x * y

    server.serve_forever()
```

The following example included in the `Lib/xmlrpc/server.py` module shows a server allowing dotted names and registering a multicall function.

Warning

Enabling the *allow_dotted_names* option allows intruders to access your module's global variables and may allow intruders to execute arbitrary code on your machine. Only use this example only within a secure, closed network.

```
import datetime

class ExampleService:
    def getData(self):
        return '42'

    class currentTime:
        @staticmethod
        def getCurrentTime():
            return datetime.datetime.now()
```

(continues on next page)

(continued from previous page)

```
with SimpleXMLRPCServer(("localhost", 8000)) as server:
    server.register_function(pow)
    server.register_function(lambda x,y: x+y, 'add')
    server.register_instance(ExampleService(), allow_dotted_names=True)
    server.register_multicall_functions()
    print('Serving XML-RPC on localhost port 8000')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("\nKeyboard interrupt received, exiting.")
        sys.exit(0)
```

This ExampleService demo can be invoked from the command line:

```
python -m xmlrpc.server
```

The client that interacts with the above server is included in `Lib/xmlrpc/client.py`:

```
server = ServerProxy("http://localhost:8000")

try:
    print(server.currentTime.getCurrentTime())
except Error as v:
    print("ERROR", v)

multi = MultiCall(server)
multi.getData()
multi.pow(2,9)
multi.add(1,2)
try:
    for response in multi():
        print(response)
except Error as v:
    print("ERROR", v)
```

This client which interacts with the demo XMLRPC server can be invoked as:

```
python -m xmlrpc.client
```

21.22.2 CGIXMLRPCRequestHandler

The *CGIXMLRPCRequestHandler* class can be used to handle XML-RPC requests sent to Python CGI scripts.

CGIXMLRPCRequestHandler.**register_function**(*function=None*, *name=None*)

Register a function that can respond to XML-RPC requests. If *name* is given, it will be the method name associated with *function*, otherwise `function.__name__` will be used. *name* is a string, and may contain characters not legal in Python identifiers, including the period character.

This method can also be used as a decorator. When used as a decorator, *name* can only be given as a keyword argument to register *function* under *name*. If no *name* is given, `function.__name__` will be used.

Changed in version 3.7: *register_function()* can be used as a decorator.

CGIXMLRPCRequestHandler.**register_instance**(*instance*)

Register an object which is used to expose method names which have not been registered using *register_function()*. If instance contains a `_dispatch()` method, it is called with the requested method name

and the parameters from the request; the return value is returned to the client as the result. If instance does not have a `_dispatch()` method, it is searched for an attribute matching the name of the requested method; if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

`CGIXMLRPCRequestHandler.`**`register_introspection_functions`**`()`

Register the XML-RPC introspection functions `system.listMethods`, `system.methodHelp` and `system.methodSignature`.

`CGIXMLRPCRequestHandler.`**`register_multicall_functions`**`()`

Register the XML-RPC multicall function `system.multicall`.

`CGIXMLRPCRequestHandler.`**`handle_request`**`(`*request_text=None*`)`

Handle an XML-RPC request. If *request_text* is given, it should be the POST data provided by the HTTP server, otherwise the contents of stdin will be used.

Example:

```
class MyFuncs:
    def mul(self, x, y):
        return x * y


handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```

21.22.3 Documenting XMLRPC server

These classes extend the above classes to serve HTML documentation in response to HTTP GET requests. Servers can either be free standing, using *DocXMLRPCServer*, or embedded in a CGI environment, using *DocCGIXMLRPCRequestHandler*.

class `xmlrpc.server.`**`DocXMLRPCServer`**`(`*addr*, *requestHandler=DocXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=True*`)`

Create a new server instance. All parameters have the same meaning as for *SimpleXMLRPCServer*; *requestHandler* defaults to *DocXMLRPCRequestHandler*.

Changed in version 3.3: The *use_builtin_types* flag was added.

class `xmlrpc.server.`**`DocCGIXMLRPCRequestHandler`**

Create a new instance to handle XML-RPC requests in a CGI environment.

class `xmlrpc.server.`**`DocXMLRPCRequestHandler`**

Create a new request handler instance. This request handler supports XML-RPC POST requests, documentation GET requests, and modifies logging so that the *logRequests* parameter to the *DocXMLRPCServer* constructor parameter is honored.

21.22.4 DocXMLRPCServer Objects

The *DocXMLRPCServer* class is derived from *SimpleXMLRPCServer* and provides a means of creating self-documenting, stand alone XML-RPC servers. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocXMLRPCServer.**set_server_title**(*server_title*)

Set the title used in the generated HTML documentation. This title will be used inside the HTML "title" element.

DocXMLRPCServer.**set_server_name**(*server_name*)

Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a "h1" element.

DocXMLRPCServer.**set_server_documentation**(*server_documentation*)

Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

21.22.5 DocCGIXMLRPCRequestHandler

The *DocCGIXMLRPCRequestHandler* class is derived from *CGIXMLRPCRequestHandler* and provides a means of creating self-documenting, XML-RPC CGI scripts. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocCGIXMLRPCRequestHandler.**set_server_title**(*server_title*)

Set the title used in the generated HTML documentation. This title will be used inside the HTML "title" element.

DocCGIXMLRPCRequestHandler.**set_server_name**(*server_name*)

Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a "h1" element.

DocCGIXMLRPCRequestHandler.**set_server_documentation**(*server_documentation*)

Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

21.23 ipaddress — IPv4/IPv6 manipulation library

Source code: Lib/ipaddress.py

---

ipaddress provides the capabilities to create, manipulate and operate on IPv4 and IPv6 addresses and networks.

The functions and classes in this module make it straightforward to handle various tasks related to IP addresses, including checking whether or not two hosts are on the same subnet, iterating over all hosts in a particular subnet, checking whether or not a string represents a valid IP address or network definition, and so on.

This is the full module API reference—for an overview and introduction, see ipaddress-howto.

Added in version 3.3.

21.23.1 Convenience factory functions

The *ipaddress* module provides factory functions to conveniently create IP addresses, networks and interfaces:

ipaddress.**ip_address**(*address*)

Return an *IPv4Address* or *IPv6Address* object depending on the IP address passed as argument. Either IPv4 or IPv6 addresses may be supplied; integers less than `2**32` will be considered to be IPv4 by default. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address.

```
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:db8::')
IPv6Address('2001:db8::')
```

ipaddress.**ip_network**(*address*, *strict=True*)

Return an *IPv4Network* or *IPv6Network* object depending on the IP address passed as argument. *address* is a string or integer representing the IP network. Either IPv4 or IPv6 networks may be supplied; integers less than `2**32` will be considered to be IPv4 by default. *strict* is passed to *IPv4Network* or *IPv6Network* constructor. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address, or if the network has host bits set.

```
>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')
```

ipaddress.**ip_interface**(*address*)

Return an *IPv4Interface* or *IPv6Interface* object depending on the IP address passed as argument. *address* is a string or integer representing the IP address. Either IPv4 or IPv6 addresses may be supplied; integers less than `2**32` will be considered to be IPv4 by default. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address.

One downside of these convenience functions is that the need to handle both IPv4 and IPv6 formats means that error messages provide minimal information on the precise error, as the functions don't know whether the IPv4 or IPv6 format was intended. More detailed error reporting can be obtained by calling the appropriate version specific class constructors directly.

21.23.2 IP Addresses

Address objects

The *IPv4Address* and *IPv6Address* objects share a lot of common attributes. Some attributes that are only meaningful for IPv6 addresses are also implemented by *IPv4Address* objects, in order to make it easier to write code that handles both IP versions correctly. Address objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.**IPv4Address**(*address*)

Construct an IPv4 address. An *AddressValueError* is raised if *address* is not a valid IPv4 address.

The following constitutes a valid IPv4 address:

1. A string in decimal-dot notation, consisting of four decimal integers in the inclusive range 0–255, separated by dots (e.g. `192.168.0.1`). Each integer represents an octet (byte) in the address. Leading zeroes are not tolerated to prevent confusion with octal notation.
2. An integer that fits into 32 bits.
3. An integer packed into a *bytes* object of length 4 (most significant octet first).

```
>>> ipaddress.IPv4Address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(3232235521)
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(b'\xC0\xA8\x00\x01')
IPv4Address('192.168.0.1')
```

Changed in version 3.8: Leading zeros are tolerated, even in ambiguous cases that look like octal notation.

Changed in version 3.9.5: Leading zeros are no longer tolerated and are treated as an error. IPv4 address strings are now parsed as strict as glibc *inet_pton()*.

version

The appropriate version number: `4` for IPv4, `6` for IPv6.

max_prefixlen

The total number of bits in the address representation for this version: `32` for IPv4, `128` for IPv6.

The prefix defines the number of leading bits in an address that are compared to determine whether or not an address is part of a network.

compressed

exploded

The string representation in dotted decimal notation. Leading zeroes are never included in the representation.

As IPv4 does not define a shorthand notation for addresses with octets set to zero, these two attributes are always the same as `str(addr)` for IPv4 addresses. Exposing these attributes makes it easier to write display code that can handle both IPv4 and IPv6 addresses.

packed

The binary representation of this address - a *bytes* object of the appropriate length (most significant octet first). This is 4 bytes for IPv4 and 16 bytes for IPv6.

reverse_pointer

The name of the reverse DNS PTR record for the IP address, e.g.:

```
>>> ipaddress.ip_address("127.0.0.1").reverse_pointer
'1.0.0.127.in-addr.arpa'
>>> ipaddress.ip_address("2001:db8::1").reverse_pointer
'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa'
```

This is the name that could be used for performing a PTR lookup, not the resolved hostname itself.

Added in version 3.5.

is_multicast

`True` if the address is reserved for multicast use. See **RFC 3171** (for IPv4) or **RFC 2373** (for IPv6).

is_private

`True` if the address is defined as not globally reachable by iana-ipv4-special-registry (for IPv4) or iana-ipv6-special-registry (for IPv6) with the following exceptions:

- `is_private` is `False` for the shared address space (`100.64.0.0/10`)
- For IPv4-mapped IPv6-addresses the `is_private` value is determined by the semantics of the underlying IPv4 addresses and the following condition holds (see *IPv6Address.ipv4_mapped*):

```
address.is_private == address.ipv4_mapped.is_private
```

`is_private` has value opposite to *`is_global`*, except for the shared address space (`100.64.0.0/10` range) where they are both `False`.

Changed in version 3.13: Fixed some false positives and false negatives.

- `192.0.0.0/24` is considered private with the exception of `192.0.0.9/32` and `192.0.0.10/32` (previously: only the `192.0.0.0/29` sub-range was considered private).
- `64:ff9b:1::/48` is considered private.
- `2002::/16` is considered private.
- There are exceptions within `2001::/23` (otherwise considered private): `2001:1::1/128`, `2001:1::2/128`, `2001:3::/32`, `2001:4:112::/48`, `2001:20::/28`, `2001:30::/28`. The exceptions are not considered private.

is_global

`True` if the address is defined as globally reachable by iana-ipv4-special-registry (for IPv4) or iana-ipv6-special-registry (for IPv6) with the following exception:

For IPv4-mapped IPv6-addresses the `is_private` value is determined by the semantics of the underlying IPv4 addresses and the following condition holds (see *`IPv6Address.ipv4_mapped`*):

```
address.is_global == address.ipv4_mapped.is_global
```

`is_global` has value opposite to *`is_private`*, except for the shared address space (`100.64.0.0/10` range) where they are both `False`.

Added in version 3.4.

Changed in version 3.13: Fixed some false positives and false negatives, see *`is_private`* for details.

is_unspecified

`True` if the address is unspecified. See **RFC 5735** (for IPv4) or **RFC 2373** (for IPv6).

is_reserved

`True` if the address is otherwise IETF reserved.

is_loopback

`True` if this is a loopback address. See **RFC 3330** (for IPv4) or **RFC 2373** (for IPv6).

is_link_local

`True` if the address is reserved for link-local usage. See **RFC 3927**.

ipv6_mapped

`IPv4Address` object representing the IPv4-mapped IPv6 address. See **RFC 4291**.

Added in version 3.13.

`IPv4Address.`**`__format__`**(*fmt*)

Returns a string representation of the IP address, controlled by an explicit format string. *fmt* can be one of the following: `'s'`, the default option, equivalent to *`str()`*, `'b'` for a zero-padded binary string, `'X'` or `'x'` for an uppercase or lowercase hexadecimal representation, or `'n'`, which is equivalent to `'b'` for IPv4 addresses and `'x'` for IPv6. For binary and hexadecimal representations, the form specifier `'#'` and the grouping option `'_'` are available. `__format__` is used by `format`, `str.format` and f-strings.

```
>>> format(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> '{:#b}'.format(ipaddress.IPv4Address('192.168.0.1'))
'0b11000000101010000000000000000001'
```

(continues on next page)

(continued from previous page)

```
>>> f'{ipaddress.IPv6Address("2001:db8::1000"):s}'
'2001:db8::1000'
>>> format(ipaddress.IPv6Address('2001:db8::1000'), '_X')
'2001_0DB8_0000_0000_0000_0000_0000_1000'
>>> '{:#_n}'.format(ipaddress.IPv6Address('2001:db8::1000'))
'0x2001_0db8_0000_0000_0000_0000_0000_1000'
```

Added in version 3.9.

class ipaddress.**IPv6Address**(*address*)

Construct an IPv6 address. An *AddressValueError* is raised if *address* is not a valid IPv6 address.

The following constitutes a valid IPv6 address:

1. A string consisting of eight groups of four hexadecimal digits, each group representing 16 bits. The groups are separated by colons. This describes an *exploded* (longhand) notation. The string can also be *compressed* (shorthand notation) by various means. See **RFC 4291** for details. For example, `"0000:0000:0000:0000:0000:0abc:0007:0def"` can be compressed to `"::abc:7:def"`.

   Optionally, the string may also have a scope zone ID, expressed with a suffix `%scope_id`. If present, the scope ID must be non-empty, and may not contain `%`. See **RFC 4007** for details. For example, `fe80::1234%1` might identify address `fe80::1234` on the first link of the node.

2. An integer that fits into 128 bits.

3. An integer packed into a *bytes* object of length 16, big-endian.

```
>>> ipaddress.IPv6Address('2001:db8::1000')
IPv6Address('2001:db8::1000')
>>> ipaddress.IPv6Address('ff02::5678%1')
IPv6Address('ff02::5678%1')
```

compressed

The short form of the address representation, with leading zeroes in groups omitted and the longest sequence of groups consisting entirely of zeroes collapsed to a single empty group.

This is also the value returned by `str(addr)` for IPv6 addresses.

exploded

The long form of the address representation, with all leading zeroes and groups consisting entirely of zeroes included.

For the following attributes and methods, see the corresponding documentation of the *IPv4Address* class:

packed

reverse_pointer

version

max_prefixlen

is_multicast

is_private

is_global

Added in version 3.4.

is_unspecified

is_reserved

is_loopback

is_link_local

is_site_local

`True` if the address is reserved for site-local usage. Note that the site-local address space has been deprecated by **RFC 3879**. Use *`is_private`* to test if this address is in the space of unique local addresses as defined by **RFC 4193**.

ipv4_mapped

For addresses that appear to be IPv4 mapped addresses (starting with `::FFFF/96`), this property will report the embedded IPv4 address. For any other address, this property will be `None`.

scope_id

For scoped addresses as defined by **RFC 4007**, this property identifies the particular zone of the address's scope that the address belongs to, as a string. When no scope zone is specified, this property will be `None`.

sixtofour

For addresses that appear to be 6to4 addresses (starting with `2002::/16`) as defined by **RFC 3056**, this property will report the embedded IPv4 address. For any other address, this property will be `None`.

teredo

For addresses that appear to be Teredo addresses (starting with `2001::/32`) as defined by **RFC 4380**, this property will report the embedded `(server, client)` IP address pair. For any other address, this property will be `None`.

`IPv6Address.`**`__format__`**(*fmt*)

Refer to the corresponding method documentation in *`IPv4Address`*.

Added in version 3.9.

Conversion to Strings and Integers

To interoperate with networking interfaces such as the socket module, addresses must be converted to strings or integers. This is handled using the *`str()`* and *`int()`* builtin functions:

```
>>> str(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> int(ipaddress.IPv4Address('192.168.0.1'))
3232235521
>>> str(ipaddress.IPv6Address('::1'))
'::1'
>>> int(ipaddress.IPv6Address('::1'))
1
```

Note that IPv6 scoped addresses are converted to integers without scope zone ID.

Operators

Address objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Comparison operators

Address objects can be compared with the usual set of comparison operators. Same IPv6 addresses with different scope zone IDs are not equal. Some examples:

```
>>> IPv4Address('127.0.0.2') > IPv4Address('127.0.0.1')
True
>>> IPv4Address('127.0.0.2') == IPv4Address('127.0.0.1')
False
>>> IPv4Address('127.0.0.2') != IPv4Address('127.0.0.1')
True
>>> IPv6Address('fe80::1234') == IPv6Address('fe80::1234%1')
False
>>> IPv6Address('fe80::1234%1') != IPv6Address('fe80::1234%2')
True
```

Arithmetic operators

Integers can be added to or subtracted from address objects. Some examples:

```
>>> IPv4Address('127.0.0.2') + 3
IPv4Address('127.0.0.5')
>>> IPv4Address('127.0.0.2') - 3
IPv4Address('126.255.255.255')
>>> IPv4Address('255.255.255.255') + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ipaddress.AddressValueError: 4294967296 (>= 2**32) is not permitted as an IPv4 address
```

21.23.3 IP Network definitions

The *IPv4Network* and *IPv6Network* objects provide a mechanism for defining and inspecting IP network definitions. A network definition consists of a *mask* and a *network address*, and as such defines a range of IP addresses that equal the network address when masked (binary AND) with the mask. For example, a network definition with the mask `255.255.255.0` and the network address `192.168.1.0` consists of IP addresses in the inclusive range `192.168.1.0` to `192.168.1.255`.

Prefix, net mask and host mask

There are several equivalent ways to specify IP network masks. A *prefix* `/<nbits>` is a notation that denotes how many high-order bits are set in the network mask. A *net mask* is an IP address with some number of high-order bits set. Thus the prefix `/24` is equivalent to the net mask `255.255.255.0` in IPv4, or `ffff:ff00::` in IPv6. In addition, a *host mask* is the logical inverse of a *net mask*, and is sometimes used (for example in Cisco access control lists) to denote a network mask. The host mask equivalent to `/24` in IPv4 is `0.0.0.255`.

Network objects

All attributes implemented by address objects are implemented by network objects as well. In addition, network objects implement additional attributes. All of these are common between *IPv4Network* and *IPv6Network*, so to avoid duplication they are only documented for *IPv4Network*. Network objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.**IPv4Network**(*address*, *strict=True*)

Construct an IPv4 network definition. *address* can be one of the following:

1. A string consisting of an IP address and an optional mask, separated by a slash (`/`). The IP address is the network address, and the mask can be either a single number, which means it's a *prefix*, or a string representation of an IPv4 address. If it's the latter, the mask is interpreted as a *net mask* if it starts with a non-zero field, or as a *host mask* if it starts with a zero field, with the single exception of an all-zero mask which is treated as a *net mask*. If no mask is provided, it's considered to be `/32`.

   For example, the following *address* specifications are equivalent: `192.168.1.0/24`, `192.168.1.0/255.255.255.0` and `192.168.1.0/0.0.0.255`.

2. An integer that fits into 32 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being `/32`.

3. An integer packed into a *bytes* object of length 4, big-endian. The interpretation is similar to an integer *address*.

4. A two-tuple of an address description and a netmask, where the address description is either a string, a 32-bits integer, a 4-bytes packed integer, or an existing IPv4Address object; and the netmask is either an integer representing the prefix length (e.g. `24`) or a string representing the prefix mask (e.g. `255.255.255.0`).

An *AddressValueError* is raised if *address* is not a valid IPv4 address. A *NetmaskValueError* is raised if the mask is not valid for an IPv4 address.

If *strict* is `True` and host bits are set in the supplied address, then *ValueError* is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

Unless stated otherwise, all network methods accepting other network/address objects will raise *TypeError* if the argument's IP version is incompatible to `self`.

Changed in version 3.5: Added the two-tuple form for the *address* constructor parameter.

version

max_prefixlen

Refer to the corresponding attribute documentation in *IPv4Address*.

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

These attributes are true for the network as a whole if they are true for both the network address and the broadcast address.

network_address

The network address for the network. The network address and the prefix length together uniquely define a network.

broadcast_address

The broadcast address for the network. Packets sent to the broadcast address should be received by every host on the network.

hostmask

The host mask, as an *IPv4Address* object.

netmask

The net mask, as an *IPv4Address* object.

with_prefixlen

compressed

exploded

A string representation of the network, with the mask in prefix notation.

`with_prefixlen` and `compressed` are always the same as `str(network)`. `exploded` uses the exploded form the network address.

with_netmask

A string representation of the network, with the mask in net mask notation.

with_hostmask

A string representation of the network, with the mask in host mask notation.

num_addresses

The total number of addresses in the network.

prefixlen

Length of the network prefix, in bits.

hosts()

Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the network address itself and the network broadcast address. For networks with a mask length of 31, the network address and network broadcast address are also included in the result. Networks with a mask of 32 will return a list containing the single host address.

```
>>> list(ip_network('192.0.2.0/29').hosts())
[IPv4Address('192.0.2.1'), IPv4Address('192.0.2.2'),
 IPv4Address('192.0.2.3'), IPv4Address('192.0.2.4'),
 IPv4Address('192.0.2.5'), IPv4Address('192.0.2.6')]
>>> list(ip_network('192.0.2.0/31').hosts())
[IPv4Address('192.0.2.0'), IPv4Address('192.0.2.1')]
>>> list(ip_network('192.0.2.1/32').hosts())
[IPv4Address('192.0.2.1')]
```

overlaps(*other*)

`True` if this network is partly or wholly contained in *other* or *other* is wholly contained in this network.

address_exclude(*network*)

Computes the network definitions resulting from removing the given *network* from this one. Returns an iterator of network objects. Raises `ValueError` if *network* is not completely contained in this network.

```
>>> n1 = ip_network('192.0.2.0/28')
>>> n2 = ip_network('192.0.2.1/32')
>>> list(n1.address_exclude(n2))
[IPv4Network('192.0.2.8/29'), IPv4Network('192.0.2.4/30'),
 IPv4Network('192.0.2.2/31'), IPv4Network('192.0.2.0/32')]
```

subnets(*prefixlen_diff=1*, *new_prefix=None*)

The subnets that join to make the current network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be increased by. *new_prefix* is the desired new prefix of the subnets; it must be larger than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns an iterator of network objects.

```
>>> list(ip_network('192.0.2.0/24').subnets())
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
>>> list(ip_network('192.0.2.0/24').subnets(prefixlen_diff=2))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=26))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=23))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise ValueError('new prefix must be longer')
ValueError: new prefix must be longer
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=25))
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
```

supernet(*prefixlen_diff=1*, *new_prefix=None*)

The supernet containing this network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be decreased by. *new_prefix* is the desired new prefix of the supernet; it must be smaller than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns a single network object.

```
>>> ip_network('192.0.2.0/24').supernet()
IPv4Network('192.0.2.0/23')
>>> ip_network('192.0.2.0/24').supernet(prefixlen_diff=2)
IPv4Network('192.0.0.0/22')
>>> ip_network('192.0.2.0/24').supernet(new_prefix=20)
IPv4Network('192.0.0.0/20')
```

subnet_of(*other*)

Return `True` if this network is a subnet of *other*.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> b.subnet_of(a)
True
```

Added in version 3.7.

supernet_of(*other*)

Return `True` if this network is a supernet of *other*.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> a.supernet_of(b)
True
```

Added in version 3.7.

compare_networks(*other*)

Compare this network to *other*. In this comparison only the network addresses are considered; host bits aren't. Returns either `-1`, `0` or `1`.

```
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.2/32'))
-1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.0/32'))
1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.1/32'))
0
```

Deprecated since version 3.7: It uses the same ordering and comparison algorithm as "<", "==", and ">"

class ipaddress.**IPv6Network**(*address*, *strict=True*)

Construct an IPv6 network definition. *address* can be one of the following:

1. A string consisting of an IP address and an optional prefix length, separated by a slash (`/`). The IP address is the network address, and the prefix length must be a single number, the *prefix*. If no prefix length is provided, it's considered to be `/128`.

   Note that currently expanded netmasks are not supported. That means `2001:db00::0/24` is a valid argument while `2001:db00::0/ffff:ff00::` is not.

2. An integer that fits into 128 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being `/128`.

3. An integer packed into a *bytes* object of length 16, big-endian. The interpretation is similar to an integer *address*.

4. A two-tuple of an address description and a netmask, where the address description is either a string, a 128-bits integer, a 16-bytes packed integer, or an existing IPv6Address object; and the netmask is an integer representing the prefix length.

An *AddressValueError* is raised if *address* is not a valid IPv6 address. A *NetmaskValueError* is raised if the mask is not valid for an IPv6 address.

If *strict* is `True` and host bits are set in the supplied address, then *ValueError* is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

Changed in version 3.5: Added the two-tuple form for the *address* constructor parameter.

version

max_prefixlen

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

network_address

broadcast_address

hostmask

netmask

with_prefixlen

compressed

exploded

with_netmask

with_hostmask

num_addresses

prefixlen

hosts()

Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the Subnet-Router anycast address. For networks with a mask length of 127, the Subnet-Router anycast address is also included in the result. Networks with a mask of 128 will return a list containing the single host address.

overlaps(*other*)

address_exclude(*network*)

subnets(*prefixlen_diff=1*, *new_prefix=None*)

supernet(*prefixlen_diff=1*, *new_prefix=None*)

subnet_of(*other*)

supernet_of(*other*)

compare_networks(*other*)

Refer to the corresponding attribute documentation in *IPv4Network*.

is_site_local

These attribute is true for the network as a whole if it is true for both the network address and the broadcast address.

Operators

Network objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Network objects can be compared with the usual set of logical operators. Network objects are ordered first by network address, then by net mask.

Iteration

Network objects can be iterated to list all the addresses belonging to the network. For iteration, *all* hosts are returned, including unusable hosts (for usable hosts, use the `hosts()` method). An example:

```
>>> for addr in IPv4Network('192.0.2.0/28'):
...     addr
...
IPv4Address('192.0.2.0')
IPv4Address('192.0.2.1')
IPv4Address('192.0.2.2')
IPv4Address('192.0.2.3')
IPv4Address('192.0.2.4')
IPv4Address('192.0.2.5')
IPv4Address('192.0.2.6')
IPv4Address('192.0.2.7')
IPv4Address('192.0.2.8')
IPv4Address('192.0.2.9')
IPv4Address('192.0.2.10')
IPv4Address('192.0.2.11')
IPv4Address('192.0.2.12')
IPv4Address('192.0.2.13')
IPv4Address('192.0.2.14')
IPv4Address('192.0.2.15')
```

Networks as containers of addresses

Network objects can act as containers of addresses. Some examples:

```
>>> IPv4Network('192.0.2.0/28')[0]
IPv4Address('192.0.2.0')
>>> IPv4Network('192.0.2.0/28')[15]
IPv4Address('192.0.2.15')
>>> IPv4Address('192.0.2.6') in IPv4Network('192.0.2.0/28')
True
>>> IPv4Address('192.0.3.6') in IPv4Network('192.0.2.0/28')
False
```

21.23.4 Interface objects

Interface objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.**IPv4Interface**(*address*)

Construct an IPv4 interface. The meaning of *address* is as in the constructor of *IPv4Network*, except that arbitrary host addresses are always accepted.

IPv4Interface is a subclass of *IPv4Address*, so it inherits all the attributes from that class. In addition, the following attributes are available:

ip

The address (*IPv4Address*) without network information.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.ip
IPv4Address('192.0.2.5')
```

network

The network (*IPv4Network*) this interface belongs to.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.network
IPv4Network('192.0.2.0/24')
```

with_prefixlen

A string representation of the interface with the mask in prefix notation.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_prefixlen
'192.0.2.5/24'
```

with_netmask

A string representation of the interface with the network as a net mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_netmask
'192.0.2.5/255.255.255.0'
```

with_hostmask

A string representation of the interface with the network as a host mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_hostmask
'192.0.2.5/0.0.0.255'
```

class ipaddress.**IPv6Interface**(*address*)

Construct an IPv6 interface. The meaning of *address* is as in the constructor of *IPv6Network*, except that arbitrary host addresses are always accepted.

IPv6Interface is a subclass of *IPv6Address*, so it inherits all the attributes from that class. In addition, the following attributes are available:

ip

network

with_prefixlen

with_netmask

with_hostmask

Refer to the corresponding attribute documentation in *IPv4Interface*.

Operators

Interface objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Interface objects can be compared with the usual set of logical operators.

For equality comparison (== and !=), both the IP address and network must be the same for the objects to be equal. An interface will not compare equal to any address or network object.

For ordering (<, >, etc) the rules are different. Interface and address objects with the same IP version can be compared, and the address objects will always sort before the interface objects. Two interface objects are first compared by their networks and, if those are the same, then by their IP addresses.

21.23.5 Other Module Level Functions

The module also provides the following module level functions:

ipaddress.**v4_int_to_packed**(*address*)

Represent an address as 4 packed bytes in network (big-endian) order. *address* is an integer representation of an IPv4 IP address. A *ValueError* is raised if the integer is negative or too large to be an IPv4 IP address.

```
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.v4_int_to_packed(3221225985)
b'\xc0\x00\x02\x01'
```

ipaddress.**v6_int_to_packed**(*address*)

Represent an address as 16 packed bytes in network (big-endian) order. *address* is an integer representation of an IPv6 IP address. A *ValueError* is raised if the integer is negative or too large to be an IPv6 IP address.

ipaddress.**summarize_address_range**(*first*, *last*)

Return an iterator of the summarized network range given the first and last IP addresses. *first* is the first *IPv4Address* or *IPv6Address* in the range and *last* is the last *IPv4Address* or *IPv6Address* in the range. A *TypeError* is raised if *first* or *last* are not IP addresses or are not of the same version. A *ValueError* is raised if *last* is not greater than *first* or if *first* address version is not 4 or 6.

```
>>> [ipaddr for ipaddr in ipaddress.summarize_address_range(
...    ipaddress.IPv4Address('192.0.2.0'),
...    ipaddress.IPv4Address('192.0.2.130'))]
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/31'), IPv4Network('192.0.2.
↪130/32')]
```

ipaddress.**collapse_addresses**(*addresses*)

Return an iterator of the collapsed *IPv4Network* or *IPv6Network* objects. *addresses* is an *iterable* of *IPv4Network* or *IPv6Network* objects. A *TypeError* is raised if *addresses* contains mixed version objects.

```
>>> [ipaddr for ipaddr in
... ipaddress.collapse_addresses([ipaddress.IPv4Network('192.0.2.0/25'),
... ipaddress.IPv4Network('192.0.2.128/25')])]
[IPv4Network('192.0.2.0/24')]
```

ipaddress.**get_mixed_type_key**(*obj*)

Return a key suitable for sorting between networks and addresses. Address and Network objects are not sortable by default; they're fundamentally different, so the expression:

```
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0/24')
```

doesn't make sense. There are some times however, where you may wish to have *ipaddress* sort these anyway. If you need to do this, you can use this function as the *key* argument to *sorted()*.

obj is either a network or address object.

21.23.6 Custom Exceptions

To support more specific error reporting from class constructors, the module defines the following exceptions:

exception ipaddress.**AddressValueError**(*ValueError*)

Any value error related to the address.

exception ipaddress.**NetmaskValueError**(*ValueError*)

Any value error related to the net mask.

CHAPTER

TWENTYTWO

MULTIMEDIA SERVICES

The modules described in this chapter implement various algorithms or interfaces that are mainly useful for multimedia applications. They are available at the discretion of the installation. Here's an overview:

22.1 `wave` — Read and write WAV files

Source code: Lib/wave.py

The *wave* module provides a convenient interface to the Waveform Audio "WAVE" (or "WAV") file format. Only uncompressed PCM encoded wave files are supported.

Changed in version 3.12: Support for `WAVE_FORMAT_EXTENSIBLE` headers was added, provided that the extended format is `KSDATAFORMAT_SUBTYPE_PCM`.

The *wave* module defines the following function and exception:

`wave.`**`open`**(*file*, *mode=None*)

If *file* is a string, open the file by that name, otherwise treat it as a file-like object. *mode* can be:

`'rb'`
Read only mode.

`'wb'`
Write only mode.

Note that it does not allow read/write WAV files.

A *mode* of `'rb'` returns a *Wave_read* object, while a *mode* of `'wb'` returns a *Wave_write* object. If *mode* is omitted and a file-like object is passed as *file*, `file.mode` is used as the default value for *mode*.

If you pass in a file-like object, the wave object will not close it when its `close()` method is called; it is the caller's responsibility to close the file object.

The *open()* function may be used in a `with` statement. When the `with` block completes, the *Wave_read.close()* or *Wave_write.close()* method is called.

Changed in version 3.4: Added support for unseekable files.

`exception` `wave.`**`Error`**

An error raised when something is impossible because it violates the WAV specification or hits an implementation deficiency.

22.1.1 Wave_read Objects

class wave.**Wave_read**

Read a WAV file.

Wave_read objects, as returned by *open()*, have the following methods:

close()

Close the stream if it was opened by *wave*, and make the instance unusable. This is called automatically on object collection.

getnchannels()

Returns number of audio channels (1 for mono, 2 for stereo).

getsampwidth()

Returns sample width in bytes.

getframerate()

Returns sampling frequency.

getnframes()

Returns number of audio frames.

getcomptype()

Returns compression type (`'NONE'` is the only supported type).

getcompname()

Human-readable version of *getcomptype()*. Usually `'not compressed'` parallels `'NONE'`.

getparams()

Returns a *namedtuple()* `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, equivalent to output of the `get*()` methods.

readframes(*n*)

Reads and returns at most *n* frames of audio, as a *bytes* object.

rewind()

Rewind the file pointer to the beginning of the audio stream.

The following two methods are defined for compatibility with the old `aifc` module, and don't do anything interesting.

getmarkers()

Returns `None`.

Deprecated since version 3.13, will be removed in version 3.15: The method only existed for compatibility with the `aifc` module which has been removed in Python 3.13.

getmark(*id*)

Raise an error.

Deprecated since version 3.13, will be removed in version 3.15: The method only existed for compatibility with the `aifc` module which has been removed in Python 3.13.

The following two methods define a term "position" which is compatible between them, and is otherwise implementation dependent.

setpos(*pos*)

Set the file pointer to the specified position.

tell()

Return current file pointer position.

22.1.2 Wave_write Objects

class wave.**Wave_write**

Write a WAV file.

Wave_write objects, as returned by *open()*.

For seekable output streams, the wave header will automatically be updated to reflect the number of frames actually written. For unseekable streams, the *nframes* value must be accurate when the first frame data is written. An accurate *nframes* value can be achieved either by calling *setnframes()* or *setparams()* with the number of frames that will be written before *close()* is called and then using *writeframesraw()* to write the frame data, or by calling *writeframes()* with all of the frame data to be written. In the latter case *writeframes()* will calculate the number of frames in the data and set *nframes* accordingly before writing the frame data.

Changed in version 3.4: Added support for unseekable files.

Wave_write objects have the following methods:

close()

Make sure *nframes* is correct, and close the file if it was opened by *wave*. This method is called upon object collection. It will raise an exception if the output stream is not seekable and *nframes* does not match the number of frames actually written.

setnchannels(*n*)

Set the number of channels.

setsampwidth(*n*)

Set the sample width to *n* bytes.

setframerate(*n*)

Set the frame rate to *n*.

Changed in version 3.2: A non-integral input to this method is rounded to the nearest integer.

setnframes(*n*)

Set the number of frames to *n*. This will be changed later if the number of frames actually written is different (this update attempt will raise an error if the output stream is not seekable).

setcomptype(*type*, *name*)

Set the compression type and description. At the moment, only compression type NONE is supported, meaning no compression.

setparams(*tuple*)

The *tuple* should be (nchannels, sampwidth, framerate, nframes, comptype, compname), with values valid for the set*() methods. Sets all parameters.

tell()

Return current position in the file, with the same disclaimer for the *Wave_read.tell()* and *Wave_read.setpos()* methods.

writeframesraw(*data*)

Write audio frames, without correcting *nframes*.

Changed in version 3.4: Any *bytes-like object* is now accepted.

writeframes(*data*)

Write audio frames and make sure *nframes* is correct. It will raise an error if the output stream is not seekable and the total number of frames that have been written after *data* has been written does not match the previously set value for *nframes*.

Changed in version 3.4: Any *bytes-like object* is now accepted.

Note that it is invalid to set any parameters after calling `writeframes()` or `writeframesraw()`, and any attempt to do so will raise `wave.Error`.

22.2 `colorsys` — Conversions between color systems

Source code: Lib/colorsys.py

The `colorsys` module defines bidirectional conversions of color values between colors expressed in the RGB (Red Green Blue) color space used in computer monitors and three other coordinate systems: YIQ, HLS (Hue Lightness Saturation) and HSV (Hue Saturation Value). Coordinates in all of these color spaces are floating-point values. In the YIQ space, the Y coordinate is between 0 and 1, but the I and Q coordinates can be positive or negative. In all other spaces, the coordinates are all between 0 and 1.

See also

More information about color spaces can be found at https://poynton.ca/ColorFAQ.html and https://www.cambridgeincolour.com/tutorials/color-spaces.htm.

The `colorsys` module defines the following functions:

colorsys.**rgb_to_yiq**(*r*, *g*, *b*)

Convert the color from RGB coordinates to YIQ coordinates.

colorsys.**yiq_to_rgb**(*y*, *i*, *q*)

Convert the color from YIQ coordinates to RGB coordinates.

colorsys.**rgb_to_hls**(*r*, *g*, *b*)

Convert the color from RGB coordinates to HLS coordinates.

colorsys.**hls_to_rgb**(*h*, *l*, *s*)

Convert the color from HLS coordinates to RGB coordinates.

colorsys.**rgb_to_hsv**(*r*, *g*, *b*)

Convert the color from RGB coordinates to HSV coordinates.

colorsys.**hsv_to_rgb**(*h*, *s*, *v*)

Convert the color from HSV coordinates to RGB coordinates.

Example:

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

CHAPTER

TWENTYTHREE

INTERNATIONALIZATION

The modules described in this chapter help you write software that is independent of language and locale by providing mechanisms for selecting a language to be used in program messages or by tailoring output to match local conventions.

The list of modules described in this chapter is:

23.1 gettext — Multilingual internationalization services

Source code: Lib/gettext.py

The *gettext* module provides internationalization (I18N) and localization (L10N) services for your Python modules and applications. It supports both the GNU **gettext** message catalog API and a higher level, class-based API that may be more appropriate for Python files. The interface described below allows you to write your module and application messages in one natural language, and provide a catalog of translated messages for running under different natural languages.

Some hints on localizing your Python modules and applications are also given.

23.1.1 GNU gettext API

The *gettext* module defines the following API, which is very similar to the GNU **gettext** API. If you use this API you will affect the translation of your entire application globally. Often this is what you want if your application is monolingual, with the choice of language dependent on the locale of your user. If you are localizing a Python module, or if your application needs to switch languages on the fly, you probably want to use the class-based API instead.

gettext.**bindtextdomain**(*domain*, *localedir=None*)

Bind the *domain* to the locale directory *localedir*. More concretely, *gettext* will look for binary `.mo` files for the given domain using the path (on Unix): *localedir*`/`*language*`/LC_MESSAGES/`*domain*`.mo`, where *language* is searched for in the environment variables `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, and `LANG` respectively.

If *localedir* is omitted or `None`, then the current binding for *domain* is returned.[1]

gettext.**textdomain**(*domain=None*)

Change or query the current global domain. If *domain* is `None`, then the current global domain is returned, otherwise the global domain is set to *domain*, which is returned.

[1] The default locale directory is system dependent; for example, on Red Hat Linux it is `/usr/share/locale`, but on Solaris it is `/usr/lib/locale`. The `gettext` module does not try to support these system dependent defaults; instead its default is *sys.base_prefix*`/share/locale` (see *sys.base_prefix*). For this reason, it is always best to call *bindtextdomain()* with an explicit absolute path at the start of your application.

gettext.**gettext**(*message*)

Return the localized translation of *message*, based on the current global domain, language, and locale directory. This function is usually aliased as `_()` in the local namespace (see examples below).

gettext.**dgettext**(*domain*, *message*)

Like *gettext()*, but look the message up in the specified *domain*.

gettext.**ngettext**(*singular*, *plural*, *n*)

Like *gettext()*, but consider plural forms. If a translation is found, apply the plural formula to *n*, and return the resulting message (some languages have more than two plural forms). If no translation is found, return *singular* if *n* is 1; return *plural* otherwise.

The Plural formula is taken from the catalog header. It is a C or Python expression that has a free variable *n*; the expression evaluates to the index of the plural in the catalog. See the GNU gettext documentation for the precise syntax to be used in `.po` files and the formulas for a variety of languages.

gettext.**dngettext**(*domain*, *singular*, *plural*, *n*)

Like *ngettext()*, but look the message up in the specified *domain*.

gettext.**pgettext**(*context*, *message*)

gettext.**dpgettext**(*domain*, *context*, *message*)

gettext.**npgettext**(*context*, *singular*, *plural*, *n*)

gettext.**dnpgettext**(*domain*, *context*, *singular*, *plural*, *n*)

Similar to the corresponding functions without the `p` in the prefix (that is, *gettext()*, *dgettext()*, *ngettext()*, *dngettext()*), but the translation is restricted to the given message *context*.

Added in version 3.8.

Note that GNU **gettext** also defines a `dcgettext()` method, but this was deemed not useful and so it is currently unimplemented.

Here's an example of typical usage for this API:

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print(_('This is a translatable string.'))
```

23.1.2 Class-based API

The class-based API of the *gettext* module gives you more flexibility and greater convenience than the GNU **gettext** API. It is the recommended way of localizing your Python applications and modules. `gettext` defines a *GNUTranslations* class which implements the parsing of GNU `.mo` format files, and has methods for returning strings. Instances of this class can also install themselves in the built-in namespace as the function `_()`.

gettext.**find**(*domain*, *localedir=None*, *languages=None*, *all=False*)

This function implements the standard `.mo` file search algorithm. It takes a *domain*, identical to what *textdomain()* takes. Optional *localedir* is as in *bindtextdomain()*. Optional *languages* is a list of strings, where each string is a language code.

If *localedir* is not given, then the default system locale directory is used.[2] If *languages* is not given, then the following environment variables are searched: `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, and `LANG`. The first one returning

[2] See the footnote for *bindtextdomain()* above.

a non-empty value is used for the *languages* variable. The environment variables should contain a colon separated list of languages, which will be split on the colon to produce the expected list of language code strings.

`find()` then expands and normalizes the languages, and then iterates through them, searching for an existing file built of these components:

localedir/*language*/LC_MESSAGES/*domain*.mo

The first such file name that exists is returned by `find()`. If no such file is found, then `None` is returned. If *all* is given, it returns a list of all file names, in the order in which they appear in the languages list or the environment variables.

gettext.**translation**(*domain*, *localedir=None*, *languages=None*, *class_=None*, *fallback=False*)

Return a `*Translations` instance based on the *domain*, *localedir*, and *languages*, which are first passed to `find()` to get a list of the associated `.mo` file paths. Instances with identical `.mo` file names are cached. The actual class instantiated is *class_* if provided, otherwise `GNUTranslations`. The class's constructor must take a single *file object* argument.

If multiple files are found, later files are used as fallbacks for earlier ones. To allow setting the fallback, `copy.copy()` is used to clone each translation object from the cache; the actual instance data is still shared with the cache.

If no `.mo` file is found, this function raises `OSError` if *fallback* is false (which is the default), and returns a `NullTranslations` instance if *fallback* is true.

Changed in version 3.3: `IOError` used to be raised, it is now an alias of `OSError`.

Changed in version 3.11: *codeset* parameter is removed.

gettext.**install**(*domain*, *localedir=None*, *, *names=None*)

This installs the function `_()` in Python's builtins namespace, based on *domain* and *localedir* which are passed to the function `translation()`.

For the *names* parameter, please see the description of the translation object's `install()` method.

As seen below, you usually mark the strings in your application that are candidates for translation, by wrapping them in a call to the `_()` function, like this:

```
print(_('This string will be translated.'))
```

For convenience, you want the `_()` function to be installed in Python's builtins namespace, so it is easily accessible in all modules of your application.

Changed in version 3.11: *names* is now a keyword-only parameter.

The `NullTranslations` class

Translation classes are what actually implement the translation of original source file message strings to translated message strings. The base class used by all translation classes is `NullTranslations`; this provides the basic interface you can use to write your own specialized translation classes. Here are the methods of `NullTranslations`:

class gettext.**NullTranslations**(*fp=None*)

Takes an optional *file object* *fp*, which is ignored by the base class. Initializes "protected" instance variables *_info* and *_charset* which are set by derived classes, as well as *_fallback*, which is set through `add_fallback()`. It then calls `self._parse(fp)` if *fp* is not `None`.

_parse(*fp*)

No-op in the base class, this method takes file object *fp*, and reads the data from the file, initializing its message catalog. If you have an unsupported message catalog file format, you should override this method to parse your format.

add_fallback(*fallback*)

Add *fallback* as the fallback object for the current translation object. A translation object should consult the fallback if it cannot provide a translation for a given message.

gettext(*message*)

If a fallback has been set, forward `gettext()` to the fallback. Otherwise, return *message*. Overridden in derived classes.

ngettext(*singular*, *plural*, *n*)

If a fallback has been set, forward `ngettext()` to the fallback. Otherwise, return *singular* if *n* is 1; return *plural* otherwise. Overridden in derived classes.

pgettext(*context*, *message*)

If a fallback has been set, forward *pgettext()* to the fallback. Otherwise, return the translated message. Overridden in derived classes.

Added in version 3.8.

npgettext(*context*, *singular*, *plural*, *n*)

If a fallback has been set, forward *npgettext()* to the fallback. Otherwise, return the translated message. Overridden in derived classes.

Added in version 3.8.

info()

Return a dictionary containing the metadata found in the message catalog file.

charset()

Return the encoding of the message catalog file.

install(*names=None*)

This method installs *gettext()* into the built-in namespace, binding it to _.

If the *names* parameter is given, it must be a sequence containing the names of functions you want to install in the builtins namespace in addition to `_()`. Supported names are `'gettext'`, `'ngettext'`, `'pgettext'`, and `'npgettext'`.

Note that this is only one way, albeit the most convenient way, to make the `_()` function available to your application. Because it affects the entire application globally, and specifically the built-in namespace, localized modules should never install `_()`. Instead, they should use this code to make `_()` available to their module:

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

This puts `_()` only in the module's global namespace and so only affects calls within this module.

Changed in version 3.8: Added `'pgettext'` and `'npgettext'`.

The `GNUTranslations` class

The `gettext` module provides one additional class derived from *`NullTranslations`*: *`GNUTranslations`*. This class overrides `_parse()` to enable reading GNU **gettext** format `.mo` files in both big-endian and little-endian format.

`GNUTranslations` parses optional metadata out of the translation catalog. It is convention with GNU **gettext** to include metadata as the translation for the empty string. This metadata is in **RFC 822**-style `key: value` pairs, and should contain the `Project-Id-Version` key. If the key `Content-Type` is found, then the `charset` property is used to initialize the "protected" `_charset` instance variable, defaulting to `None` if not found. If the charset encoding is specified, then all message ids and message strings read from the catalog are converted to Unicode using this encoding, else ASCII is assumed.

Since message ids are read as Unicode strings too, all `*gettext()` methods will assume message ids as Unicode strings, not byte strings.

The entire set of key/value pairs are placed into a dictionary and set as the "protected" `_info` instance variable.

If the `.mo` file's magic number is invalid, the major version number is unexpected, or if other problems occur while reading the file, instantiating a *`GNUTranslations`* class can raise *`OSError`*.

class gettext.**GNUTranslations**

The following methods are overridden from the base class implementation:

gettext(*message*)

Look up the *message* id in the catalog and return the corresponding message string, as a Unicode string. If there is no entry in the catalog for the *message* id, and a fallback has been set, the look up is forwarded to the fallback's *`gettext()`* method. Otherwise, the *message* id is returned.

ngettext(*singular*, *plural*, *n*)

Do a plural-forms lookup of a message id. *singular* is used as the message id for purposes of lookup in the catalog, while *n* is used to determine which plural form to use. The returned message string is a Unicode string.

If the message id is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's *`ngettext()`* method. Otherwise, when *n* is 1 *singular* is returned, and *plural* is returned in all other cases.

Here is an example:

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ngettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

pgettext(*context*, *message*)

Look up the *context* and *message* id in the catalog and return the corresponding message string, as a Unicode string. If there is no entry in the catalog for the *message* id and *context*, and a fallback has been set, the look up is forwarded to the fallback's *`pgettext()`* method. Otherwise, the *message* id is returned.

Added in version 3.8.

npgettext(*context*, *singular*, *plural*, *n*)

Do a plural-forms lookup of a message id. *singular* is used as the message id for purposes of lookup in the catalog, while *n* is used to determine which plural form to use.

If the message id for *context* is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's *npgettext()* method. Otherwise, when *n* is 1 *singular* is returned, and *plural* is returned in all other cases.

Added in version 3.8.

Solaris message catalog support

The Solaris operating system defines its own binary `.mo` file format, but since no documentation can be found on this format, it is not supported at this time.

The Catalog constructor

GNOME uses a version of the *gettext* module by James Henstridge, but this version has a slightly different API. Its documented usage was:

```
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print(_('hello world'))
```

For compatibility with this older module, the function `Catalog()` is an alias for the *translation()* function described above.

One difference between this module and Henstridge's: his catalog objects supported access through a mapping API, but this appears to be unused and so is not currently supported.

23.1.3 Internationalizing your programs and modules

Internationalization (I18N) refers to the operation by which a program is made aware of multiple languages. Localization (L10N) refers to the adaptation of your program, once internationalized, to the local language and cultural habits. In order to provide multilingual messages for your Python programs, you need to take the following steps:

1. prepare your program or module by specially marking translatable strings
2. run a suite of tools over your marked files to generate raw messages catalogs
3. create language-specific translations of the message catalogs
4. use the *gettext* module so that message strings are properly translated

In order to prepare your code for I18N, you need to look at all the strings in your files. Any string that needs to be translated should be marked by wrapping it in `_('...')` — that is, a call to the function `_`. For example:

```
filename = 'mylog.txt'
message = _('writing a log message')
with open(filename, 'w') as fp:
    fp.write(message)
```

In this example, the string `'writing a log message'` is marked as a candidate for translation, while the strings `'mylog.txt'` and `'w'` are not.

There are a few tools to extract the strings meant for translation. The original GNU **gettext** only supported C or C++ source code but its extended version **xgettext** scans code written in a number of languages, including Python, to find strings marked as translatable. Babel is a Python internationalization library that includes a `pybabel` script to extract and compile message catalogs. François Pinard's program called **xpot** does a similar job and is available as part of his po-utils package.

(Python also includes pure-Python versions of these programs, called **pygettext.py** and **msgfmt.py**; some Python distributions will install them for you. **pygettext.py** is similar to **xgettext**, but only understands Python source code and cannot handle other programming languages such as C or C++. **pygettext.py** supports a command-line interface similar to **xgettext**; for details on its use, run `pygettext.py --help`. **msgfmt.py** is binary compatible with GNU **msgfmt**. With these two programs, you may not need the GNU **gettext** package to internationalize your Python applications.)

xgettext, **pygettext**, and similar tools generate `.po` files that are message catalogs. They are structured human-readable files that contain every marked string in the source code, along with a placeholder for the translated versions of these strings.

Copies of these `.po` files are then handed over to the individual human translators who write translations for every supported natural language. They send back the completed language-specific versions as a `<language-name>.po` file that's compiled into a machine-readable `.mo` binary catalog file using the **msgfmt** program. The `.mo` files are used by the *gettext* module for the actual translation processing at run-time.

How you use the *gettext* module in your code depends on whether you are internationalizing a single module or your entire application. The next two sections will discuss each case.

Localizing your module

If you are localizing your module, you must take care not to make global changes, e.g. to the built-in namespace. You should not use the GNU **gettext** API but instead the class-based API.

Let's say your module is called "spam" and the module's various natural language translation `.mo` files reside in `/usr/share/locale` in GNU **gettext** format. Here's what you would put at the top of your module:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.gettext
```

Localizing your application

If you are localizing your application, you can install the `_()` function globally into the built-in namespace, usually in the main driver file of your application. This will let all your application-specific files just use `_('...')` without having to explicitly install it in each file.

In the simple case then, you need only add the following bit of code to the main driver file of your application:

```
import gettext
gettext.install('myapplication')
```

If you need to set the locale directory, you can pass it into the *install()* function:

```
import gettext
gettext.install('myapplication', '/usr/share/locale')
```

Changing languages on the fly

If your program needs to support many languages at the same time, you may want to create multiple translation instances and then switch between them explicitly, like so:

```
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

Deferred translations

In most coding situations, strings are translated where they are coded. Occasionally however, you need to mark strings for translation, but defer actual translation until later. A classic example is:

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python', ]
# ...
for a in animals:
    print(a)
```

Here, you want to mark the strings in the `animals` list as being translatable, but you don't actually want to translate them until they are printed.

Here is one way you can handle this situation:

```
def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
    print(_(a))
```

This works because the dummy definition of `_()` simply returns the string unchanged. And this dummy definition will temporarily override any definition of `_()` in the built-in namespace (until the `del` command). Take care, though if you have a previous definition of `_()` in the local namespace.

Note that the second use of `_()` will not identify "a" as being translatable to the **gettext** program, because the parameter is not a string literal.

Another way to handle this is with the following example:

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print(_(a))
```

In this case, you are marking translatable strings with the function `N_()`, which won't conflict with any definition of `_()`. However, you will need to teach your message extraction program to look for translatable strings marked with `N_()`. **xgettext**, **pygettext**, `pybabel extract`, and **xpot** all support this through the use of the `-k` command-line switch. The choice of `N_()` here is totally arbitrary; it could have just as easily been `MarkThisStringForTranslation()`.

23.1.4 Acknowledgements

The following people contributed code, feedback, design suggestions, previous implementations, and valuable experience to the creation of this module:

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw
- Gustavo Niemeyer

23.2 `locale` — Internationalization services

Source code: Lib/locale.py

The *locale* module opens access to the POSIX locale database and functionality. The POSIX locale mechanism allows programmers to deal with certain cultural issues in an application, without requiring the programmer to know all the specifics of each country where the software is executed.

The *locale* module is implemented on top of the `_locale` module, which in turn uses an ANSI C locale implementation if available.

The *locale* module defines the following exception and functions:

exception locale.**Error**

Exception raised when the locale passed to *setlocale()* is not recognized.

locale.**setlocale**(*category*, *locale=None*)

If *locale* is given and not None, *setlocale()* modifies the locale setting for the *category*. The available categories are listed in the data description below. *locale* may be a string, or an iterable of two strings (language code and encoding). If it's an iterable, it's converted to a locale name using the locale aliasing engine. An empty string specifies the user's default settings. If the modification of the locale fails, the exception *Error* is raised. If successful, the new locale setting is returned.

If *locale* is omitted or None, the current setting for *category* is returned.

setlocale() is not thread-safe on most systems. Applications typically start with a call of

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

This sets the locale for all categories to the user's default setting (typically specified in the LANG environment variable). If the locale is not changed thereafter, using multithreading should not cause problems.

locale.**localeconv**()

Returns the database of the local conventions as a dictionary. This dictionary has the following strings as keys:

Category	Key	Meaning
LC_NUMERIC	'decimal_point'	Decimal point character.
	'grouping'	Sequence of numbers specifying which relative positions the 'thousands_sep' is expected. If the sequence is terminated with *CHAR_MAX*, no further grouping is performed. If the sequence terminates with a 0, the last group size is repeatedly used.
	'thousands_sep'	Character used between groups.
LC_MONETARY	'int_curr_symbol'	International currency symbol.
	'currency_symbol'	Local currency symbol.
	'p_cs_precedes/n_cs_precedes'	Whether the currency symbol precedes the value (for positive resp. negative values).
	'p_sep_by_space/n_sep_by_space'	Whether the currency symbol is separated from the value by a space (for positive resp. negative values).
	'mon_decimal_point'	Decimal point used for monetary values.
	'frac_digits'	Number of fractional digits used in local formatting of monetary values.
	'int_frac_digits'	Number of fractional digits used in international formatting of monetary values.
	'mon_thousands_sep'	Group separator used for monetary values.
	'mon_grouping'	Equivalent to 'grouping', used for monetary values.
	'positive_sign'	Symbol used to annotate a positive monetary value.
	'negative_sign'	Symbol used to annotate a negative monetary value.
	'p_sign_posn/n_sign_posn'	The position of the sign (for positive resp. negative values), see below.

All numeric values can be set to *CHAR_MAX* to indicate that there is no value specified in this locale.

The possible values for 'p_sign_posn' and 'n_sign_posn' are given below.

Value	Explanation
0	Currency and value are surrounded by parentheses.
1	The sign should precede the value and currency symbol.
2	The sign should follow the value and currency symbol.
3	The sign should immediately precede the value.
4	The sign should immediately follow the value.
CHAR_MAX	Nothing is specified in this locale.

The function temporarily sets the LC_CTYPE locale to the LC_NUMERIC locale or the LC_MONETARY locale if locales are different and numeric or monetary strings are non-ASCII. This temporary change affects other threads.

Changed in version 3.7: The function now temporarily sets the `LC_CTYPE` locale to the `LC_NUMERIC` locale in some cases.

`locale.`**`nl_langinfo`**(*option*)

Return some locale-specific information as a string. This function is not available on all systems, and the set of possible options might also vary across platforms. The possible argument values are numbers, for which symbolic constants are available in the locale module.

The *`nl_langinfo()`* function accepts one of the following keys. Most descriptions are taken from the corresponding description in the GNU C library.

`locale.`**`CODESET`**

Get a string with the name of the character encoding used in the selected locale.

`locale.`**`D_T_FMT`**

Get a string that can be used as a format string for *`time.strftime()`* to represent date and time in a locale-specific way.

`locale.`**`D_FMT`**

Get a string that can be used as a format string for *`time.strftime()`* to represent a date in a locale-specific way.

`locale.`**`T_FMT`**

Get a string that can be used as a format string for *`time.strftime()`* to represent a time in a locale-specific way.

`locale.`**`T_FMT_AMPM`**

Get a format string for *`time.strftime()`* to represent time in the am/pm format.

`locale.`**`DAY_1`**

`locale.`**`DAY_2`**

`locale.`**`DAY_3`**

`locale.`**`DAY_4`**

`locale.`**`DAY_5`**

`locale.`**`DAY_6`**

`locale.`**`DAY_7`**

Get the name of the n-th day of the week.

> **Note**
>
> This follows the US convention of *`DAY_1`* being Sunday, not the international convention (ISO 8601) that Monday is the first day of the week.

`locale.`**`ABDAY_1`**

`locale.`**`ABDAY_2`**

`locale.`**`ABDAY_3`**

`locale.`**`ABDAY_4`**

`locale.`**`ABDAY_5`**

`locale.`**`ABDAY_6`**

`locale.`**`ABDAY_7`**

Get the abbreviated name of the n-th day of the week.

`locale.`**`MON_1`**

locale.**MON_2**
locale.**MON_3**
locale.**MON_4**
locale.**MON_5**
locale.**MON_6**
locale.**MON_7**
locale.**MON_8**
locale.**MON_9**
locale.**MON_10**
locale.**MON_11**
locale.**MON_12**

Get the name of the n-th month.

locale.**ABMON_1**
locale.**ABMON_2**
locale.**ABMON_3**
locale.**ABMON_4**
locale.**ABMON_5**
locale.**ABMON_6**
locale.**ABMON_7**
locale.**ABMON_8**
locale.**ABMON_9**
locale.**ABMON_10**
locale.**ABMON_11**
locale.**ABMON_12**

Get the abbreviated name of the n-th month.

locale.**RADIXCHAR**

Get the radix character (decimal dot, decimal comma, etc.).

locale.**THOUSEP**

Get the separator character for thousands (groups of three digits).

locale.**YESEXPR**

Get a regular expression that can be used with the regex function to recognize a positive response to a yes/no question.

locale.**NOEXPR**

Get a regular expression that can be used with the `regex(3)` function to recognize a negative response to a yes/no question.

> **Note**
>
> The regular expressions for *YESEXPR* and *NOEXPR* use syntax suitable for the `regex` function from the C library, which might differ from the syntax used in *re*.

locale.**CRNCYSTR**

Get the currency symbol, preceded by "-" if the symbol should appear before the value, "+" if the symbol should appear after the value, or "." if the symbol should replace the radix character.

locale.**ERA**

Get a string that represents the era used in the current locale.

Most locales do not define this value. An example of a locale which does define this value is the Japanese one. In Japan, the traditional representation of dates includes the name of the era corresponding to the then-emperor's reign.

Normally it should not be necessary to use this value directly. Specifying the `E` modifier in their format strings causes the *time.strftime()* function to use this information. The format of the returned string is not specified, and therefore you should not assume knowledge of it on different systems.

locale.**ERA_D_T_FMT**

Get a format string for *time.strftime()* to represent date and time in a locale-specific era-based way.

locale.**ERA_D_FMT**

Get a format string for *time.strftime()* to represent a date in a locale-specific era-based way.

locale.**ERA_T_FMT**

Get a format string for *time.strftime()* to represent a time in a locale-specific era-based way.

locale.**ALT_DIGITS**

Get a representation of up to 100 values used to represent the values 0 to 99.

locale.**getdefaultlocale**([*envvars*])

Tries to determine the default locale settings and returns them as a tuple of the form `(language code, encoding)`.

According to POSIX, a program which has not called `setlocale(LC_ALL, '')` runs using the portable `'C'` locale. Calling `setlocale(LC_ALL, '')` lets it use the default locale as defined by the `LANG` variable. Since we do not want to interfere with the current locale setting we thus emulate the behavior in the way described above.

To maintain compatibility with other platforms, not only the `LANG` variable is tested, but a list of variables given as envvars parameter. The first found to be defined will be used. *envvars* defaults to the search path used in GNU gettext; it must always contain the variable name `'LANG'`. The GNU gettext search path contains `'LC_ALL'`, `'LC_CTYPE'`, `'LANG'` and `'LANGUAGE'`, in that order.

Except for the code `'C'`, the language code corresponds to **RFC 1766**. *language code* and *encoding* may be `None` if their values cannot be determined.

Deprecated since version 3.11, will be removed in version 3.15.

locale.**getlocale**(*category=LC_CTYPE*)

Returns the current setting for the given locale category as sequence containing *language code*, *encoding*. *category* may be one of the `LC_*` values except *LC_ALL*. It defaults to *LC_CTYPE*.

Except for the code `'C'`, the language code corresponds to **RFC 1766**. *language code* and *encoding* may be `None` if their values cannot be determined.

locale.**getpreferredencoding**(*do_setlocale=True*)

Return the *locale encoding* used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess.

On some systems, it is necessary to invoke *setlocale()* to obtain the user preferences, so this function is not thread-safe. If invoking setlocale is not necessary or desired, *do_setlocale* should be set to `False`.

On Android or if the *Python UTF-8 Mode* is enabled, always return `'utf-8'`, the *locale encoding* and the *do_setlocale* argument are ignored.

The Python preinitialization configures the LC_CTYPE locale. See also the *filesystem encoding and error handler*.

Changed in version 3.7: The function now always returns `"utf-8"` on Android or if the *Python UTF-8 Mode* is enabled.

`locale.`**`getencoding`**`()`

Get the current *locale encoding*:

- On Android and VxWorks, return `"utf-8"`.
- On Unix, return the encoding of the current *LC_CTYPE* locale. Return `"utf-8"` if `nl_langinfo(CODESET)` returns an empty string: for example, if the current LC_CTYPE locale is not supported.
- On Windows, return the ANSI code page.

The Python preinitialization configures the LC_CTYPE locale. See also the *filesystem encoding and error handler*.

This function is similar to *getpreferredencoding(False)* except this function ignores the *Python UTF-8 Mode*.

Added in version 3.11.

`locale.`**`normalize`**`(`*localename*`)`

Returns a normalized locale code for the given locale name. The returned locale code is formatted for use with *setlocale()*. If normalization fails, the original name is returned unchanged.

If the given encoding is not known, the function defaults to the default encoding for the locale code just like *setlocale()*.

`locale.`**`strcoll`**`(`*string1*, *string2*`)`

Compares two strings according to the current *LC_COLLATE* setting. As any other compare function, returns a negative, or a positive value, or `0`, depending on whether *string1* collates before or after *string2* or is equal to it.

`locale.`**`strxfrm`**`(`*string*`)`

Transforms a string to one that can be used in locale-aware comparisons. For example, `strxfrm(s1) < strxfrm(s2)` is equivalent to `strcoll(s1, s2) < 0`. This function can be used when the same string is compared repeatedly, e.g. when collating a sequence of strings.

`locale.`**`format_string`**`(`*format*, *val*, *grouping=False*, *monetary=False*`)`

Formats a number *val* according to the current *LC_NUMERIC* setting. The format follows the conventions of the `%` operator. For floating-point values, the decimal point is modified if appropriate. If *grouping* is `True`, also takes the grouping into account.

If *monetary* is true, the conversion uses monetary thousands separator and grouping strings.

Processes formatting specifiers as in `format % val`, but takes the current locale settings into account.

Changed in version 3.7: The *monetary* keyword parameter was added.

`locale.`**`currency`**`(`*val*, *symbol=True*, *grouping=False*, *international=False*`)`

Formats a number *val* according to the current *LC_MONETARY* settings.

The returned string includes the currency symbol if *symbol* is true, which is the default. If *grouping* is `True` (which is not the default), grouping is done with the value. If *international* is `True` (which is not the default), the international currency symbol is used.

> **Note**
>
> This function will not work with the 'C' locale, so you have to set a locale via *setlocale()* first.

locale.**str**(*float*)

Formats a floating-point number using the same format as the built-in function `str(float)`, but takes the decimal point into account.

locale.**delocalize**(*string*)

Converts a string into a normalized number string, following the *LC_NUMERIC* settings.

Added in version 3.5.

locale.**localize**(*string*, *grouping=False*, *monetary=False*)

Converts a normalized number string into a formatted string following the *LC_NUMERIC* settings.

Added in version 3.10.

locale.**atof**(*string*, *func=float*)

Converts a string to a number, following the *LC_NUMERIC* settings, by calling *func* on the result of calling *delocalize()* on *string*.

locale.**atoi**(*string*)

Converts a string to an integer, following the *LC_NUMERIC* conventions.

locale.**LC_CTYPE**

Locale category for the character type functions. Most importantly, this category defines the text encoding, i.e. how bytes are interpreted as Unicode codepoints. See **PEP 538** and **PEP 540** for how this variable might be automatically coerced to `C.UTF-8` to avoid issues created by invalid settings in containers or incompatible settings passed over remote SSH connections.

Python doesn't internally use locale-dependent character transformation functions from `ctype.h`. Instead, an internal `pyctype.h` provides locale-independent equivalents like `Py_TOLOWER`.

locale.**LC_COLLATE**

Locale category for sorting strings. The functions *strcoll()* and *strxfrm()* of the *locale* module are affected.

locale.**LC_TIME**

Locale category for the formatting of time. The function *time.strftime()* follows these conventions.

locale.**LC_MONETARY**

Locale category for formatting of monetary values. The available options are available from the *localeconv()* function.

locale.**LC_MESSAGES**

Locale category for message display. Python currently does not support application specific locale-aware messages. Messages displayed by the operating system, like those returned by *os.strerror()* might be affected by this category.

This value may not be available on operating systems not conforming to the POSIX standard, most notably Windows.

locale.**LC_NUMERIC**

Locale category for formatting numbers. The functions *format_string()*, *atoi()*, *atof()* and *str()* of the *locale* module are affected by that category. All other numeric formatting operations are not affected.

locale.**LC_ALL**

Combination of all locale settings. If this flag is used when the locale is changed, setting the locale for all categories is attempted. If that fails for any category, no category is changed at all. When the locale is retrieved using this flag, a string indicating the setting for all categories is returned. This string can be later used to restore the settings.

locale.**CHAR_MAX**

This is a symbolic constant used for different values returned by *localeconv()*.

Example:

```
>>> import locale
>>> loc = locale.getlocale()  # get current locale
# use German locale; name might vary with platform
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> locale.strcoll('f\xe4n', 'foo')  # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '')    # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C')   # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc)   # restore saved locale
```

23.2.1 Background, details, hints, tips and caveats

The C standard defines the locale as a program-wide property that may be relatively expensive to change. On top of that, some implementations are broken in such a way that frequent locale changes may cause core dumps. This makes the locale somewhat painful to use correctly.

Initially, when a program is started, the locale is the C locale, no matter what the user's preferred locale is. There is one exception: the *LC_CTYPE* category is changed at startup to set the current locale encoding to the user's preferred locale encoding. The program must explicitly say that it wants the user's preferred locale settings for other categories by calling `setlocale(LC_ALL, '')`.

It is generally a bad idea to call *setlocale()* in some library routine, since as a side effect it affects the entire program. Saving and restoring it is almost as bad: it is expensive and affects other threads that happen to run before the settings have been restored.

If, when coding a module for general use, you need a locale independent version of an operation that is affected by the locale (such as certain formats used with *time.strftime()*), you will have to find a way to do it without using the standard library routine. Even better is convincing yourself that using locale settings is okay. Only as a last resort should you document that your module is not compatible with non-C locale settings.

The only way to perform numeric operations according to the locale is to use the special functions defined by this module: *atof()*, *atoi()*, *format_string()*, *str()*.

There is no way to perform case conversions and character classifications according to the locale. For (Unicode) text strings these are done according to the character value only, while for byte strings, the conversions and classifications are done according to the ASCII value of the byte, and bytes whose high bit is set (i.e., non-ASCII bytes) are never converted or considered part of a character class such as letter or whitespace.

23.2.2 For extension writers and programs that embed Python

Extension modules should never call *setlocale()*, except to find out what the current locale is. But since the return value can only be used portably to restore it, that is not very useful (except perhaps to find out whether or not the locale is C).

When Python code uses the *locale* module to change the locale, this also affects the embedding application. If the embedding application doesn't want this to happen, it should remove the `_locale` extension module (which does all the work) from the table of built-in modules in the `config.c` file, and make sure that the `_locale` module is not accessible as a shared library.

23.2.3 Access to message catalogs

locale.**gettext**(*msg*)

locale.**dgettext**(*domain*, *msg*)

locale.**dcgettext**(*domain*, *msg*, *category*)

locale.**textdomain**(*domain*)

locale.**bindtextdomain**(*domain*, *dir*)

locale.**bind_textdomain_codeset**(*domain*, *codeset*)

The locale module exposes the C library's gettext interface on systems that provide this interface. It consists of the functions *gettext()*, *dgettext()*, *dcgettext()*, *textdomain()*, *bindtextdomain()*, and *bind_textdomain_codeset()*. These are similar to the same functions in the *gettext* module, but use the C library's binary format for message catalogs, and the C library's search algorithms for locating message catalogs.

Python applications should normally find no need to invoke these functions, and should use *gettext* instead. A known exception to this rule are applications that link with additional C libraries which internally invoke C functions `gettext` or `dcgettext`. For these applications, it may be necessary to bind the text domain, so that the libraries can properly locate their message catalogs.

CHAPTER

TWENTYFOUR

PROGRAM FRAMEWORKS

The modules described in this chapter are frameworks that will largely dictate the structure of your program. Currently the modules described here are all oriented toward writing command-line interfaces.

The full list of modules described in this chapter is:

24.1 `turtle` — Turtle graphics

Source code: Lib/turtle.py

24.1.1 Introduction

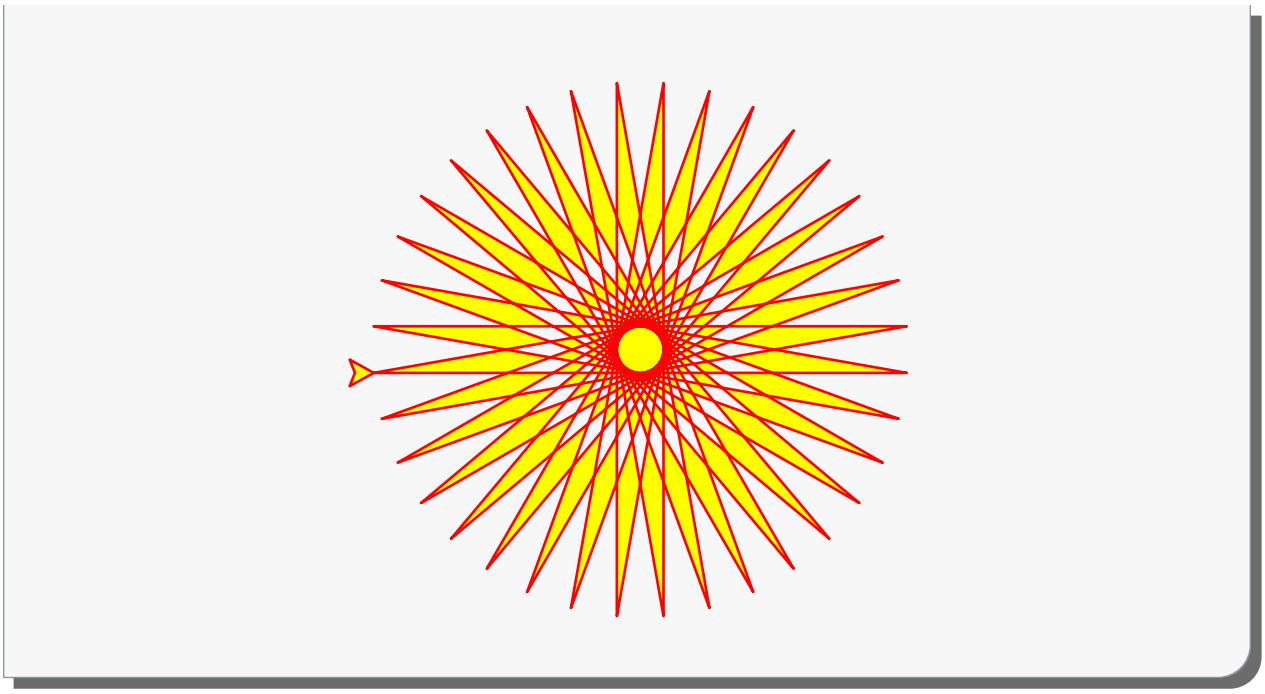
Turtle graphics is an implementation of the popular geometric drawing tools introduced in Logo, developed by Wally Feurzeig, Seymour Papert and Cynthia Solomon in 1967.

24.1.2 Get started

Imagine a robotic turtle starting at (0, 0) in the x-y plane. After an `import turtle`, give it the command `turtle.forward(15)`, and it moves (on-screen!) 15 pixels in the direction it is facing, drawing a line as it moves. Give it the command `turtle.right(25)`, and it rotates in-place 25 degrees clockwise.

Turtle star

Turtle can draw intricate shapes using programs that repeat simple moves.



In Python, turtle graphics provides a representation of a physical "turtle" (a little robot with a pen) that draws on a sheet of paper on the floor.

It's an effective and well-proven way for learners to encounter programming concepts and interaction with software, as it provides instant, visible feedback. It also provides convenient access to graphical output in general.

Turtle drawing was originally created as an educational tool, to be used by teachers in the classroom. For the programmer who needs to produce some graphical output it can be a way to do that without the overhead of introducing more complex or external libraries into their work.

24.1.3 Tutorial

New users should start here. In this tutorial we'll explore some of the basics of turtle drawing.

Starting a turtle environment

In a Python shell, import all the objects of the `turtle` module:

```
from turtle import *
```

If you run into a `No module named '_tkinter'` error, you'll have to install the *Tk interface package* on your system.

Basic drawing

Send the turtle forward 100 steps:

```
forward(100)
```

You should see (most likely, in a new window on your display) a line drawn by the turtle, heading East. Change the direction of the turtle, so that it turns 120 degrees left (anti-clockwise):

```
left(120)
```

Let's continue by drawing a triangle:

```
forward(100)
left(120)
forward(100)
```

Notice how the turtle, represented by an arrow, points in different directions as you steer it.

Experiment with those commands, and also with `backward()` and `right()`.

Pen control

Try changing the color - for example, `color('blue')` - and width of the line - for example, `width(3)` - and then drawing again.

You can also move the turtle around without drawing, by lifting up the pen: `up()` before moving. To start drawing again, use `down()`.

The turtle's position

Send your turtle back to its starting-point (useful if it has disappeared off-screen):

```
home()
```

The home position is at the center of the turtle's screen. If you ever need to know them, get the turtle's x-y coordinates with:

```
pos()
```

Home is at `(0, 0)`.

And after a while, it will probably help to clear the window so we can start anew:

```
clearscreen()
```

Making algorithmic patterns

Using loops, it's possible to build up geometric patterns:

```
for steps in range(100):
    for c in ('blue', 'red', 'green'):
        color(c)
        forward(steps)
        right(30)
```

- which of course, are limited only by the imagination!

Let's draw the star shape at the top of this page. We want red lines, filled in with yellow:

```
color('red')
fillcolor('yellow')
```

Just as `up()` and `down()` determine whether lines will be drawn, filling can be turned on and off:

```
begin_fill()
```

Next we'll create a loop:

```
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
```

`abs(pos()) < 1` is a good way to know when the turtle is back at its home position.

Finally, complete the filling:

```
end_fill()
```

(Note that filling only actually takes place when you give the `end_fill()` command.)

24.1.4 How to...

This section covers some typical turtle use-cases and approaches.

Get started as quickly as possible

One of the joys of turtle graphics is the immediate, visual feedback that's available from simple commands - it's an excellent way to introduce children to programming ideas, with a minimum of overhead (not just children, of course).

The turtle module makes this possible by exposing all its basic functionality as functions, available with `from turtle import *`. The *turtle graphics tutorial* covers this approach.

It's worth noting that many of the turtle commands also have even more terse equivalents, such as `fd()` for *`forward()`*. These are especially useful when working with learners for whom typing is not a skill.

> You'll need to have the *`Tk interface package`* installed on your system for turtle graphics to work. Be warned that this is not always straightforward, so check this in advance if you're planning to use turtle graphics with a learner.

Use the `turtle` module namespace

Using `from turtle import *` is convenient - but be warned that it imports a rather large collection of objects, and if you're doing anything but turtle graphics you run the risk of a name conflict (this becomes even more an issue if you're using turtle graphics in a script where other modules might be imported).

The solution is to use `import turtle` - `fd()` becomes `turtle.fd()`, `width()` becomes `turtle.width()` and so on. (If typing "turtle" over and over again becomes tedious, use for example `import turtle as t` instead.)

Use turtle graphics in a script

It's recommended to use the `turtle` module namespace as described immediately above, for example:

```
import turtle as t
from random import random

for i in range(100):
    steps = int(random() * 100)
    angle = int(random() * 360)
    t.right(angle)
    t.fd(steps)
```

Another step is also required though - as soon as the script ends, Python will also close the turtle's window. Add:

```
t.mainloop()
```

to the end of the script. The script will now wait to be dismissed and will not exit until it is terminated, for example by closing the turtle graphics window.

Use object-oriented turtle graphics

See also

Explanation of the object-oriented interface

Other than for very basic introductory purposes, or for trying things out as quickly as possible, it's more usual and much more powerful to use the object-oriented approach to turtle graphics. For example, this allows multiple turtles on screen at once.

In this approach, the various turtle commands are methods of objects (mostly of `Turtle` objects). You *can* use the object-oriented approach in the shell, but it would be more typical in a Python script.

The example above then becomes:

```
from turtle import Turtle
from random import random

t = Turtle()
for i in range(100):
    steps = int(random() * 100)
    angle = int(random() * 360)
    t.right(angle)
    t.fd(steps)
```

(continues on next page)

(continued from previous page)

```
t.screen.mainloop()
```

Note the last line. `t.screen` is an instance of the *Screen* that a Turtle instance exists on; it's created automatically along with the turtle.

The turtle's screen can be customised, for example:

```
t.screen.title('Object-oriented turtle demo')
t.screen.bgcolor("orange")
```

24.1.5 Turtle graphics reference

> **Note**
>
> In the following documentation the argument list for functions is given. Methods, of course, have the additional first argument *self* which is omitted here.

Turtle methods

Turtle motion

Move and draw

forward() | *fd()*
backward() | *bk()* | *back()*
right() | *rt()*
left() | *lt()*
goto() | *setpos()* | *setposition()*
teleport()
setx()
sety()
setheading() | *seth()*
home()
circle()
dot()
stamp()
clearstamp()
clearstamps()
undo()
speed()

Tell Turtle's state

position() | *pos()*
towards()
xcor()
ycor()
heading()
distance()

Setting and measurement

degrees()
radians()

Pen control

Drawing state

pendown() | *pd()* | *down()*
penup() | *pu()* | *up()*
pensize() | *width()*
pen()
isdown()

Color control

color()
pencolor()
fillcolor()

Filling

filling()
begin_fill()
end_fill()

More drawing control

reset()
clear()
write()

Turtle state

Visibility

showturtle() | *st()*
hideturtle() | *ht()*
isvisible()

Appearance

shape()
resizemode()
shapesize() | *turtlesize()*
shearfactor()
tiltangle()
tilt()
shapetransform()
get_shapepoly()

Using events

onclick()
onrelease()
ondrag()

Special Turtle methods

begin_poly()
end_poly()
get_poly()
clone()
getturtle() | *getpen()*
getscreen()
setundobuffer()
undobufferentries()

Methods of TurtleScreen/Screen

Window control

bgcolor()
bgpic()
clearscreen()
resetscreen()
screensize()
setworldcoordinates()

Animation control

delay()
tracer()
update()

Using screen events

listen()
onkey() | *onkeyrelease()*
onkeypress()
onclick() | *onscreenclick()*
ontimer()
mainloop() | *done()*

Settings and special methods

mode()
colormode()
getcanvas()
getshapes()
register_shape() | *addshape()*
turtles()
window_height()
window_width()

Input methods

textinput()
numinput()

Methods specific to Screen

bye()

exitonclick()
setup()
title()

24.1.6 Methods of RawTurtle/Turtle and corresponding functions

Most of the examples in this section refer to a Turtle instance called `turtle`.

Turtle motion

`turtle.`**`forward`**(*distance*)
`turtle.`**`fd`**(*distance*)

Parameters
`distance` – a number (integer or float)

Move the turtle forward by the specified *distance*, in the direction the turtle is headed.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

`turtle.`**`back`**(*distance*)
`turtle.`**`bk`**(*distance*)
`turtle.`**`backward`**(*distance*)

Parameters
`distance` – a number

Move the turtle backward by *distance*, opposite to the direction the turtle is headed. Do not change the turtle's heading.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

`turtle.`**`right`**(*angle*)
`turtle.`**`rt`**(*angle*)

Parameters
`angle` – a number (integer or float)

Turn turtle right by *angle* units. (Units are by default degrees, but can be set via the *degrees()* and *radians()* functions.) Angle orientation depends on the turtle mode, see *mode()*.

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

turtle.**left**(*angle*)

turtle.**lt**(*angle*)

Parameters

angle – a number (integer or float)

Turn turtle left by *angle* units. (Units are by default degrees, but can be set via the *degrees()* and *radians()* functions.) Angle orientation depends on the turtle mode, see *mode()*.

```
>>> turtle.heading()
22.0
>>> turtle.left(45)
>>> turtle.heading()
67.0
```

turtle.**goto**(*x*, *y=None*)

turtle.**setpos**(*x*, *y=None*)

turtle.**setposition**(*x*, *y=None*)

Parameters

- **x** – a number or a pair/vector of numbers
- **y** – a number or None

If *y* is None, *x* must be a pair of coordinates or a *Vec2D* (e.g. as returned by *pos()*).

Move turtle to an absolute position. If the pen is down, draw line. Do not change the turtle's orientation.

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00,0.00)
```

turtle.**teleport**(*x*, *y=None*, *, *fill_gap=False*)

Parameters

- **x** – a number or None
- **y** – a number or None
- **fill_gap** – a boolean

Move turtle to an absolute position. Unlike goto(x, y), a line will not be drawn. The turtle's orientation does not change. If currently filling, the polygon(s) teleported from will be filled after leaving, and filling will begin

again after teleporting. This can be disabled with fill_gap=True, which makes the imaginary line traveled during teleporting act as a fill barrier like in goto(x, y).

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.teleport(60)
>>> turtle.pos()
(60.00,0.00)
>>> turtle.teleport(y=10)
>>> turtle.pos()
(60.00,10.00)
>>> turtle.teleport(20, 30)
>>> turtle.pos()
(20.00,30.00)
```

Added in version 3.12.

turtle.**setx**(*x*)

Parameters
x – a number (integer or float)

Set the turtle's first coordinate to *x*, leave second coordinate unchanged.

```
>>> turtle.position()
(0.00,240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00,240.00)
```

turtle.**sety**(*y*)

Parameters
y – a number (integer or float)

Set the turtle's second coordinate to *y*, leave first coordinate unchanged.

```
>>> turtle.position()
(0.00,40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```

turtle.**setheading**(*to_angle*)
turtle.**seth**(*to_angle*)

Parameters
to_angle – a number (integer or float)

Set the orientation of the turtle to *to_angle*. Here are some common directions in degrees:

standard mode	logo mode
0 - east	0 - north
90 - north	90 - east
180 - west	180 - south
270 - south	270 - west

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

turtle.**home**()

Move turtle to the origin – coordinates (0,0) – and set its heading to its start-orientation (which depends on the mode, see *mode()*).

```
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00,-10.00)
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**circle**(*radius*, *extent=None*, *steps=None*)

Parameters

- **radius** – a number
- **extent** – a number (or None)
- **steps** – an integer (or None)

Draw a circle with given *radius*. The center is *radius* units left of the turtle; *extent* – an angle – determines which part of the circle is drawn. If *extent* is not given, draw the entire circle. If *extent* is not a full circle, one endpoint of the arc is the current pen position. Draw the arc in counterclockwise direction if *radius* is positive, otherwise in clockwise direction. Finally the direction of the turtle is changed by the amount of *extent*.

As the circle is approximated by an inscribed regular polygon, *steps* determines the number of steps to use. If not given, it will be calculated automatically. May be used to draw regular polygons.

```
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180)  # draw a semicircle
>>> turtle.position()
(0.00,240.00)
>>> turtle.heading()
180.0
```

turtle.**dot**(*size=None*, **color*)

Parameters

- **size** – an integer >= 1 (if given)
- **color** – a colorstring or a numeric color tuple

Draw a circular dot with diameter *size*, using *color*. If *size* is not given, the maximum of pensize+4 and 2*pensize is used.

```
>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
(100.00,-0.00)
>>> turtle.heading()
0.0
```

turtle.**stamp**()

Stamp a copy of the turtle shape onto the canvas at the current turtle position. Return a stamp_id for that stamp, which can be used to delete it by calling `clearstamp(stamp_id)`.

```
>>> turtle.color("blue")
>>> stamp_id = turtle.stamp()
>>> turtle.fd(50)
```

turtle.**clearstamp**(*stampid*)

Parameters

stampid – an integer, must be return value of previous *stamp()* call

Delete stamp with given *stampid*.

```
>>> turtle.position()
(150.00,-0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00,-0.00)
>>> turtle.clearstamp(astamp)
>>> turtle.position()
(200.00,-0.00)
```

turtle.**clearstamps**(*n=None*)

Parameters

n – an integer (or `None`)

Delete all or first/last *n* of turtle's stamps. If *n* is `None`, delete all stamps, if $n > 0$ delete first *n* stamps, else if $n < 0$ delete last *n* stamps.

```
>>> for i in range(8):
...     unused_stamp_id = turtle.stamp()
...     turtle.fd(30)
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

turtle.**undo**()

Undo (repeatedly) the last turtle action(s). Number of available undo actions is determined by the size of the undobuffer.

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

turtle.**speed**(*speed=None*)

Parameters
speed – an integer in the range 0..10 or a speedstring (see below)

Set the turtle's speed to an integer value in the range 0..10. If no argument is given, return current speed.

If input is a number greater than 10 or smaller than 0.5, speed is set to 0. Speedstrings are mapped to speedvalues as follows:

- "fastest": 0
- "fast": 10
- "normal": 6
- "slow": 3
- "slowest": 1

Speeds from 1 to 10 enforce increasingly faster animation of line drawing and turtle turning.

Attention: *speed* = 0 means that *no* animation takes place. forward/back makes turtle jump and likewise left/right make the turtle turn instantly.

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

Tell Turtle's state

turtle.**position**()

turtle.**pos**()

Return the turtle's current location (x,y) (as a *Vec2D* vector).

```
>>> turtle.pos()
(440.00,-0.00)
```

turtle.**towards**(*x*, *y=None*)

Parameters

- **x** – a number or a pair/vector of numbers or a turtle instance
- **y** – a number if *x* is a number, else None

Return the angle between the line from turtle position to position specified by (x,y), the vector or the other turtle. This depends on the turtle's start orientation which depends on the mode - "standard"/"world" or "logo".

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0,0)
225.0
```

turtle.**xcor**()

Return the turtle's x coordinate.

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28,76.60)
>>> print(round(turtle.xcor(), 5))
64.27876
```

turtle.**ycor**()

Return the turtle's y coordinate.

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print(turtle.pos())
(50.00,86.60)
>>> print(round(turtle.ycor(), 5))
86.60254
```

turtle.**heading**()

Return the turtle's current heading (value depends on the turtle mode, see *mode()*).

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

turtle.**distance**(*x*, *y=None*)

Parameters

- **x** – a number or a pair/vector of numbers or a turtle instance
- **y** – a number if *x* is a number, else None

Return the distance from the turtle to (x,y), the given vector, or the given other turtle, in turtle step units.

```
>>> turtle.home()
>>> turtle.distance(30,40)
50.0
>>> turtle.distance((30,40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

Settings for measurement

turtle.**degrees**(*fullcircle=360.0*)

Parameters
fullcircle – a number

Set angle measurement units, i.e. set number of "degrees" for a full circle. Default value is 360 degrees.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0

Change angle measurement unit to grad (also known as gon,
grade, or gradian and equals 1/100-th of the right angle.)
>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0
```

turtle.**radians**()

Set the angle measurement units to radians. Equivalent to `degrees(2*math.pi)`.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

Pen control

Drawing state

turtle.**pendown**()
turtle.**pd**()
turtle.**down**()

Pull the pen down – drawing when moving.

turtle.**penup**()
turtle.**pu**()
turtle.**up**()

Pull the pen up – no drawing when moving.

turtle.**pensize**(*width=None*)
turtle.**width**(*width=None*)

Parameters
width – a positive number

Set the line thickness to *width* or return it. If resizemode is set to "auto" and turtleshape is a polygon, that polygon is drawn with the same line thickness. If no argument is given, the current pensize is returned.

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)   # from here on lines of width 10 are drawn
```

`turtle.`**`pen`**(*pen=None*, ***pendict*)

Parameters

- **`pen`** – a dictionary with some or all of the below listed keys
- **`pendict`** – one or more keyword-arguments with the below listed keys as keywords

Return or set the pen's attributes in a "pen-dictionary" with the following key/value pairs:

- "shown": True/False
- "pendown": True/False
- "pencolor": color-string or color-tuple
- "fillcolor": color-string or color-tuple
- "pensize": positive number
- "speed": number in range 0..10
- "resizemode": "auto" or "user" or "noresize"
- "stretchfactor": (positive number, positive number)
- "outline": positive number
- "tilt": number

This dictionary can be used as argument for a subsequent call to *pen()* to restore the former pen-state. Moreover one or more of these attributes can be provided as keyword-arguments. This can be used to set several pen attributes in one statement.

```
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shearfactor', 0.0), ('shown', True), ('speed', 9),
 ('stretchfactor', (1.0, 1.0)), ('tilt', 0.0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", "")
>>> turtle.penup()
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow')]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red')]
```

`turtle.`**`isdown`**()

Return `True` if pen is down, `False` if it's up.

```
>>> turtle.penup()
>>> turtle.isdown()
False
```

(continues on next page)

(continued from previous page)

```
>>> turtle.pendown()
>>> turtle.isdown()
True
```

Color control

turtle.**pencolor**(**args*)

Return or set the pencolor.

Four input formats are allowed:

pencolor()
Return the current pencolor as color specification string or as a tuple (see example). May be used as input to another color/pencolor/fillcolor call.

pencolor(colorstring)
Set pencolor to *colorstring*, which is a Tk color specification string, such as `"red"`, `"yellow"`, or `"#33cc8c"`.

pencolor((r, g, b))
Set pencolor to the RGB color represented by the tuple of *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode, where colormode is either 1.0 or 255 (see *colormode()*).

pencolor(r, g, b)
Set pencolor to the RGB color represented by *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode.

If turtleshape is a polygon, the outline of that polygon is drawn with the newly set pencolor.

```
>>> colormode()
1.0
>>> turtle.pencolor()
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
>>> colormode(255)
>>> turtle.pencolor()
(51.0, 204.0, 140.0)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
```

turtle.**fillcolor**(**args*)

Return or set the fillcolor.

Four input formats are allowed:

fillcolor()
Return the current fillcolor as color specification string, possibly in tuple format (see example). May be used as input to another color/pencolor/fillcolor call.

fillcolor(colorstring)
Set fillcolor to *colorstring*, which is a Tk color specification string, such as `"red"`, `"yellow"`, or `"#33cc8c"`.

fillcolor((r, g, b))
Set fillcolor to the RGB color represented by the tuple of *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode, where colormode is either 1.0 or 255 (see *colormode()*).

fillcolor(r, g, b)
Set fillcolor to the RGB color represented by *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode.

If turtleshape is a polygon, the interior of that polygon is drawn with the newly set fillcolor.

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor((50, 193, 143))  # Integers, not floats
>>> turtle.fillcolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255.0, 255.0, 255.0)
```

turtle.**color**(**args*)
Return or set pencolor and fillcolor.

Several input formats are allowed. They use 0 to 3 arguments as follows:

color()
Return the current pencolor and the current fillcolor as a pair of color specification strings or tuples as returned by *pencolor()* and *fillcolor()*.

color(colorstring), color((r,g,b)), color(r,g,b)
Inputs as in *pencolor()*, set both, fillcolor and pencolor, to the given value.

color(colorstring1, colorstring2), color((r1,g1,b1), (r2,g2,b2))
Equivalent to `pencolor(colorstring1)` and `fillcolor(colorstring2)` and analogously if the other input format is used.

If turtleshape is a polygon, outline and interior of that polygon is drawn with the newly set colors.

```
>>> turtle.color("red", "green")
>>> turtle.color()
('red', 'green')
>>> color("#285078", "#a0c8f0")
>>> color()
((40.0, 80.0, 120.0), (160.0, 200.0, 240.0))
```

See also: Screen method *colormode()*.

Filling

turtle.**filling**()

Return fillstate (`True` if filling, `False` else).

```
>>> turtle.begin_fill()
>>> if turtle.filling():
...     turtle.pensize(5)
... else:
...     turtle.pensize(3)
```

turtle.**begin_fill**()

To be called just before drawing a shape to be filled.

turtle.**end_fill**()

Fill the shape drawn after the last call to *begin_fill()*.

Whether or not overlap regions for self-intersecting polygons or multiple shapes are filled depends on the operating system graphics, type of overlap, and number of overlaps. For example, the Turtle star above may be either all yellow or have some white regions.

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

More drawing control

turtle.**reset**()

Delete the turtle's drawings from the screen, re-center the turtle and set variables to the default values.

```
>>> turtle.goto(0,-22)
>>> turtle.left(100)
>>> turtle.position()
(0.00,-22.00)
>>> turtle.heading()
100.0
>>> turtle.reset()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**clear**()

Delete the turtle's drawings from the screen. Do not move turtle. State and position of the turtle as well as drawings of other turtles are not affected.

turtle.**write**(*arg*, *move=False*, *align='left'*, *font=('Arial', 8, 'normal')*)

Parameters

- **arg** – object to be written to the TurtleScreen
- **move** – True/False
- **align** – one of the strings "left", "center" or right"

- **font** – a triple (fontname, fontsize, fonttype)

Write text - the string representation of *arg* - at the current turtle position according to *align* ("left", "center" or "right") and with the given font. If *move* is true, the pen is moved to the bottom-right corner of the text. By default, *move* is `False`.

```
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

Turtle state

Visibility

`turtle.`**`hideturtle`**`()`

`turtle.`**`ht`**`()`

Make the turtle invisible. It's a good idea to do this while you're in the middle of doing some complex drawing, because hiding the turtle speeds up the drawing observably.

```
>>> turtle.hideturtle()
```

`turtle.`**`showturtle`**`()`

`turtle.`**`st`**`()`

Make the turtle visible.

```
>>> turtle.showturtle()
```

`turtle.`**`isvisible`**`()`

Return `True` if the Turtle is shown, `False` if it's hidden.

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

Appearance

`turtle.`**`shape`**`(`*name=None*`)`

Parameters

name – a string which is a valid shapename

Set turtle shape to shape with given *name* or, if name is not given, return name of current shape. Shape with *name* must exist in the TurtleScreen's shape dictionary. Initially there are the following polygon shapes: "arrow", "turtle", "circle", "square", "triangle", "classic". To learn about how to deal with shapes see Screen method *register_shape()*.

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

turtle.**resizemode**(*rmode=None*)

Parameters
rmode – one of the strings “auto”, “user”, “noresize”

Set resizemode to one of the values: “auto”, “user”, “noresize”. If *rmode* is not given, return current resizemode. Different resizemodes have the following effects:

- “auto”: adapts the appearance of the turtle corresponding to the value of pensize.
- “user”: adapts the appearance of the turtle according to the values of stretchfactor and outlinewidth (outline), which are set by *shapesize()*.
- “noresize”: no adaption of the turtle’s appearance takes place.

`resizemode("user")` is called by *shapesize()* when used with arguments.

```
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

turtle.**shapesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)
turtle.**turtlesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)

Parameters

- **stretch_wid** – positive number
- **stretch_len** – positive number
- **outline** – positive number

Return or set the pen’s attributes x/y-stretchfactors and/or outline. Set resizemode to “user”. If and only if resizemode is set to “user”, the turtle will be displayed stretched according to its stretchfactors: *stretch_wid* is stretchfactor perpendicular to its orientation, *stretch_len* is stretchfactor in direction of its orientation, *outline* determines the width of the shape’s outline.

```
>>> turtle.shapesize()
(1.0, 1.0, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

turtle.**shearfactor**(*shear=None*)

Parameters
shear – number (optional)

Set or return the current shearfactor. Shear the turtleshape according to the given shearfactor shear, which is the tangent of the shear angle. Do *not* change the turtle’s heading (direction of movement). If shear is not given: return the current shearfactor, i. e. the tangent of the shear angle, by which lines parallel to the heading of the turtle are sheared.

```
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.shearfactor(0.5)
>>> turtle.shearfactor()
0.5
```

turtle.**tilt**(*angle*)

Parameters
angle – a number

Rotate the turtleshape by *angle* from its current tilt-angle, but do *not* change the turtle's heading (direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

turtle.**tiltangle**(*angle=None*)

Parameters
angle – a number (optional)

Set or return the current tilt-angle. If angle is given, rotate the turtleshape to point in the direction specified by angle, regardless of its current tilt-angle. Do *not* change the turtle's heading (direction of movement). If angle is not given: return the current tilt-angle, i. e. the angle between the orientation of the turtleshape and the heading of the turtle (its direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

turtle.**shapetransform**(*t11=None*, *t12=None*, *t21=None*, *t22=None*)

Parameters

- **t11** – a number (optional)
- **t12** – a number (optional)
- **t21** – a number (optional)
- **t12** – a number (optional)

Set or return the current transformation matrix of the turtle shape.

If none of the matrix elements are given, return the transformation matrix as a tuple of 4 elements. Otherwise set the given elements and transform the turtleshape according to the matrix consisting of first row t11, t12 and second row t21, t22. The determinant t11 * t22 - t12 * t21 must not be zero, otherwise an error is raised. Modify stretchfactor, shearfactor and tiltangle according to the given matrix.

```
>>> turtle = Turtle()
>>> turtle.shape("square")
>>> turtle.shapesize(4,2)
>>> turtle.shearfactor(-0.5)
>>> turtle.shapetransform()
(4.0, -1.0, -0.0, 2.0)
```

turtle.**get_shapepoly**()

Return the current shape polygon as tuple of coordinate pairs. This can be used to define a new shape or components of a compound shape.

```
>>> turtle.shape("square")
>>> turtle.shapetransform(4, -1, 0, 2)
>>> turtle.get_shapepoly()
((50, -20), (30, 20), (-50, 20), (-30, -20))
```

Using events

turtle.**onclick**(*fun*, *btn=1*, *add=None*)

Parameters

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – `True` or `False` – if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-click events on this turtle. If *fun* is `None`, existing bindings are removed. Example for the anonymous turtle, i.e. the procedural way:

```
>>> def turn(x, y):
...     left(180)
...
>>> onclick(turn)  # Now clicking into the turtle will turn it.
>>> onclick(None)  # event-binding will be removed
```

turtle.**onrelease**(*fun*, *btn=1*, *add=None*)

Parameters

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – `True` or `False` – if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-button-release events on this turtle. If *fun* is `None`, existing bindings are removed.

```
>>> class MyTurtle(Turtle):
...     def glow(self,x,y):
...         self.fillcolor("red")
...     def unglow(self,x,y):
...         self.fillcolor("")
```

(continues on next page)

(continued from previous page)

```
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)      # clicking on turtle turns fillcolor red,
>>> turtle.onrelease(turtle.unglow) # releasing turns it to transparent.
```

turtle.**ondrag**(*fun*, *btn=1*, *add=None*)

Parameters

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – `True` or `False` – if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-move events on this turtle. If *fun* is `None`, existing bindings are removed.

Remark: Every sequence of mouse-move-events on a turtle is preceded by a mouse-click event on that turtle.

```
>>> turtle.ondrag(turtle.goto)
```

Subsequently, clicking and dragging the Turtle will move it across the screen thereby producing handdrawings (if pen is down).

Special Turtle methods

turtle.**begin_poly**()

Start recording the vertices of a polygon. Current turtle position is first vertex of polygon.

turtle.**end_poly**()

Stop recording the vertices of a polygon. Current turtle position is last vertex of polygon. This will be connected with the first vertex.

turtle.**get_poly**()

Return the last recorded polygon.

```
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

turtle.**clone**()

Create and return a clone of the turtle with same position, heading and turtle properties.

```
>>> mick = Turtle()
>>> joe = mick.clone()
```

turtle.**getturtle**()

turtle.**getpen**()

Return the Turtle object itself. Only reasonable use: as a function to return the "anonymous turtle":

```
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

turtle.**getscreen**()

Return the *TurtleScreen* object the turtle is drawing on. TurtleScreen methods can then be called for that object.

```
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

turtle.**setundobuffer**(*size*)

Parameters

size – an integer or None

Set or disable undobuffer. If *size* is an integer, an empty undobuffer of given size is installed. *size* gives the maximum number of turtle actions that can be undone by the *undo()* method/function. If *size* is None, the undobuffer is disabled.

```
>>> turtle.setundobuffer(42)
```

turtle.**undobufferentries**()

Return number of entries in the undobuffer.

```
>>> while undobufferentries():
...     undo()
```

Compound shapes

To use compound turtle shapes, which consist of several polygons of different color, you must use the helper class *Shape* explicitly as described below:

1. Create an empty Shape object of type "compound".
2. Add as many components to this object as desired, using the *addcomponent()* method.

   For example:

```
>>> s = Shape("compound")
>>> poly1 = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s.addcomponent(poly1, "red", "blue")
>>> poly2 = ((0,0),(10,-5),(-10,-5))
>>> s.addcomponent(poly2, "blue", "red")
```

3. Now add the Shape to the Screen's shapelist and use it:

```
>>> register_shape("myshape", s)
>>> shape("myshape")
```

Note

The *Shape* class is used internally by the *register_shape()* method in different ways. The application programmer has to deal with the Shape class *only* when using compound shapes like shown above!

24.1.7 Methods of TurtleScreen/Screen and corresponding functions

Most of the examples in this section refer to a TurtleScreen instance called `screen`.

Window control

`turtle.`**`bgcolor`**`(`*args*`)`

Parameters
`args` – a color string or three numbers in the range 0..colormode or a 3-tuple of such numbers

Set or return background color of the TurtleScreen.

```
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128.0, 0.0, 128.0)
```

`turtle.`**`bgpic`**`(`*picname=None*`)`

Parameters
`picname` – a string, name of a gif-file or `"nopic"`, or `None`

Set background image or return name of current backgroundimage. If *picname* is a filename, set the corresponding image as background. If *picname* is `"nopic"`, delete background image, if present. If *picname* is `None`, return the filename of the current backgroundimage.

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```

`turtle.`**`clear`**`()`

Note

This TurtleScreen method is available as a global function only under the name `clearscreen`. The global function `clear` is a different one derived from the Turtle method `clear`.

`turtle.`**`clearscreen`**`()`

Delete all drawings and all turtles from the TurtleScreen. Reset the now empty TurtleScreen to its initial state: white background, no background image, no event bindings and tracing on.

turtle.**reset**()

> **Note**
>
> This TurtleScreen method is available as a global function only under the name `resetscreen`. The global function `reset` is another one derived from the Turtle method `reset`.

turtle.**resetscreen**()

Reset all Turtles on the Screen to their initial state.

turtle.**screensize**(*canvwidth=None*, *canvheight=None*, *bg=None*)

Parameters

- **canvwidth** – positive integer, new width of canvas in pixels
- **canvheight** – positive integer, new height of canvas in pixels
- **bg** – colorstring or color-tuple, new background color

If no arguments are given, return current (canvaswidth, canvasheight). Else resize the canvas the turtles are drawing on. Do not alter the drawing window. To observe hidden parts of the canvas, use the scrollbars. With this method, one can make visible those parts of a drawing which were outside the canvas before.

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500)
```

e.g. to search for an erroneously escaped turtle ;-)

turtle.**setworldcoordinates**(*llx*, *lly*, *urx*, *ury*)

Parameters

- **llx** – a number, x-coordinate of lower left corner of canvas
- **lly** – a number, y-coordinate of lower left corner of canvas
- **urx** – a number, x-coordinate of upper right corner of canvas
- **ury** – a number, y-coordinate of upper right corner of canvas

Set up user-defined coordinate system and switch to mode “world” if necessary. This performs a `screen.reset()`. If mode “world” is already active, all drawings are redrawn according to the new coordinates.

ATTENTION: in user-defined coordinate systems angles may appear distorted.

```
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...     left(10)
...
>>> for _ in range(8):
...     left(45); fd(2)   # a regular octagon
```

Animation control

turtle.**delay**(*delay=None*)

Parameters
delay – positive integer

Set or return the drawing *delay* in milliseconds. (This is approximately the time interval between two consecutive canvas updates.) The longer the drawing delay, the slower the animation.

Optional argument:

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

turtle.**tracer**(*n=None*, *delay=None*)

Parameters

- **n** – nonnegative integer
- **delay** – nonnegative integer

Turn turtle animation on/off and set delay for update drawings. If *n* is given, only each n-th regular screen update is really performed. (Can be used to accelerate the drawing of complex graphics.) When called without arguments, returns the currently stored value of n. Second argument sets delay value (see *delay()*).

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

turtle.**update**()

Perform a TurtleScreen update. To be used when tracer is turned off.

See also the RawTurtle/Turtle method *speed()*.

Using screen events

turtle.**listen**(*xdummy=None*, *ydummy=None*)

Set focus on TurtleScreen (in order to collect key-events). Dummy arguments are provided in order to be able to pass *listen()* to the onclick method.

turtle.**onkey**(*fun*, *key*)

turtle.**onkeyrelease**(*fun*, *key*)

Parameters

- **fun** – a function with no arguments or `None`
- **key** – a string: key (e.g. “a”) or key-symbol (e.g. “space”)

Bind *fun* to key-release event of key. If *fun* is `None`, event bindings are removed. Remark: in order to be able to register key-events, TurtleScreen must have the focus. (See method *listen()*.)

```
>>> def f():
...     fd(50)
...     lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onkeypress**(*fun*, *key=None*)

Parameters

- **fun** – a function with no arguments or `None`
- **key** – a string: key (e.g. "a") or key-symbol (e.g. "space")

Bind *fun* to key-press event of key if key is given, or to any key-press-event if no key is given. Remark: in order to be able to register key-events, TurtleScreen must have focus. (See method *listen()*.)

```
>>> def f():
...     fd(50)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onclick**(*fun*, *btn=1*, *add=None*)

turtle.**onscreenclick**(*fun*, *btn=1*, *add=None*)

Parameters

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – `True` or `False` – if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-click events on this screen. If *fun* is `None`, existing bindings are removed.

Example for a TurtleScreen instance named `screen` and a Turtle instance named `turtle`:

```
>>> screen.onclick(turtle.goto) # Subsequently clicking into the TurtleScreen will
>>>                             # make the turtle move to the clicked point.
>>> screen.onclick(None)        # remove event binding again
```

Note

This TurtleScreen method is available as a global function only under the name `onscreenclick`. The global function `onclick` is another one derived from the Turtle method `onclick`.

turtle.**ontimer**(*fun*, *t=0*)

Parameters

- **fun** – a function with no arguments
- **t** – a number >= 0

Install a timer that calls *fun* after *t* milliseconds.

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()   ### makes the turtle march around
>>> running = False
```

turtle.**mainloop**()

turtle.**done**()

Starts event loop - calling Tkinter's mainloop function. Must be the last statement in a turtle graphics program. Must *not* be used if a script is run from within IDLE in -n mode (No subprocess) - for interactive use of turtle graphics.

```
>>> screen.mainloop()
```

Input methods

turtle.**textinput**(*title*, *prompt*)

Parameters

- **title** – string
- **prompt** – string

Pop up a dialog window for input of a string. Parameter title is the title of the dialog window, prompt is a text mostly describing what information to input. Return the string input. If the dialog is canceled, return `None`.

```
>>> screen.textinput("NIM", "Name of first player:")
```

turtle.**numinput**(*title*, *prompt*, *default=None*, *minval=None*, *maxval=None*)

Parameters

- **title** – string
- **prompt** – string
- **default** – number (optional)
- **minval** – number (optional)
- **maxval** – number (optional)

Pop up a dialog window for input of a number. title is the title of the dialog window, prompt is a text mostly describing what numerical information to input. default: default value, minval: minimum value for input, maxval: maximum value for input. The number input must be in the range minval .. maxval if these are given. If not, a hint is issued and the dialog remains open for correction. Return the number input. If the dialog is canceled, return `None`.

```
>>> screen.numinput("Poker", "Your stakes:", 1000, minval=10, maxval=10000)
```

Settings and special methods

turtle.**mode**(*mode=None*)

Parameters
mode – one of the strings “standard”, “logo” or “world”

Set turtle mode (“standard”, “logo” or “world”) and perform reset. If mode is not given, current mode is returned.

Mode “standard” is compatible with old *turtle*. Mode “logo” is compatible with most Logo turtle graphics. Mode “world” uses user-defined “world coordinates”. **Attention**: in this mode angles appear distorted if x/y unit-ratio doesn’t equal 1.

Mode	Initial turtle heading	positive angles
“standard”	to the right (east)	counterclockwise
“logo”	upward (north)	clockwise

```
>>> mode("logo")   # resets turtle heading to north
>>> mode()
'logo'
```

turtle.**colormode**(*cmode=None*)

Parameters
cmode – one of the values 1.0 or 255

Return the colormode or set it to 1.0 or 255. Subsequently *r*, *g*, *b* values of color triples have to be in the range 0..*cmode*.

```
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
     ...
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

turtle.**getcanvas**()

Return the Canvas of this TurtleScreen. Useful for insiders who know what to do with a Tkinter Canvas.

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas object ...>
```

turtle.**getshapes**()

Return a list of names of all currently available turtle shapes.

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

turtle.**register_shape**(*name*, *shape=None*)

turtle.**addshape**(*name*, *shape=None*)

There are three different ways to call this function:

(1) *name* is the name of a gif-file and *shape* is `None`: Install the corresponding image shape.

```
>>> screen.register_shape("turtle.gif")
```

> **Note**
>
> Image shapes *do not* rotate when turning the turtle, so they do not display the heading of the turtle!

(2) *name* is an arbitrary string and *shape* is a tuple of pairs of coordinates: Install the corresponding polygon shape.

```
>>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3)))
```

(3) *name* is an arbitrary string and *shape* is a (compound) *Shape* object: Install the corresponding compound shape.

Add a turtle shape to TurtleScreen's shapelist. Only thusly registered shapes can be used by issuing the command `shape(shapename)`.

turtle.**turtles**()

Return the list of turtles on the screen.

```
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

turtle.**window_height**()

Return the height of the turtle window.

```
>>> screen.window_height()
480
```

turtle.**window_width**()

Return the width of the turtle window.

```
>>> screen.window_width()
640
```

Methods specific to Screen, not inherited from TurtleScreen

turtle.**bye**()

Shut the turtlegraphics window.

turtle.**exitonclick**()

Bind `bye()` method to mouse clicks on the Screen.

If the value "using_IDLE" in the configuration dictionary is `False` (default value), also enter mainloop. Remark: If IDLE with the `-n` switch (no subprocess) is used, this value should be set to `True` in `turtle.cfg`. In this case IDLE's own mainloop is active also for the client script.

turtle.**setup**(*width=_CFG['width']*, *height=_CFG['height']*, *startx=_CFG['leftright']*, *starty=_CFG['topbottom']*)

Set the size and position of the main window. Default values of arguments are stored in the configuration dictionary and can be changed via a `turtle.cfg` file.

Parameters

- **width** – if an integer, a size in pixels, if a float, a fraction of the screen; default is 50% of screen
- **height** – if an integer, the height in pixels, if a float, a fraction of the screen; default is 75% of screen
- **startx** – if positive, starting position in pixels from the left edge of the screen, if negative from the right edge, if `None`, center window horizontally
- **starty** – if positive, starting position in pixels from the top edge of the screen, if negative from the bottom edge, if `None`, center window vertically

```
>>> screen.setup (width=200, height=200, startx=0, starty=0)
>>>              # sets window to 200x200 pixels, in upper left of screen
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>>              # sets window to 75% of screen by 50% of screen and centers
```

turtle.**title**(*titlestring*)

Parameters

titlestring – a string that is shown in the titlebar of the turtle graphics window

Set title of turtle window to *titlestring*.

```
>>> screen.title("Welcome to the turtle zoo!")
```

24.1.8 Public classes

class turtle.**RawTurtle**(*canvas*)

class turtle.**RawPen**(*canvas*)

Parameters

canvas – a `tkinter.Canvas`, a *ScrolledCanvas* or a *TurtleScreen*

Create a turtle. The turtle has all methods described above as "methods of Turtle/RawTurtle".

class turtle.**Turtle**

Subclass of RawTurtle, has the same interface but draws on a default *Screen* object created automatically when needed for the first time.

class turtle.**TurtleScreen**(*cv*)

Parameters

cv – a `tkinter.Canvas`

Provides screen oriented methods like *bgcolor()* etc. that are described above.

class turtle.**Screen**

Subclass of TurtleScreen, with *four methods added*.

class turtle.**ScrolledCanvas**(*master*)

Parameters
master – some Tkinter widget to contain the ScrolledCanvas, i.e. a Tkinter-canvas with scrollbars added

Used by class Screen, which thus automatically provides a ScrolledCanvas as playground for the turtles.

class turtle.**Shape**(*type_*, *data*)

Parameters
type_ – one of the strings "polygon", "image", "compound"

Data structure modeling shapes. The pair `(type_, data)` must follow this specification:

*type_*	*data*
"polygon"	a polygon-tuple, i.e. a tuple of pairs of coordinates
"image"	an image (in this form only used internally!)
"compound"	`None` (a compound shape has to be constructed using the *`addcomponent()`* method)

addcomponent(*poly*, *fill*, *outline=None*)

Parameters

- **poly** – a polygon, i.e. a tuple of pairs of numbers
- **fill** – a color the *poly* will be filled with
- **outline** – a color for the poly's outline (if given)

Example:

```
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... add more components and then use register_shape()
```

See *Compound shapes*.

class turtle.**Vec2D**(*x*, *y*)

A two-dimensional vector class, used as a helper class for implementing turtle graphics. May be useful for turtle graphics programs too. Derived from tuple, so a vector is a tuple!

Provides (for *a*, *b* vectors, *k* number):

- `a + b` vector addition
- `a - b` vector subtraction
- `a * b` inner product
- `k * a` and `a * k` multiplication with scalar
- `abs(a)` absolute value of a
- `a.rotate(angle)` rotation

24.1.9 Explanation

A turtle object draws on a screen object, and there a number of key classes in the turtle object-oriented interface that can be used to create them and relate them to each other.

A *Turtle* instance will automatically create a *Screen* instance if one is not already present.

Turtle is a subclass of *RawTurtle*, which *doesn't* automatically create a drawing surface - a *canvas* will need to be provided or created for it. The *canvas* can be a tkinter.Canvas, *ScrolledCanvas* or *TurtleScreen*.

TurtleScreen is the basic drawing surface for a turtle. *Screen* is a subclass of TurtleScreen, and includes *some additional methods* for managing its appearance (including size and title) and behaviour. TurtleScreen's constructor needs a tkinter.Canvas or a *ScrolledCanvas* as an argument.

The functional interface for turtle graphics uses the various methods of Turtle and TurtleScreen/Screen. Behind the scenes, a screen object is automatically created whenever a function derived from a Screen method is called. Similarly, a turtle object is automatically created whenever any of the functions derived from a Turtle method is called.

To use multiple turtles on a screen, the object-oriented interface must be used.

24.1.10 Help and configuration

How to use help

The public methods of the Screen and Turtle classes are documented extensively via docstrings. So these can be used as online-help via the Python help facilities:

- When using IDLE, tooltips show the signatures and first lines of the docstrings of typed in function-/method calls.
- Calling *help()* on methods or functions displays the docstrings:

```
>>> help(Screen.bgcolor)
Help on method bgcolor in module turtle:

bgcolor(self, *args) unbound turtle.Screen method
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.


    >>> screen.bgcolor("orange")
    >>> screen.bgcolor()
    "orange"
    >>> screen.bgcolor(0.5,0,0.5)
    >>> screen.bgcolor()
    "#800080"

>>> help(Turtle.penup)
Help on method penup in module turtle:

penup(self) unbound turtle.Turtle method
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument
```

(continues on next page)

(continued from previous page)

```
    >>> turtle.penup()
```

- The docstrings of the functions which are derived from methods have a modified form:

```
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.

    Example::

      >>> bgcolor("orange")
      >>> bgcolor()
      "orange"
      >>> bgcolor(0.5,0,0.5)
      >>> bgcolor()
      "#800080"

>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    Example:
    >>> penup()
```

These modified docstrings are created automatically together with the function definitions that are derived from the methods at import time.

Translation of docstrings into different languages

There is a utility to create a dictionary the keys of which are the method names and the values of which are the docstrings of the public methods of the classes Screen and Turtle.

turtle.**write_docstringdict**(*filename='turtle_docstringdict'*)

Parameters
filename – a string, used as filename

Create and write docstring-dictionary to a Python script with the given filename. This function has to be called explicitly (it is not used by the turtle graphics classes). The docstring dictionary will be written to the Python script *filename*.py. It is intended to serve as a template for translation of the docstrings into different languages.

If you (or your students) want to use *turtle* with online help in your native language, you have to translate the docstrings and save the resulting file as e.g. turtle_docstringdict_german.py.

If you have an appropriate entry in your `turtle.cfg` file this dictionary will be read in at import time and will replace the original English docstrings.

At the time of this writing there are docstring dictionaries in German and in Italian. (Requests please to glingl@aon.at.)

How to configure Screen and Turtles

The built-in default configuration mimics the appearance and behaviour of the old turtle module in order to retain best possible compatibility with it.

If you want to use a different configuration which better reflects the features of this module or which better fits to your needs, e.g. for use in a classroom, you can prepare a configuration file `turtle.cfg` which will be read at import time and modify the configuration according to its settings.

The built in configuration would correspond to the following `turtle.cfg`:

```
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

Short explanation of selected entries:

- The first four lines correspond to the arguments of the *Screen.setup* method.
- Line 5 and 6 correspond to the arguments of the method *Screen.screensize*.
- *shape* can be any of the built-in shapes, e.g: arrow, turtle, etc. For more info try `help(shape)`.
- If you want to use no fill color (i.e. make the turtle transparent), you have to write `fillcolor = ""` (but all nonempty strings must not have quotes in the cfg file).
- If you want to reflect the turtle its state, you have to use `resizemode = auto`.
- If you set e.g. `language = italian` the docstringdict `turtle_docstringdict_italian.py` will be loaded at import time (if present on the import path, e.g. in the same directory as *turtle*).
- The entries *exampleturtle* and *examplescreen* define the names of these objects as they occur in the docstrings. The transformation of method-docstrings to function-docstrings will delete these names from the docstrings.
- *using_IDLE*: Set this to `True` if you regularly work with IDLE and its `-n` switch ("no subprocess"). This will prevent *exitonclick()* to enter the mainloop.

There can be a `turtle.cfg` file in the directory where *turtle* is stored and an additional one in the current working directory. The latter will override the settings of the first one.

The `Lib/turtledemo` directory contains a `turtle.cfg` file. You can study it as an example and see its effects when running the demos (preferably not from within the demo-viewer).

24.1.11 `turtledemo` — Demo scripts

The *turtledemo* package includes a set of demo scripts. These scripts can be run and viewed using the supplied demo viewer as follows:

```
python -m turtledemo
```

Alternatively, you can run the demo scripts individually. For example,

```
python -m turtledemo.bytedesign
```

The *turtledemo* package directory contains:

- A demo viewer `__main__.py` which can be used to view the sourcecode of the scripts and run them at the same time.
- Multiple scripts demonstrating different features of the *turtle* module. Examples can be accessed via the Examples menu. They can also be run standalone.
- A `turtle.cfg` file which serves as an example of how to write and use such files.

The demo scripts are:

Name	Description	Features
bytedesign	complex classical turtle graphics pattern	*tracer()*, delay, *update()*
chaos	graphs Verhulst dynamics, shows that computer's computations can generate results sometimes against the common sense expectations	world coordinates
clock	analog clock showing time of your computer	turtles as clock's hands, ontimer
colormixer	experiment with r, g, b	*ondrag()*
forest	3 breadth-first trees	randomization
fractalcurves	Hilbert & Koch curves	recursion
lindenmayer	ethnomathematics (indian kolams)	L-System
minimal_hanoi	Towers of Hanoi	Rectangular Turtles as Hanoi discs (shape, shapesize)
nim	play the classical nim game with three heaps of sticks against the computer.	turtles as nimsticks, event driven (mouse, keyboard)
paint	super minimalistic drawing program	*onclick()*
peace	elementary	turtle: appearance and animation
penrose	aperiodic tiling with kites and darts	*stamp()*
planet_and_moon	simulation of gravitational system	compound shapes, *Vec2D*
rosette	a pattern from the wikipedia article on turtle graphics	*clone()*, *undo()*
round_dance	dancing turtles rotating pairwise in opposite direction	compound shapes, clone shapesize, tilt, get_shapepoly, update
sorting_animate	visual demonstration of different sorting methods	simple alignment, randomization
tree	a (graphical) breadth first tree (using generators)	*clone()*
two_canvases	simple design	turtles on two canvases
yinyang	another elementary example	*circle()*

Have fun!

24.1.12 Changes since Python 2.6

- The methods *Turtle.tracer*, *Turtle.window_width* and *Turtle.window_height* have been eliminated. Methods with these names and functionality are now available only as methods of *Screen*. The functions derived from these remain available. (In fact already in Python 2.6 these methods were merely duplications of the corresponding *TurtleScreen*/*Screen* methods.)
- The method `Turtle.fill()` has been eliminated. The behaviour of *begin_fill()* and *end_fill()* have changed slightly: now every filling process must be completed with an `end_fill()` call.
- A method *Turtle.filling* has been added. It returns a boolean value: `True` if a filling process is under way, `False` otherwise. This behaviour corresponds to a `fill()` call without arguments in Python 2.6.

24.1.13 Changes since Python 3.0

- The *Turtle* methods *shearfactor()*, *shapetransform()* and *get_shapepoly()* have been added. Thus the full range of regular linear transforms is now available for transforming turtle shapes. *tiltangle()* has been enhanced in functionality: it now can be used to get or set the tilt angle.
- The *Screen* method *onkeypress()* has been added as a complement to *onkey()*. As the latter binds actions to the key release event, an alias: *onkeyrelease()* was also added for it.
- The method *Screen.mainloop* has been added, so there is no longer a need to use the standalone *mainloop()* function when working with *Screen* and *Turtle* objects.
- Two input methods have been added: *Screen.textinput* and *Screen.numinput*. These pop up input dialogs and return strings and numbers respectively.
- Two example scripts `tdemo_nim.py` and `tdemo_round_dance.py` have been added to the `Lib/turtledemo` directory.

24.2 cmd — Support for line-oriented command interpreters

Source code: Lib/cmd.py

The *Cmd* class provides a simple framework for writing line-oriented command interpreters. These are often useful for test harnesses, administrative tools, and prototypes that will later be wrapped in a more sophisticated interface.

class cmd.**Cmd**(*completekey='tab'*, *stdin=None*, *stdout=None*)

A *Cmd* instance or subclass instance is a line-oriented interpreter framework. There is no good reason to instantiate *Cmd* itself; rather, it's useful as a superclass of an interpreter class you define yourself in order to inherit *Cmd*'s methods and encapsulate action methods.

The optional argument *completekey* is the *readline* name of a completion key; it defaults to `Tab`. If *completekey* is not *None* and *readline* is available, command completion is done automatically.

The default, `'tab'`, is treated specially, so that it refers to the `Tab` key on every *readline.backend*. Specifically, if *readline.backend* is `editline`, `Cmd` will use `'^I'` instead of `'tab'`. Note that other values are not treated this way, and might only work with a specific backend.

The optional arguments *stdin* and *stdout* specify the input and output file objects that the Cmd instance or subclass instance will use for input and output. If not specified, they will default to *sys.stdin* and *sys.stdout*.

If you want a given *stdin* to be used, make sure to set the instance's *use_rawinput* attribute to `False`, otherwise *stdin* will be ignored.

Changed in version 3.13: `completekey='tab'` is replaced by `'^I'` for `editline`.

24.2.1 Cmd Objects

A *Cmd* instance has the following methods:

Cmd.**cmdloop**(*intro=None*)

Repeatedly issue a prompt, accept input, parse an initial prefix off the received input, and dispatch to action methods, passing them the remainder of the line as argument.

The optional argument is a banner or intro string to be issued before the first prompt (this overrides the *intro* class attribute).

If the *readline* module is loaded, input will automatically inherit **bash**-like history-list editing (e.g. `Control-P` scrolls back to the last command, `Control-N` forward to the next one, `Control-F` moves the cursor to the right non-destructively, `Control-B` moves the cursor to the left non-destructively, etc.).

An end-of-file on input is passed back as the string `'EOF'`.

An interpreter instance will recognize a command name `foo` if and only if it has a method `do_foo()`. As a special case, a line beginning with the character `'?'` is dispatched to the method *do_help()*. As another special case, a line beginning with the character `'!'` is dispatched to the method `do_shell()` (if such a method is defined).

This method will return when the *postcmd()* method returns a true value. The *stop* argument to *postcmd()* is the return value from the command's corresponding `do_*()` method.

If completion is enabled, completing commands will be done automatically, and completing of commands args is done by calling `complete_foo()` with arguments *text*, *line*, *begidx*, and *endidx*. *text* is the string prefix we are attempting to match: all returned matches must begin with it. *line* is the current input line with leading whitespace removed, *begidx* and *endidx* are the beginning and ending indexes of the prefix text, which could be used to provide different completion depending upon which position the argument is in.

`Cmd.`**`do_help`**(*arg*)

All subclasses of *Cmd* inherit a predefined `do_help()`. This method, called with an argument `'bar'`, invokes the corresponding method `help_bar()`, and if that is not present, prints the docstring of `do_bar()`, if available. With no argument, `do_help()` lists all available help topics (that is, all commands with corresponding `help_*()` methods or commands that have docstrings), and also lists any undocumented commands.

`Cmd.`**`onecmd`**(*str*)

Interpret the argument as though it had been typed in response to the prompt. This may be overridden, but should not normally need to be; see the *precmd()* and *postcmd()* methods for useful execution hooks. The return value is a flag indicating whether interpretation of commands by the interpreter should stop. If there is a `do_*()` method for the command *str*, the return value of that method is returned, otherwise the return value from the *default()* method is returned.

`Cmd.`**`emptyline`**()

Method called when an empty line is entered in response to the prompt. If this method is not overridden, it repeats the last nonempty command entered.

`Cmd.`**`default`**(*line*)

Method called on an input line when the command prefix is not recognized. If this method is not overridden, it prints an error message and returns.

`Cmd.`**`completedefault`**(*text*, *line*, *begidx*, *endidx*)

Method called to complete an input line when no command-specific `complete_*()` method is available. By default, it returns an empty list.

`Cmd.`**`columnize`**(*list*, *displaywidth=80*)

Method called to display a list of strings as a compact set of columns. Each column is only as wide as necessary. Columns are separated by two spaces for readability.

`Cmd.`**`precmd`**(*line*)

Hook method executed just before the command line *line* is interpreted, but after the input prompt is generated and issued. This method is a stub in *Cmd*; it exists to be overridden by subclasses. The return value is used as the command which will be executed by the *onecmd()* method; the *precmd()* implementation may re-write the command or simply return *line* unchanged.

`Cmd.`**`postcmd`**(*stop*, *line*)

Hook method executed just after a command dispatch is finished. This method is a stub in *Cmd*; it exists to be overridden by subclasses. *line* is the command line which was executed, and *stop* is a flag which indicates whether

execution will be terminated after the call to *postcmd()*; this will be the return value of the *onecmd()* method. The return value of this method will be used as the new value for the internal flag which corresponds to *stop*; returning false will cause interpretation to continue.

Cmd.**preloop**()

Hook method executed once when *cmdloop()* is called. This method is a stub in *Cmd*; it exists to be overridden by subclasses.

Cmd.**postloop**()

Hook method executed once when *cmdloop()* is about to return. This method is a stub in *Cmd*; it exists to be overridden by subclasses.

Instances of *Cmd* subclasses have some public instance variables:

Cmd.**prompt**

The prompt issued to solicit input.

Cmd.**identchars**

The string of characters accepted for the command prefix.

Cmd.**lastcmd**

The last nonempty command prefix seen.

Cmd.**cmdqueue**

A list of queued input lines. The cmdqueue list is checked in *cmdloop()* when new input is needed; if it is nonempty, its elements will be processed in order, as if entered at the prompt.

Cmd.**intro**

A string to issue as an intro or banner. May be overridden by giving the *cmdloop()* method an argument.

Cmd.**doc_header**

The header to issue if the help output has a section for documented commands.

Cmd.**misc_header**

The header to issue if the help output has a section for miscellaneous help topics (that is, there are `help_*()` methods without corresponding `do_*()` methods).

Cmd.**undoc_header**

The header to issue if the help output has a section for undocumented commands (that is, there are `do_*()` methods without corresponding `help_*()` methods).

Cmd.**ruler**

The character used to draw separator lines under the help-message headers. If empty, no ruler line is drawn. It defaults to `'='`.

Cmd.**use_rawinput**

A flag, defaulting to true. If true, *cmdloop()* uses *input()* to display a prompt and read the next command; if false, *sys.stdout.write()* and *sys.stdin.readline()* are used. (This means that by importing *readline*, on systems that support it, the interpreter will automatically support **Emacs**-like line editing and command-history keystrokes.)

24.2.2 Cmd Example

The *cmd* module is mainly useful for building custom shells that let a user work with a program interactively.

This section presents a simple example of how to build a shell around a few of the commands in the *turtle* module.

Basic turtle commands such as *forward()* are added to a *Cmd* subclass with method named `do_forward()`. The argument is converted to a number and dispatched to the turtle module. The docstring is used in the help utility provided by the shell.

The example also includes a basic record and playback facility implemented with the *precmd()* method which is responsible for converting the input to lowercase and writing the commands to a file. The `do_playback()` method reads the file and adds the recorded commands to the *cmdqueue* for immediate playback:

```
import cmd, sys
from turtle import *

class TurtleShell(cmd.Cmd):
    intro = 'Welcome to the turtle shell.   Type help or ? to list commands.\n'
    prompt = '(turtle) '
    file = None

    # ----- basic turtle commands -----
    def do_forward(self, arg):
        'Move the turtle forward by the specified distance:  FORWARD 10'
        forward(*parse(arg))
    def do_right(self, arg):
        'Turn turtle right by given number of degrees:  RIGHT 20'
        right(*parse(arg))
    def do_left(self, arg):
        'Turn turtle left by given number of degrees:  LEFT 90'
        left(*parse(arg))
    def do_goto(self, arg):
        'Move turtle to an absolute position with changing orientation.  GOTO 100 200'
        goto(*parse(arg))
    def do_home(self, arg):
        'Return turtle to the home position:  HOME'
        home()
    def do_circle(self, arg):
        'Draw circle with given radius an options extent and steps:  CIRCLE 50'
        circle(*parse(arg))
    def do_position(self, arg):
        'Print the current turtle position:  POSITION'
        print('Current position is %d %d\n' % position())
    def do_heading(self, arg):
        'Print the current turtle heading in degrees:  HEADING'
        print('Current heading is %d\n' % (heading(),))
    def do_color(self, arg):
        'Set the color:  COLOR BLUE'
        color(arg.lower())
    def do_undo(self, arg):
        'Undo (repeatedly) the last turtle action(s):  UNDO'
    def do_reset(self, arg):
        'Clear the screen and return turtle to center:  RESET'
        reset()
    def do_bye(self, arg):
        'Stop recording, close the turtle window, and exit:  BYE'
        print('Thank you for using Turtle')
        self.close()
```

(continues on next page)

(continued from previous page)

```
        bye()
        return True

    # ----- record and playback -----
    def do_record(self, arg):
        'Save future commands to filename:  RECORD rose.cmd'
        self.file = open(arg, 'w')
    def do_playback(self, arg):
        'Playback commands from a file:  PLAYBACK rose.cmd'
        self.close()
        with open(arg) as f:
            self.cmdqueue.extend(f.read().splitlines())
    def precmd(self, line):
        line = line.lower()
        if self.file and 'playback' not in line:
            print(line, file=self.file)
        return line
    def close(self):
        if self.file:
            self.file.close()
            self.file = None

def parse(arg):
    'Convert a series of zero or more numbers to an argument tuple'
    return tuple(map(int, arg.split()))

if __name__ == '__main__':
    TurtleShell().cmdloop()
```

Here is a sample session with the turtle shell showing the help functions, using blank lines to repeat commands, and the simple record and playback facility:

```
Welcome to the turtle shell.   Type help or ? to list commands.

(turtle) ?

Documented commands (type help <topic>):
========================================
bye     color    goto     home  playback  record  right
circle  forward  heading  left  position  reset   undo

(turtle) help forward
Move the turtle forward by the specified distance:  FORWARD 10
(turtle) record spiral.cmd
(turtle) position
Current position is 0 0

(turtle) heading
Current heading is 0

(turtle) reset
(turtle) circle 20
(turtle) right 30
(turtle) circle 40
(turtle) right 30
(turtle) circle 60
(turtle) right 30
```

(continues on next page)

(continued from previous page)

```
(turtle) circle 80
(turtle) right 30
(turtle) circle 100
(turtle) right 30
(turtle) circle 120
(turtle) right 30
(turtle) circle 120
(turtle) heading
Current heading is 180

(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 500
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 300
(turtle) playback spiral.cmd
Current position is 0 0

Current heading is 0

Current heading is 180

(turtle) bye
Thank you for using Turtle
```

24.3 `shlex` — Simple lexical analysis

Source code: Lib/shlex.py

The *shlex* class makes it easy to write lexical analyzers for simple syntaxes resembling that of the Unix shell. This will often be useful for writing minilanguages, (for example, in run control files for Python applications) or for parsing quoted strings.

The *shlex* module defines the following functions:

shlex.**split**(*s*, *comments=False*, *posix=True*)

Split the string *s* using shell-like syntax. If *comments* is *False* (the default), the parsing of comments in the given string will be disabled (setting the *commenters* attribute of the *shlex* instance to the empty string). This function operates in POSIX mode by default, but uses non-POSIX mode if the *posix* argument is false.

Changed in version 3.12: Passing `None` for *s* argument now raises an exception, rather than reading *sys.stdin*.

shlex.**join**(*split_command*)

Concatenate the tokens of the list *split_command* and return a string. This function is the inverse of *split()*.

```
>>> from shlex import join
>>> print(join(['echo', '-n', 'Multiple words']))
echo -n 'Multiple words'
```

The returned value is shell-escaped to protect against injection vulnerabilities (see *quote()*).

Added in version 3.8.

shlex.**quote**(*s*)

Return a shell-escaped version of the string *s*. The returned value is a string that can safely be used as one token in a shell command line, for cases where you cannot use a list.

Warning

The shlex module is **only designed for Unix shells**.

The *quote()* function is not guaranteed to be correct on non-POSIX compliant shells or shells from other operating systems such as Windows. Executing commands quoted by this module on such shells can open up the possibility of a command injection vulnerability.

Consider using functions that pass command arguments with lists such as *subprocess.run()* with shell=False.

This idiom would be unsafe:

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command)  # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

quote() lets you plug the security hole:

```
>>> from shlex import quote
>>> command = 'ls -l {}'.format(quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print(remote_command)
ssh home 'ls -l '"'"'somefile; rm -rf ~'"'"''
```

The quoting is compatible with UNIX shells and with *split()*:

```
>>> from shlex import split
>>> remote_command = split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'"]
>>> command = split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

Added in version 3.3.

The *shlex* module defines the following class:

class shlex.**shlex**(*instream=None*, *infile=None*, *posix=False*, *punctuation_chars=False*)

A *shlex* instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file-/stream-like object with *read()* and *readline()* methods,

or a string. If no argument is given, input will be taken from `sys.stdin`. The second optional argument is a filename string, which sets the initial value of the *infile* attribute. If the *instream* argument is omitted or equal to `sys.stdin`, this second argument defaults to "stdin". The *posix* argument defines the operational mode: when *posix* is not true (default), the *shlex* instance will operate in compatibility mode. When operating in POSIX mode, *shlex* will try to be as close as possible to the POSIX shell parsing rules. The *punctuation_chars* argument provides a way to make the behaviour even closer to how real shells parse. This can take a number of values: the default value, `False`, preserves the behaviour seen under Python 3.5 and earlier. If set to `True`, then parsing of the characters `();<>|&` is changed: any run of these characters (considered punctuation characters) is returned as a single token. If set to a non-empty string of characters, those characters will be used as the punctuation characters. Any characters in the *wordchars* attribute that appear in *punctuation_chars* will be removed from *wordchars*. See *Improved Compatibility with Shells* for more information. *punctuation_chars* can be set only upon *shlex* instance creation and can't be modified later.

Changed in version 3.6: The *punctuation_chars* parameter was added.

See also

Module *configparser*
Parser for configuration files similar to the Windows `.ini` files.

24.3.1 shlex Objects

A *shlex* instance has the following methods:

`shlex.`**`get_token`**`()`

Return a token. If tokens have been stacked using *push_token()*, pop a token off the stack. Otherwise, read one from the input stream. If reading encounters an immediate end-of-file, *eof* is returned (the empty string (`''`) in non-POSIX mode, and `None` in POSIX mode).

`shlex.`**`push_token`**`(`*str*`)`

Push the argument onto the token stack.

`shlex.`**`read_token`**`()`

Read a raw token. Ignore the pushback stack, and do not interpret source requests. (This is not ordinarily a useful entry point, and is documented here only for the sake of completeness.)

`shlex.`**`sourcehook`**`(`*filename*`)`

When *shlex* detects a source request (see *source* below) this method is given the following token as argument, and expected to return a tuple consisting of a filename and an open file-like object.

Normally, this method first strips any quotes off the argument. If the result is an absolute pathname, or there was no previous source request in effect, or the previous source was a stream (such as `sys.stdin`), the result is left alone. Otherwise, if the result is a relative pathname, the directory part of the name of the file immediately before it on the source inclusion stack is prepended (this behavior is like the way the C preprocessor handles `#include "file.h"`).

The result of the manipulations is treated as a filename, and returned as the first component of the tuple, with *open()* called on it to yield the second component. (Note: this is the reverse of the order of arguments in instance initialization!)

This hook is exposed so that you can use it to implement directory search paths, addition of file extensions, and other namespace hacks. There is no corresponding 'close' hook, but a shlex instance will call the *close()* method of the sourced input stream when it returns EOF.

For more explicit control of source stacking, use the *push_source()* and *pop_source()* methods.

shlex.**push_source**(*newstream*, *newfile=None*)

Push an input source stream onto the input stack. If the filename argument is specified it will later be available for use in error messages. This is the same method used internally by the *sourcehook()* method.

shlex.**pop_source**()

Pop the last-pushed input source from the input stack. This is the same method used internally when the lexer reaches EOF on a stacked input stream.

shlex.**error_leader**(*infile=None*, *lineno=None*)

This method generates an error message leader in the format of a Unix C compiler error label; the format is `'"%s", line %d: '`, where the `%s` is replaced with the name of the current source file and the `%d` with the current input line number (the optional arguments can be used to override these).

This convenience is provided to encourage *shlex* users to generate error messages in the standard, parseable format understood by Emacs and other Unix tools.

Instances of *shlex* subclasses have some public instance variables which either control lexical analysis or can be used for debugging:

shlex.**commenters**

The string of characters that are recognized as comment beginners. All characters from the comment beginner to end of line are ignored. Includes just `'#'` by default.

shlex.**wordchars**

The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumerics and underscore. In POSIX mode, the accented characters in the Latin-1 set are also included. If *punctuation_chars* is not empty, the characters `~-./*?=`, which can appear in filename specifications and command line parameters, will also be included in this attribute, and any characters which appear in `punctuation_chars` will be removed from `wordchars` if they are present there. If *whitespace_split* is set to `True`, this will have no effect.

shlex.**whitespace**

Characters that will be considered whitespace and skipped. Whitespace bounds tokens. By default, includes space, tab, linefeed and carriage-return.

shlex.**escape**

Characters that will be considered as escape. This will be only used in POSIX mode, and includes just `'\'` by default.

shlex.**quotes**

Characters that will be considered string quotes. The token accumulates until the same quote is encountered again (thus, different quote types protect each other as in the shell.) By default, includes ASCII single and double quotes.

shlex.**escapedquotes**

Characters in *quotes* that will interpret escape characters defined in *escape*. This is only used in POSIX mode, and includes just `'"'` by default.

shlex.**whitespace_split**

If `True`, tokens will only be split in whitespaces. This is useful, for example, for parsing command lines with *shlex*, getting tokens in a similar way to shell arguments. When used in combination with *punctuation_chars*, tokens will be split on whitespace in addition to those characters.

Changed in version 3.8: The *punctuation_chars* attribute was made compatible with the *whitespace_split* attribute.

shlex.**infile**

The name of the current input file, as initially set at class instantiation time or stacked by later source requests. It may be useful to examine this when constructing error messages.

shlex.**instream**

The input stream from which this *shlex* instance is reading characters.

shlex.**source**

This attribute is None by default. If you assign a string to it, that string will be recognized as a lexical-level inclusion request similar to the source keyword in various shells. That is, the immediately following token will be opened as a filename and input will be taken from that stream until EOF, at which point the *close()* method of that stream will be called and the input source will again become the original input stream. Source requests may be stacked any number of levels deep.

shlex.**debug**

If this attribute is numeric and 1 or more, a *shlex* instance will print verbose progress output on its behavior. If you need to use this, you can read the module source code to learn the details.

shlex.**lineno**

Source line number (count of newlines seen so far plus one).

shlex.**token**

The token buffer. It may be useful to examine this when catching exceptions.

shlex.**eof**

Token used to determine end of file. This will be set to the empty string (''), in non-POSIX mode, and to None in POSIX mode.

shlex.**punctuation_chars**

A read-only property. Characters that will be considered punctuation. Runs of punctuation characters will be returned as a single token. However, note that no semantic validity checking will be performed: for example, '>>>' could be returned as a token, even though it may not be recognised as such by shells.

Added in version 3.6.

24.3.2 Parsing Rules

When operating in non-POSIX mode, *shlex* will try to obey to the following rules.

- Quote characters are not recognized within words (Do"Not"Separate is parsed as the single word Do"Not"Separate);
- Escape characters are not recognized;
- Enclosing characters in quotes preserve the literal value of all characters within the quotes;
- Closing quotes separate words ("Do"Separate is parsed as "Do" and Separate);
- If *whitespace_split* is False, any character not declared to be a word character, whitespace, or a quote will be returned as a single-character token. If it is True, *shlex* will only split words in whitespaces;
- EOF is signaled with an empty string (''');
- It's not possible to parse empty strings, even if quoted.

When operating in POSIX mode, *shlex* will try to obey to the following parsing rules.

- Quotes are stripped out, and do not separate words ("Do"Not"Separate" is parsed as the single word DoNot-Separate);
- Non-quoted escape characters (e.g. '\') preserve the literal value of the next character that follows;
- Enclosing characters in quotes which are not part of *escapedquotes* (e.g. "'") preserve the literal value of all characters within the quotes;

- Enclosing characters in quotes which are part of *escapedquotes* (e.g. `'"'`) preserves the literal value of all characters within the quotes, with the exception of the characters mentioned in *escape*. The escape characters retain its special meaning only when followed by the quote in use, or the escape character itself. Otherwise the escape character will be considered a normal character.
- EOF is signaled with a *None* value;
- Quoted empty strings (`''`) are allowed.

24.3.3 Improved Compatibility with Shells

Added in version 3.6.

The *shlex* class provides compatibility with the parsing performed by common Unix shells like `bash`, `dash`, and `sh`. To take advantage of this compatibility, specify the `punctuation_chars` argument in the constructor. This defaults to `False`, which preserves pre-3.6 behaviour. However, if it is set to `True`, then parsing of the characters `();<>|&` is changed: any run of these characters is returned as a single token. While this is short of a full parser for shells (which would be out of scope for the standard library, given the multiplicity of shells out there), it does allow you to perform processing of command lines more easily than you could otherwise. To illustrate, you can see the difference in the following snippet:

```
>>> import shlex
>>> text = "a && b; c && d || e; f >'abc'; (def \"ghi\")"
>>> s = shlex.shlex(text, posix=True)
>>> s.whitespace_split = True
>>> list(s)
['a', '&&', 'b;', 'c', '&&', 'd', '||', 'e;', 'f', '>abc;', '(def', 'ghi)']
>>> s = shlex.shlex(text, posix=True, punctuation_chars=True)
>>> s.whitespace_split = True
>>> list(s)
['a', '&&', 'b', ';', 'c', '&&', 'd', '||', 'e', ';', 'f', '>', 'abc', ';',
'(', 'def', 'ghi', ')']
```

Of course, tokens will be returned which are not valid for shells, and you'll need to implement your own error checks on the returned tokens.

Instead of passing `True` as the value for the punctuation_chars parameter, you can pass a string with specific characters, which will be used to determine which characters constitute punctuation. For example:

```
>>> import shlex
>>> s = shlex.shlex("a && b || c", punctuation_chars="|")
>>> list(s)
['a', '&', '&', 'b', '||', 'c']
```

Note

When `punctuation_chars` is specified, the *wordchars* attribute is augmented with the characters `~-./*?=`. That is because these characters can appear in file names (including wildcards) and command-line arguments (e.g. `--color=auto`). Hence:

```
>>> import shlex
>>> s = shlex.shlex('~/a && b-c --color=auto || d *.py?',
...                 punctuation_chars=True)
>>> list(s)
['~/a', '&&', 'b-c', '--color=auto', '||', 'd', '*.py?']
```

However, to match the shell as closely as possible, it is recommended to always use `posix` and *whites-*

pace_split when using *punctuation_chars*, which will negate *wordchars* entirely.

For best effect, `punctuation_chars` should be set in conjunction with `posix=True`. (Note that `posix=False` is the default for *shlex*.)

CHAPTER

TWENTYFIVE

GRAPHICAL USER INTERFACES WITH TK

Tk/Tcl has long been an integral part of Python. It provides a robust and platform independent windowing toolkit, that is available to Python programmers using the *tkinter* package, and its extension, the *tkinter.ttk* module.

The *tkinter* package is a thin object-oriented layer on top of Tcl/Tk. To use *tkinter*, you don't need to write Tcl code, but you will need to consult the Tk documentation, and occasionally the Tcl documentation. *tkinter* is a set of wrappers that implement the Tk widgets as Python classes.

tkinter's chief virtues are that it is fast, and that it usually comes bundled with Python. Although its standard documentation is weak, good material is available, which includes: references, tutorials, a book and others. *tkinter* is also famous for having an outdated look and feel, which has been vastly improved in Tk 8.5. Nevertheless, there are many other GUI libraries that you could be interested in. The Python wiki lists several alternative GUI frameworks and tools.

25.1 tkinter — Python interface to Tcl/Tk

Source code: Lib/tkinter/__init__.py

The *tkinter* package ("Tk interface") is the standard Python interface to the Tcl/Tk GUI toolkit. Both Tk and *tkinter* are available on most Unix platforms, including macOS, as well as on Windows systems.

Running `python -m tkinter` from the command line should open a window demonstrating a simple Tk interface, letting you know that *tkinter* is properly installed on your system, and also showing what version of Tcl/Tk is installed, so you can read the Tcl/Tk documentation specific to that version.

Tkinter supports a range of Tcl/Tk versions, built either with or without thread support. The official Python binary release bundles Tcl/Tk 8.6 threaded. See the source code for the *_tkinter* module for more information about supported versions.

Tkinter is not a thin wrapper, but adds a fair amount of its own logic to make the experience more pythonic. This documentation will concentrate on these additions and changes, and refer to the official Tcl/Tk documentation for details that are unchanged.

> **Note**
>
> Tcl/Tk 8.5 (2007) introduced a modern set of themed user interface components along with a new API to use them. Both old and new APIs are still available. Most documentation you will find online still uses the old API and can be woefully outdated.

See also

- **TkDocs**
  Extensive tutorial on creating user interfaces with Tkinter. Explains key concepts, and illustrates recommended approaches using the modern API.
- **Tkinter 8.5 reference: a GUI for Python**
  Reference documentation for Tkinter 8.5 detailing available classes, methods, and options.

Tcl/Tk Resources:

- **Tk commands**
  Comprehensive reference to each of the underlying Tcl/Tk commands used by Tkinter.
- **Tcl/Tk Home Page**
  Additional documentation, and links to Tcl/Tk core development.

Books:

- **Modern Tkinter for Busy Python Developers**
  By Mark Roseman. (ISBN 978-1999149567)
- **Python GUI programming with Tkinter**
  By Alan D. Moore. (ISBN 978-1788835886)
- **Programming Python**
  By Mark Lutz; has excellent coverage of Tkinter. (ISBN 978-0596158101)
- **Tcl and the Tk Toolkit (2nd edition)**
  By John Ousterhout, inventor of Tcl/Tk, and Ken Jones; does not cover Tkinter. (ISBN 978-0321336330)

25.1.1 Architecture

Tcl/Tk is not a single library but rather consists of a few distinct modules, each with separate functionality and its own official documentation. Python's binary releases also ship an add-on module together with it.

Tcl
Tcl is a dynamic interpreted programming language, just like Python. Though it can be used on its own as a general-purpose programming language, it is most commonly embedded into C applications as a scripting engine or an interface to the Tk toolkit. The Tcl library has a C interface to create and manage one or more instances of a Tcl interpreter, run Tcl commands and scripts in those instances, and add custom commands implemented in either Tcl or C. Each interpreter has an event queue, and there are facilities to send events to it and process them. Unlike Python, Tcl's execution model is designed around cooperative multitasking, and Tkinter bridges this difference (see *Threading model* for details).

Tk
Tk is a Tcl package implemented in C that adds custom commands to create and manipulate GUI widgets. Each `Tk` object embeds its own Tcl interpreter instance with Tk loaded into it. Tk's widgets are very customizable, though at the cost of a dated appearance. Tk uses Tcl's event queue to generate and process GUI events.

Ttk
Themed Tk (Ttk) is a newer family of Tk widgets that provide a much better appearance on different platforms than many of the classic Tk widgets. Ttk is distributed as part of Tk, starting with Tk version 8.5. Python bindings are provided in a separate module, `tkinter.ttk`.

Internally, Tk and Ttk use facilities of the underlying operating system, i.e., Xlib on Unix/X11, Cocoa on macOS, GDI on Windows.

When your Python application uses a class in Tkinter, e.g., to create a widget, the `tkinter` module first assembles a

Tcl/Tk command string. It passes that Tcl command string to an internal *_tkinter* binary module, which then calls the Tcl interpreter to evaluate it. The Tcl interpreter will then call into the Tk and/or Ttk packages, which will in turn make calls to Xlib, Cocoa, or GDI.

25.1.2 Tkinter Modules

Support for Tkinter is spread across several modules. Most applications will need the main *tkinter* module, as well as the *tkinter.ttk* module, which provides the modern themed widget set and API:

```
from tkinter import *
from tkinter import ttk
```

class tkinter.**Tk**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=True*, *sync=False*, *use=None*)

Construct a toplevel Tk widget, which is usually the main window of an application, and initialize a Tcl interpreter for this widget. Each instance has its own associated Tcl interpreter.

The *Tk* class is typically instantiated using all default values. However, the following keyword arguments are currently recognized:

screenName
When given (as a string), sets the DISPLAY environment variable. (X11 only)

baseName
Name of the profile file. By default, *baseName* is derived from the program name (sys.argv[0]).

className
Name of the widget class. Used as a profile file and also as the name with which Tcl is invoked (*argv0* in *interp*).

useTk
If True, initialize the Tk subsystem. The *tkinter.Tcl()* function sets this to False.

sync
If True, execute all X server commands synchronously, so that errors are reported immediately. Can be used for debugging. (X11 only)

use
Specifies the *id* of the window in which to embed the application, instead of it being created as an independent toplevel window. *id* must be specified in the same way as the value for the -use option for toplevel widgets (that is, it has a form like that returned by winfo_id()).

Note that on some platforms this will only work correctly if *id* refers to a Tk frame or toplevel that has its -container option enabled.

Tk reads and interprets profile files, named *.className*.tcl and *.baseName*.tcl, into the Tcl interpreter and calls *exec()* on the contents of *.className*.py and *.baseName*.py. The path for the profile files is the HOME environment variable or, if that isn't defined, then *os.curdir*.

tk
The Tk application object created by instantiating *Tk*. This provides access to the Tcl interpreter. Each widget that is attached the same instance of *Tk* has the same value for its *tk* attribute.

master
The widget object that contains this widget. For *Tk*, the *master* is *None* because it is the main window. The terms *master* and *parent* are similar and sometimes used interchangeably as argument names; however, calling winfo_parent() returns a string of the widget name whereas *master* returns the object. *parent*/*child* reflects the tree-like relationship while *master*/*slave* reflects the container structure.

children

The immediate descendants of this widget as a *dict* with the child widget names as the keys and the child instance objects as the values.

tkinter.**Tcl**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=False*)

The *Tcl()* function is a factory function which creates an object much like that created by the *Tk* class, except that it does not initialize the Tk subsystem. This is most often useful when driving the Tcl interpreter in an environment where one doesn't want to create extraneous toplevel windows, or where one cannot (such as Unix/Linux systems without an X server). An object created by the *Tcl()* object can have a Toplevel window created (and the Tk subsystem initialized) by calling its `loadtk()` method.

The modules that provide Tk support include:

tkinter
Main Tkinter module.

tkinter.colorchooser
Dialog to let the user choose a color.

tkinter.commondialog
Base class for the dialogs defined in the other modules listed here.

tkinter.filedialog
Common dialogs to allow the user to specify a file to open or save.

tkinter.font
Utilities to help work with fonts.

tkinter.messagebox
Access to standard Tk dialog boxes.

tkinter.scrolledtext
Text widget with a vertical scroll bar built in.

tkinter.simpledialog
Basic dialogs and convenience functions.

tkinter.ttk
Themed widget set introduced in Tk 8.5, providing modern alternatives for many of the classic widgets in the main *tkinter* module.

Additional modules:

_tkinter
A binary module that contains the low-level interface to Tcl/Tk. It is automatically imported by the main *tkinter* module, and should never be used directly by application programmers. It is usually a shared library (or DLL), but might in some cases be statically linked with the Python interpreter.

idlelib
Python's Integrated Development and Learning Environment (IDLE). Based on *tkinter*.

tkinter.constants
Symbolic constants that can be used in place of strings when passing various parameters to Tkinter calls. Automatically imported by the main *tkinter* module.

tkinter.dnd
(experimental) Drag-and-drop support for *tkinter*. This will become deprecated when it is replaced with the Tk DND.

turtle
Turtle graphics in a Tk window.

25.1.3 Tkinter Life Preserver

This section is not designed to be an exhaustive tutorial on either Tk or Tkinter. For that, refer to one of the external resources noted earlier. Instead, this section provides a very quick orientation to what a Tkinter application looks like, identifies foundational Tk concepts, and explains how the Tkinter wrapper is structured.

The remainder of this section will help you to identify the classes, methods, and options you'll need in your Tkinter application, and where to find more detailed documentation on them, including in the official Tcl/Tk reference manual.

A Hello World Program

We'll start by walking through a "Hello World" application in Tkinter. This isn't the smallest one we could write, but has enough to illustrate some key concepts you'll need to know.

```
from tkinter import *
from tkinter import ttk
root = Tk()
frm = ttk.Frame(root, padding=10)
frm.grid()
ttk.Label(frm, text="Hello World!").grid(column=0, row=0)
ttk.Button(frm, text="Quit", command=root.destroy).grid(column=1, row=0)
root.mainloop()
```

After the imports, the next line creates an instance of the `Tk` class, which initializes Tk and creates its associated Tcl interpreter. It also creates a toplevel window, known as the root window, which serves as the main window of the application.

The following line creates a frame widget, which in this case will contain a label and a button we'll create next. The frame is fit inside the root window.

The next line creates a label widget holding a static text string. The `grid()` method is used to specify the relative layout (position) of the label within its containing frame widget, similar to how tables in HTML work.

A button widget is then created, and placed to the right of the label. When pressed, it will call the `destroy()` method of the root window.

Finally, the `mainloop()` method puts everything on the display, and responds to user input until the program terminates.

Important Tk Concepts

Even this simple program illustrates the following key Tk concepts:

widgets
: A Tkinter user interface is made up of individual *widgets*. Each widget is represented as a Python object, instantiated from classes like `ttk.Frame`, `ttk.Label`, and `ttk.Button`.

widget hierarchy
: Widgets are arranged in a *hierarchy*. The label and button were contained within a frame, which in turn was contained within the root window. When creating each *child* widget, its *parent* widget is passed as the first argument to the widget constructor.

configuration options
: Widgets have *configuration options*, which modify their appearance and behavior, such as the text to display in a label or button. Different classes of widgets will have different sets of options.

geometry management
: Widgets aren't automatically added to the user interface when they are created. A *geometry manager* like `grid` controls where in the user interface they are placed.

event loop
: Tkinter reacts to user input, changes from your program, and even refreshes the display only when actively running an *event loop*. If your program isn't running the event loop, your user interface won't update.

Understanding How Tkinter Wraps Tcl/Tk

When your application uses Tkinter's classes and methods, internally Tkinter is assembling strings representing Tcl/Tk commands, and executing those commands in the Tcl interpreter attached to your application's `Tk` instance.

Whether it's trying to navigate reference documentation, trying to find the right method or option, adapting some existing code, or debugging your Tkinter application, there are times that it will be useful to understand what those underlying Tcl/Tk commands look like.

To illustrate, here is the Tcl/Tk equivalent of the main part of the Tkinter script above.

```
ttk::frame .frm -padding 10
grid .frm
grid [ttk::label .frm.lbl -text "Hello World!"] -column 0 -row 0
grid [ttk::button .frm.btn -text "Quit" -command "destroy ."] -column 1 -row 0
```

Tcl's syntax is similar to many shell languages, where the first word is the command to be executed, with arguments to that command following it, separated by spaces. Without getting into too many details, notice the following:

- The commands used to create widgets (like `ttk::frame`) correspond to widget classes in Tkinter.
- Tcl widget options (like `-text`) correspond to keyword arguments in Tkinter.
- Widgets are referred to by a *pathname* in Tcl (like `.frm.btn`), whereas Tkinter doesn't use names but object references.
- A widget's place in the widget hierarchy is encoded in its (hierarchical) pathname, which uses a `.` (dot) as a path separator. The pathname for the root window is just `.` (dot). In Tkinter, the hierarchy is defined not by pathname but by specifying the parent widget when creating each child widget.
- Operations which are implemented as separate *commands* in Tcl (like `grid` or `destroy`) are represented as *methods* on Tkinter widget objects. As you'll see shortly, at other times Tcl uses what appear to be method calls on widget objects, which more closely mirror what would is used in Tkinter.

How do I...? What option does...?

If you're not sure how to do something in Tkinter, and you can't immediately find it in the tutorial or reference documentation you're using, there are a few strategies that can be helpful.

First, remember that the details of how individual widgets work may vary across different versions of both Tkinter and Tcl/Tk. If you're searching documentation, make sure it corresponds to the Python and Tcl/Tk versions installed on your system.

When searching for how to use an API, it helps to know the exact name of the class, option, or method that you're using. Introspection, either in an interactive Python shell or with *`print()`*, can help you identify what you need.

To find out what configuration options are available on any widget, call its `configure()` method, which returns a dictionary containing a variety of information about each object, including its default and current values. Use `keys()` to get just the names of each option.

```
btn = ttk.Button(frm, ...)
print(btn.configure().keys())
```

As most widgets have many configuration options in common, it can be useful to find out which are specific to a particular widget class. Comparing the list of options to that of a simpler widget, like a frame, is one way to do that.

```
print(set(btn.configure().keys()) - set(frm.configure().keys()))
```

Similarly, you can find the available methods for a widget object using the standard *dir()* function. If you try it, you'll see there are over 200 common widget methods, so again identifying those specific to a widget class is helpful.

```
print(dir(btn))
print(set(dir(btn)) - set(dir(frm)))
```

Navigating the Tcl/Tk Reference Manual

As noted, the official Tk commands reference manual (man pages) is often the most accurate description of what specific operations on widgets do. Even when you know the name of the option or method that you need, you may still have a few places to look.

While all operations in Tkinter are implemented as method calls on widget objects, you've seen that many Tcl/Tk operations appear as commands that take a widget pathname as its first parameter, followed by optional parameters, e.g.

```
destroy .
grid .frm.btn -column 0 -row 0
```

Others, however, look more like methods called on a widget object (in fact, when you create a widget in Tcl/Tk, it creates a Tcl command with the name of the widget pathname, with the first parameter to that command being the name of a method to call).

```
.frm.btn invoke
.frm.lbl configure -text "Goodbye"
```

In the official Tcl/Tk reference documentation, you'll find most operations that look like method calls on the man page for a specific widget (e.g., you'll find the `invoke()` method on the ttk::button man page), while functions that take a widget as a parameter often have their own man page (e.g., grid).

You'll find many common options and methods in the options or ttk::widget man pages, while others are found in the man page for a specific widget class.

You'll also find that many Tkinter methods have compound names, e.g., `winfo_x()`, `winfo_height()`, `winfo_viewable()`. You'd find documentation for all of these in the winfo man page.

Note

Somewhat confusingly, there are also methods on all Tkinter widgets that don't actually operate on the widget, but operate at a global scope, independent of any widget. Examples are methods for accessing the clipboard or the system bell. (They happen to be implemented as methods in the base `Widget` class that all Tkinter widgets inherit from).

25.1.4 Threading model

Python and Tcl/Tk have very different threading models, which *tkinter* tries to bridge. If you use threads, you may need to be aware of this.

A Python interpreter may have many threads associated with it. In Tcl, multiple threads can be created, but each thread has a separate Tcl interpreter instance associated with it. Threads can also create more than one interpreter instance, though each interpreter instance can be used only by the one thread that created it.

Each `Tk` object created by *tkinter* contains a Tcl interpreter. It also keeps track of which thread created that interpreter. Calls to *tkinter* can be made from any Python thread. Internally, if a call comes from a thread other than the one that created the `Tk` object, an event is posted to the interpreter's event queue, and when executed, the result is returned to the calling Python thread.

Tcl/Tk applications are normally event-driven, meaning that after initialization, the interpreter runs an event loop (i.e. `Tk.mainloop()`) and responds to events. Because it is single-threaded, event handlers must respond quickly, otherwise they will block other events from being processed. To avoid this, any long-running computations should not run in an event handler, but are either broken into smaller pieces using timers, or run in another thread. This is different from many GUI toolkits where the GUI runs in a completely separate thread from all application code including event handlers.

If the Tcl interpreter is not running the event loop and processing events, any *tkinter* calls made from threads other than the one running the Tcl interpreter will fail.

A number of special cases exist:

- Tcl/Tk libraries can be built so they are not thread-aware. In this case, *tkinter* calls the library from the originating Python thread, even if this is different than the thread that created the Tcl interpreter. A global lock ensures only one call occurs at a time.
- While *tkinter* allows you to create more than one instance of a `Tk` object (with its own interpreter), all interpreters that are part of the same thread share a common event queue, which gets ugly fast. In practice, don't create more than one instance of `Tk` at a time. Otherwise, it's best to create them in separate threads and ensure you're running a thread-aware Tcl/Tk build.
- Blocking event handlers are not the only way to prevent the Tcl interpreter from reentering the event loop. It is even possible to run multiple nested event loops or abandon the event loop entirely. If you're doing anything tricky when it comes to events or threads, be aware of these possibilities.
- There are a few select *tkinter* functions that presently work only when called from the thread that created the Tcl interpreter.

25.1.5 Handy Reference

Setting Options

Options control things like the color and border width of a widget. Options can be set in three ways:

At object creation time, using keyword arguments

```
fred = Button(self, fg="red", bg="blue")
```

After object creation, treating the option name like a dictionary index

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

Use the config() method to update multiple attrs subsequent to object creation

```
fred.config(fg="red", bg="blue")
```

For a complete explanation of a given option and its behavior, see the Tk man pages for the widget in question.

Note that the man pages list "STANDARD OPTIONS" and "WIDGET SPECIFIC OPTIONS" for each widget. The former is a list of options that are common to many widgets, the latter are the options that are idiosyncratic to that particular widget. The Standard Options are documented on the *options(3)* man page.

No distinction between standard and widget-specific options is made in this document. Some options don't apply to some kinds of widgets. Whether a given widget responds to a particular option depends on the class of the widget; buttons have a `command` option, labels do not.

The options supported by a given widget are listed in that widget's man page, or can be queried at runtime by calling the `config()` method without arguments, or by calling the `keys()` method on that widget. The return value of these calls is a dictionary whose key is the name of the option as a string (for example, `'relief'`) and whose values are 5-tuples.

Some options, like `bg` are synonyms for common options with long names (`bg` is shorthand for "background"). Passing the `config()` method the name of a shorthand option will return a 2-tuple, not 5-tuple. The 2-tuple passed back will contain the name of the synonym and the "real" option (such as `('bg', 'background')`).

Index	Meaning	Example
0	option name	`'relief'`
1	option name for database lookup	`'relief'`
2	option class for database lookup	`'Relief'`
3	default value	`'raised'`
4	current value	`'groove'`

Example:

```
>>> print(fred.config())
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

Of course, the dictionary printed will include all the options available and their values. This is meant only as an example.

The Packer

The packer is one of Tk's geometry-management mechanisms. Geometry managers are used to specify the relative positioning of widgets within their container - their mutual *master*. In contrast to the more cumbersome *placer* (which is used less commonly, and we do not cover here), the packer takes qualitative relationship specification - *above*, *to the left of*, *filling*, etc - and works everything out to determine the exact placement coordinates for you.

The size of any *master* widget is determined by the size of the "slave widgets" inside. The packer is used to control where slave widgets appear inside the master into which they are packed. You can pack widgets into frames, and frames into other frames, in order to achieve the kind of layout you desire. Additionally, the arrangement is dynamically adjusted to accommodate incremental changes to the configuration, once it is packed.

Note that widgets do not appear until they have had their geometry specified with a geometry manager. It's a common early mistake to leave out the geometry specification, and then be surprised when the widget is created but nothing appears. A widget will appear only after it has had, for example, the packer's `pack()` method applied to it.

The pack() method can be called with keyword-option/value pairs that control where the widget is to appear within its container, and how it is to behave when the main application window is resized. Here are some examples:

```
fred.pack()                     # defaults to side = "top"
fred.pack(side="left")
fred.pack(expand=1)
```

Packer Options

For more extensive information on the packer and the options that it can take, see the man pages and page 183 of John Ousterhout's book.

anchor
Anchor type. Denotes where the packer is to place each slave in its parcel.

expand
Boolean, `0` or `1`.

fill
Legal values: `'x'`, `'y'`, `'both'`, `'none'`.

ipadx and ipady
A distance - designating internal padding on each side of the slave widget.

padx and pady
A distance - designating external padding on each side of the slave widget.

side
Legal values are: `'left'`, `'right'`, `'top'`, `'bottom'`.

Coupling Widget Variables

The current-value setting of some widgets (like text entry widgets) can be connected directly to application variables by using special options. These options are `variable`, `textvariable`, `onvalue`, `offvalue`, and `value`. This connection works both ways: if the variable changes for any reason, the widget it's connected to will be updated to reflect the new value.

Unfortunately, in the current implementation of *tkinter* it is not possible to hand over an arbitrary Python variable to a widget through a `variable` or `textvariable` option. The only kinds of variables for which this works are variables that are subclassed from a class called Variable, defined in *tkinter*.

There are many useful subclasses of Variable already defined: `StringVar`, `IntVar`, `DoubleVar`, and `BooleanVar`. To read the current value of such a variable, call the `get()` method on it, and to change its value you call the `set()` method. If you follow this protocol, the widget will always track the value of the variable, with no further intervention on your part.

For example:

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.pack()

        self.entrythingy = tk.Entry()
        self.entrythingy.pack()

        # Create the application variable.
```

(continues on next page)

(continued from previous page)

```
        self.contents = tk.StringVar()
        # Set it to some value.
        self.contents.set("this is a variable")
        # Tell the entry widget to watch this variable.
        self.entrythingy["textvariable"] = self.contents

        # Define a callback for when the user hits return.
        # It prints the current value of the variable.
        self.entrythingy.bind('<Key-Return>',
                              self.print_contents)

    def print_contents(self, event):
        print("Hi. The current entry content is:",
              self.contents.get())

root = tk.Tk()
myapp = App(root)
myapp.mainloop()
```

The Window Manager

In Tk, there is a utility command, `wm`, for interacting with the window manager. Options to the `wm` command allow you to control things like titles, placement, icon bitmaps, and the like. In *`tkinter`*, these commands have been implemented as methods on the `Wm` class. Toplevel widgets are subclassed from the `Wm` class, and so can call the `Wm` methods directly.

To get at the toplevel window that contains a given widget, you can often just refer to the widget's master. Of course if the widget has been packed inside of a frame, the master won't represent a toplevel window. To get at the toplevel window that contains an arbitrary widget, you can call the `_root()` method. This method begins with an underscore to denote the fact that this function is part of the implementation, and not an interface to Tk functionality.

Here are some examples of typical usage:

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

# create the application
myapp = App()

#
# here are method calls to the window manager class
#
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# start the program
myapp.mainloop()
```

Tk Option Data Types

anchor
Legal values are points of the compass: `"n"`, `"ne"`, `"e"`, `"se"`, `"s"`, `"sw"`, `"w"`, `"nw"`, and also `"center"`.

bitmap
There are eight built-in, named bitmaps: `'error'`, `'gray25'`, `'gray50'`, `'hourglass'`, `'info'`, `'questhead'`, `'question'`, `'warning'`. To specify an X bitmap filename, give the full path to the file, preceded with an `@`, as in `"@/usr/contrib/bitmap/gumby.bit"`.

boolean
You can pass integers 0 or 1 or the strings `"yes"` or `"no"`.

callback
This is any Python function that takes no arguments. For example:

```
def print_it():
    print("hi there")
fred["command"] = print_it
```

color
Colors can be given as the names of X colors in the rgb.txt file, or as strings representing RGB values in 4 bit: `"#RGB"`, 8 bit: `"#RRGGBB"`, 12 bit: `"#RRRGGGBBB"`, or 16 bit: `"#RRRRGGGGBBBB"` ranges, where R,G,B here represent any legal hex digit. See page 160 of Ousterhout's book for details.

cursor
The standard X cursor names from `cursorfont.h` can be used, without the `XC_` prefix. For example to get a hand cursor (`XC_hand2`), use the string `"hand2"`. You can also specify a bitmap and mask file of your own. See page 179 of Ousterhout's book.

distance
Screen distances can be specified in either pixels or absolute distances. Pixels are given as numbers and absolute distances as strings, with the trailing character denoting units: `c` for centimetres, `i` for inches, `m` for millimetres, `p` for printer's points. For example, 3.5 inches is expressed as `"3.5i"`.

font
Tk uses a list font name format, such as `{courier 10 bold}`. Font sizes with positive numbers are measured in points; sizes with negative numbers are measured in pixels.

geometry
This is a string of the form `widthxheight`, where width and height are measured in pixels for most widgets (in characters for widgets displaying text). For example: `fred["geometry"] = "200x100"`.

justify
Legal values are the strings: `"left"`, `"center"`, `"right"`, and `"fill"`.

region
This is a string with four space-delimited elements, each of which is a legal distance (see above). For example: `"2 3 4 5"` and `"3i 2i 4.5i 2i"` and `"3c 2c 4c 10.43c"` are all legal regions.

relief
Determines what the border style of a widget will be. Legal values are: `"raised"`, `"sunken"`, `"flat"`, `"groove"`, and `"ridge"`.

scrollcommand
This is almost always the `set()` method of some scrollbar widget, but can be any widget method that takes a single argument.

wrap
Must be one of: `"none"`, `"char"`, or `"word"`.

Bindings and Events

The bind method from the widget command allows you to watch for certain events and to have a callback function trigger when that event type occurs. The form of the bind method is:

```
def bind(self, sequence, func, add=''):
```

where:

sequence
: is a string that denotes the target kind of event. (See the *bind(3tk)* man page, and page 201 of John Ousterhout's book, *Tcl and the Tk Toolkit (2nd edition)*, for details).

func
: is a Python function, taking one argument, to be invoked when the event occurs. An Event instance will be passed as the argument. (Functions deployed this way are commonly known as *callbacks*.)

add
: is optional, either `''` or `'+'`. Passing an empty string denotes that this binding is to replace any other bindings that this event is associated with. Passing a `'+'` means that this function is to be added to the list of functions bound to this event type.

For example:

```
def turn_red(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turn_red)
```

Notice how the widget field of the event is being accessed in the `turn_red()` callback. This field contains the widget that caught the X event. The following table lists the other event fields you can access, and how they are denoted in Tk, which can be useful when referring to the Tk man pages.

Tk	Tkinter Event Field	Tk	Tkinter Event Field
%f	focus	%A	char
%h	height	%E	send_event
%k	keycode	%K	keysym
%s	state	%N	keysym_num
%t	time	%T	type
%w	width	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

The index Parameter

A number of widgets require "index" parameters to be passed. These are used to point at a specific place in a Text widget, or to particular characters in an Entry widget, or to particular menu items in a Menu widget.

Entry widget indexes (index, view index, etc.)
: Entry widgets have options that refer to character positions in the text being displayed. You can use these *tkinter* functions to access these special points in text widgets:

Text widget indexes
: The index notation for Text widgets is very rich and is best described in the Tk man pages.

Menu indexes (menu.invoke(), menu.entryconfig(), etc.)
Some options and methods for menus manipulate specific menu entries. Anytime a menu index is needed for an option or a parameter, you may pass in:

- an integer which refers to the numeric position of the entry in the widget, counted from the top, starting with 0;
- the string `"active"`, which refers to the menu position that is currently under the cursor;
- the string `"last"` which refers to the last menu item;
- An integer preceded by `@`, as in `@6`, where the integer is interpreted as a y pixel coordinate in the menu's coordinate system;
- the string `"none"`, which indicates no menu entry at all, most often used with menu.activate() to deactivate all entries, and finally,
- a text string that is pattern matched against the label of the menu entry, as scanned from the top of the menu to the bottom. Note that this index type is considered after all the others, which means that matches for menu items labelled `last`, `active`, or `none` may be interpreted as the above literals, instead.

Images

Images of different formats can be created through the corresponding subclass of `tkinter.Image`:

- `BitmapImage` for images in XBM format.
- `PhotoImage` for images in PGM, PPM, GIF and PNG formats. The latter is supported starting with Tk 8.6.

Either type of image is created through either the `file` or the `data` option (other options are available as well).

Changed in version 3.13: Added the `PhotoImage` method `copy_replace()` to copy a region from one image to other image, possibly with pixel zooming and/or subsampling. Add *from_coords* parameter to `PhotoImage` methods `copy()`, `zoom()` and `subsample()`. Add *zoom* and *subsample* parameters to `PhotoImage` method `copy()`.

The image object can then be used wherever an `image` option is supported by some widget (e.g. labels, buttons, menus). In these cases, Tk will not keep a reference to the image. When the last Python reference to the image object is deleted, the image data is deleted as well, and Tk will display an empty box wherever the image was used.

See also

The Pillow package adds support for formats such as BMP, JPEG, TIFF, and WebP, among others.

25.1.6 File Handlers

Tk allows you to register and unregister a callback function which will be called from the Tk mainloop when I/O is possible on a file descriptor. Only one handler may be registered per file descriptor. Example code:

```
import tkinter
widget = tkinter.Tk()
mask = tkinter.READABLE | tkinter.WRITABLE
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

This feature is not available on Windows.

Since you don't know how many bytes are available for reading, you may not want to use the *BufferedIOBase* or *TextIOBase* *read()* or *readline()* methods, since these will insist on reading a predefined number of bytes. For sockets, the *recv()* or *recvfrom()* methods will work fine; for other files, use raw reads or `os.read(file.fileno(), maxbytecount)`.

`Widget.tk.`**`createfilehandler`**(*file*, *mask*, *func*)

Registers the file handler callback function *func*. The *file* argument may either be an object with a *fileno()* method (such as a file or socket object), or an integer file descriptor. The *mask* argument is an ORed combination of any of the three constants below. The callback is called as follows:

```
callback(file, mask)
```

`Widget.tk.`**`deletefilehandler`**(*file*)

Unregisters a file handler.

`_tkinter.`**`READABLE`**

`_tkinter.`**`WRITABLE`**

`_tkinter.`**`EXCEPTION`**

Constants used in the *mask* arguments.

25.2 `tkinter.colorchooser` — Color choosing dialog

Source code: Lib/tkinter/colorchooser.py

The *tkinter.colorchooser* module provides the *Chooser* class as an interface to the native color picker dialog. `Chooser` implements a modal color choosing dialog window. The `Chooser` class inherits from the *Dialog* class.

`class` `tkinter.colorchooser.`**`Chooser`**(*master=None*, ***options*)

`tkinter.colorchooser.`**`askcolor`**(*color=None*, ***options*)

Create a color choosing dialog. A call to this method will show the window, wait for the user to make a selection, and return the selected color (or `None`) to the caller.

See also

Module ***tkinter.commondialog***
Tkinter standard dialog module

25.3 `tkinter.font` — Tkinter font wrapper

Source code: Lib/tkinter/font.py

The *tkinter.font* module provides the *Font* class for creating and using named fonts.

The different font weights and slants are:

`tkinter.font.`**`NORMAL`**

`tkinter.font.`**`BOLD`**

`tkinter.font.`**`ITALIC`**

tkinter.font.**ROMAN**

class tkinter.font.**Font**(*root=None*, *font=None*, *name=None*, *exists=False*, ***options*)

The *Font* class represents a named font. *Font* instances are given unique names and can be specified by their family, size, and style configuration. Named fonts are Tk's method of creating and identifying fonts as a single object, rather than specifying a font by its attributes with each occurrence.

arguments:

font - font specifier tuple (family, size, options)
name - unique font name
exists - self points to existing named font if true

additional keyword options (ignored if *font* is specified):

family - font family i.e. Courier, Times
size - font size
If *size* is positive it is interpreted as size in points.
If *size* is a negative number its absolute value is treated as size in pixels.
weight - font emphasis (NORMAL, BOLD)
slant - ROMAN, ITALIC
underline - font underlining (0 - none, 1 - underline)
overstrike - font strikeout (0 - none, 1 - strikeout)

actual(*option=None*, *displayof=None*)

Return the attributes of the font.

cget(*option*)

Retrieve an attribute of the font.

config(***options*)

Modify attributes of the font.

copy()

Return new instance of the current font.

measure(*text*, *displayof=None*)

Return amount of space the text would occupy on the specified display when formatted in the current font. If no display is specified then the main application window is assumed.

metrics(**options*, ***kw*)

Return font-specific data. Options include:

***ascent* - distance between baseline and highest point that a**
character of the font can occupy

***descent* - distance between baseline and lowest point that a**
character of the font can occupy

***linespace* - minimum vertical separation necessary between any two**
characters of the font that ensures no vertical overlap between lines.

fixed - 1 if font is fixed-width else 0

tkinter.font.**families**(*root=None*, *displayof=None*)

Return the different font families.

tkinter.font.**names**(*root=None*)

Return the names of defined fonts.

tkinter.font.**nametofont**(*name*, *root=None*)

Return a *Font* representation of a tk named font.

Changed in version 3.10: The *root* parameter was added.

25.4 Tkinter Dialogs

25.4.1 tkinter.simpledialog — Standard Tkinter input dialogs

Source code: Lib/tkinter/simpledialog.py

The *tkinter.simpledialog* module contains convenience classes and functions for creating simple modal dialogs to get a value from the user.

tkinter.simpledialog.**askfloat**(*title*, *prompt*, ***kw*)

tkinter.simpledialog.**askinteger**(*title*, *prompt*, ***kw*)

tkinter.simpledialog.**askstring**(*title*, *prompt*, ***kw*)

The above three functions provide dialogs that prompt the user to enter a value of the desired type.

class tkinter.simpledialog.**Dialog**(*parent*, *title=None*)

The base class for custom dialogs.

body(*master*)

Override to construct the dialog's interface and return the widget that should have initial focus.

buttonbox()

Default behaviour adds OK and Cancel buttons. Override for custom button layouts.

25.4.2 tkinter.filedialog — File selection dialogs

Source code: Lib/tkinter/filedialog.py

The *tkinter.filedialog* module provides classes and factory functions for creating file/directory selection windows.

Native Load/Save Dialogs

The following classes and functions provide file dialog windows that combine a native look-and-feel with configuration options to customize behaviour. The following keyword arguments are applicable to the classes and functions listed below:

parent - the window to place the dialog on top of

title - the title of the window

initialdir - the directory that the dialog starts in

initialfile - the file selected upon opening of the dialog

filetypes - a sequence of (label, pattern) tuples, '*' wildcard is allowed

defaultextension - default extension to append to file (save dialogs)

multiple - when true, selection of multiple items is allowed

Static factory functions

The below functions when called create a modal, native look-and-feel dialog, wait for the user's selection, then return the selected value(s) or `None` to the caller.

`tkinter.filedialog.`**`askopenfile`**(*mode='r'*, ***options*)

`tkinter.filedialog.`**`askopenfiles`**(*mode='r'*, ***options*)

The above two functions create an *Open* dialog and return the opened file object(s) in read-only mode.

`tkinter.filedialog.`**`asksaveasfile`**(*mode='w'*, ***options*)

Create a *SaveAs* dialog and return a file object opened in write-only mode.

`tkinter.filedialog.`**`askopenfilename`**(***options*)

`tkinter.filedialog.`**`askopenfilenames`**(***options*)

The above two functions create an *Open* dialog and return the selected filename(s) that correspond to existing file(s).

`tkinter.filedialog.`**`asksaveasfilename`**(***options*)

Create a *SaveAs* dialog and return the selected filename.

`tkinter.filedialog.`**`askdirectory`**(***options*)

Prompt user to select a directory.

Additional keyword option:

mustexist - determines if selection must be an existing directory.

`class` `tkinter.filedialog.`**`Open`**(*master=None*, ***options*)

`class` `tkinter.filedialog.`**`SaveAs`**(*master=None*, ***options*)

The above two classes provide native dialog windows for saving and loading files.

Convenience classes

The below classes are used for creating file/directory windows from scratch. These do not emulate the native look-and-feel of the platform.

`class` `tkinter.filedialog.`**`Directory`**(*master=None*, ***options*)

Create a dialog prompting the user to select a directory.

Note

The *FileDialog* class should be subclassed for custom event handling and behaviour.

class tkinter.filedialog.**FileDialog**(*master*, *title=None*)

Create a basic file selection dialog.

cancel_command(*event=None*)

Trigger the termination of the dialog window.

dirs_double_event(*event*)

Event handler for double-click event on directory.

dirs_select_event(*event*)

Event handler for click event on directory.

files_double_event(*event*)

Event handler for double-click event on file.

files_select_event(*event*)

Event handler for single-click event on file.

filter_command(*event=None*)

Filter the files by directory.

get_filter()

Retrieve the file filter currently in use.

get_selection()

Retrieve the currently selected item.

go(*dir_or_file=os.curdir*, *pattern='*'*, *default=''*, *key=None*)

Render dialog and start event loop.

ok_event(*event*)

Exit dialog returning current selection.

quit(*how=None*)

Exit dialog returning filename, if any.

set_filter(*dir*, *pat*)

Set the file filter.

set_selection(*file*)

Update the current file selection to *file*.

class tkinter.filedialog.**LoadFileDialog**(*master*, *title=None*)

A subclass of FileDialog that creates a dialog window for selecting an existing file.

ok_command()

Test that a file is provided and that the selection indicates an already existing file.

class tkinter.filedialog.**SaveFileDialog**(*master*, *title=None*)

A subclass of FileDialog that creates a dialog window for selecting a destination file.

ok_command()

Test whether or not the selection points to a valid file that is not a directory. Confirmation is required if an already existing file is selected.

25.4.3 `tkinter.commondialog` — Dialog window templates

Source code: Lib/tkinter/commondialog.py

The *tkinter.commondialog* module provides the *Dialog* class that is the base class for dialogs defined in other supporting modules.

class tkinter.commondialog.**Dialog**(*master=None*, ***options*)

show(*color=None*, ***options*)

Render the Dialog window.

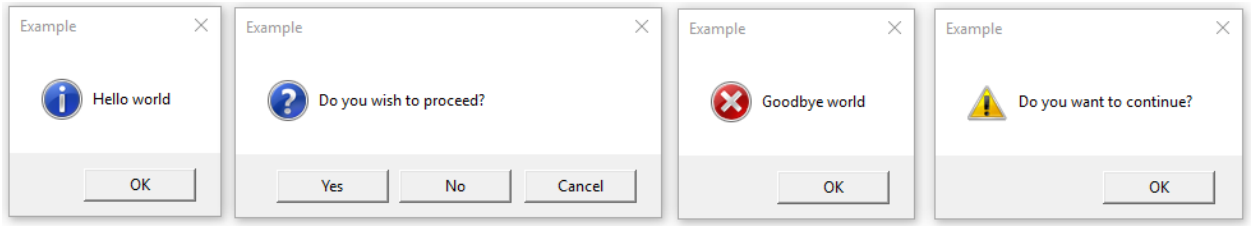
See also

Modules *tkinter.messagebox*, tut-files

25.5 `tkinter.messagebox` — Tkinter message prompts

Source code: Lib/tkinter/messagebox.py

The *tkinter.messagebox* module provides a template base class as well as a variety of convenience methods for commonly used configurations. The message boxes are modal and will return a subset of (`True`, `False`, `None`, *OK*, *CANCEL*, *YES*, *NO*) based on the user's selection. Common message box styles and layouts include but are not limited to:



class tkinter.messagebox.**Message**(*master=None*, ***options*)

Create a message window with an application-specified message, an icon and a set of buttons. Each of the buttons in the message window is identified by a unique symbolic name (see the *type* options).

The following options are supported:

command

Specifies the function to invoke when the user closes the dialog. The name of the button clicked by the user to close the dialog is passed as argument. This is only available on macOS.

default

Gives the *symbolic name* of the default button for this message window (*OK*, *CANCEL*, and so on). If this option is not specified, the first button in the dialog will be made the default.

detail

Specifies an auxiliary message to the main message given by the *message* option. The message detail will be presented beneath the main message and, where supported by the OS, in a less emphasized font than the main message.

icon
Specifies an *icon* to display. If this option is not specified, then the `INFO` icon will be displayed.

message
Specifies the message to display in this message box. The default value is an empty string.

parent
Makes the specified window the logical parent of the message box. The message box is displayed on top of its parent window.

title
Specifies a string to display as the title of the message box. This option is ignored on macOS, where platform guidelines forbid the use of a title on this kind of dialog.

type
Arranges for a *predefined set of buttons* to be displayed.

show (***options*)

Display a message window and wait for the user to select one of the buttons. Then return the symbolic name of the selected button. Keyword arguments can override options specified in the constructor.

Information message box

tkinter.messagebox.**showinfo**(*title=None*, *message=None*, ***options*)

Creates and displays an information message box with the specified title and message.

Warning message boxes

tkinter.messagebox.**showwarning**(*title=None*, *message=None*, ***options*)

Creates and displays a warning message box with the specified title and message.

tkinter.messagebox.**showerror**(*title=None*, *message=None*, ***options*)

Creates and displays an error message box with the specified title and message.

Question message boxes

tkinter.messagebox.**askquestion**(*title=None*, *message=None*, ***, *type=YESNO*, ***options*)

Ask a question. By default shows buttons `YES` and `NO`. Returns the symbolic name of the selected button.

tkinter.messagebox.**askokcancel**(*title=None*, *message=None*, ***options*)

Ask if operation should proceed. Shows buttons `OK` and `CANCEL`. Returns `True` if the answer is ok and `False` otherwise.

tkinter.messagebox.**askretrycancel**(*title=None*, *message=None*, ***options*)

Ask if operation should be retried. Shows buttons `RETRY` and `CANCEL`. Return `True` if the answer is yes and `False` otherwise.

tkinter.messagebox.**askyesno**(*title=None*, *message=None*, ***options*)

Ask a question. Shows buttons `YES` and `NO`. Returns `True` if the answer is yes and `False` otherwise.

tkinter.messagebox.**askyesnocancel**(*title=None*, *message=None*, ***options*)

Ask a question. Shows buttons `YES`, `NO` and `CANCEL`. Return `True` if the answer is yes, `None` if cancelled, and `False` otherwise.

Symbolic names of buttons:

tkinter.messagebox.**ABORT = 'abort'**

tkinter.messagebox.**RETRY = 'retry'**

tkinter.messagebox.**IGNORE = 'ignore'**

tkinter.messagebox.**OK = 'ok'**

tkinter.messagebox.**CANCEL = 'cancel'**

tkinter.messagebox.**YES = 'yes'**

tkinter.messagebox.**NO = 'no'**

Predefined sets of buttons:

tkinter.messagebox.**ABORTRETRYIGNORE = 'abortretryignore'**
: Displays three buttons whose symbolic names are *ABORT*, *RETRY* and *IGNORE*.

tkinter.messagebox.**OK = 'ok'**
: Displays one button whose symbolic name is *OK*.

tkinter.messagebox.**OKCANCEL = 'okcancel'**
: Displays two buttons whose symbolic names are *OK* and *CANCEL*.

tkinter.messagebox.**RETRYCANCEL = 'retrycancel'**
: Displays two buttons whose symbolic names are *RETRY* and *CANCEL*.

tkinter.messagebox.**YESNO = 'yesno'**
: Displays two buttons whose symbolic names are *YES* and *NO*.

tkinter.messagebox.**YESNOCANCEL = 'yesnocancel'**
: Displays three buttons whose symbolic names are *YES*, *NO* and *CANCEL*.

Icon images:

tkinter.messagebox.**ERROR = 'error'**

tkinter.messagebox.**INFO = 'info'**

tkinter.messagebox.**QUESTION = 'question'**

tkinter.messagebox.**WARNING = 'warning'**

25.6 tkinter.scrolledtext — Scrolled Text Widget

Source code: Lib/tkinter/scrolledtext.py

The *tkinter.scrolledtext* module provides a class of the same name which implements a basic text widget which has a vertical scroll bar configured to do the "right thing." Using the *ScrolledText* class is a lot easier than setting up a text widget and scroll bar directly.

The text widget and scrollbar are packed together in a `Frame`, and the methods of the `Grid` and `Pack` geometry managers are acquired from the `Frame` object. This allows the *ScrolledText* widget to be used directly to achieve most normal geometry management behavior.

Should more specific control be necessary, the following attributes are available:

class tkinter.scrolledtext.**ScrolledText**(*master=None*, ***kw*)

frame

The frame which surrounds the text and scroll bar widgets.

vbar

The scroll bar widget.

25.7 `tkinter.dnd` — Drag and drop support

Source code: Lib/tkinter/dnd.py

> **Note**
>
> This is experimental and due to be deprecated when it is replaced with the Tk DND.

The `tkinter.dnd` module provides drag-and-drop support for objects within a single application, within the same window or between windows. To enable an object to be dragged, you must create an event binding for it that starts the drag-and-drop process. Typically, you bind a ButtonPress event to a callback function that you write (see *Bindings and Events*). The function should call `dnd_start()`, where 'source' is the object to be dragged, and 'event' is the event that invoked the call (the argument to your callback function).

Selection of a target object occurs as follows:

1. Top-down search of area under mouse for target widget
   - Target widget should have a callable *dnd_accept* attribute
   - If *dnd_accept* is not present or returns `None`, search moves to parent widget
   - If no target widget is found, then the target object is `None`
2. Call to *<old_target>.dnd_leave(source, event)*
3. Call to *<new_target>.dnd_enter(source, event)*
4. Call to *<target>.dnd_commit(source, event)* to notify of drop
5. Call to *<source>.dnd_end(target, event)* to signal end of drag-and-drop

class `tkinter.dnd.`**DndHandler**(*source*, *event*)

The *DndHandler* class handles drag-and-drop events tracking Motion and ButtonRelease events on the root of the event widget.

cancel(*event=None*)

Cancel the drag-and-drop process.

finish(*event*, *commit=0*)

Execute end of drag-and-drop functions.

on_motion(*event*)

Inspect area below mouse for target objects while drag is performed.

on_release(*event*)

Signal end of drag when the release pattern is triggered.

tkinter.dnd.**dnd_start**(*source*, *event*)

Factory function for drag-and-drop process.

See also

Bindings and Events

25.8 tkinter.ttk — Tk themed widgets

Source code: Lib/tkinter/ttk.py

The *tkinter.ttk* module provides access to the Tk themed widget set, introduced in Tk 8.5. It provides additional benefits including anti-aliased font rendering under X11 and window transparency (requiring a composition window manager on X11).

The basic idea for *tkinter.ttk* is to separate, to the extent possible, the code implementing a widget's behavior from the code implementing its appearance.

See also

Tk Widget Styling Support
A document introducing theming support for Tk

25.8.1 Using Ttk

To start using Ttk, import its module:

```
from tkinter import ttk
```

To override the basic Tk widgets, the import should follow the Tk import:

```
from tkinter import *
from tkinter.ttk import *
```

That code causes several *tkinter.ttk* widgets (`Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale` and `Scrollbar`) to automatically replace the Tk widgets.

This has the direct benefit of using the new widgets which gives a better look and feel across platforms; however, the replacement widgets are not completely compatible. The main difference is that widget options such as "fg", "bg" and others related to widget styling are no longer present in Ttk widgets. Instead, use the `ttk.Style` class for improved styling effects.

See also

Converting existing applications to use Tile widgets
A monograph (using Tcl terminology) about differences typically encountered when moving applications to use the new widgets.

25.8.2 Ttk Widgets

Ttk comes with 18 widgets, twelve of which already existed in tkinter: `Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale`, `Scrollbar`, and *`Spinbox`*. The other six are new: *`Combobox`*, *`Notebook`*, *`Progressbar`*, `Separator`, `Sizegrip` and *`Treeview`*. And all them are subclasses of *`Widget`*.

Using the Ttk widgets gives the application an improved look and feel. As discussed above, there are differences in how the styling is coded.

Tk code:

```
l1 = tkinter.Label(text="Test", fg="black", bg="white")
l2 = tkinter.Label(text="Test", fg="black", bg="white")
```

Ttk code:

```
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")

l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

For more information about *TtkStyling*, see the *`Style`* class documentation.

25.8.3 Widget

`ttk.Widget` defines standard options and methods supported by Tk themed widgets and is not supposed to be directly instantiated.

Standard Options

All the `ttk` Widgets accept the following options:

Option	Description
class	Specifies the window class. The class is used when querying the option database for the window's other options, to determine the default bindtags for the window, and to select the widget's default layout and style. This option is read-only, and may only be specified when the window is created.
cursor	Specifies the mouse cursor to be used for the widget. If set to the empty string (the default), the cursor is inherited for the parent widget.
takefocus	Determines whether the window accepts the focus during keyboard traversal. 0, 1 or an empty string is returned. If 0 is returned, it means that the window should be skipped entirely during keyboard traversal. If 1, it means that the window should receive the input focus as long as it is viewable. And an empty string means that the traversal scripts make the decision about whether or not to focus on the window.
style	May be used to specify a custom widget style.

Scrollable Widget Options

The following options are supported by widgets that are controlled by a scrollbar.

Option	Description
xscrollcommand	Used to communicate with horizontal scrollbars. When the view in the widget's window change, the widget will generate a Tcl command based on the scrollcommand. Usually this option consists of the method `Scrollbar.set()` of some scrollbar. This will cause the scrollbar to be updated whenever the view in the window changes.
yscrollcommand	Used to communicate with vertical scrollbars. For some more information, see above.

Label Options

The following options are supported by labels, buttons and other button-like widgets.

Option	Description
text	Specifies a text string to be displayed inside the widget.
textvariable	Specifies a name whose value will be used in place of the text option resource.
underline	If set, specifies the index (0-based) of a character to underline in the text string. The underline character is used for mnemonic activation.
image	Specifies an image to display. This is a list of 1 or more elements. The first element is the default image name. The rest of the list if a sequence of statespec/value pairs as defined by *`Style.map()`*, specifying different images to use when the widget is in a particular state or a combination of states. All images in the list should have the same size.
compound	Specifies how to display the image relative to the text, in the case both text and images options are present. Valid values are: • text: display text only • image: display image only • top, bottom, left, right: display image above, below, left of, or right of the text, respectively. • none: the default. display the image if present, otherwise the text.
width	If greater than zero, specifies how much space, in character widths, to allocate for the text label, if less than zero, specifies a minimum width. If zero or unspecified, the natural width of the text label is used.

Compatibility Options

Option	Description
state	May be set to "normal" or "disabled" to control the "disabled" state bit. This is a write-only option: setting it changes the widget state, but the *`Widget.state()`* method does not affect this option.

Widget States

The widget state is a bitmap of independent state flags.

Flag	Description
active	The mouse cursor is over the widget and pressing a mouse button will cause some action to occur
disabled	Widget is disabled under program control
focus	Widget has keyboard focus
pressed	Widget is being pressed
selected	"On", "true", or "current" for things like Checkbuttons and radiobuttons
background	Windows and Mac have a notion of an "active" or foreground window. The *background* state is set for widgets in a background window, and cleared for those in the foreground window
readonly	Widget should not allow user modification
alternate	A widget-specific alternate display format
invalid	The widget's value is invalid

A state specification is a sequence of state names, optionally prefixed with an exclamation point indicating that the bit is off.

ttk.Widget

Besides the methods described below, the `ttk.Widget` supports the methods `tkinter.Widget.cget()` and `tkinter.Widget.configure()`.

class tkinter.ttk.**Widget**

identify(*x*, *y*)

Returns the name of the element at position *x* *y*, or the empty string if the point does not lie within any element.

x and *y* are pixel coordinates relative to the widget.

instate(*statespec*, *callback=None*, **args*, ***kw*)

Test the widget's state. If a callback is not specified, returns `True` if the widget state matches *statespec* and `False` otherwise. If callback is specified then it is called with args if widget state matches *statespec*.

state(*statespec=None*)

Modify or inquire widget state. If *statespec* is specified, sets the widget state according to it and return a new *statespec* indicating which flags were changed. If *statespec* is not specified, returns the currently enabled state flags.

statespec will usually be a list or a tuple.

25.8.4 Combobox

The `ttk.Combobox` widget combines a text field with a pop-down list of values. This widget is a subclass of `Entry`.

Besides the methods inherited from *Widget*: `Widget.cget()`, `Widget.configure()`, *`Widget.identify()`*, *`Widget.instate()`* and *`Widget.state()`*, and the following inherited from `Entry`: `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.selection()`, `Entry.xview()`, it has some other methods, described at `ttk.Combobox`.

Options

This widget accepts the following specific options:

Option	Description
exportselection	Boolean value. If set, the widget selection is linked to the Window Manager selection (which can be returned by invoking Misc.selection_get, for example).
justify	Specifies how the text is aligned within the widget. One of "left", "center", or "right".
height	Specifies the height of the pop-down listbox, in rows.
postcommand	A script (possibly registered with Misc.register) that is called immediately before displaying the values. It may specify which values to display.
state	One of "normal", "readonly", or "disabled". In the "readonly" state, the value may not be edited directly, and the user can only selection of the values from the dropdown list. In the "normal" state, the text field is directly editable. In the "disabled" state, no interaction is possible.
textvariable	Specifies a name whose value is linked to the widget value. Whenever the value associated with that name changes, the widget value is updated, and vice versa. See `tkinter.StringVar`.
values	Specifies the list of values to display in the drop-down listbox.
width	Specifies an integer value indicating the desired width of the entry window, in average-size characters of the widget's font.

Virtual events

The combobox widgets generates a **<<ComboboxSelected>>** virtual event when the user selects an element from the list of values.

ttk.Combobox

class tkinter.ttk.**Combobox**

current(*newindex=None*)

If *newindex* is specified, sets the combobox value to the element position *newindex*. Otherwise, returns the index of the current value or -1 if the current value is not in the values list.

get()

Returns the current value of the combobox.

set(*value*)

Sets the value of the combobox to *value*.

25.8.5 Spinbox

The `ttk.Spinbox` widget is a `ttk.Entry` enhanced with increment and decrement arrows. It can be used for numbers or lists of string values. This widget is a subclass of `Entry`.

Besides the methods inherited from *Widget*: `Widget.cget()`, `Widget.configure()`, *Widget.identify()*, *Widget.instate()* and *Widget.state()*, and the following inherited from `Entry`: `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.xview()`, it has some other methods, described at `ttk.Spinbox`.

Options

This widget accepts the following specific options:

Option	Description
from	Float value. If set, this is the minimum value to which the decrement button will decrement. Must be spelled as `from_` when used as an argument, since `from` is a Python keyword.
to	Float value. If set, this is the maximum value to which the increment button will increment.
increment	Float value. Specifies the amount which the increment/decrement buttons change the value. Defaults to 1.0.
values	Sequence of string or float values. If specified, the increment/decrement buttons will cycle through the items in this sequence rather than incrementing or decrementing numbers.
wrap	Boolean value. If `True`, increment and decrement buttons will cycle from the `to` value to the `from` value or the `from` value to the `to` value, respectively.
format	String value. This specifies the format of numbers set by the increment/decrement buttons. It must be in the form "%W.Pf", where W is the padded width of the value, P is the precision, and '%' and 'f' are literal.
command	Python callable. Will be called with no arguments whenever either of the increment or decrement buttons are pressed.

Virtual events

The spinbox widget generates an **<<Increment>>** virtual event when the user presses <Up>, and a **<<Decrement>>** virtual event when the user presses <Down>.

ttk.Spinbox

class tkinter.ttk.**Spinbox**

get()

Returns the current value of the spinbox.

set(*value*)

Sets the value of the spinbox to *value*.

25.8.6 Notebook

Ttk Notebook widget manages a collection of windows and displays a single one at a time. Each child window is associated with a tab, which the user may select to change the currently displayed window.

Options

This widget accepts the following specific options:

Option	Description
height	If present and greater than zero, specifies the desired height of the pane area (not including internal padding or tabs). Otherwise, the maximum height of all panes is used.
padding	Specifies the amount of extra space to add around the outside of the notebook. The padding is a list up to four length specifications left top right bottom. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left.
width	If present and greater than zero, specified the desired width of the pane area (not including internal padding). Otherwise, the maximum width of all panes is used.

Tab Options

There are also specific options for tabs:

Option	Description
state	Either “normal”, “disabled” or “hidden”. If “disabled”, then the tab is not selectable. If “hidden”, then the tab is not shown.
sticky	Specifies how the child window is positioned within the pane area. Value is a string containing zero or more of the characters “n”, “s”, “e” or “w”. Each letter refers to a side (north, south, east or west) that the child window will stick to, as per the `grid()` geometry manager.
padding	Specifies the amount of extra space to add between the notebook and this pane. Syntax is the same as for the option padding used by this widget.
text	Specifies a text to be displayed in the tab.
image	Specifies an image to display in the tab. See the option image described in *`Widget`*.
compound	Specifies how to display the image relative to the text, in the case both options text and image are present. See *Label Options* for legal values.
underline	Specifies the index (0-based) of a character to underline in the text string. The underlined character is used for mnemonic activation if *`Notebook.enable_traversal()`* is called.

Tab Identifiers

The tab_id present in several methods of `ttk.Notebook` may take any of the following forms:

- An integer between zero and the number of tabs
- The name of a child window
- A positional specification of the form “@x,y”, which identifies the tab
- The literal string “current”, which identifies the currently selected tab
- The literal string “end”, which returns the number of tabs (only valid for *`Notebook.index()`*)

Virtual Events

This widget generates a **<<NotebookTabChanged>>** virtual event after a new tab is selected.

ttk.Notebook

class tkinter.ttk.**Notebook**

add(*child*, ***kw*)

Adds a new tab to the notebook.

If window is currently managed by the notebook but hidden, it is restored to its previous position.

See *Tab Options* for the list of available options.

forget(*tab_id*)

Removes the tab specified by *tab_id*, unmaps and unmanages the associated window.

hide(*tab_id*)

Hides the tab specified by *tab_id*.

The tab will not be displayed, but the associated window remains managed by the notebook and its configuration remembered. Hidden tabs may be restored with the `add()` command.

identify(*x*, *y*)

Returns the name of the tab element at position *x*, *y*, or the empty string if none.

index(*tab_id*)

Returns the numeric index of the tab specified by *tab_id*, or the total number of tabs if *tab_id* is the string "end".

insert(*pos*, *child*, ***kw*)

Inserts a pane at the specified position.

pos is either the string "end", an integer index, or the name of a managed child. If *child* is already managed by the notebook, moves it to the specified position.

See *Tab Options* for the list of available options.

select(*tab_id=None*)

Selects the specified *tab_id*.

The associated child window will be displayed, and the previously selected window (if different) is unmapped. If *tab_id* is omitted, returns the widget name of the currently selected pane.

tab(*tab_id*, *option=None*, ***kw*)

Query or modify the options of the specific *tab_id*.

If *kw* is not given, returns a dictionary of the tab option values. If *option* is specified, returns the value of that *option*. Otherwise, sets the options to the corresponding values.

tabs()

Returns a list of windows managed by the notebook.

enable_traversal()

Enable keyboard traversal for a toplevel window containing this notebook.

This will extend the bindings for the toplevel window containing the notebook as follows:

- `Control-Tab`: selects the tab following the currently selected one.

- `Shift-Control-Tab`: selects the tab preceding the currently selected one.
- `Alt-K`: where *K* is the mnemonic (underlined) character of any tab, will select that tab.

Multiple notebooks in a single toplevel may be enabled for traversal, including nested notebooks. However, notebook traversal only works properly if all panes have the notebook they are in as master.

25.8.7 Progressbar

The `ttk.Progressbar` widget shows the status of a long-running operation. It can operate in two modes: 1) the determinate mode which shows the amount completed relative to the total amount of work to be done and 2) the indeterminate mode which provides an animated display to let the user know that work is progressing.

Options

This widget accepts the following specific options:

Option	Description
orient	One of "horizontal" or "vertical". Specifies the orientation of the progress bar.
length	Specifies the length of the long axis of the progress bar (width if horizontal, height if vertical).
mode	One of "determinate" or "indeterminate".
maximum	A number specifying the maximum value. Defaults to 100.
value	The current value of the progress bar. In "determinate" mode, this represents the amount of work completed. In "indeterminate" mode, it is interpreted as modulo *maximum*; that is, the progress bar completes one "cycle" when its value increases by *maximum*.
variable	A name which is linked to the option value. If specified, the value of the progress bar is automatically set to the value of this name whenever the latter is modified.
phase	Read-only option. The widget periodically increments the value of this option whenever its value is greater than 0 and, in determinate mode, less than maximum. This option may be used by the current theme to provide additional animation effects.

ttk.Progressbar

class tkinter.ttk.**Progressbar**

start(*interval=None*)

Begin autoincrement mode: schedules a recurring timer event that calls *Progressbar.step()* every *interval* milliseconds. If omitted, *interval* defaults to 50 milliseconds.

step(*amount=None*)

Increments the progress bar's value by *amount*.

amount defaults to 1.0 if omitted.

stop()

Stop autoincrement mode: cancels any recurring timer event initiated by *Progressbar.start()* for this progress bar.

25.8.8 Separator

The `ttk.Separator` widget displays a horizontal or vertical separator bar.

It has no other methods besides the ones inherited from `ttk.Widget`.

Options

This widget accepts the following specific option:

Option	Description
orient	One of "horizontal" or "vertical". Specifies the orientation of the separator.

25.8.9 Sizegrip

The `ttk.Sizegrip` widget (also known as a grow box) allows the user to resize the containing toplevel window by pressing and dragging the grip.

This widget has neither specific options nor specific methods, besides the ones inherited from `ttk.Widget`.

Platform-specific notes

- On macOS, toplevel windows automatically include a built-in size grip by default. Adding a `Sizegrip` is harmless, since the built-in grip will just mask the widget.

Bugs

- If the containing toplevel's position was specified relative to the right or bottom of the screen (e.g. ....), the `Sizegrip` widget will not resize the window.
- This widget supports only "southeast" resizing.

25.8.10 Treeview

The `ttk.Treeview` widget displays a hierarchical collection of items. Each item has a textual label, an optional image, and an optional list of data values. The data values are displayed in successive columns after the tree label.

The order in which data values are displayed may be controlled by setting the widget option `displaycolumns`. The tree widget can also display column headings. Columns may be accessed by number or symbolic names listed in the widget option columns. See *Column Identifiers*.

Each item is identified by a unique name. The widget will generate item IDs if they are not supplied by the caller. There is a distinguished root item, named `{}`. The root item itself is not displayed; its children appear at the top level of the hierarchy.

Each item also has a list of tags, which can be used to associate event bindings with individual items and control the appearance of the item.

The Treeview widget supports horizontal and vertical scrolling, according to the options described in *Scrollable Widget Options* and the methods *`Treeview.xview()`* and *`Treeview.yview()`*.

Options

This widget accepts the following specific options:

Option	Description
columns	A list of column identifiers, specifying the number of columns and their names.
displaycolumns	A list of column identifiers (either symbolic or integer indices) specifying which data columns are displayed and the order in which they appear, or the string "#all".
height	Specifies the number of rows which should be visible. Note: the requested width is determined from the sum of the column widths.
padding	Specifies the internal padding for the widget. The padding is a list of up to four length specifications.
selectmode	Controls how the built-in class bindings manage the selection. One of "extended", "browse" or "none". If set to "extended" (the default), multiple items may be selected. If "browse", only a single item will be selected at a time. If "none", the selection will not be changed. Note that the application code and tag bindings can set the selection however they wish, regardless of the value of this option.
show	A list containing zero or more of the following values, specifying which elements of the tree to display. • tree: display tree labels in column #0. • headings: display the heading row. The default is "tree headings", i.e., show all elements. **Note**: Column #0 always refers to the tree column, even if show="tree" is not specified.

Item Options

The following item options may be specified for items in the insert and item widget commands.

Option	Description
text	The textual label to display for the item.
image	A Tk Image, displayed to the left of the label.
values	The list of values associated with the item. Each item should have the same number of values as the widget option columns. If there are fewer values than columns, the remaining values are assumed empty. If there are more values than columns, the extra values are ignored.
open	`True`/`False` value indicating whether the item's children should be displayed or hidden.
tags	A list of tags associated with this item.

Tag Options

The following options may be specified on tags:

Option	Description
foreground	Specifies the text foreground color.
background	Specifies the cell or item background color.
font	Specifies the font to use when drawing text.
image	Specifies the item image, in case the item's image option is empty.

Column Identifiers

Column identifiers take any of the following forms:

- A symbolic name from the list of columns option.
- An integer n, specifying the nth data column.
- A string of the form #n, where n is an integer, specifying the nth display column.

Notes:

- Item's option values may be displayed in a different order than the order in which they are stored.
- Column #0 always refers to the tree column, even if show="tree" is not specified.

A data column number is an index into an item's option values list; a display column number is the column number in the tree where the values are displayed. Tree labels are displayed in column #0. If option displaycolumns is not set, then data column n is displayed in column #n+1. Again, **column #0 always refers to the tree column**.

Virtual Events

The Treeview widget generates the following virtual events.

Event	Description
<<TreeviewSelect>>	Generated whenever the selection changes.
<<TreeviewOpen>>	Generated just before settings the focus item to open=True.
<<TreeviewClose>>	Generated just after setting the focus item to open=False.

The `Treeview.focus()` and `Treeview.selection()` methods can be used to determine the affected item or items.

ttk.Treeview

class tkinter.ttk.**Treeview**

bbox(*item*, *column=None*)

Returns the bounding box (relative to the treeview widget's window) of the specified *item* in the form (x, y, width, height).

If *column* is specified, returns the bounding box of that cell. If the *item* is not visible (i.e., if it is a descendant of a closed item or is scrolled offscreen), returns an empty string.

get_children(*item=None*)

Returns the list of children belonging to *item*.

If *item* is not specified, returns root children.

set_children(*item*, **newchildren*)

Replaces *item*'s child with *newchildren*.

Children present in *item* that are not present in *newchildren* are detached from the tree. No items in *newchildren* may be an ancestor of *item*. Note that not specifying *newchildren* results in detaching *item*'s children.

column(*column*, *option=None*, ***kw*)

Query or modify the options for the specified *column*.

If *kw* is not given, returns a dict of the column option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

id
Returns the column name. This is a read-only option.

***anchor*: One of the standard Tk anchor values.**
Specifies how the text in this column should be aligned with respect to the cell.

***minwidth*: width**
The minimum width of the column in pixels. The treeview widget will not make the column any smaller than specified by this option when the widget is resized or the user drags a column.

***stretch*: `True`/`False`**
Specifies whether the column's width should be adjusted when the widget is resized.

***width*: width**
The width of the column in pixels.

To configure the tree column, call this with column = "#0"

delete(**items*)

Delete all specified *items* and all their descendants.

The root item may not be deleted.

detach(**items*)

Unlinks all of the specified *items* from the tree.

The items and all of their descendants are still present, and may be reinserted at another point in the tree, but will not be displayed.

The root item may not be detached.

exists(*item*)

Returns `True` if the specified *item* is present in the tree.

focus(*item=None*)

If *item* is specified, sets the focus item to *item*. Otherwise, returns the current focus item, or '' if there is none.

heading(*column*, *option=None*, ***kw*)

Query or modify the heading options for the specified *column*.

If *kw* is not given, returns a dict of the heading option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

***text*: text**
The text to display in the column heading.

***image*: imageName**
Specifies an image to display to the right of the column heading.

***anchor*: anchor**
Specifies how the heading text should be aligned. One of the standard Tk anchor values.

***command*: callback**
A callback to be invoked when the heading label is pressed.

To configure the tree column heading, call this with column = "#0".

identify (*component*, *x*, *y*)

Returns a description of the specified *component* under the point given by *x* and *y*, or the empty string if no such *component* is present at that position.

identify_row (*y*)

Returns the item ID of the item at position *y*.

identify_column (*x*)

Returns the data column identifier of the cell at position *x*.

The tree column has ID #0.

identify_region (*x*, *y*)

Returns one of:

region	meaning
heading	Tree heading area.
separator	Space between two columns headings.
tree	The tree area.
cell	A data cell.

Availability: Tk 8.6.

identify_element (*x*, *y*)

Returns the element at position *x*, *y*.

Availability: Tk 8.6.

index (*item*)

Returns the integer index of *item* within its parent's list of children.

insert (*parent*, *index*, *iid=None*, ***kw*)

Creates a new item and returns the item identifier of the newly created item.

parent is the item ID of the parent item, or the empty string to create a new top-level item. *index* is an integer, or the value "end", specifying where in the list of parent's children to insert the new item. If *index* is less than or equal to zero, the new node is inserted at the beginning; if *index* is greater than or equal to the current number of children, it is inserted at the end. If *iid* is specified, it is used as the item identifier; *iid* must not already exist in the tree. Otherwise, a new unique identifier is generated.

See *Item Options* for the list of available options.

item (*item*, *option=None*, ***kw*)

Query or modify the options for the specified *item*.

If no options are given, a dict with options/values for the item is returned. If *option* is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values as given by *kw*.

move (*item*, *parent*, *index*)

Moves *item* to position *index* in *parent*'s list of children.

It is illegal to move an item under one of its descendants. If *index* is less than or equal to zero, *item* is moved to the beginning; if greater than or equal to the number of children, it is moved to the end. If *item* was detached it is reattached.

next (*item*)

Returns the identifier of *item*'s next sibling, or '' if *item* is the last child of its parent.

parent (*item*)

Returns the ID of the parent of *item*, or '' if *item* is at the top level of the hierarchy.

prev (*item*)

Returns the identifier of *item*'s previous sibling, or '' if *item* is the first child of its parent.

reattach (*item*, *parent*, *index*)

An alias for *Treeview.move()*.

see (*item*)

Ensure that *item* is visible.

Sets all of *item*'s ancestors open option to `True`, and scrolls the widget if necessary so that *item* is within the visible portion of the tree.

selection ()

Returns a tuple of selected items.

Changed in version 3.8: `selection()` no longer takes arguments. For changing the selection state use the following selection methods.

selection_set (**items*)

items becomes the new selection.

Changed in version 3.6: *items* can be passed as separate arguments, not just as a single tuple.

selection_add (**items*)

Add *items* to the selection.

Changed in version 3.6: *items* can be passed as separate arguments, not just as a single tuple.

selection_remove (**items*)

Remove *items* from the selection.

Changed in version 3.6: *items* can be passed as separate arguments, not just as a single tuple.

selection_toggle (**items*)

Toggle the selection state of each item in *items*.

Changed in version 3.6: *items* can be passed as separate arguments, not just as a single tuple.

set (*item*, *column=None*, *value=None*)

With one argument, returns a dictionary of column/value pairs for the specified *item*. With two arguments, returns the current value of the specified *column*. With three arguments, sets the value of given *column* in given *item* to the specified *value*.

tag_bind (*tagname*, *sequence=None*, *callback=None*)

Bind a callback for the given event *sequence* to the tag *tagname*. When an event is delivered to an item, the callbacks for each of the item's tags option are called.

tag_configure (*tagname*, *option=None*, ***kw*)

Query or modify the options for the specified *tagname*.

If *kw* is not given, returns a dict of the option settings for *tagname*. If *option* is specified, returns the value for that *option* for the specified *tagname*. Otherwise, sets the options to the corresponding values for the given *tagname*.

tag_has (*tagname*, *item=None*)

If *item* is specified, returns 1 or 0 depending on whether the specified *item* has the given *tagname*. Otherwise, returns a list of all items that have the specified tag.

Availability: Tk 8.6

xview (**args*)

Query or modify horizontal position of the treeview.

yview (**args*)

Query or modify vertical position of the treeview.

25.8.11 Ttk Styling

Each widget in `ttk` is assigned a style, which specifies the set of elements making up the widget and how they are arranged, along with dynamic and default settings for element options. By default the style name is the same as the widget's class name, but it may be overridden by the widget's style option. If you don't know the class name of a widget, use the method `Misc.winfo_class()` (somewidget.winfo_class()).

See also

Tcl'2004 conference presentation
This document explains how the theme engine works

class tkinter.ttk.**Style**

This class is used to manipulate the style database.

configure (*style*, *query_opt=None*, ***kw*)

Query or set the default value of the specified option(s) in *style*.

Each key in *kw* is an option and each value is a string identifying the value for that option.

For example, to change every default button to be a flat button with some padding and a different background color:

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

ttk.Style().configure("TButton", padding=6, relief="flat",
   background="#ccc")

btn = ttk.Button(text="Sample")
```

(continues on next page)

(continued from previous page)

```
btn.pack()

root.mainloop()
```

map (*style*, *query_opt=None*, ***kw*)

Query or sets dynamic values of the specified option(s) in *style*.

Each key in *kw* is an option and each value should be a list or a tuple (usually) containing statespecs grouped in tuples, lists, or some other preference. A statespec is a compound of one or more states and then a value.

An example may make it more understandable:

```
import tkinter
from tkinter import ttk

root = tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
    background=[('pressed', '!disabled', 'black'), ('active', 'white')]
    )

colored_btn = ttk.Button(text="Test", style="C.TButton").pack()

root.mainloop()
```

Note that the order of the (states, value) sequences for an option does matter, if the order is changed to `[('active', 'blue'), ('pressed', 'red')]` in the foreground option, for example, the result would be a blue foreground when the widget were in active or pressed states.

lookup (*style*, *option*, *state=None*, *default=None*)

Returns the value specified for *option* in *style*.

If *state* is specified, it is expected to be a sequence of one or more states. If the *default* argument is set, it is used as a fallback value in case no specification for option is found.

To check what font a Button uses by default:

```
from tkinter import ttk

print(ttk.Style().lookup("TButton", "font"))
```

layout (*style*, *layoutspec=None*)

Define the widget layout for given *style*. If *layoutspec* is omitted, return the layout specification for given style.

layoutspec, if specified, is expected to be a list or some other sequence type (excluding strings), where each item should be a tuple and the first item is the layout name and the second item should have the format described in *Layouts*.

To understand the format, see the following example (it is not intended to do anything useful):

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()
```

(continues on next page)

(continued from previous page)

```
style = ttk.Style()
style.layout("TMenubutton", [
    ("Menubutton.background", None),
    ("Menubutton.button", {"children":
        [("Menubutton.focus", {"children":
            [("Menubutton.padding", {"children":
                [("Menubutton.label", {"side": "left", "expand": 1})]
            })]
        })]
    }),
])

mbtn = ttk.Menubutton(text='Text')
mbtn.pack()
root.mainloop()
```

element_create (*elementname*, *etype*, **args*, ***kw*)

Create a new element in the current theme, of the given *etype* which is expected to be either "image", "from" or "vsapi". The latter is only available in Tk 8.6 on Windows.

If "image" is used, *args* should contain the default image name followed by statespec/value pairs (this is the imagespec), and *kw* may have the following options:

border=padding
: padding is a list of up to four integers, specifying the left, top, right, and bottom borders, respectively.

height=height
: Specifies a minimum height for the element. If less than zero, the base image's height is used as a default.

padding=padding
: Specifies the element's interior padding. Defaults to border's value if not specified.

sticky=spec
: Specifies how the image is placed within the final parcel. spec contains zero or more characters "n", "s", "w", or "e".

width=width
: Specifies a minimum width for the element. If less than zero, the base image's width is used as a default.

Example:

```
img1 = tkinter.PhotoImage(master=root, file='button.png')
img1 = tkinter.PhotoImage(master=root, file='button-pressed.png')
img1 = tkinter.PhotoImage(master=root, file='button-active.png')
style = ttk.Style(root)
style.element_create('Button.button', 'image',
                     img1, ('pressed', img2), ('active', img3),
                     border=(2, 4), sticky='we')
```

If "from" is used as the value of *etype*, `element_create()` will clone an existing element. *args* is expected to contain a themename, from which the element will be cloned, and optionally an element to clone from. If this element to clone from is not specified, an empty element will be used. *kw* is discarded.

Example:

```
style = ttk.Style(root)
style.element_create('plain.background', 'from', 'default')
```

If "vsapi" is used as the value of *etype*, `element_create()` will create a new element in the current theme whose visual appearance is drawn using the Microsoft Visual Styles API which is responsible for the themed styles on Windows XP and Vista. *args* is expected to contain the Visual Styles class and part as given in the Microsoft documentation followed by an optional sequence of tuples of ttk states and the corresponding Visual Styles API state value. *kw* may have the following options:

padding=padding
Specify the element's interior padding. *padding* is a list of up to four integers specifying the left, top, right and bottom padding quantities respectively. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left. In other words, a list of three numbers specify the left, vertical, and right padding; a list of two numbers specify the horizontal and the vertical padding; a single number specifies the same padding all the way around the widget. This option may not be mixed with any other options.

margins=padding
Specifies the elements exterior padding. *padding* is a list of up to four integers specifying the left, top, right and bottom padding quantities respectively. This option may not be mixed with any other options.

width=width
Specifies the width for the element. If this option is set then the Visual Styles API will not be queried for the recommended size or the part. If this option is set then *height* should also be set. The *width* and *height* options cannot be mixed with the *padding* or *margins* options.

height=height
Specifies the height of the element. See the comments for *width*.

Example:

```
style = ttk.Style(root)
style.element_create('pin', 'vsapi', 'EXPLORERBAR', 3, [
                     ('pressed', '!selected', 3),
                     ('active', '!selected', 2),
                     ('pressed', 'selected', 6),
                     ('active', 'selected', 5),
                     ('selected', 4),
                     ('', 1)])
style.layout('Explorer.Pin',
             [('Explorer.Pin.pin', {'sticky': 'news'})])
pin = ttk.Checkbutton(style='Explorer.Pin')
pin.pack(expand=True, fill='both')
```

Changed in version 3.13: Added support of the "vsapi" element factory.

element_names()
Returns the list of elements defined in the current theme.

element_options(*elementname*)
Returns the list of *elementname*'s options.

theme_create(*themename*, *parent=None*, *settings=None*)
Create a new theme.

It is an error if *themename* already exists. If *parent* is specified, the new theme will inherit styles, elements and layouts from the parent theme. If *settings* are present they are expected to have the same syntax used for `theme_settings()`.

theme_settings(*themename*, *settings*)
Temporarily sets the current theme to *themename*, apply specified *settings* and then restore the previous theme.

Each key in *settings* is a style and each value may contain the keys 'configure', 'map', 'layout' and 'element create' and they are expected to have the same format as specified by the methods `Style.configure()`, `Style.map()`, `Style.layout()` and `Style.element_create()` respectively.

As an example, let's change the Combobox for the default theme a bit:

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
   "TCombobox": {
       "configure": {"padding": 5},
       "map": {
           "background": [("active", "green2"),
                          ("!disabled", "green4")],
           "fieldbackground": [("!disabled", "green3")],
           "foreground": [("focus", "OliveDrab1"),
                          ("!disabled", "OliveDrab2")]
       }
   }
})

combo = ttk.Combobox().pack()

root.mainloop()
```

theme_names()

Returns a list of all known themes.

theme_use(*themename=None*)

If *themename* is not given, returns the theme in use. Otherwise, sets the current theme to *themename*, refreshes all widgets and emits a <<ThemeChanged>> event.

Layouts

A layout can be just `None`, if it takes no options, or a dict of options specifying how to arrange the element. The layout mechanism uses a simplified version of the pack geometry manager: given an initial cavity, each element is allocated a parcel.

The valid options/values are:

***side*: whichside**

Specifies which side of the cavity to place the element; one of top, right, bottom or left. If omitted, the element occupies the entire cavity.

***sticky*: nswe**

Specifies where the element is placed inside its allocated parcel.

***unit*: 0 or 1**

If set to 1, causes the element and all of its descendants to be treated as a single element for the purposes of `Widget.identify()` et al. It's used for things like scrollbar thumbs with grips.

***children*: [sublayout...]**

Specifies a list of elements to place inside the element. Each element is a tuple (or other sequence type) where the first item is the layout name, and the other is a *Layout*.

25.9 IDLE

Source code: Lib/idlelib/

IDLE is Python's Integrated Development and Learning Environment.

IDLE has the following features:

- cross-platform: works mostly the same on Windows, Unix, and macOS
- Python shell window (interactive interpreter) with colorizing of code input, output, and error messages
- multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto completion, and other features
- search within any window, replace within editor windows, and search through multiple files (grep)
- debugger with persistent breakpoints, stepping, and viewing of global and local namespaces
- configuration, browsers, and other dialogs

25.9.1 Menus

IDLE has two main window types, the Shell window and the Editor window. It is possible to have multiple editor windows simultaneously. On Windows and Linux, each has its own top menu. Each menu documented below indicates which window type it is associated with.

Output windows, such as used for Edit => Find in Files, are a subtype of editor window. They currently have the same top menu but a different default title and context menu.

On macOS, there is one application menu. It dynamically changes according to the window currently selected. It has an IDLE menu, and some entries described below are moved around to conform to Apple guidelines.

File menu (Shell and Editor)

New File
: Create a new file editing window.

Open…
: Open an existing file with an Open dialog.

Open Module…
: Open an existing module (searches sys.path).

Recent Files
: Open a list of recent files. Click one to open it.

Module Browser
: Show functions, classes, and methods in the current Editor file in a tree structure. In the shell, open a module first.

Path Browser
: Show sys.path directories, modules, functions, classes and methods in a tree structure.

Save
: Save the current window to the associated file, if there is one. Windows that have been changed since being opened or last saved have a * before and after the window title. If there is no associated file, do Save As instead.

Save As…
Save the current window with a Save As dialog. The file saved becomes the new associated file for the window. (If your file namager is set to hide extensions, the current extension will be omitted in the file name box. If the new filename has no '.', '.py' and '.txt' will be added for Python and text files, except that on macOS Aqua,'.py' is added for all files.)

Save Copy As…
Save the current window to different file without changing the associated file. (See Save As note above about filename extensions.)

Print Window
Print the current window to the default printer.

Close Window
Close the current window (if an unsaved editor, ask to save; if an unsaved Shell, ask to quit execution). Calling `exit()` or `close()` in the Shell window also closes Shell. If this is the only window, also exit IDLE.

Exit IDLE
Close all windows and quit IDLE (ask to save unsaved edit windows).

Edit menu (Shell and Editor)

Undo
Undo the last change to the current window. A maximum of 1000 changes may be undone.

Redo
Redo the last undone change to the current window.

Select All
Select the entire contents of the current window.

Cut
Copy selection into the system-wide clipboard; then delete the selection.

Copy
Copy selection into the system-wide clipboard.

Paste
Insert contents of the system-wide clipboard into the current window.

The clipboard functions are also available in context menus.

Find…
Open a search dialog with many options

Find Again
Repeat the last search, if there is one.

Find Selection
Search for the currently selected string, if there is one.

Find in Files…
Open a file search dialog. Put results in a new output window.

Replace…
Open a search-and-replace dialog.

Go to Line
Move the cursor to the beginning of the line requested and make that line visible. A request past the end of the file goes to the end. Clear any selection and update the line and column status.

Show Completions
Open a scrollable list allowing selection of existing names. See *Completions* in the Editing and navigation section below.

Expand Word
Expand a prefix you have typed to match a full word in the same window; repeat to get a different expansion.

Show Call Tip
After an unclosed parenthesis for a function, open a small window with function parameter hints. See *Calltips* in the Editing and navigation section below.

Show Surrounding Parens
Highlight the surrounding parenthesis.

Format menu (Editor window only)

Format Paragraph
Reformat the current blank-line-delimited paragraph in comment block or multiline string or selected line in a string. All lines in the paragraph will be formatted to less than N columns, where N defaults to 72.

Indent Region
Shift selected lines right by the indent width (default 4 spaces).

Dedent Region
Shift selected lines left by the indent width (default 4 spaces).

Comment Out Region
Insert ## in front of selected lines.

Uncomment Region
Remove leading # or ## from selected lines.

Tabify Region
Turn *leading* stretches of spaces into tabs. (Note: We recommend using 4 space blocks to indent Python code.)

Untabify Region
Turn *all* tabs into the correct number of spaces.

Toggle Tabs
Open a dialog to switch between indenting with spaces and tabs.

New Indent Width
Open a dialog to change indent width. The accepted default by the Python community is 4 spaces.

Strip Trailing Chitespace
Remove trailing space and other whitespace characters after the last non-whitespace character of a line by applying str.rstrip to each line, including lines within multiline strings. Except for Shell windows, remove extra newlines at the end of the file.

Run menu (Editor window only)

Run Module
Do *Check Module*. If no error, restart the shell to clean the environment, then execute the module. Output is displayed in the Shell window. Note that output requires use of `print` or `write`. When execution is complete, the Shell retains focus and displays a prompt. At this point, one may interactively explore the result of execution. This is similar to executing a file with `python -i file` at a command line.

Run… Customized
Same as *Run Module*, but run the module with customized settings. *Command Line Arguments* extend `sys.argv` as if passed on a command line. The module can be run in the Shell without restarting.

Check Module
Check the syntax of the module currently open in the Editor window. If the module has not been saved IDLE will either prompt the user to save or autosave, as selected in the General tab of the Idle Settings dialog. If there is a syntax error, the approximate location is indicated in the Editor window.

Python Shell
Open or wake up the Python Shell window.

Shell menu (Shell window only)

View Last Restart
Scroll the shell window to the last Shell restart.

Restart Shell
Restart the shell to clean the environment and reset display and exception handling.

Previous History
Cycle through earlier commands in history which match the current entry.

Next History
Cycle through later commands in history which match the current entry.

Interrupt Execution
Stop a running program.

Debug menu (Shell window only)

Go to File/Line
Look on the current line. with the cursor, and the line above for a filename and line number. If found, open the file if not already open, and show the line. Use this to view source lines referenced in an exception traceback and lines found by Find in Files. Also available in the context menu of the Shell window and Output windows.

Debugger (toggle)
When activated, code entered in the Shell or run from an Editor will run under the debugger. In the Editor, breakpoints can be set with the context menu. This feature is still incomplete and somewhat experimental.

Stack Viewer
Show the stack traceback of the last exception in a tree widget, with access to locals and globals.

Auto-open Stack Viewer
Toggle automatically opening the stack viewer on an unhandled exception.

Options menu (Shell and Editor)

Configure IDLE
Open a configuration dialog and change preferences for the following: fonts, indentation, keybindings, text color themes, startup windows and size, additional help sources, and extensions. On macOS, open the configuration dialog by selecting Preferences in the application menu. For more details, see *Setting preferences* under Help and preferences.

Most configuration options apply to all windows or all future windows. The option items below only apply to the active window.

Show/Hide Code Context (Editor Window only)
Open a pane at the top of the edit window which shows the block context of the code which has scrolled above the top of the window. See *Code Context* in the Editing and Navigation section below.

Show/Hide Line Numbers (Editor Window only)
Open a column to the left of the edit window which shows the number of each line of text. The default is off, which may be changed in the preferences (see *Setting preferences*).

Zoom/Restore Height
Toggles the window between normal size and maximum height. The initial size defaults to 40 lines by 80 chars unless changed on the General tab of the Configure IDLE dialog. The maximum height for a screen is determined by momentarily maximizing a window the first time one is zoomed on the screen. Changing screen settings may invalidate the saved height. This toggle has no effect when a window is maximized.

Window menu (Shell and Editor)

Lists the names of all open windows; select one to bring it to the foreground (deiconifying it if necessary).

Help menu (Shell and Editor)

About IDLE
Display version, copyright, license, credits, and more.

IDLE Help
Display this IDLE document, detailing the menu options, basic editing and navigation, and other tips.

Python Docs
Access local Python documentation, if installed, or start a web browser and open docs.python.org showing the latest Python documentation.

Turtle Demo
Run the turtledemo module with example Python code and turtle drawings.

Additional help sources may be added here with the Configure IDLE dialog under the General tab. See the *Help sources* subsection below for more on Help menu choices.

Context menus

Open a context menu by right-clicking in a window (Control-click on macOS). Context menus have the standard clipboard functions also on the Edit menu.

Cut
Copy selection into the system-wide clipboard; then delete the selection.

Copy
Copy selection into the system-wide clipboard.

Paste
Insert contents of the system-wide clipboard into the current window.

Editor windows also have breakpoint functions. Lines with a breakpoint set are specially marked. Breakpoints only have an effect when running under the debugger. Breakpoints for a file are saved in the user's `.idlerc` directory.

Set Breakpoint
Set a breakpoint on the current line.

Clear Breakpoint
Clear the breakpoint on that line.

Shell and Output windows also have the following.

Go to file/line
Same as in Debug menu.

The Shell window also has an output squeezing facility explained in the *Python Shell window* subsection below.

Squeeze
If the cursor is over an output line, squeeze all the output between the code above and the prompt below down to a 'Squeezed text' label.

25.9.2 Editing and Navigation

Editor windows

IDLE may open editor windows when it starts, depending on settings and how you start IDLE. Thereafter, use the File menu. There can be only one open editor window for a given file.

The title bar contains the name of the file, the full path, and the version of Python and IDLE running the window. The status bar contains the line number ('Ln') and column number ('Col'). Line numbers start with 1; column numbers with 0.

IDLE assumes that files with a known .py* extension contain Python code and that other files do not. Run Python code with the Run menu.

Key bindings

The IDLE insertion cursor is a thin vertical bar between character positions. When characters are entered, the insertion cursor and everything to its right moves right one character and the new character is entered in the new space.

Several non-character keys move the cursor and possibly delete characters. Deletion does not puts text on the clipboard, but IDLE has an undo list. Wherever this doc discusses keys, 'C' refers to the `Control` key on Windows and Unix and the `Command` key on macOS. (And all such discussions assume that the keys have not been re-bound to something else.)

- Arrow keys move the cursor one character or line.

- C-LeftArrow and C-RightArrow moves left or right one word.
- Home and End go to the beginning or end of the line.
- Page Up and Page Down go up or down one screen.
- C-Home and C-End go to beginning or end of the file.
- Backspace and Del (or C-d) delete the previous or next character.
- C-Backspace and C-Del delete one word left or right.
- C-k deletes ('kills') everything to the right.

Standard keybindings (like C-c to copy and C-v to paste) may work. Keybindings are selected in the Configure IDLE dialog.

Automatic indentation

After a block-opening statement, the next line is indented by 4 spaces (in the Python Shell window by one tab). After certain keywords (break, return etc.) the next line is dedented. In leading indentation, Backspace deletes up to 4 spaces if they are there. Tab inserts spaces (in the Python Shell window one tab), number depends on Indent width. Currently, tabs are restricted to four spaces due to Tcl/Tk limitations.

See also the indent/dedent region commands on the *Format menu*.

Search and Replace

Any selection becomes a search target. However, only selections within a line work because searches are only performed within lines with the terminal newline removed. If [x] Regular expression is checked, the target is interpreted according to the Python re module.

Completions

Completions are supplied, when requested and available, for module names, attributes of classes or functions, or filenames. Each request method displays a completion box with existing names. (See tab completions below for an exception.) For any box, change the name being completed and the item highlighted in the box by typing and deleting characters; by hitting Up, Down, PageUp, PageDown, Home, and End keys; and by a single click within the box. Close the box with Escape, Enter, and double Tab keys or clicks outside the box. A double click within the box selects and closes.

One way to open a box is to type a key character and wait for a predefined interval. This defaults to 2 seconds; customize it in the settings dialog. (To prevent auto popups, set the delay to a large number of milliseconds, such as 100000000.) For imported module names or class or function attributes, type '.'. For filenames in the root directory, type *os.sep* or *os.altsep* immediately after an opening quote. (On Windows, one can specify a drive first.) Move into subdirectories by typing a directory name and a separator.

Instead of waiting, or after a box is closed, open a completion box immediately with Show Completions on the Edit menu. The default hot key is C-space. If one types a prefix for the desired name before opening the box, the first match or near miss is made visible. The result is the same as if one enters a prefix after the box is displayed. Show Completions after a quote completes filenames in the current directory instead of a root directory.

Hitting Tab after a prefix usually has the same effect as Show Completions. (With no prefix, it indents.) However, if there is only one match to the prefix, that match is immediately added to the editor text without opening a box.

Invoking 'Show Completions', or hitting Tab after a prefix, outside of a string and without a preceding '.' opens a box with keywords, builtin names, and available module-level names.

When editing code in an editor (as oppose to Shell), increase the available module-level names by running your code and not restarting the Shell thereafter. This is especially useful after adding imports at the top of a file. This also increases possible attribute completions.

Completion boxes initially exclude names beginning with '_' or, for modules, not included in '__all__'. The hidden names can be accessed by typing '_' after '.', either before or after the box is opened.

Calltips

A calltip is shown automatically when one types `(` after the name of an *accessible* function. A function name expression may include dots and subscripts. A calltip remains until it is clicked, the cursor is moved out of the argument area, or `)` is typed. Whenever the cursor is in the argument part of a definition, select Edit and "Show Call Tip" on the menu or enter its shortcut to display a calltip.

The calltip consists of the function's signature and docstring up to the latter's first blank line or the fifth non-blank line. (Some builtin functions lack an accessible signature.) A '/' or '*' in the signature indicates that the preceding or following arguments are passed by position or name (keyword) only. Details are subject to change.

In Shell, the accessible functions depends on what modules have been imported into the user process, including those imported by Idle itself, and which definitions have been run, all since the last restart.

For example, restart the Shell and enter `itertools.count(`. A calltip appears because Idle imports itertools into the user process for its own use. (This could change.) Enter `turtle.write(` and nothing appears. Idle does not itself import turtle. The menu entry and shortcut also do nothing. Enter `import turtle`. Thereafter, `turtle.write(` will display a calltip.

In an editor, import statements have no effect until one runs the file. One might want to run a file after writing import statements, after adding function definitions, or after opening an existing file.

Code Context

Within an editor window containing Python code, code context can be toggled in order to show or hide a pane at the top of the window. When shown, this pane freezes the opening lines for block code, such as those beginning with `class`, `def`, or `if` keywords, that would have otherwise scrolled out of view. The size of the pane will be expanded and contracted as needed to show the all current levels of context, up to the maximum number of lines defined in the Configure IDLE dialog (which defaults to 15). If there are no current context lines and the feature is toggled on, a single blank line will display. Clicking on a line in the context pane will move that line to the top of the editor.

The text and background colors for the context pane can be configured under the Highlights tab in the Configure IDLE dialog.

Shell window

In IDLE's Shell, enter, edit, and recall complete statements. (Most consoles and terminals only work with a single physical line at a time).

Submit a single-line statement for execution by hitting `Return` with the cursor anywhere on the line. If a line is extended with Backslash (`\`), the cursor must be on the last physical line. Submit a multi-line compound statement by entering a blank line after the statement.

When one pastes code into Shell, it is not compiled and possibly executed until one hits `Return`, as specified above. One may edit pasted code first. If one pastes more than one statement into Shell, the result will be a *`SyntaxError`* when multiple statements are compiled as if they were one.

Lines containing `RESTART` mean that the user execution process has been re-started. This occurs when the user execution process has crashed, when one requests a restart on the Shell menu, or when one runs code in an editor window.

The editing features described in previous subsections work when entering code interactively. IDLE's Shell window also responds to the following:

- C-c attempts to interrupt statement execution (but may fail).
- C-d closes Shell if typed at a >>> prompt.
- Alt-p and Alt-n (C-p and C-n on macOS) retrieve to the current prompt the previous or next previously entered statement that matches anything already typed.
- Return while the cursor is on any previous statement appends the latter to anything already typed at the prompt.

Text colors

Idle defaults to black on white text, but colors text with special meanings. For the shell, these are shell output, shell error, user output, and user error. For Python code, at the shell prompt or in an editor, these are keywords, builtin class and function names, names following `class` and `def`, strings, and comments. For any text window, these are the cursor (when present), found text (when possible), and selected text.

IDLE also highlights the soft keywords `match`, `case`, and _ in pattern-matching statements. However, this highlighting is not perfect and will be incorrect in some rare cases, including some _-s in `case` patterns.

Text coloring is done in the background, so uncolorized text is occasionally visible. To change the color scheme, use the Configure IDLE dialog Highlighting tab. The marking of debugger breakpoint lines in the editor and text in popups and dialogs is not user-configurable.

25.9.3 Startup and Code Execution

Upon startup with the -s option, IDLE will execute the file referenced by the environment variables `IDLESTARTUP` or `PYTHONSTARTUP`. IDLE first checks for `IDLESTARTUP`; if `IDLESTARTUP` is present the file referenced is run. If `IDLESTARTUP` is not present, IDLE checks for `PYTHONSTARTUP`. Files referenced by these environment variables are convenient places to store functions that are used frequently from the IDLE shell, or for executing import statements to import common modules.

In addition, `Tk` also loads a startup file if it is present. Note that the Tk file is loaded unconditionally. This additional file is `.Idle.py` and is looked for in the user's home directory. Statements in this file will be executed in the Tk namespace, so this file is not useful for importing functions to be used from IDLE's Python shell.

Command line usage

```
idle.py [-c command] [-d] [-e] [-h] [-i] [-r file] [-s] [-t title] [-] [arg] ...

-c command  run command in the shell window
-d          enable debugger and open shell window
-e          open editor window
-h          print help message with legal combinations and exit
-i          open shell window
-r file     run file in shell window
-s          run $IDLESTARTUP or $PYTHONSTARTUP first, in shell window
-t title    set title of shell window
-           run stdin in shell (- must be last option before args)
```

If there are arguments:

- If -, -c, or r is used, all arguments are placed in `sys.argv[1:...]` and `sys.argv[0]` is set to `''`, `'-c'`, or `'-r'`. No editor window is opened, even if that is the default set in the Options dialog.

- Otherwise, arguments are files opened for editing and `sys.argv` reflects the arguments passed to IDLE itself.

Startup failure

IDLE uses a socket to communicate between the IDLE GUI process and the user code execution process. A connection must be established whenever the Shell starts or restarts. (The latter is indicated by a divider line that says 'RESTART'). If the user process fails to connect to the GUI process, it usually displays a `Tk` error box with a 'cannot connect' message that directs the user here. It then exits.

One specific connection failure on Unix systems results from misconfigured masquerading rules somewhere in a system's network setup. When IDLE is started from a terminal, one will see a message starting with `** Invalid host:`. The valid value is `127.0.0.1 (idlelib.rpc.LOCALHOST)`. One can diagnose with `tcpconnect -irv 127.0.0.1 6543` in one terminal window and `tcplisten <same args>` in another.

A common cause of failure is a user-written file with the same name as a standard library module, such as *random.py* and *tkinter.py*. When such a file is located in the same directory as a file that is about to be run, IDLE cannot import the stdlib file. The current fix is to rename the user file.

Though less common than in the past, an antivirus or firewall program may stop the connection. If the program cannot be taught to allow the connection, then it must be turned off for IDLE to work. It is safe to allow this internal connection because no data is visible on external ports. A similar problem is a network mis-configuration that blocks connections.

Python installation issues occasionally stop IDLE: multiple versions can clash, or a single installation might need admin access. If one undo the clash, or cannot or does not want to run as admin, it might be easiest to completely remove Python and start over.

A zombie pythonw.exe process could be a problem. On Windows, use Task Manager to check for one and stop it if there is. Sometimes a restart initiated by a program crash or Keyboard Interrupt (control-C) may fail to connect. Dismissing the error box or using Restart Shell on the Shell menu may fix a temporary problem.

When IDLE first starts, it attempts to read user configuration files in `~/.idlerc/` (~ is one's home directory). If there is a problem, an error message should be displayed. Leaving aside random disk glitches, this can be prevented by never editing the files by hand. Instead, use the configuration dialog, under Options. Once there is an error in a user configuration file, the best solution may be to delete it and start over with the settings dialog.

If IDLE quits with no message, and it was not started from a console, try starting it from a console or terminal (`python -m idlelib`) and see if this results in an error message.

On Unix-based systems with tcl/tk older than `8.6.11` (see `About IDLE`) certain characters of certain fonts can cause a tk failure with a message to the terminal. This can happen either if one starts IDLE to edit a file with such a character or later when entering such a character. If one cannot upgrade tcl/tk, then re-configure IDLE to use a font that works better.

Running user code

With rare exceptions, the result of executing Python code with IDLE is intended to be the same as executing the same code by the default method, directly with Python in a text-mode system console or terminal window. However, the different interface and operation occasionally affect visible results. For instance, `sys.modules` starts with more entries, and `threading.active_count()` returns 2 instead of 1.

By default, IDLE runs user code in a separate OS process rather than in the user interface process that runs the shell and editor. In the execution process, it replaces `sys.stdin`, `sys.stdout`, and `sys.stderr` with objects that get input from and send output to the Shell window. The original values stored in `sys.__stdin__`, `sys.__stdout__`, and `sys.__stderr__` are not touched, but may be `None`.

Sending print output from one process to a text widget in another is slower than printing to a system terminal in the same process. This has the most effect when printing multiple arguments, as the string for each argument, each separator, the newline are sent separately. For development, this is usually not a problem, but if one wants to print faster in IDLE,

format and join together everything one wants displayed together and then print a single string. Both format strings and *str.join()* can help combine fields and lines.

IDLE's standard stream replacements are not inherited by subprocesses created in the execution process, whether directly by user code or by modules such as multiprocessing. If such subprocess use `input` from sys.stdin or `print` or `write` to sys.stdout or sys.stderr, IDLE should be started in a command line window. (On Windows, use `python` or `py` rather than `pythonw` or `pyw`.) The secondary subprocess will then be attached to that window for input and output.

If `sys` is reset by user code, such as with `importlib.reload(sys)`, IDLE's changes are lost and input from the keyboard and output to the screen will not work correctly.

When Shell has the focus, it controls the keyboard and screen. This is normally transparent, but functions that directly access the keyboard and screen will not work. These include system-specific functions that determine whether a key has been pressed and if so, which.

The IDLE code running in the execution process adds frames to the call stack that would not be there otherwise. IDLE wraps `sys.getrecursionlimit` and `sys.setrecursionlimit` to reduce the effect of the additional stack frames.

When user code raises SystemExit either directly or by calling sys.exit, IDLE returns to a Shell prompt instead of exiting.

User output in Shell

When a program outputs text, the result is determined by the corresponding output device. When IDLE executes user code, `sys.stdout` and `sys.stderr` are connected to the display area of IDLE's Shell. Some of its features are inherited from the underlying Tk Text widget. Others are programmed additions. Where it matters, Shell is designed for development rather than production runs.

For instance, Shell never throws away output. A program that sends unlimited output to Shell will eventually fill memory, resulting in a memory error. In contrast, some system text windows only keep the last n lines of output. A Windows console, for instance, keeps a user-settable 1 to 9999 lines, with 300 the default.

A Tk Text widget, and hence IDLE's Shell, displays characters (codepoints) in the BMP (Basic Multilingual Plane) subset of Unicode. Which characters are displayed with a proper glyph and which with a replacement box depends on the operating system and installed fonts. Tab characters cause the following text to begin after the next tab stop. (They occur every 8 'characters'). Newline characters cause following text to appear on a new line. Other control characters are ignored or displayed as a space, box, or something else, depending on the operating system and font. (Moving the text cursor through such output with arrow keys may exhibit some surprising spacing behavior.)

```
>>> s = 'a\tb\a<\x02><\r>\bc\nd'  # Enter 22 chars.
>>> len(s)
14
>>> s  # Display repr(s)
'a\tb\x07<\x02><\r>\x08c\nd'
>>> print(s, end='')  # Display s as is.
# Result varies by OS and font.  Try it.
```

The `repr` function is used for interactive echo of expression values. It returns an altered version of the input string in which control codes, some BMP codepoints, and all non-BMP codepoints are replaced with escape codes. As demonstrated above, it allows one to identify the characters in a string, regardless of how they are displayed.

Normal and error output are generally kept separate (on separate lines) from code input and each other. They each get different highlight colors.

For SyntaxError tracebacks, the normal '^' marking where the error was detected is replaced by coloring the text with an error highlight. When code run from a file causes other exceptions, one may right click on a traceback line to jump to the corresponding line in an IDLE editor. The file will be opened if necessary.

Shell has a special facility for squeezing output lines down to a 'Squeezed text' label. This is done automatically for output over N lines (N = 50 by default). N can be changed in the PyShell section of the General page of the Settings dialog. Output with fewer lines can be squeezed by right clicking on the output. This can be useful lines long enough to slow down scrolling.

Squeezed output is expanded in place by double-clicking the label. It can also be sent to the clipboard or a separate view window by right-clicking the label.

Developing tkinter applications

IDLE is intentionally different from standard Python in order to facilitate development of tkinter programs. Enter `import tkinter as tk; root = tk.Tk()` in standard Python and nothing appears. Enter the same in IDLE and a tk window appears. In standard Python, one must also enter `root.update()` to see the window. IDLE does the equivalent in the background, about 20 times a second, which is about every 50 milliseconds. Next enter `b = tk.Button(root, text='button'); b.pack()`. Again, nothing visibly changes in standard Python until one enters `root.update()`.

Most tkinter programs run `root.mainloop()`, which usually does not return until the tk app is destroyed. If the program is run with `python -i` or from an IDLE editor, a `>>>` shell prompt does not appear until `mainloop()` returns, at which time there is nothing left to interact with.

When running a tkinter program from an IDLE editor, one can comment out the mainloop call. One then gets a shell prompt immediately and can interact with the live application. One just has to remember to re-enable the mainloop call when running in standard Python.

Running without a subprocess

By default, IDLE executes user code in a separate subprocess via a socket, which uses the internal loopback interface. This connection is not externally visible and no data is sent to or received from the internet. If firewall software complains anyway, you can ignore it.

If the attempt to make the socket connection fails, Idle will notify you. Such failures are sometimes transient, but if persistent, the problem may be either a firewall blocking the connection or misconfiguration of a particular system. Until the problem is fixed, one can run Idle with the -n command line switch.

If IDLE is started with the -n command line switch it will run in a single process and will not create the subprocess which runs the RPC Python execution server. This can be useful if Python cannot create the subprocess or the RPC socket interface on your platform. However, in this mode user code is not isolated from IDLE itself. Also, the environment is not restarted when Run/Run Module (F5) is selected. If your code has been modified, you must reload() the affected modules and re-import any specific items (e.g. from foo import baz) if the changes are to take effect. For these reasons, it is preferable to run IDLE with the default subprocess if at all possible.

Deprecated since version 3.4.

25.9.4 Help and Preferences

Help sources

Help menu entry "IDLE Help" displays a formatted html version of the IDLE chapter of the Library Reference. The result, in a read-only tkinter text window, is close to what one sees in a web browser. Navigate through the text with a mousewheel, the scrollbar, or up and down arrow keys held down. Or click the TOC (Table of Contents) button and select a section header in the opened box.

Help menu entry "Python Docs" opens the extensive sources of help, including tutorials, available at `docs.python.org/x.y`, where 'x.y' is the currently running Python version. If your system has an off-line copy of the docs (this may be an installation option), that will be opened instead.

Selected URLs can be added or removed from the help menu at any time using the General tab of the Configure IDLE dialog.

Setting preferences

The font preferences, highlighting, keys, and general preferences can be changed via Configure IDLE on the Option menu. Non-default user settings are saved in a `.idlerc` directory in the user's home directory. Problems caused by bad user configuration files are solved by editing or deleting one or more of the files in `.idlerc`.

On the Font tab, see the text sample for the effect of font face and size on multiple characters in multiple languages. Edit the sample to add other characters of personal interest. Use the sample to select monospaced fonts. If particular characters have problems in Shell or an editor, add them to the top of the sample and try changing first size and then font.

On the Highlights and Keys tab, select a built-in or custom color theme and key set. To use a newer built-in color theme or key set with older IDLEs, save it as a new custom theme or key set and it well be accessible to older IDLEs.

IDLE on macOS

Under System Preferences: Dock, one can set "Prefer tabs when opening documents" to "Always". This setting is not compatible with the tk/tkinter GUI framework used by IDLE, and it breaks a few IDLE features.

Extensions

IDLE contains an extension facility. Preferences for extensions can be changed with the Extensions tab of the preferences dialog. See the beginning of config-extensions.def in the idlelib directory for further information. The only current default extension is zzdummy, an example also used for testing.

25.9.5 idlelib

Source code: Lib/idlelib

The Lib/idlelib package implements the IDLE application. See the rest of this page for how to use IDLE.

The files in idlelib are described in idlelib/README.txt. Access it either in idlelib or click Help => About IDLE on the IDLE menu. This file also maps IDLE menu items to the code that implements the item. Except for files listed under 'Startup', the idlelib code is 'private' in sense that feature changes can be backported (see **PEP 434**).

CHAPTER

TWENTYSIX

DEVELOPMENT TOOLS

The modules described in this chapter help you write software. For example, the *pydoc* module takes a module and generates documentation based on the module's contents. The *doctest* and *unittest* modules contains frameworks for writing unit tests that automatically exercise code and verify that the expected output is produced.

The list of modules described in this chapter is:

26.1 `typing` — Support for type hints

Added in version 3.5.

Source code: Lib/typing.py

Note

The Python runtime does not enforce function and variable type annotations. They can be used by third party tools such as *type checkers*, IDEs, linters, etc.

This module provides runtime support for type hints.

Consider the function below:

```
def surface_area_of_cube(edge_length: float) -> str:
    return f"The surface area of the cube is {6 * edge_length ** 2}."
```

The function `surface_area_of_cube` takes an argument expected to be an instance of *float*, as indicated by the *type hint* `edge_length: float`. The function is expected to return an instance of *str*, as indicated by the `-> str` hint.

While type hints can be simple classes like *float* or *str*, they can also be more complex. The *typing* module provides a vocabulary of more advanced type hints.

New features are frequently added to the `typing` module. The typing_extensions package provides backports of these new features to older versions of Python.

See also

"Typing cheat sheet"
A quick overview of type hints (hosted at the mypy docs)

"Type System Reference" section of the mypy docs
The Python typing system is standardised via PEPs, so this reference should broadly apply to most Python type checkers. (Some parts may still be specific to mypy.)

"Static Typing with Python"
Type-checker-agnostic documentation written by the community detailing type system features, useful typing related tools and typing best practices.

26.1.1 Specification for the Python Type System

The canonical, up-to-date specification of the Python type system can be found at "Specification for the Python type system".

26.1.2 Type aliases

A type alias is defined using the `type` statement, which creates an instance of *TypeAliasType*. In this example, `Vector` and `list[float]` will be treated equivalently by static type checkers:

```
type Vector = list[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# passes type checking; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

Type aliases are useful for simplifying complex type signatures. For example:

```
from collections.abc import Sequence

type ConnectionOptions = dict[str, str]
type Address = tuple[str, int]
type Server = tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: Sequence[Server]) -> None:
    ...

# The static type checker will treat the previous type signature as
# being exactly equivalent to this one.
def broadcast_message(
    message: str,
    servers: Sequence[tuple[tuple[str, int], dict[str, str]]]
) -> None:
    ...
```

The `type` statement is new in Python 3.12. For backwards compatibility, type aliases can also be created through simple assignment:

```
Vector = list[float]
```

Or marked with *TypeAlias* to make it explicit that this is a type alias, not a normal variable assignment:

```
from typing import TypeAlias

Vector: TypeAlias = list[float]
```

26.1.3 NewType

Use the *NewType* helper to create distinct types:

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

The static type checker will treat the new type as if it were a subclass of the original type. This is useful in helping catch logical errors:

```
def get_user_name(user_id: UserId) -> str:
    ...

# passes type checking
user_a = get_user_name(UserId(42351))

# fails type checking; an int is not a UserId
user_b = get_user_name(-1)
```

You may still perform all `int` operations on a variable of type `UserId`, but the result will always be of type `int`. This lets you pass in a `UserId` wherever an `int` might be expected, but will prevent you from accidentally creating a `UserId` in an invalid way:

```
# 'output' is of type 'int', not 'UserId'
output = UserId(23413) + UserId(54341)
```

Note that these checks are enforced only by the static type checker. At runtime, the statement `Derived = NewType('Derived', Base)` will make `Derived` a callable that immediately returns whatever parameter you pass it. That means the expression `Derived(some_value)` does not create a new class or introduce much overhead beyond that of a regular function call.

More precisely, the expression `some_value is Derived(some_value)` is always true at runtime.

It is invalid to create a subtype of `Derived`:

```
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not pass type checking
class AdminUserId(UserId): pass
```

However, it is possible to create a *NewType* based on a 'derived' `NewType`:

```
from typing import NewType

UserId = NewType('UserId', int)

ProUserId = NewType('ProUserId', UserId)
```

and typechecking for `ProUserId` will work as expected.

See **PEP 484** for more details.

> **Note**
>
> Recall that the use of a type alias declares two types to be *equivalent* to one another. Doing `type Alias = Original` will make the static type checker treat `Alias` as being *exactly equivalent* to `Original` in all cases. This is useful when you want to simplify complex type signatures.
>
> In contrast, `NewType` declares one type to be a *subtype* of another. Doing `Derived = NewType('Derived', Original)` will make the static type checker treat `Derived` as a *subclass* of `Original`, which means a value of type `Original` cannot be used in places where a value of type `Derived` is expected. This is useful when you want to prevent logic errors with minimal runtime cost.

Added in version 3.5.2.

Changed in version 3.10: `NewType` is now a class rather than a function. As a result, there is some additional runtime cost when calling `NewType` over a regular function.

Changed in version 3.11: The performance of calling `NewType` has been restored to its level in Python 3.9.

26.1.4 Annotating callable objects

Functions – or other *callable* objects – can be annotated using *collections.abc.Callable* or *typing.Callable*. `Callable[[int], str]` signifies a function that takes a single parameter of type *int* and returns a *str*.

For example:

```
from collections.abc import Callable, Awaitable

def feeder(get_next_item: Callable[[], str]) -> None:
    ...  # Body

def async_query(on_success: Callable[[int], None],
                on_error: Callable[[int, Exception], None]) -> None:
    ...  # Body

async def on_update(value: str) -> None:
    ...  # Body

callback: Callable[[str], Awaitable[None]] = on_update
```

The subscription syntax must always be used with exactly two values: the argument list and the return type. The argument list must be a list of types, a *ParamSpec*, *Concatenate*, or an ellipsis. The return type must be a single type.

If a literal ellipsis `...` is given as the argument list, it indicates that a callable with any arbitrary parameter list would be acceptable:

```
def concat(x: str, y: str) -> str:
    return x + y

x: Callable[..., str]
x = str     # OK
x = concat  # Also OK
```

`Callable` cannot express complex signatures such as functions that take a variadic number of arguments, *overloaded functions*, or functions that have keyword-only parameters. However, these signatures can be expressed by defining a *`Protocol`* class with a `__call__()` method:

```
from collections.abc import Iterable
from typing import Protocol

class Combiner(Protocol):
    def __call__(self, *vals: bytes, maxlen: int | None = None) -> list[bytes]: ...

def batch_proc(data: Iterable[bytes], cb_results: Combiner) -> bytes:
    for item in data:
        ...

def good_cb(*vals: bytes, maxlen: int | None = None) -> list[bytes]:
    ...
def bad_cb(*vals: bytes, maxitems: int | None) -> list[bytes]:
    ...

batch_proc([], good_cb)  # OK
batch_proc([], bad_cb)   # Error! Argument 2 has incompatible type because of
                         # different name and kind in the callback
```

Callables which take other callables as arguments may indicate that their parameter types are dependent on each other using *`ParamSpec`*. Additionally, if that callable adds or removes arguments from other callables, the *`Concatenate`* operator may be used. They take the form `Callable[ParamSpecVariable, ReturnType]` and `Callable[Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable], ReturnType]` respectively.

Changed in version 3.10: `Callable` now supports *`ParamSpec`* and *`Concatenate`*. See **PEP 612** for more details.

See also

The documentation for *`ParamSpec`* and *`Concatenate`* provides examples of usage in `Callable`.

26.1.5 Generics

Since type information about objects kept in containers cannot be statically inferred in a generic way, many container classes in the standard library support subscription to denote the expected types of container elements.

```
from collections.abc import Mapping, Sequence

class Employee: ...

# Sequence[Employee] indicates that all elements in the sequence
# must be instances of "Employee".
# Mapping[str, str] indicates that all keys and all values in the mapping
# must be strings.
def notify_by_email(employees: Sequence[Employee],
                    overrides: Mapping[str, str]) -> None: ...
```

Generic functions and classes can be parameterized by using type parameter syntax:

```
from collections.abc import Sequence

```

(continues on next page)

(continued from previous page)

```
def first[T](l: Sequence[T]) -> T:  # Function is generic over the TypeVar "T"
    return l[0]
```

Or by using the *TypeVar* factory directly:

```
from collections.abc import Sequence
from typing import TypeVar

U = TypeVar('U')                  # Declare type variable "U"

def second(l: Sequence[U]) -> U:  # Function is generic over the TypeVar "U"
    return l[1]
```

Changed in version 3.12: Syntactic support for generics is new in Python 3.12.

26.1.6 Annotating tuples

For most containers in Python, the typing system assumes that all elements in the container will be of the same type. For example:

```
from collections.abc import Mapping

# Type checker will infer that all elements in ``x`` are meant to be ints
x: list[int] = []

# Type checker error: ``list`` only accepts a single type argument:
y: list[int, str] = [1, 'foo']

# Type checker will infer that all keys in ``z`` are meant to be strings,
# and that all values in ``z`` are meant to be either strings or ints
z: Mapping[str, str | int] = {}
```

list only accepts one type argument, so a type checker would emit an error on the `y` assignment above. Similarly, *Mapping* only accepts two type arguments: the first indicates the type of the keys, and the second indicates the type of the values.

Unlike most other Python containers, however, it is common in idiomatic Python code for tuples to have elements which are not all of the same type. For this reason, tuples are special-cased in Python's typing system. *tuple* accepts *any number* of type arguments:

```
# OK: ``x`` is assigned to a tuple of length 1 where the sole element is an int
x: tuple[int] = (5,)

# OK: ``y`` is assigned to a tuple of length 2;
# element 1 is an int, element 2 is a str
y: tuple[int, str] = (5, "foo")

# Error: the type annotation indicates a tuple of length 1,
# but ``z`` has been assigned to a tuple of length 3
z: tuple[int] = (1, 2, 3)
```

To denote a tuple which could be of *any* length, and in which all elements are of the same type `T`, use `tuple[T, ...]`. To denote an empty tuple, use `tuple[()]`. Using plain `tuple` as an annotation is equivalent to using `tuple[Any, ...]`:

```
x: tuple[int, ...] = (1, 2)
# These reassignments are OK: ``tuple[int, ...]`` indicates x can be of any length
x = (1, 2, 3)
x = ()
# This reassignment is an error: all elements in ``x`` must be ints
x = ("foo", "bar")

# ``y`` can only ever be assigned to an empty tuple
y: tuple[()] = ()

z: tuple = ("foo", "bar")
# These reassignments are OK: plain ``tuple`` is equivalent to ``tuple[Any, ...]``
z = (1, 2, 3)
z = ()
```

26.1.7 The type of class objects

A variable annotated with `C` may accept a value of type `C`. In contrast, a variable annotated with `type[C]` (or *`typing.Type[C]`*) may accept values that are classes themselves – specifically, it will accept the *class object* of `C`. For example:

```
a = 3         # Has type ``int``
b = int       # Has type ``type[int]``
c = type(a)   # Also has type ``type[int]``
```

Note that `type[C]` is covariant:

```
class User: ...
class ProUser(User): ...
class TeamUser(User): ...

def make_new_user(user_class: type[User]) -> User:
    # ...
    return user_class()

make_new_user(User)      # OK
make_new_user(ProUser)   # Also OK: ``type[ProUser]`` is a subtype of ``type[User]``
make_new_user(TeamUser)  # Still fine
make_new_user(User())    # Error: expected ``type[User]`` but got ``User``
make_new_user(int)       # Error: ``type[int]`` is not a subtype of ``type[User]``
```

The only legal parameters for *`type`* are classes, *`Any`*, *type variables*, and unions of any of these types. For example:

```
def new_non_team_user(user_class: type[BasicUser | ProUser]): ...

new_non_team_user(BasicUser)  # OK
new_non_team_user(ProUser)    # OK
new_non_team_user(TeamUser)   # Error: ``type[TeamUser]`` is not a subtype
                              # of ``type[BasicUser | ProUser]``
new_non_team_user(User)       # Also an error
```

`type[Any]` is equivalent to *`type`*, which is the root of Python's metaclass hierarchy.

26.1.8 User-defined generic types

A user-defined class can be defined as a generic class.

```
from logging import Logger

class LoggedVar[T]:
    def __init__(self, value: T, name: str, logger: Logger) -> None:
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('%s: %s', self.name, message)
```

This syntax indicates that the class `LoggedVar` is parameterised around a single *type variable* `T` . This also makes `T` valid as a type within the class body.

Generic classes implicitly inherit from *Generic*. For compatibility with Python 3.11 and lower, it is also possible to inherit explicitly from *Generic* to indicate a generic class:

```
from typing import TypeVar, Generic

T = TypeVar('T')

class LoggedVar(Generic[T]):
    ...
```

Generic classes have `__class_getitem__()` methods, meaning they can be parameterised at runtime (e.g. `LoggedVar[int]` below):

```
from collections.abc import Iterable

def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

A generic type can have any number of type variables. All varieties of *TypeVar* are permissible as parameters for a generic type:

```
from typing import TypeVar, Generic, Sequence

class WeirdTrio[T, B: Sequence[bytes], S: (int, str)]:
    ...

OldT = TypeVar('OldT', contravariant=True)
OldB = TypeVar('OldB', bound=Sequence[bytes], covariant=True)
OldS = TypeVar('OldS', int, str)
```

(continues on next page)

(continued from previous page)

```
class OldWeirdTrio(Generic[OldT, OldB, OldS]):
    ...
```

Each type variable argument to *Generic* must be distinct. This is thus invalid:

```
from typing import TypeVar, Generic
...

class Pair[M, M]:  # SyntaxError
    ...

T = TypeVar('T')

class Pair(Generic[T, T]):   # INVALID
    ...
```

Generic classes can also inherit from other classes:

```
from collections.abc import Sized

class LinkedList[T](Sized):
    ...
```

When inheriting from generic classes, some type parameters could be fixed:

```
from collections.abc import Mapping

class MyDict[T](Mapping[str, T]):
    ...
```

In this case `MyDict` has a single parameter, `T`.

Using a generic class without specifying type parameters assumes *Any* for each position. In the following example, `MyIterable` is not generic but implicitly inherits from `Iterable[Any]`:

```
from collections.abc import Iterable

class MyIterable(Iterable): # Same as Iterable[Any]
    ...
```

User-defined generic type aliases are also supported. Examples:

```
from collections.abc import Iterable

type Response[S] = Iterable[S] | int

# Return type here is same as Iterable[str] | int
def response(query: str) -> Response[str]:
    ...

type Vec[T] = Iterable[tuple[T, T]]

def inproduct[T: (int, float, complex)](v: Vec[T]) -> T: # Same as Iterable[tuple[T,␣
↪T]]
    return sum(x*y for x, y in v)
```

For backward compatibility, generic type aliases can also be created through a simple assignment:

```
from collections.abc import Iterable
from typing import TypeVar

S = TypeVar("S")
Response = Iterable[S] | int
```

Changed in version 3.7: *Generic* no longer has a custom metaclass.

Changed in version 3.12: Syntactic support for generics and type aliases is new in version 3.12. Previously, generic classes had to explicitly inherit from *Generic* or contain a type variable in one of their bases.

User-defined generics for parameter expressions are also supported via parameter specification variables in the form `[**P]`. The behavior is consistent with type variables' described above as parameter specification variables are treated by the typing module as a specialized type variable. The one exception to this is that a list of types can be used to substitute a *ParamSpec*:

```
>>> class Z[T, **P]: ...  # T is a TypeVar; P is a ParamSpec
...
>>> Z[int, [dict, float]]
__main__.Z[int, [dict, float]]
```

Classes generic over a *ParamSpec* can also be created using explicit inheritance from *Generic*. In this case, `**` is not used:

```
from typing import ParamSpec, Generic

P = ParamSpec('P')

class Z(Generic[P]):
    ...
```

Another difference between *TypeVar* and *ParamSpec* is that a generic with only one parameter specification variable will accept parameter lists in the forms `X[[Type1, Type2, ...]]` and also `X[Type1, Type2, ...]` for aesthetic reasons. Internally, the latter is converted to the former, so the following are equivalent:

```
>>> class X[**P]: ...
...
>>> X[int, str]
__main__.X[[int, str]]
>>> X[[int, str]]
__main__.X[[int, str]]
```

Note that generics with *ParamSpec* may not have correct `__parameters__` after substitution in some cases because they are intended primarily for static type checking.

Changed in version 3.10: *Generic* can now be parameterized over parameter expressions. See *ParamSpec* and **PEP 612** for more details.

A user-defined generic class can have ABCs as base classes without a metaclass conflict. Generic metaclasses are not supported. The outcome of parameterizing generics is cached, and most types in the typing module are *hashable* and comparable for equality.

26.1.9 The `Any` type

A special kind of type is *Any*. A static type checker will treat every type as being compatible with *Any* and *Any* as being compatible with every type.

This means that it is possible to perform any operation or method call on a value of type *Any* and assign it to any variable:

```
from typing import Any

a: Any = None
a = []          # OK
a = 2           # OK

s: str = ''
s = a           # OK

def foo(item: Any) -> int:
    # Passes type checking; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

Notice that no type checking is performed when assigning a value of type *Any* to a more precise type. For example, the static type checker did not report an error when assigning `a` to `s` even though `s` was declared to be of type *str* and receives an *int* value at runtime!

Furthermore, all functions without a return type or parameter types will implicitly default to using *Any*:

```
def legacy_parser(text):
    ...
    return data

# A static type checker will treat the above
# as having the same signature as:
def legacy_parser(text: Any) -> Any:
    ...
    return data
```

This behavior allows *Any* to be used as an *escape hatch* when you need to mix dynamically and statically typed code.

Contrast the behavior of *Any* with the behavior of *object*. Similar to *Any*, every type is a subtype of *object*. However, unlike *Any*, the reverse is not true: *object* is *not* a subtype of every other type.

That means when the type of a value is *object*, a type checker will reject almost all operations on it, and assigning it to a variable (or using it as a return value) of a more specialized type is a type error. For example:

```
def hash_a(item: object) -> int:
    # Fails type checking; an object does not have a 'magic' method.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Passes type checking
    item.magic()
    ...

# Passes type checking, since ints and strs are subclasses of object
hash_a(42)
hash_a("foo")
```

(continues on next page)

(continued from previous page)

```
# Passes type checking, since Any is compatible with all types
hash_b(42)
hash_b("foo")
```

Use *object* to indicate that a value could be any type in a typesafe manner. Use *Any* to indicate that a value is dynamically typed.

26.1.10 Nominal vs structural subtyping

Initially **PEP 484** defined the Python static type system as using *nominal subtyping*. This means that a class `A` is allowed where a class `B` is expected if and only if `A` is a subclass of `B`.

This requirement previously also applied to abstract base classes, such as *Iterable*. The problem with this approach is that a class had to be explicitly marked to support them, which is unpythonic and unlike what one would normally do in idiomatic dynamically typed Python code. For example, this conforms to **PEP 484**:

```
from collections.abc import Sized, Iterable, Iterator

class Bucket(Sized, Iterable[int]):
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...
```

PEP 544 allows to solve this problem by allowing users to write the above code without explicit base classes in the class definition, allowing `Bucket` to be implicitly considered a subtype of both `Sized` and `Iterable[int]` by static type checkers. This is known as *structural subtyping* (or static duck-typing):

```
from collections.abc import Iterator, Iterable

class Bucket:  # Note: no base classes
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...

def collect(items: Iterable[int]) -> int: ...
result = collect(Bucket())  # Passes type check
```

Moreover, by subclassing a special class *Protocol*, a user can define new custom protocols to fully enjoy structural subtyping (see examples below).

26.1.11 Module contents

The `typing` module defines the following classes, functions and decorators.

Special typing primitives

Special types

These can be used as types in annotations. They do not support subscription using `[]`.

`typing.`**`Any`**

Special type indicating an unconstrained type.

- Every type is compatible with *Any*.
- *Any* is compatible with every type.

Changed in version 3.11: *Any* can now be used as a base class. This can be useful for avoiding type checker errors with classes that can duck type anywhere or are highly dynamic.

`typing.`**`AnyStr`**

A *constrained type variable*.

Definition:

```
AnyStr = TypeVar('AnyStr', str, bytes)
```

`AnyStr` is meant to be used for functions that may accept *str* or *bytes* arguments but cannot allow the two to mix.

For example:

```
def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat("foo", "bar")    # OK, output has type 'str'
concat(b"foo", b"bar")  # OK, output has type 'bytes'
concat("foo", b"bar")   # Error, cannot mix str and bytes
```

Note that, despite its name, `AnyStr` has nothing to do with the *Any* type, nor does it mean "any string". In particular, `AnyStr` and `str | bytes` are different from each other and have different use cases:

```
# Invalid use of AnyStr:
# The type variable is used only once in the function signature,
# so cannot be "solved" by the type checker
def greet_bad(cond: bool) -> AnyStr:
    return "hi there!" if cond else b"greetings!"

# The better way of annotating this function:
def greet_proper(cond: bool) -> str | bytes:
    return "hi there!" if cond else b"greetings!"
```

Deprecated since version 3.13, will be removed in version 3.18: Deprecated in favor of the new type parameter syntax. Use `class A[T: (str, bytes)]: ...` instead of importing `AnyStr`. See **PEP 695** for more details.

In Python 3.16, `AnyStr` will be removed from `typing.__all__`, and deprecation warnings will be emitted at runtime when it is accessed or imported from `typing`. `AnyStr` will be removed from `typing` in Python 3.18.

`typing.`**`LiteralString`**

Special type that includes only literal strings.

Any string literal is compatible with `LiteralString`, as is another `LiteralString`. However, an object typed as just `str` is not. A string created by composing `LiteralString`-typed objects is also acceptable as a `LiteralString`.

Example:

```
def run_query(sql: LiteralString) -> None:
    ...

def caller(arbitrary_string: str, literal_string: LiteralString) -> None:
    run_query("SELECT * FROM students")  # OK
    run_query(literal_string)  # OK
    run_query("SELECT * FROM " + literal_string)  # OK
    run_query(arbitrary_string)  # type checker error
    run_query(  # type checker error
        f"SELECT * FROM students WHERE name = {arbitrary_string}"
    )
```

`LiteralString` is useful for sensitive APIs where arbitrary user-generated strings could generate problems. For example, the two cases above that generate type checker errors could be vulnerable to an SQL injection attack.

See **PEP 675** for more details.

Added in version 3.11.

`typing.`**`Never`**

`typing.`**`NoReturn`**

`Never` and `NoReturn` represent the bottom type, a type that has no members.

They can be used to indicate that a function never returns, such as *`sys.exit()`*:

```
from typing import Never  # or NoReturn

def stop() -> Never:
    raise RuntimeError('no way')
```

Or to define a function that should never be called, as there are no valid arguments, such as *`assert_never()`*:

```
from typing import Never  # or NoReturn

def never_call_me(arg: Never) -> None:
    pass

def int_or_str(arg: int | str) -> None:
    never_call_me(arg)  # type checker error
    match arg:
        case int():
            print("It's an int")
        case str():
            print("It's a str")
        case _:
            never_call_me(arg)  # OK, arg is of type Never (or NoReturn)
```

`Never` and `NoReturn` have the same meaning in the type system and static type checkers treat both equivalently.

Added in version 3.6.2: Added *`NoReturn`*.

Added in version 3.11: Added *`Never`*.

typing.**Self**

Special type to represent the current enclosed class.

For example:

```
from typing import Self, reveal_type

class Foo:
    def return_self(self) -> Self:
        ...
        return self

class SubclassOfFoo(Foo): pass

reveal_type(Foo().return_self())  # Revealed type is "Foo"
reveal_type(SubclassOfFoo().return_self())  # Revealed type is "SubclassOfFoo"
```

This annotation is semantically equivalent to the following, albeit in a more succinct fashion:

```
from typing import TypeVar

Self = TypeVar("Self", bound="Foo")

class Foo:
    def return_self(self: Self) -> Self:
        ...
        return self
```

In general, if something returns `self`, as in the above examples, you should use `Self` as the return annotation. If `Foo.return_self` was annotated as returning `"Foo"`, then the type checker would infer the object returned from `SubclassOfFoo.return_self` as being of type `Foo` rather than `SubclassOfFoo`.

Other common use cases include:

- *`classmethod`*s that are used as alternative constructors and return instances of the `cls` parameter.
- Annotating an `__enter__()` method which returns self.

You should not use `Self` as the return annotation if the method is not guaranteed to return an instance of a subclass when the class is subclassed:

```
class Eggs:
    # Self would be an incorrect return annotation here,
    # as the object returned is always an instance of Eggs,
    # even in subclasses
    def returns_eggs(self) -> "Eggs":
        return Eggs()
```

See **PEP 673** for more details.

Added in version 3.11.

typing.**TypeAlias**

Special annotation for explicitly declaring a *type alias*.

For example:

```
from typing import TypeAlias

Factors: TypeAlias = list[int]
```

`TypeAlias` is particularly useful on older Python versions for annotating aliases that make use of forward references, as it can be hard for type checkers to distinguish these from normal variable assignments:

```
from typing import Generic, TypeAlias, TypeVar

T = TypeVar("T")

# "Box" does not exist yet,
# so we have to use quotes for the forward reference on Python <3.12.
# Using ``TypeAlias`` tells the type checker that this is a type alias␣
↪declaration,
# not a variable assignment to a string.
BoxOfStrings: TypeAlias = "Box[str]"

class Box(Generic[T]):
    @classmethod
    def make_box_of_strings(cls) -> BoxOfStrings: ...
```

See **PEP 613** for more details.

Added in version 3.10.

Deprecated since version 3.12: *TypeAlias* is deprecated in favor of the `type` statement, which creates instances of *TypeAliasType* and which natively supports forward references. Note that while *TypeAlias* and *TypeAliasType* serve similar purposes and have similar names, they are distinct and the latter is not the type of the former. Removal of *TypeAlias* is not currently planned, but users are encouraged to migrate to `type` statements.

Special forms

These can be used as types in annotations. They all support subscription using `[]`, but each has a unique syntax.

`typing.`**`Union`**

Union type; `Union[X, Y]` is equivalent to `X | Y` and means either X or Y.

To define a union, use e.g. `Union[int, str]` or the shorthand `int | str`. Using that shorthand is recommended. Details:

- The arguments must be types and there must be at least one.
- Unions of unions are flattened, e.g.:

```
Union[Union[int, str], float] == Union[int, str, float]
```

- Unions of a single argument vanish, e.g.:

```
Union[int] == int  # The constructor actually returns int
```

- Redundant arguments are skipped, e.g.:

```
Union[int, str, int] == Union[int, str] == int | str
```

- When comparing unions, the argument order is ignored, e.g.:

```
Union[int, str] == Union[str, int]
```

- You cannot subclass or instantiate a `Union`.
- You cannot write `Union[X][Y]`.

Changed in version 3.7: Don't remove explicit subclasses from unions at runtime.

Changed in version 3.10: Unions can now be written as `X | Y`. See *union type expressions*.

typing.**Optional**

`Optional[X]` is equivalent to `X | None` (or `Union[X, None]`).

Note that this is not the same concept as an optional argument, which is one that has a default. An optional argument with a default does not require the `Optional` qualifier on its type annotation just because it is optional. For example:

```
def foo(arg: int = 0) -> None:
    ...
```

On the other hand, if an explicit value of `None` is allowed, the use of `Optional` is appropriate, whether the argument is optional or not. For example:

```
def foo(arg: Optional[int] = None) -> None:
    ...
```

Changed in version 3.10: Optional can now be written as `X | None`. See *union type expressions*.

typing.**Concatenate**

Special form for annotating higher-order functions.

`Concatenate` can be used in conjunction with *Callable* and *ParamSpec* to annotate a higher-order callable which adds, removes, or transforms parameters of another callable. Usage is in the form `Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable]`. `Concatenate` is currently only valid when used as the first argument to a *Callable*. The last parameter to `Concatenate` must be a *ParamSpec* or ellipsis (`...`).

For example, to annotate a decorator `with_lock` which provides a *threading.Lock* to the decorated function, `Concatenate` can be used to indicate that `with_lock` expects a callable which takes in a `Lock` as the first argument, and returns a callable with a different type signature. In this case, the *ParamSpec* indicates that the returned callable's parameter types are dependent on the parameter types of the callable being passed in:

```
from collections.abc import Callable
from threading import Lock
from typing import Concatenate

# Use this lock to ensure that only one thread is executing a function
# at any time.
my_lock = Lock()

def with_lock[**P, R](f: Callable[Concatenate[Lock, P], R]) -> Callable[P, R]:
    '''A type-safe decorator which provides a lock.'''
    def inner(*args: P.args, **kwargs: P.kwargs) -> R:
        # Provide the lock as the first argument.
        return f(my_lock, *args, **kwargs)
    return inner

@with_lock
def sum_threadsafe(lock: Lock, numbers: list[float]) -> float:
    '''Add a list of numbers together in a thread-safe manner.'''
    with lock:
        return sum(numbers)

# We don't need to pass in the lock ourselves thanks to the decorator.
sum_threadsafe([1.1, 2.2, 3.3])
```

Added in version 3.10.

See also

- **PEP 612** – Parameter Specification Variables (the PEP which introduced `ParamSpec` and `Concatenate`)
- *ParamSpec*
- *Annotating callable objects*

`typing.`**`Literal`**

Special typing form to define "literal types".

`Literal` can be used to indicate to type checkers that the annotated object has a value equivalent to one of the provided literals.

For example:

```
def validate_simple(data: Any) -> Literal[True]:  # always returns True
    ...

type Mode = Literal['r', 'rb', 'w', 'wb']
def open_helper(file: str, mode: Mode) -> str:
    ...

open_helper('/some/path', 'r')      # Passes type check
open_helper('/other/path', 'typo')  # Error in type checker
```

`Literal[...]` cannot be subclassed. At runtime, an arbitrary value is allowed as type argument to `Literal[...]`, but type checkers may impose restrictions. See **PEP 586** for more details about literal types.

Added in version 3.8.

Changed in version 3.9.1: `Literal` now de-duplicates parameters. Equality comparisons of `Literal` objects are no longer order dependent. `Literal` objects will now raise a *TypeError* exception during equality comparisons if one of their parameters are not *hashable*.

`typing.`**`ClassVar`**

Special type construct to mark class variables.

As introduced in **PEP 526**, a variable annotation wrapped in ClassVar indicates that a given attribute is intended to be used as a class variable and should not be set on instances of that class. Usage:

```
class Starship:
    stats: ClassVar[dict[str, int]] = {} # class variable
    damage: int = 10                     # instance variable
```

ClassVar accepts only types and cannot be further subscribed.

ClassVar is not a class itself, and should not be used with *isinstance()* or *issubclass()*. *ClassVar* does not change Python runtime behavior, but it can be used by third-party type checkers. For example, a type checker might flag the following code as an error:

```
enterprise_d = Starship(3000)
enterprise_d.stats = {} # Error, setting class variable on instance
Starship.stats = {}     # This is OK
```

Added in version 3.5.3.

Changed in version 3.13: *ClassVar* can now be nested in *Final* and vice versa.

typing.**Final**

Special typing construct to indicate final names to type checkers.

Final names cannot be reassigned in any scope. Final names declared in class scopes cannot be overridden in subclasses.

For example:

```
MAX_SIZE: Final = 9000
MAX_SIZE += 1  # Error reported by type checker

class Connection:
    TIMEOUT: Final[int] = 10

class FastConnector(Connection):
    TIMEOUT = 1  # Error reported by type checker
```

There is no runtime checking of these properties. See **PEP 591** for more details.

Added in version 3.8.

Changed in version 3.13: *Final* can now be nested in *ClassVar* and vice versa.

typing.**Required**

Special typing construct to mark a *TypedDict* key as required.

This is mainly useful for `total=False` TypedDicts. See *TypedDict* and **PEP 655** for more details.

Added in version 3.11.

typing.**NotRequired**

Special typing construct to mark a *TypedDict* key as potentially missing.

See *TypedDict* and **PEP 655** for more details.

Added in version 3.11.

typing.**ReadOnly**

A special typing construct to mark an item of a *TypedDict* as read-only.

For example:

```
class Movie(TypedDict):
   title: ReadOnly[str]
   year: int

def mutate_movie(m: Movie) -> None:
   m["year"] = 1992  # allowed
   m["title"] = "The Matrix"  # typechecker error
```

There is no runtime checking for this property.

See *TypedDict* and **PEP 705** for more details.

Added in version 3.13.

typing.**Annotated**

Special typing form to add context-specific metadata to an annotation.

Add metadata `x` to a given type `T` by using the annotation `Annotated[T, x]`. Metadata added using `Annotated` can be used by static analysis tools or at runtime. At runtime, the metadata is stored in a `__metadata__` attribute.

If a library or tool encounters an annotation `Annotated[T, x]` and has no special logic for the metadata, it should ignore the metadata and simply treat the annotation as `T`. As such, `Annotated` can be useful for code that wants to use annotations for purposes outside Python's static typing system.

Using `Annotated[T, x]` as an annotation still allows for static typechecking of `T`, as type checkers will simply ignore the metadata `x`. In this way, `Annotated` differs from the *`@no_type_check`* decorator, which can also be used for adding annotations outside the scope of the typing system, but completely disables typechecking for a function or class.

The responsibility of how to interpret the metadata lies with the tool or library encountering an `Annotated` annotation. A tool or library encountering an `Annotated` type can scan through the metadata elements to determine if they are of interest (e.g., using *`isinstance()`*).

`Annotated[<type>, <metadata>]`

Here is an example of how you might use `Annotated` to add metadata to type annotations if you were doing range analysis:

```
@dataclass
class ValueRange:
    lo: int
    hi: int

T1 = Annotated[int, ValueRange(-10, 5)]
T2 = Annotated[T1, ValueRange(-20, 3)]
```

Details of the syntax:

- The first argument to `Annotated` must be a valid type
- Multiple metadata elements can be supplied (`Annotated` supports variadic arguments):

  ```
  @dataclass
  class ctype:
      kind: str

  Annotated[int, ValueRange(3, 10), ctype("char")]
  ```

  It is up to the tool consuming the annotations to decide whether the client is allowed to add multiple metadata elements to one annotation and how to merge those annotations.

- `Annotated` must be subscripted with at least two arguments (`Annotated[int]` is not valid)
- The order of the metadata elements is preserved and matters for equality checks:

  ```
  assert Annotated[int, ValueRange(3, 10), ctype("char")] != Annotated[
      int, ctype("char"), ValueRange(3, 10)
  ]
  ```

- Nested `Annotated` types are flattened. The order of the metadata elements starts with the innermost annotation:

```
assert Annotated[Annotated[int, ValueRange(3, 10)], ctype("char")] ==␣
↪Annotated[
    int, ValueRange(3, 10), ctype("char")
]
```

- Duplicated metadata elements are not removed:

```
assert Annotated[int, ValueRange(3, 10)] != Annotated[
    int, ValueRange(3, 10), ValueRange(3, 10)
]
```

- `Annotated` can be used with nested and generic aliases:

```
@dataclass
class MaxLen:
    value: int

type Vec[T] = Annotated[list[tuple[T, T]], MaxLen(10)]

# When used in a type annotation, a type checker will treat "V" the same as
# ``Annotated[list[tuple[int, int]], MaxLen(10)]``:
type V = Vec[int]
```

- `Annotated` cannot be used with an unpacked *TypeVarTuple*:

```
type Variadic[*Ts] = Annotated[*Ts, Ann1]  # NOT valid
```

  This would be equivalent to:

```
Annotated[T1, T2, T3, ..., Ann1]
```

  where `T1`, `T2`, etc. are *TypeVars*. This would be invalid: only one type should be passed to Annotated.

- By default, *get_type_hints()* strips the metadata from annotations. Pass `include_extras=True` to have the metadata preserved:

```
>>> from typing import Annotated, get_type_hints
>>> def func(x: Annotated[int, "metadata"]) -> None: pass
...
>>> get_type_hints(func)
{'x': <class 'int'>, 'return': <class 'NoneType'>}
>>> get_type_hints(func, include_extras=True)
{'x': typing.Annotated[int, 'metadata'], 'return': <class 'NoneType'>}
```

- At runtime, the metadata associated with an `Annotated` type can be retrieved via the `__metadata__` attribute:

```
>>> from typing import Annotated
>>> X = Annotated[int, "very", "important", "metadata"]
>>> X
typing.Annotated[int, 'very', 'important', 'metadata']
>>> X.__metadata__
('very', 'important', 'metadata')
```

See also

> **PEP 593 - Flexible function and variable annotations**
> The PEP introducing `Annotated` to the standard library.

Added in version 3.9.

`typing.`**`TypeIs`**

Special typing construct for marking user-defined type predicate functions.

`TypeIs` can be used to annotate the return type of a user-defined type predicate function. `TypeIs` only accepts a single type argument. At runtime, functions marked this way should return a boolean and take at least one positional argument.

`TypeIs` aims to benefit *type narrowing* – a technique used by static type checkers to determine a more precise type of an expression within a program's code flow. Usually type narrowing is done by analyzing conditional code flow and applying the narrowing to a block of code. The conditional expression here is sometimes referred to as a "type predicate":

```
def is_str(val: str | float):
    # "isinstance" type predicate
    if isinstance(val, str):
        # Type of ``val`` is narrowed to ``str``
        ...
    else:
        # Else, type of ``val`` is narrowed to ``float``.
        ...
```

Sometimes it would be convenient to use a user-defined boolean function as a type predicate. Such a function should use `TypeIs[...]` or *TypeGuard* as its return type to alert static type checkers to this intention. `TypeIs` usually has more intuitive behavior than `TypeGuard`, but it cannot be used when the input and output types are incompatible (e.g., `list[object]` to `list[int]`) or when the function does not return `True` for all instances of the narrowed type.

Using `-> TypeIs[NarrowedType]` tells the static type checker that for a given function:

1. The return value is a boolean.
2. If the return value is `True`, the type of its argument is the intersection of the argument's original type and `NarrowedType`.
3. If the return value is `False`, the type of its argument is narrowed to exclude `NarrowedType`.

For example:

```
from typing import assert_type, final, TypeIs

class Parent: pass
class Child(Parent): pass
@final
class Unrelated: pass

def is_parent(val: object) -> TypeIs[Parent]:
    return isinstance(val, Parent)

def run(arg: Child | Unrelated):
    if is_parent(arg):
        # Type of ``arg`` is narrowed to the intersection
        # of ``Parent`` and ``Child``, which is equivalent to
        # ``Child``.
```

(continues on next page)

(continued from previous page)

```
        assert_type(arg, Child)
    else:
        # Type of ``arg`` is narrowed to exclude ``Parent``,
        # so only ``Unrelated`` is left.
        assert_type(arg, Unrelated)
```

The type inside `TypeIs` must be consistent with the type of the function's argument; if it is not, static type checkers will raise an error. An incorrectly written `TypeIs` function can lead to unsound behavior in the type system; it is the user's responsibility to write such functions in a type-safe manner.

If a `TypeIs` function is a class or instance method, then the type in `TypeIs` maps to the type of the second parameter (after `cls` or `self`).

In short, the form `def foo(arg: TypeA) -> TypeIs[TypeB]: ...`, means that if `foo(arg)` returns `True`, then `arg` is an instance of `TypeB`, and if it returns `False`, it is not an instance of `TypeB`.

`TypeIs` also works with type variables. For more information, see **PEP 742** (Narrowing types with `TypeIs`).

Added in version 3.13.

typing.**TypeGuard**

Special typing construct for marking user-defined type predicate functions.

Type predicate functions are user-defined functions that return whether their argument is an instance of a particular type. `TypeGuard` works similarly to *`TypeIs`*, but has subtly different effects on type checking behavior (see below).

Using `-> TypeGuard` tells the static type checker that for a given function:

1. The return value is a boolean.
2. If the return value is `True`, the type of its argument is the type inside `TypeGuard`.

`TypeGuard` also works with type variables. See **PEP 647** for more details.

For example:

```
def is_str_list(val: list[object]) -> TypeGuard[list[str]]:
    '''Determines whether all objects in the list are strings'''
    return all(isinstance(x, str) for x in val)

def func1(val: list[object]):
    if is_str_list(val):
        # Type of ``val`` is narrowed to ``list[str]``.
        print(" ".join(val))
    else:
        # Type of ``val`` remains as ``list[object]``.
        print("Not a list of strings!")
```

`TypeIs` and `TypeGuard` differ in the following ways:

- `TypeIs` requires the narrowed type to be a subtype of the input type, while `TypeGuard` does not. The main reason is to allow for things like narrowing `list[object]` to `list[str]` even though the latter is not a subtype of the former, since `list` is invariant.
- When a `TypeGuard` function returns `True`, type checkers narrow the type of the variable to exactly the `TypeGuard` type. When a `TypeIs` function returns `True`, type checkers can infer a more precise type combining the previously known type of the variable with the `TypeIs` type. (Technically, this is known as an intersection type.)

- When a `TypeGuard` function returns `False`, type checkers cannot narrow the type of the variable at all. When a `TypeIs` function returns `False`, type checkers can narrow the type of the variable to exclude the `TypeIs` type.

Added in version 3.10.

`typing.`**`Unpack`**

Typing operator to conceptually mark an object as having been unpacked.

For example, using the unpack operator `*` on a *type variable tuple* is equivalent to using `Unpack` to mark the type variable tuple as having been unpacked:

```
Ts = TypeVarTuple('Ts')
tup: tuple[*Ts]
# Effectively does:
tup: tuple[Unpack[Ts]]
```

In fact, `Unpack` can be used interchangeably with `*` in the context of *typing.TypeVarTuple* and *builtins.tuple* types. You might see `Unpack` being used explicitly in older versions of Python, where `*` couldn't be used in certain places:

```
# In older versions of Python, TypeVarTuple and Unpack
# are located in the `typing_extensions` backports package.
from typing_extensions import TypeVarTuple, Unpack

Ts = TypeVarTuple('Ts')
tup: tuple[*Ts]         # Syntax error on Python <= 3.10!
tup: tuple[Unpack[Ts]]  # Semantically equivalent, and backwards-compatible
```

`Unpack` can also be used along with *typing.TypedDict* for typing `**kwargs` in a function signature:

```
from typing import TypedDict, Unpack

class Movie(TypedDict):
    name: str
    year: int

# This function expects two keyword arguments - `name` of type `str`
# and `year` of type `int`.
def foo(**kwargs: Unpack[Movie]): ...
```

See **PEP 692** for more details on using `Unpack` for `**kwargs` typing.

Added in version 3.11.

Building generic types and type aliases

The following classes should not be used directly as annotations. Their intended purpose is to be building blocks for creating generic types and type aliases.

These objects can be created through special syntax (type parameter lists and the `type` statement). For compatibility with Python 3.11 and earlier, they can also be created without the dedicated syntax, as documented below.

`class` `typing.`**`Generic`**

Abstract base class for generic types.

A generic type is typically declared by adding a list of type parameters after the class name:

```
class Mapping[KT, VT]:
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

Such a class implicitly inherits from `Generic`. The runtime semantics of this syntax are discussed in the Language Reference.

This class can then be used as follows:

```
def lookup_name[X, Y](mapping: Mapping[X, Y], key: X, default: Y) -> Y:
    try:
        return mapping[key]
    except KeyError:
        return default
```

Here the brackets after the function name indicate a generic function.

For backwards compatibility, generic classes can also be declared by explicitly inheriting from `Generic`. In this case, the type parameters must be declared separately:

```
KT = TypeVar('KT')
VT = TypeVar('VT')

class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

class typing.**TypeVar**(*name*, **constraints*, *bound=None*, *covariant=False*, *contravariant=False*, *infer_variance=False*, *default=typing.NoDefault*)

Type variable.

The preferred way to construct a type variable is via the dedicated syntax for generic functions, generic classes, and generic type aliases:

```
class Sequence[T]:  # T is a TypeVar
    ...
```

This syntax can also be used to create bound and constrained type variables:

```
class StrSequence[S: str]:  # S is a TypeVar bound to str
    ...


class StrOrBytesSequence[A: (str, bytes)]:  # A is a TypeVar constrained to str␣
↪or bytes
    ...
```

However, if desired, reusable type variables can also be constructed manually, like so:

```
T = TypeVar('T')  # Can be anything
S = TypeVar('S', bound=str)  # Can be any subtype of str
A = TypeVar('A', str, bytes)  # Must be exactly str or bytes
```

Type variables exist primarily for the benefit of static type checkers. They serve as the parameters for generic types as well as for generic function and type alias definitions. See *Generic* for more information on generic types. Generic functions work as follows:

```
def repeat[T](x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x."""
    return [x]*n


def print_capitalized[S: str](x: S) -> S:
    """Print x capitalized, and return x."""
    print(x.capitalize())
    return x


def concatenate[A: (str, bytes)](x: A, y: A) -> A:
    """Add two strings or bytes objects together."""
    return x + y
```

Note that type variables can be *bound*, *constrained*, or neither, but cannot be both bound *and* constrained.

The variance of type variables is inferred by type checkers when they are created through the type parameter syntax or when `infer_variance=True` is passed. Manually created type variables may be explicitly marked covariant or contravariant by passing `covariant=True` or `contravariant=True`. By default, manually created type variables are invariant. See **PEP 484** and **PEP 695** for more details.

Bound type variables and constrained type variables have different semantics in several important ways. Using a *bound* type variable means that the `TypeVar` will be solved using the most specific type possible:

```
x = print_capitalized('a string')
reveal_type(x)  # revealed type is str

class StringSubclass(str):
    pass

y = print_capitalized(StringSubclass('another string'))
reveal_type(y)  # revealed type is StringSubclass

z = print_capitalized(45)  # error: int is not a subtype of str
```

Type variables can be bound to concrete types, abstract types (ABCs or protocols), and even unions of types:

```
# Can be anything with an __abs__ method
def print_abs[T: SupportsAbs](arg: T) -> None:
    print("Absolute value:", abs(arg))

U = TypeVar('U', bound=str|bytes)  # Can be any subtype of the union str|bytes
V = TypeVar('V', bound=SupportsAbs)  # Can be anything with an __abs__ method
```

Using a *constrained* type variable, however, means that the `TypeVar` can only ever be solved as being exactly one of the constraints given:

```
a = concatenate('one', 'two')
reveal_type(a)  # revealed type is str

b = concatenate(StringSubclass('one'), StringSubclass('two'))
reveal_type(b)  # revealed type is str, despite StringSubclass being passed in

c = concatenate('one', b'two')  # error: type variable 'A' can be either str or␣
↪bytes in a function call, but not both
```

At runtime, `isinstance(x, T)` will raise *TypeError*.

__name__

The name of the type variable.

__covariant__

Whether the type var has been explicitly marked as covariant.

__contravariant__

Whether the type var has been explicitly marked as contravariant.

__infer_variance__

Whether the type variable's variance should be inferred by type checkers.

Added in version 3.12.

__bound__

The bound of the type variable, if any.

Changed in version 3.12: For type variables created through type parameter syntax, the bound is evaluated only when the attribute is accessed, not when the type variable is created (see lazy-evaluation).

__constraints__

A tuple containing the constraints of the type variable, if any.

Changed in version 3.12: For type variables created through type parameter syntax, the constraints are evaluated only when the attribute is accessed, not when the type variable is created (see lazy-evaluation).

__default__

The default value of the type variable, or *typing.NoDefault* if it has no default.

Added in version 3.13.

has_default()

Return whether or not the type variable has a default value. This is equivalent to checking whether *__default__* is not the *typing.NoDefault* singleton, except that it does not force evaluation of the lazily evaluated default value.

Added in version 3.13.

Changed in version 3.12: Type variables can now be declared using the type parameter syntax introduced by **PEP 695**. The `infer_variance` parameter was added.

Changed in version 3.13: Support for default values was added.

class typing.**TypeVarTuple**(*name*, *, *default=typing.NoDefault*)

Type variable tuple. A specialized form of *type variable* that enables *variadic* generics.

Type variable tuples can be declared in type parameter lists using a single asterisk (`*`) before the name:

```
def move_first_element_to_last[T, *Ts](tup: tuple[T, *Ts]) -> tuple[*Ts, T]:
    return (*tup[1:], tup[0])
```

Or by explicitly invoking the `TypeVarTuple` constructor:

```
T = TypeVar("T")
Ts = TypeVarTuple("Ts")

def move_first_element_to_last(tup: tuple[T, *Ts]) -> tuple[*Ts, T]:
    return (*tup[1:], tup[0])
```

A normal type variable enables parameterization with a single type. A type variable tuple, in contrast, allows parameterization with an *arbitrary* number of types by acting like an *arbitrary* number of type variables wrapped in a tuple. For example:

```
# T is bound to int, Ts is bound to ()
# Return value is (1,), which has type tuple[int]
move_first_element_to_last(tup=(1,))

# T is bound to int, Ts is bound to (str,)
# Return value is ('spam', 1), which has type tuple[str, int]
move_first_element_to_last(tup=(1, 'spam'))

# T is bound to int, Ts is bound to (str, float)
# Return value is ('spam', 3.0, 1), which has type tuple[str, float, int]
move_first_element_to_last(tup=(1, 'spam', 3.0))

# This fails to type check (and fails at runtime)
# because tuple[()] is not compatible with tuple[T, *Ts]
# (at least one element is required)
move_first_element_to_last(tup=())
```

Note the use of the unpacking operator `*` in `tuple[T, *Ts]`. Conceptually, you can think of `Ts` as a tuple of type variables `(T1, T2, ...)`. `tuple[T, *Ts]` would then become `tuple[T, *(T1, T2, ...)]`, which is equivalent to `tuple[T, T1, T2, ...]`. (Note that in older versions of Python, you might see this written using *`Unpack`* instead, as `Unpack[Ts]`.)

Type variable tuples must *always* be unpacked. This helps distinguish type variable tuples from normal type variables:

```
x: Ts          # Not valid
x: tuple[Ts]   # Not valid
x: tuple[*Ts]  # The correct way to do it
```

Type variable tuples can be used in the same contexts as normal type variables. For example, in class definitions, arguments, and return types:

```
class Array[*Shape]:
    def __getitem__(self, key: tuple[*Shape]) -> float: ...
    def __abs__(self) -> "Array[*Shape]": ...
    def get_shape(self) -> tuple[*Shape]: ...
```

Type variable tuples can be happily combined with normal type variables:

```
class Array[DType, *Shape]:  # This is fine
    pass

class Array2[*Shape, DType]:  # This would also be fine
    pass

class Height: ...
class Width: ...

float_array_1d: Array[float, Height] = Array()      # Totally fine
int_array_2d: Array[int, Height, Width] = Array()  # Yup, fine too
```

However, note that at most one type variable tuple may appear in a single list of type arguments or type parameters:

```
x: tuple[*Ts, *Ts]            # Not valid
class Array[*Shape, *Shape]:  # Not valid
    pass
```

Finally, an unpacked type variable tuple can be used as the type annotation of `*args`:

```
def call_soon[*Ts](
    callback: Callable[[*Ts], None],
    *args: *Ts
) -> None:
    ...
    callback(*args)
```

In contrast to non-unpacked annotations of `*args` - e.g. `*args: int`, which would specify that *all* arguments are `int` - `*args: *Ts` enables reference to the types of the *individual* arguments in `*args`. Here, this allows us to ensure the types of the `*args` passed to `call_soon` match the types of the (positional) arguments of `callback`.

See **PEP 646** for more details on type variable tuples.

__name__

The name of the type variable tuple.

__default__

The default value of the type variable tuple, or *typing.NoDefault* if it has no default.

Added in version 3.13.

has_default()

Return whether or not the type variable tuple has a default value. This is equivalent to checking whether *__default__* is not the *typing.NoDefault* singleton, except that it does not force evaluation of the lazily evaluated default value.

Added in version 3.13.

Added in version 3.11.

Changed in version 3.12: Type variable tuples can now be declared using the type parameter syntax introduced by **PEP 695**.

Changed in version 3.13: Support for default values was added.

class typing.**ParamSpec**(*name*, *, *bound=None*, *covariant=False*, *contravariant=False*, *default=typing.NoDefault*)

Parameter specification variable. A specialized version of *type variables*.

In type parameter lists, parameter specifications can be declared with two asterisks (`**`):

```
type IntFunc[**P] = Callable[P, int]
```

For compatibility with Python 3.11 and earlier, `ParamSpec` objects can also be created as follows:

```
P = ParamSpec('P')
```

Parameter specification variables exist primarily for the benefit of static type checkers. They are used to forward the parameter types of one callable to another callable – a pattern commonly found in higher order functions and decorators. They are only valid when used in `Concatenate`, or as the first argument to `Callable`, or as parameters for user-defined Generics. See *Generic* for more information on generic types.

For example, to add basic logging to a function, one can create a decorator `add_logging` to log function calls. The parameter specification variable tells the type checker that the callable passed into the decorator and the new callable returned by it have inter-dependent type parameters:

```
from collections.abc import Callable
import logging

def add_logging[T, **P](f: Callable[P, T]) -> Callable[P, T]:
    '''A type-safe decorator to add logging to a function.'''
    def inner(*args: P.args, **kwargs: P.kwargs) -> T:
        logging.info(f'{f.__name__} was called')
        return f(*args, **kwargs)
    return inner

@add_logging
def add_two(x: float, y: float) -> float:
    '''Add two numbers together.'''
    return x + y
```

Without `ParamSpec`, the simplest way to annotate this previously was to use a *`TypeVar`* with bound `Callable[..., Any]`. However this causes two problems:

1. The type checker can't type check the `inner` function because `*args` and `**kwargs` have to be typed *`Any`*.
2. *`cast()`* may be required in the body of the `add_logging` decorator when returning the `inner` function, or the static type checker must be told to ignore the `return inner`.

args

kwargs

Since `ParamSpec` captures both positional and keyword parameters, `P.args` and `P.kwargs` can be used to split a `ParamSpec` into its components. `P.args` represents the tuple of positional parameters in a given call and should only be used to annotate `*args`. `P.kwargs` represents the mapping of keyword parameters to their values in a given call, and should be only be used to annotate `**kwargs`. Both attributes require the annotated parameter to be in scope. At runtime, `P.args` and `P.kwargs` are instances respectively of *`ParamSpecArgs`* and *`ParamSpecKwargs`*.

__name__

The name of the parameter specification.

__default__

The default value of the parameter specification, or *`typing.NoDefault`* if it has no default.

Added in version 3.13.

has_default()

Return whether or not the parameter specification has a default value. This is equivalent to checking whether *`__default__`* is not the *`typing.NoDefault`* singleton, except that it does not force evaluation of the lazily evaluated default value.

Added in version 3.13.

Parameter specification variables created with `covariant=True` or `contravariant=True` can be used to declare covariant or contravariant generic types. The `bound` argument is also accepted, similar to *`TypeVar`*. However the actual semantics of these keywords are yet to be decided.

Added in version 3.10.

Changed in version 3.12: Parameter specifications can now be declared using the type parameter syntax introduced by **PEP 695**.

Changed in version 3.13: Support for default values was added.

Note

Only parameter specification variables defined in global scope can be pickled.

See also

- **PEP 612** – Parameter Specification Variables (the PEP which introduced `ParamSpec` and `Concatenate`)
- *`Concatenate`*
- *Annotating callable objects*

`typing.`**`ParamSpecArgs`**

`typing.`**`ParamSpecKwargs`**

Arguments and keyword arguments attributes of a *`ParamSpec`*. The `P.args` attribute of a `ParamSpec` is an instance of `ParamSpecArgs`, and `P.kwargs` is an instance of `ParamSpecKwargs`. They are intended for runtime introspection and have no special meaning to static type checkers.

Calling *`get_origin()`* on either of these objects will return the original `ParamSpec`:

```
>>> from typing import ParamSpec, get_origin
>>> P = ParamSpec("P")
>>> get_origin(P.args) is P
True
>>> get_origin(P.kwargs) is P
True
```

Added in version 3.10.

`class` `typing.`**`TypeAliasType`**(*name*, *value*, *, *type_params=()*)

The type of type aliases created through the `type` statement.

Example:

```
>>> type Alias = int
>>> type(Alias)
<class 'typing.TypeAliasType'>
```

Added in version 3.12.

`__name__`

The name of the type alias:

```
>>> type Alias = int
>>> Alias.__name__
'Alias'
```

`__module__`

The module in which the type alias was defined:

```
>>> type Alias = int
>>> Alias.__module__
'__main__'
```

__type_params__

The type parameters of the type alias, or an empty tuple if the alias is not generic:

```
>>> type ListOrSet[T] = list[T] | set[T]
>>> ListOrSet.__type_params__
(T,)
>>> type NotGeneric = int
>>> NotGeneric.__type_params__
()
```

__value__

The type alias's value. This is lazily evaluated, so names used in the definition of the alias are not resolved until the `__value__` attribute is accessed:

```
>>> type Mutually = Recursive
>>> type Recursive = Mutually
>>> Mutually
Mutually
>>> Recursive
Recursive
>>> Mutually.__value__
Recursive
>>> Recursive.__value__
Mutually
```

Other special directives

These functions and classes should not be used directly as annotations. Their intended purpose is to be building blocks for creating and declaring types.

class typing.**NamedTuple**

Typed version of *collections.namedtuple()*.

Usage:

```
class Employee(NamedTuple):
    name: str
    id: int
```

This is equivalent to:

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

To give a field a default value, you can assign to it in the class body:

```
class Employee(NamedTuple):
    name: str
    id: int = 3

employee = Employee('Guido')
assert employee.id == 3
```

Fields with a default value must come after any fields without a default.

The resulting class has an extra attribute `__annotations__` giving a dict that maps the field names to the field types. (The field names are in the `_fields` attribute and the default values are in the `_field_defaults` attribute, both of which are part of the *namedtuple()* API.)

`NamedTuple` subclasses can also have docstrings and methods:

```
class Employee(NamedTuple):
    """Represents an employee."""
    name: str
    id: int = 3

    def __repr__(self) -> str:
        return f'<Employee {self.name}, id={self.id}>'
```

`NamedTuple` subclasses can be generic:

```
class Group[T](NamedTuple):
    key: T
    group: list[T]
```

Backward-compatible usage:

```
# For creating a generic NamedTuple on Python 3.11 or lower
class Group(NamedTuple, Generic[T]):
    key: T
    group: list[T]

# A functional syntax is also supported
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

Changed in version 3.6: Added support for **PEP 526** variable annotation syntax.

Changed in version 3.6.1: Added support for default values, methods, and docstrings.

Changed in version 3.8: The `_field_types` and `__annotations__` attributes are now regular dictionaries instead of instances of `OrderedDict`.

Changed in version 3.9: Removed the `_field_types` attribute in favor of the more standard `__annotations__` attribute which has the same information.

Changed in version 3.11: Added support for generic namedtuples.

Deprecated since version 3.13, will be removed in version 3.15: The undocumented keyword argument syntax for creating NamedTuple classes (`NT = NamedTuple("NT", x=int)`) is deprecated, and will be disallowed in 3.15. Use the class-based syntax or the functional syntax instead.

Deprecated since version 3.13, will be removed in version 3.15: When using the functional syntax to create a NamedTuple class, failing to pass a value to the 'fields' parameter (`NT = NamedTuple("NT")`) is deprecated. Passing `None` to the 'fields' parameter (`NT = NamedTuple("NT", None)`) is also deprecated. Both will be disallowed in Python 3.15. To create a NamedTuple class with 0 fields, use `class NT(NamedTuple): pass` or `NT = NamedTuple("NT", [])`.

class typing.**NewType**(*name*, *tp*)

Helper class to create low-overhead *distinct types*.

A `NewType` is considered a distinct type by a typechecker. At runtime, however, calling a `NewType` returns its argument unchanged.

Usage:

```
UserId = NewType('UserId', int)  # Declare the NewType "UserId"
first_user = UserId(1)  # "UserId" returns the argument unchanged at runtime
```

__module__

The module in which the new type is defined.

__name__

The name of the new type.

__supertype__

The type that the new type is based on.

Added in version 3.5.2.

Changed in version 3.10: `NewType` is now a class rather than a function.

class typing.**Protocol**(*Generic*)

Base class for protocol classes.

Protocol classes are defined like this:

```
class Proto(Protocol):
    def meth(self) -> int:
        ...
```

Such classes are primarily used with static type checkers that recognize structural subtyping (static duck-typing), for example:

```
class C:
    def meth(self) -> int:
        return 0

def func(x: Proto) -> int:
    return x.meth()

func(C())  # Passes static type check
```

See **PEP 544** for more details. Protocol classes decorated with *runtime_checkable()* (described later) act as simple-minded runtime protocols that check only the presence of given attributes, ignoring their type signatures.

Protocol classes can be generic, for example:

```
class GenProto[T](Protocol):
    def meth(self) -> T:
        ...
```

In code that needs to be compatible with Python 3.11 or older, generic Protocols can be written as follows:

```
T = TypeVar("T")

class GenProto(Protocol[T]):
    def meth(self) -> T:
        ...
```

Added in version 3.8.

@typing.**runtime_checkable**

Mark a protocol class as a runtime protocol.

Such a protocol can be used with *isinstance()* and *issubclass()*. This raises *TypeError* when applied to a non-protocol class. This allows a simple-minded structural check, very similar to "one trick ponies" in *collections.abc* such as *Iterable*. For example:

```
@runtime_checkable
class Closable(Protocol):
    def close(self): ...

assert isinstance(open('/some/file'), Closable)

@runtime_checkable
class Named(Protocol):
    name: str

import threading
assert isinstance(threading.Thread(name='Bob'), Named)
```

Note

`runtime_checkable()` will check only the presence of the required methods or attributes, not their type signatures or types. For example, *ssl.SSLObject* is a class, therefore it passes an *issubclass()* check against *Callable*. However, the `ssl.SSLObject.__init__` method exists only to raise a *TypeError* with a more informative message, therefore making it impossible to call (instantiate) *ssl.SSLObject*.

Note

An *isinstance()* check against a runtime-checkable protocol can be surprisingly slow compared to an `isinstance()` check against a non-protocol class. Consider using alternative idioms such as *hasattr()* calls for structural checks in performance-sensitive code.

Added in version 3.8.

Changed in version 3.12: The internal implementation of *isinstance()* checks against runtime-checkable protocols now uses *inspect.getattr_static()* to look up attributes (previously, *hasattr()* was used). As a result, some objects which used to be considered instances of a runtime-checkable protocol may no longer be considered instances of that protocol on Python 3.12+, and vice versa. Most users are unlikely to be affected by this change.

Changed in version 3.12: The members of a runtime-checkable protocol are now considered "frozen" at runtime as soon as the class has been created. Monkey-patching attributes onto a runtime-checkable protocol will still work, but will have no impact on *isinstance()* checks comparing objects to the protocol. See "What's new in Python 3.12" for more details.

class typing.**TypedDict**(*dict*)

Special construct to add type hints to a dictionary. At runtime it is a plain *dict*.

`TypedDict` declares a dictionary type that expects all of its instances to have a certain set of keys, where each key is associated with a value of a consistent type. This expectation is not checked at runtime but is only enforced by type checkers. Usage:

```
class Point2D(TypedDict):
    x: int
    y: int
    label: str
```

(continues on next page)

(continued from previous page)

```
a: Point2D = {'x': 1, 'y': 2, 'label': 'good'}  # OK
b: Point2D = {'z': 3, 'label': 'bad'}           # Fails type check

assert Point2D(x=1, y=2, label='first') == dict(x=1, y=2, label='first')
```

An alternative way to create a `TypedDict` is by using function-call syntax. The second argument must be a literal *dict*:

```
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label': str})
```

This functional syntax allows defining keys which are not valid identifiers, for example because they are keywords or contain hyphens:

```
# raises SyntaxError
class Point2D(TypedDict):
    in: int  # 'in' is a keyword
    x-y: int  # name with hyphens

# OK, functional syntax
Point2D = TypedDict('Point2D', {'in': int, 'x-y': int})
```

By default, all keys must be present in a `TypedDict`. It is possible to mark individual keys as non-required using *NotRequired*:

```
class Point2D(TypedDict):
    x: int
    y: int
    label: NotRequired[str]

# Alternative syntax
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label': NotRequired[str]})
```

This means that a `Point2D` `TypedDict` can have the `label` key omitted.

It is also possible to mark all keys as non-required by default by specifying a totality of `False`:

```
class Point2D(TypedDict, total=False):
    x: int
    y: int

# Alternative syntax
Point2D = TypedDict('Point2D', {'x': int, 'y': int}, total=False)
```

This means that a `Point2D` `TypedDict` can have any of the keys omitted. A type checker is only expected to support a literal `False` or `True` as the value of the `total` argument. `True` is the default, and makes all items defined in the class body required.

Individual keys of a `total=False` `TypedDict` can be marked as required using *Required*:

```
class Point2D(TypedDict, total=False):
    x: Required[int]
    y: Required[int]
    label: str

# Alternative syntax
Point2D = TypedDict('Point2D', {
```

(continues on next page)

(continued from previous page)

```
    'x': Required[int],
    'y': Required[int],
    'label': str
}, total=False)
```

It is possible for a `TypedDict` type to inherit from one or more other `TypedDict` types using the class-based syntax. Usage:

```
class Point3D(Point2D):
    z: int
```

`Point3D` has three items: `x`, `y` and `z`. It is equivalent to this definition:

```
class Point3D(TypedDict):
    x: int
    y: int
    z: int
```

A `TypedDict` cannot inherit from a non-`TypedDict` class, except for *`Generic`*. For example:

```
class X(TypedDict):
    x: int

class Y(TypedDict):
    y: int

class Z(object): pass  # A non-TypedDict class

class XY(X, Y): pass  # OK

class XZ(X, Z): pass  # raises TypeError
```

A `TypedDict` can be generic:

```
class Group[T](TypedDict):
    key: T
    group: list[T]
```

To create a generic `TypedDict` that is compatible with Python 3.11 or lower, inherit from *`Generic`* explicitly:

```
T = TypeVar("T")

class Group(TypedDict, Generic[T]):
    key: T
    group: list[T]
```

A `TypedDict` can be introspected via annotations dicts (see annotations-howto for more information on annotations best practices), *`__total__`*, *`__required_keys__`*, and *`__optional_keys__`*.

`__total__`

`Point2D.__total__` gives the value of the `total` argument. Example:

```
>>> from typing import TypedDict
>>> class Point2D(TypedDict): pass
>>> Point2D.__total__
True
```

(continues on next page)

(continued from previous page)

```
>>> class Point2D(TypedDict, total=False): pass
>>> Point2D.__total__
False
>>> class Point3D(Point2D): pass
>>> Point3D.__total__
True
```

This attribute reflects *only* the value of the `total` argument to the current `TypedDict` class, not whether the class is semantically total. For example, a `TypedDict` with `__total__` set to `True` may have keys marked with *`NotRequired`*, or it may inherit from another `TypedDict` with `total=False`. Therefore, it is generally better to use *`__required_keys__`* and *`__optional_keys__`* for introspection.

`__required_keys__`

Added in version 3.9.

`__optional_keys__`

`Point2D.__required_keys__` and `Point2D.__optional_keys__` return *`frozenset`* objects containing required and non-required keys, respectively.

Keys marked with *`Required`* will always appear in `__required_keys__` and keys marked with *`NotRequired`* will always appear in `__optional_keys__`.

For backwards compatibility with Python 3.10 and below, it is also possible to use inheritance to declare both required and non-required keys in the same `TypedDict` . This is done by declaring a `TypedDict` with one value for the `total` argument and then inheriting from it in another `TypedDict` with a different value for `total`:

```
>>> class Point2D(TypedDict, total=False):
...     x: int
...     y: int
...
>>> class Point3D(Point2D):
...     z: int
...
>>> Point3D.__required_keys__ == frozenset({'z'})
True
>>> Point3D.__optional_keys__ == frozenset({'x', 'y'})
True
```

Added in version 3.9.

Note

If `from __future__ import annotations` is used or if annotations are given as strings, annotations are not evaluated when the `TypedDict` is defined. Therefore, the runtime introspection that `__required_keys__` and `__optional_keys__` rely on may not work properly, and the values of the attributes may be incorrect.

Support for *`ReadOnly`* is reflected in the following attributes:

`__readonly_keys__`

A *`frozenset`* containing the names of all read-only keys. Keys are read-only if they carry the *`ReadOnly`* qualifier.

Added in version 3.13.

__mutable_keys__

A *frozenset* containing the names of all mutable keys. Keys are mutable if they do not carry the *ReadOnly* qualifier.

Added in version 3.13.

See **PEP 589** for more examples and detailed rules of using `TypedDict`.

Added in version 3.8.

Changed in version 3.11: Added support for marking individual keys as *Required* or *NotRequired*. See **PEP 655**.

Changed in version 3.11: Added support for generic `TypedDicts`.

Changed in version 3.13: Removed support for the keyword-argument method of creating `TypedDicts`.

Changed in version 3.13: Support for the *ReadOnly* qualifier was added.

Deprecated since version 3.13, will be removed in version 3.15: When using the functional syntax to create a TypedDict class, failing to pass a value to the 'fields' parameter (`TD = TypedDict("TD")`) is deprecated. Passing `None` to the 'fields' parameter (`TD = TypedDict("TD", None)`) is also deprecated. Both will be disallowed in Python 3.15. To create a TypedDict class with 0 fields, use `class TD(TypedDict): pass` or `TD = TypedDict("TD", {})`.

Protocols

The following protocols are provided by the typing module. All are decorated with *@runtime_checkable*.

class typing.**SupportsAbs**

An ABC with one abstract method `__abs__` that is covariant in its return type.

class typing.**SupportsBytes**

An ABC with one abstract method `__bytes__`.

class typing.**SupportsComplex**

An ABC with one abstract method `__complex__`.

class typing.**SupportsFloat**

An ABC with one abstract method `__float__`.

class typing.**SupportsIndex**

An ABC with one abstract method `__index__`.

Added in version 3.8.

class typing.**SupportsInt**

An ABC with one abstract method `__int__`.

class typing.**SupportsRound**

An ABC with one abstract method `__round__` that is covariant in its return type.

ABCs for working with IO

class typing.**IO**

class typing.**TextIO**

class typing.**BinaryIO**

Generic type `IO[AnyStr]` and its subclasses `TextIO(IO[str])` and `BinaryIO(IO[bytes])` represent the types of I/O streams such as returned by *open()*.

Functions and decorators

typing.**cast**(*typ*, *val*)

Cast a value to a type.

This returns the value unchanged. To the type checker this signals that the return value has the designated type, but at runtime we intentionally don't check anything (we want this to be as fast as possible).

typing.**assert_type**(*val*, *typ*, */*)

Ask a static type checker to confirm that *val* has an inferred type of *typ*.

At runtime this does nothing: it returns the first argument unchanged with no checks or side effects, no matter the actual type of the argument.

When a static type checker encounters a call to `assert_type()`, it emits an error if the value is not of the specified type:

```
def greet(name: str) -> None:
    assert_type(name, str)  # OK, inferred type of `name` is `str`
    assert_type(name, int)  # type checker error
```

This function is useful for ensuring the type checker's understanding of a script is in line with the developer's intentions:

```
def complex_function(arg: object):
    # Do some complex type-narrowing logic,
    # after which we hope the inferred type will be `int`
    ...
    # Test whether the type checker correctly understands our function
    assert_type(arg, int)
```

Added in version 3.11.

typing.**assert_never**(*arg*, */*)

Ask a static type checker to confirm that a line of code is unreachable.

Example:

```
def int_or_str(arg: int | str) -> None:
    match arg:
        case int():
            print("It's an int")
        case str():
            print("It's a str")
        case _ as unreachable:
            assert_never(unreachable)
```

Here, the annotations allow the type checker to infer that the last case can never execute, because `arg` is either an *int* or a *str*, and both options are covered by earlier cases.

If a type checker finds that a call to `assert_never()` is reachable, it will emit an error. For example, if the type annotation for `arg` was instead `int | str | float`, the type checker would emit an error pointing out that `unreachable` is of type *float*. For a call to `assert_never` to pass type checking, the inferred type of the argument passed in must be the bottom type, *Never*, and nothing else.

At runtime, this throws an exception when called.

See also

Unreachable Code and Exhaustiveness Checking has more information about exhaustiveness checking with static typing.

Added in version 3.11.

`typing.`**`reveal_type`**(*obj*, */*)

Ask a static type checker to reveal the inferred type of an expression.

When a static type checker encounters a call to this function, it emits a diagnostic with the inferred type of the argument. For example:

```
x: int = 1
reveal_type(x)  # Revealed type is "builtins.int"
```

This can be useful when you want to debug how your type checker handles a particular piece of code.

At runtime, this function prints the runtime type of its argument to *sys.stderr* and returns the argument unchanged (allowing the call to be used within an expression):

```
x = reveal_type(1)  # prints "Runtime type is int"
print(x)  # prints "1"
```

Note that the runtime type may be different from (more or less specific than) the type statically inferred by a type checker.

Most type checkers support `reveal_type()` anywhere, even if the name is not imported from `typing`. Importing the name from `typing`, however, allows your code to run without runtime errors and communicates intent more clearly.

Added in version 3.11.

`@typing.`**`dataclass_transform`**(**, eq_default=True, order_default=False, kw_only_default=False, frozen_default=False, field_specifiers=(), **kwargs*)

Decorator to mark an object as providing *dataclass*-like behavior.

`dataclass_transform` may be used to decorate a class, metaclass, or a function that is itself a decorator. The presence of `@dataclass_transform()` tells a static type checker that the decorated object performs runtime "magic" that transforms a class in a similar way to *@dataclasses.dataclass*.

Example usage with a decorator function:

```
@dataclass_transform()
def create_model[T](cls: type[T]) -> type[T]:
    ...
    return cls

@create_model
class CustomerModel:
```

(continues on next page)

(continued from previous page)

```
    id: int
    name: str
```

On a base class:

```
@dataclass_transform()
class ModelBase: ...

class CustomerModel(ModelBase):
    id: int
    name: str
```

On a metaclass:

```
@dataclass_transform()
class ModelMeta(type): ...

class ModelBase(metaclass=ModelMeta): ...

class CustomerModel(ModelBase):
    id: int
    name: str
```

The `CustomerModel` classes defined above will be treated by type checkers similarly to classes created with *`@dataclasses.dataclass`*. For example, type checkers will assume these classes have `__init__` methods that accept `id` and `name`.

The decorated class, metaclass, or function may accept the following bool arguments which type checkers will assume have the same effect as they would have on the *`@dataclasses.dataclass`* decorator: `init`, `eq`, `order`, `unsafe_hash`, `frozen`, `match_args`, `kw_only`, and `slots`. It must be possible for the value of these arguments (`True` or `False`) to be statically evaluated.

The arguments to the `dataclass_transform` decorator can be used to customize the default behaviors of the decorated class, metaclass, or function:

Parameters

- **`eq_default`** (`bool`) – Indicates whether the `eq` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `True`.
- **`order_default`** (`bool`) – Indicates whether the `order` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `False`.
- **`kw_only_default`** (`bool`) – Indicates whether the `kw_only` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `False`.
- **`frozen_default`** (`bool`) – Indicates whether the `frozen` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `False`.

  Added in version 3.12.
- **`field_specifiers`** (*`tuple[Callable[..., Any], ...]`*) – Specifies a static list of supported classes or functions that describe fields, similar to *`dataclasses.field()`*. Defaults to `()`.
- **`**kwargs`** (*`Any`*) – Arbitrary other keyword arguments are accepted in order to allow for possible future extensions.

Type checkers recognize the following optional parameters on field specifiers:

Table 1: **Recognised parameters for field specifiers**

Parameter name	Description
`init`	Indicates whether the field should be included in the synthesized `__init__` method. If unspecified, `init` defaults to `True`.
`default`	Provides the default value for the field.
`default_factory`	Provides a runtime callback that returns the default value for the field. If neither `default` nor `default_factory` are specified, the field is assumed to have no default value and must be provided a value when the class is instantiated.
`factory`	An alias for the `default_factory` parameter on field specifiers.
`kw_only`	Indicates whether the field should be marked as keyword-only. If `True`, the field will be keyword-only. If `False`, it will not be keyword-only. If unspecified, the value of the `kw_only` parameter on the object decorated with `dataclass_transform` will be used, or if that is unspecified, the value of `kw_only_default` on `dataclass_transform` will be used.
`alias`	Provides an alternative name for the field. This alternative name is used in the synthesized `__init__` method.

At runtime, this decorator records its arguments in the `__dataclass_transform__` attribute on the decorated object. It has no other runtime effect.

See **PEP 681** for more details.

Added in version 3.11.

`@typing.`**`overload`**

Decorator for creating overloaded functions and methods.

The `@overload` decorator allows describing functions and methods that support multiple different combinations of argument types. A series of `@overload`-decorated definitions must be followed by exactly one non-`@overload`-decorated definition (for the same function/method).

`@overload`-decorated definitions are for the benefit of the type checker only, since they will be overwritten by the non-`@overload`-decorated definition. The non-`@overload`-decorated definition, meanwhile, will be used at runtime but should be ignored by a type checker. At runtime, calling an `@overload`-decorated function directly will raise *`NotImplementedError`*.

An example of overload that gives a more precise type than can be expressed using a union or a type variable:

```
@overload
def process(response: None) -> None:
    ...
@overload
def process(response: int) -> tuple[int, str]:
    ...
@overload
def process(response: bytes) -> str:
    ...
def process(response):
    ...  # actual implementation goes here
```

See **PEP 484** for more details and comparison with other typing semantics.

Changed in version 3.11: Overloaded functions can now be introspected at runtime using *`get_overloads()`*.

`typing.`**`get_overloads`**`(`*`func`*`)`

Return a sequence of *`@overload`*-decorated definitions for *func*.

func is the function object for the implementation of the overloaded function. For example, given the definition of `process` in the documentation for *`@overload`*, `get_overloads(process)` will return a sequence of three function objects for the three defined overloads. If called on a function with no overloads, `get_overloads()` returns an empty sequence.

`get_overloads()` can be used for introspecting an overloaded function at runtime.

Added in version 3.11.

`typing.`**`clear_overloads`**`()`

Clear all registered overloads in the internal registry.

This can be used to reclaim the memory used by the registry.

Added in version 3.11.

`@typing.`**`final`**

Decorator to indicate final methods and final classes.

Decorating a method with `@final` indicates to a type checker that the method cannot be overridden in a subclass. Decorating a class with `@final` indicates that it cannot be subclassed.

For example:

```
class Base:
    @final
    def done(self) -> None:
        ...
class Sub(Base):
    def done(self) -> None:  # Error reported by type checker
        ...

@final
class Leaf:
    ...
class Other(Leaf):  # Error reported by type checker
    ...
```

There is no runtime checking of these properties. See **PEP 591** for more details.

Added in version 3.8.

Changed in version 3.11: The decorator will now attempt to set a `__final__` attribute to `True` on the decorated object. Thus, a check like `if getattr(obj, "__final__", False)` can be used at runtime to determine whether an object `obj` has been marked as final. If the decorated object does not support setting attributes, the decorator returns the object unchanged without raising an exception.

`@typing.`**`no_type_check`**

Decorator to indicate that annotations are not type hints.

This works as a class or function *decorator*. With a class, it applies recursively to all methods and classes defined in that class (but not to methods defined in its superclasses or subclasses). Type checkers will ignore all annotations in a function or class with this decorator.

`@no_type_check` mutates the decorated object in place.

`@typing.`**`no_type_check_decorator`**

Decorator to give another decorator the *`no_type_check()`* effect.

This wraps the decorator with something that wraps the decorated function in *`no_type_check()`*.

Deprecated since version 3.13, will be removed in version 3.15: No type checker ever added support for `@no_type_check_decorator`. It is therefore deprecated, and will be removed in Python 3.15.

@typing.**override**

Decorator to indicate that a method in a subclass is intended to override a method or attribute in a superclass.

Type checkers should emit an error if a method decorated with `@override` does not, in fact, override anything. This helps prevent bugs that may occur when a base class is changed without an equivalent change to a child class.

For example:

```
class Base:
    def log_status(self) -> None:
        ...

class Sub(Base):
    @override
    def log_status(self) -> None:  # Okay: overrides Base.log_status
        ...

    @override
    def done(self) -> None:  # Error reported by type checker
        ...
```

There is no runtime checking of this property.

The decorator will attempt to set an `__override__` attribute to `True` on the decorated object. Thus, a check like `if getattr(obj, "__override__", False)` can be used at runtime to determine whether an object `obj` has been marked as an override. If the decorated object does not support setting attributes, the decorator returns the object unchanged without raising an exception.

See **PEP 698** for more details.

Added in version 3.12.

@typing.**type_check_only**

Decorator to mark a class or function as unavailable at runtime.

This decorator is itself not available at runtime. It is mainly intended to mark classes that are defined in type stub files if an implementation returns an instance of a private class:

```
@type_check_only
class Response:  # private or not available at runtime
    code: int
    def get_header(self, name: str) -> str: ...

def fetch_response() -> Response: ...
```

Note that returning instances of private classes is not recommended. It is usually preferable to make such classes public.

Introspection helpers

typing.**get_type_hints**(*obj*, *globalns=None*, *localns=None*, *include_extras=False*)

Return a dictionary containing type hints for a function, method, module or class object.

This is often the same as `obj.__annotations__`, but this function makes the following changes to the annotations dictionary:

- Forward references encoded as string literals or *ForwardRef* objects are handled by evaluating them in *globalns*, *localns*, and (where applicable) *obj*'s type parameter namespace. If *globalns* or *localns* is not given, appropriate namespace dictionaries are inferred from *obj*.

- `None` is replaced with *types.NoneType*.
- If *@no_type_check* has been applied to *obj*, an empty dictionary is returned.
- If *obj* is a class `C`, the function returns a dictionary that merges annotations from `C`'s base classes with those on `C` directly. This is done by traversing `C.__mro__` and iteratively combining `__annotations__` dictionaries. Annotations on classes appearing earlier in the *method resolution order* always take precedence over annotations on classes appearing later in the method resolution order.
- The function recursively replaces all occurrences of `Annotated[T, ...]` with `T`, unless *include_extras* is set to `True` (see *Annotated* for more information).

See also *inspect.get_annotations()*, a lower-level function that returns annotations more directly.

Note

If any forward references in the annotations of *obj* are not resolvable or are not valid Python code, this function will raise an exception such as *NameError*. For example, this can happen with imported *type aliases* that include forward references, or with names imported under *if TYPE_CHECKING*.

Changed in version 3.9: Added `include_extras` parameter as part of **PEP 593**. See the documentation on *Annotated* for more information.

Changed in version 3.11: Previously, `Optional[t]` was added for function and method annotations if a default value equal to `None` was set. Now the annotation is returned unchanged.

`typing.`**`get_origin`**`(`*tp*`)`

Get the unsubscripted version of a type: for a typing object of the form `X[Y, Z, ...]` return `X`.

If `X` is a typing-module alias for a builtin or *collections* class, it will be normalized to the original class. If `X` is an instance of *ParamSpecArgs* or *ParamSpecKwargs*, return the underlying *ParamSpec*. Return `None` for unsupported objects.

Examples:

```
assert get_origin(str) is None
assert get_origin(Dict[str, int]) is dict
assert get_origin(Union[int, str]) is Union
P = ParamSpec('P')
assert get_origin(P.args) is P
assert get_origin(P.kwargs) is P
```

Added in version 3.8.

`typing.`**`get_args`**`(`*tp*`)`

Get type arguments with all substitutions performed: for a typing object of the form `X[Y, Z, ...]` return `(Y, Z, ...)`.

If `X` is a union or *Literal* contained in another generic type, the order of `(Y, Z, ...)` may be different from the order of the original arguments `[Y, Z, ...]` due to type caching. Return `()` for unsupported objects.

Examples:

```
assert get_args(int) == ()
assert get_args(Dict[int, str]) == (int, str)
assert get_args(Union[int, str]) == (int, str)
```

Added in version 3.8.

typing.**get_protocol_members**(*tp*)

Return the set of members defined in a *Protocol*.

```
>>> from typing import Protocol, get_protocol_members
>>> class P(Protocol):
...     def a(self) -> str: ...
...     b: int
>>> get_protocol_members(P) == frozenset({'a', 'b'})
True
```

Raise *TypeError* for arguments that are not Protocols.

Added in version 3.13.

typing.**is_protocol**(*tp*)

Determine if a type is a *Protocol*.

For example:

```
class P(Protocol):
    def a(self) -> str: ...
    b: int

is_protocol(P)     # => True
is_protocol(int)   # => False
```

Added in version 3.13.

typing.**is_typeddict**(*tp*)

Check if a type is a *TypedDict*.

For example:

```
class Film(TypedDict):
    title: str
    year: int

assert is_typeddict(Film)
assert not is_typeddict(list | str)

# TypedDict is a factory for creating typed dicts,
# not a typed dict itself
assert not is_typeddict(TypedDict)
```

Added in version 3.10.

class typing.**ForwardRef**

Class used for internal typing representation of string forward references.

For example, `List["SomeClass"]` is implicitly transformed into `List[ForwardRef("SomeClass")]`. `ForwardRef` should not be instantiated by a user, but may be used by introspection tools.

Note

PEP 585 generic types such as `list["SomeClass"]` will not be implicitly transformed into `list[ForwardRef("SomeClass")]` and thus will not automatically resolve to `list[SomeClass]`.

Added in version 3.7.4.

typing.**NoDefault**

A sentinel object used to indicate that a type parameter has no default value. For example:

```
>>> T = TypeVar("T")
>>> T.__default__ is typing.NoDefault
True
>>> S = TypeVar("S", default=None)
>>> S.__default__ is None
True
```

Added in version 3.13.

Constant

typing.**TYPE_CHECKING**

A special constant that is assumed to be `True` by 3rd party static type checkers. It is `False` at runtime.

Usage:

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: 'expensive_mod.SomeType') -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

The first type annotation must be enclosed in quotes, making it a "forward reference", to hide the `expensive_mod` reference from the interpreter runtime. Type annotations for local variables are not evaluated, so the second annotation does not need to be enclosed in quotes.

Note

If `from __future__ import annotations` is used, annotations are not evaluated at function definition time. Instead, they are stored as strings in `__annotations__`. This makes it unnecessary to use quotes around the annotation (see **PEP 563**).

Added in version 3.5.2.

Deprecated aliases

This module defines several deprecated aliases to pre-existing standard library classes. These were originally included in the typing module in order to support parameterizing these generic classes using `[]`. However, the aliases became redundant in Python 3.9 when the corresponding pre-existing classes were enhanced to support `[]` (see **PEP 585**).

The redundant types are deprecated as of Python 3.9. However, while the aliases may be removed at some point, removal of these aliases is not currently planned. As such, no deprecation warnings are currently issued by the interpreter for these aliases.

If at some point it is decided to remove these deprecated aliases, a deprecation warning will be issued by the interpreter for at least two releases prior to removal. The aliases are guaranteed to remain in the typing module without deprecation warnings until at least Python 3.14.

Type checkers are encouraged to flag uses of the deprecated types if the program they are checking targets a minimum Python version of 3.9 or newer.

Aliases to built-in types

class typing.**Dict**(*dict, MutableMapping[KT, VT]*)

Deprecated alias to *dict*.

Note that to annotate arguments, it is preferred to use an abstract collection type such as *Mapping* rather than to use *dict* or typing.Dict.

This type can be used as follows:

```
def count_words(text: str) -> Dict[str, int]:
    ...
```

Deprecated since version 3.9: *builtins.dict* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

class typing.**List**(*list, MutableSequence[T]*)

Deprecated alias to *list*.

Note that to annotate arguments, it is preferred to use an abstract collection type such as *Sequence* or *Iterable* rather than to use *list* or typing.List.

This type may be used as follows:

```
def vec2[T: (int, float)](x: T, y: T) -> List[T]:
    return [x, y]

def keep_positives[T: (int, float)](vector: Sequence[T]) -> List[T]:
    return [item for item in vector if item > 0]
```

Deprecated since version 3.9: *builtins.list* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

class typing.**Set**(*set, MutableSet[T]*)

Deprecated alias to *builtins.set*.

Note that to annotate arguments, it is preferred to use an abstract collection type such as *AbstractSet* rather than to use *set* or typing.Set.

Deprecated since version 3.9: *builtins.set* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

class typing.**FrozenSet**(*frozenset, AbstractSet[T_co]*)

Deprecated alias to *builtins.frozenset*.

Deprecated since version 3.9: *builtins.frozenset* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

typing.**Tuple**

Deprecated alias for *tuple*.

tuple and Tuple are special-cased in the type system; see *Annotating tuples* for more details.

Deprecated since version 3.9: *builtins.tuple* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

class typing.**Type**(*Generic[CT_co]*)

Deprecated alias to *type*.

See *The type of class objects* for details on using *type* or typing.Type in type annotations.

Added in version 3.5.2.

Deprecated since version 3.9: *builtins.type* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

Aliases to types in collections

class typing.**DefaultDict**(*collections.defaultdict, MutableMapping[KT, VT]*)

Deprecated alias to *collections.defaultdict*.

Added in version 3.5.2.

Deprecated since version 3.9: *collections.defaultdict* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

class typing.**OrderedDict**(*collections.OrderedDict, MutableMapping[KT, VT]*)

Deprecated alias to *collections.OrderedDict*.

Added in version 3.7.2.

Deprecated since version 3.9: *collections.OrderedDict* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

class typing.**ChainMap**(*collections.ChainMap, MutableMapping[KT, VT]*)

Deprecated alias to *collections.ChainMap*.

Added in version 3.6.1.

Deprecated since version 3.9: *collections.ChainMap* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

class typing.**Counter**(*collections.Counter, Dict[T, int]*)

Deprecated alias to *collections.Counter*.

Added in version 3.6.1.

Deprecated since version 3.9: *collections.Counter* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

class typing.**Deque**(*deque, MutableSequence[T]*)

Deprecated alias to *collections.deque*.

Added in version 3.6.1.

Deprecated since version 3.9: *collections.deque* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

Aliases to other concrete types

class typing.**Pattern**

class typing.**Match**

Deprecated aliases corresponding to the return types from *re.compile()* and *re.match()*.

These types (and the corresponding functions) are generic over *AnyStr*. Pattern can be specialised as Pattern[str] or Pattern[bytes]; Match can be specialised as Match[str] or Match[bytes].

Deprecated since version 3.9: Classes Pattern and Match from *re* now support []. See **PEP 585** and *Generic Alias Type*.

class typing.**Text**

Deprecated alias for *str*.

`Text` is provided to supply a forward compatible path for Python 2 code: in Python 2, `Text` is an alias for `unicode`.

Use `Text` to indicate that a value must contain a unicode string in a manner that is compatible with both Python 2 and Python 3:

```
def add_unicode_checkmark(text: Text) -> Text:
    return text + u' \u2713'
```

Added in version 3.5.2.

Deprecated since version 3.11: Python 2 is no longer supported, and most type checkers also no longer support type checking Python 2 code. Removal of the alias is not currently planned, but users are encouraged to use *str* instead of `Text`.

Aliases to container ABCs in `collections.abc`

class typing.**AbstractSet**(*Collection[T_co]*)

Deprecated alias to *collections.abc.Set*.

Deprecated since version 3.9: *collections.abc.Set* now supports subscripting (`[]`). See **PEP 585** and *Generic Alias Type*.

class typing.**ByteString**(*Sequence[int]*)

This type represents the types *bytes*, *bytearray*, and *memoryview* of byte sequences.

Deprecated since version 3.9, will be removed in version 3.14: Prefer *collections.abc.Buffer*, or a union like `bytes | bytearray | memoryview`.

class typing.**Collection**(*Sized, Iterable[T_co], Container[T_co]*)

Deprecated alias to *collections.abc.Collection*.

Added in version 3.6.

Deprecated since version 3.9: *collections.abc.Collection* now supports subscripting (`[]`). See **PEP 585** and *Generic Alias Type*.

class typing.**Container**(*Generic[T_co]*)

Deprecated alias to *collections.abc.Container*.

Deprecated since version 3.9: *collections.abc.Container* now supports subscripting (`[]`). See **PEP 585** and *Generic Alias Type*.

class typing.**ItemsView**(*MappingView, AbstractSet[tuple[KT_co, VT_co]]*)

Deprecated alias to *collections.abc.ItemsView*.

Deprecated since version 3.9: *collections.abc.ItemsView* now supports subscripting (`[]`). See **PEP 585** and *Generic Alias Type*.

class typing.**KeysView**(*MappingView, AbstractSet[KT_co]*)

Deprecated alias to *collections.abc.KeysView*.

Deprecated since version 3.9: *collections.abc.KeysView* now supports subscripting (`[]`). See **PEP 585** and *Generic Alias Type*.

class typing.**Mapping**(*Collection[KT], Generic[KT, VT_co]*)

Deprecated alias to *collections.abc.Mapping*.

This type can be used as follows:

```
def get_position_in_index(word_list: Mapping[str, int], word: str) -> int:
    return word_list[word]
```

Deprecated since version 3.9: *collections.abc.Mapping* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

class typing.**MappingView**(*Sized*)

Deprecated alias to *collections.abc.MappingView*.

Deprecated since version 3.9: *collections.abc.MappingView* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

class typing.**MutableMapping**(*Mapping[KT, VT]*)

Deprecated alias to *collections.abc.MutableMapping*.

Deprecated since version 3.9: *collections.abc.MutableMapping* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

class typing.**MutableSequence**(*Sequence[T]*)

Deprecated alias to *collections.abc.MutableSequence*.

Deprecated since version 3.9: *collections.abc.MutableSequence* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

class typing.**MutableSet**(*AbstractSet[T]*)

Deprecated alias to *collections.abc.MutableSet*.

Deprecated since version 3.9: *collections.abc.MutableSet* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

class typing.**Sequence**(*Reversible[T_co], Collection[T_co]*)

Deprecated alias to *collections.abc.Sequence*.

Deprecated since version 3.9: *collections.abc.Sequence* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

class typing.**ValuesView**(*MappingView, Collection[_VT_co]*)

Deprecated alias to *collections.abc.ValuesView*.

Deprecated since version 3.9: *collections.abc.ValuesView* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

Aliases to asynchronous ABCs in collections.abc

class typing.**Coroutine**(*Awaitable[ReturnType], Generic[YieldType, SendType, ReturnType]*)

Deprecated alias to *collections.abc.Coroutine*.

The variance and order of type variables correspond to those of *Generator*, for example:

```
from collections.abc import Coroutine
c: Coroutine[list[str], str, int]  # Some coroutine defined elsewhere
x = c.send('hi')                   # Inferred type of 'x' is list[str]
async def bar() -> None:
    y = await c                    # Inferred type of 'y' is int
```

Added in version 3.5.3.

Deprecated since version 3.9: *collections.abc.Coroutine* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

class typing.**AsyncGenerator**(*AsyncIterator[YieldType], Generic[YieldType, SendType]*)

Deprecated alias to *collections.abc.AsyncGenerator*.

An async generator can be annotated by the generic type `AsyncGenerator[YieldType, SendType]`. For example:

```
async def echo_round() -> AsyncGenerator[int, float]:
    sent = yield 0
    while sent >= 0.0:
        rounded = await round(sent)
        sent = yield rounded
```

Unlike normal generators, async generators cannot return a value, so there is no `ReturnType` type parameter. As with *Generator*, the `SendType` behaves contravariantly.

The `SendType` defaults to `None`:

```
async def infinite_stream(start: int) -> AsyncGenerator[int]:
    while True:
        yield start
        start = await increment(start)
```

It is also possible to set this type explicitly:

```
async def infinite_stream(start: int) -> AsyncGenerator[int, None]:
    while True:
        yield start
        start = await increment(start)
```

Alternatively, annotate your generator as having a return type of either `AsyncIterable[YieldType]` or `AsyncIterator[YieldType]`:

```
async def infinite_stream(start: int) -> AsyncIterator[int]:
    while True:
        yield start
        start = await increment(start)
```

Added in version 3.6.1.

Deprecated since version 3.9: *collections.abc.AsyncGenerator* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

Changed in version 3.13: The `SendType` parameter now has a default.

class typing.**AsyncIterable**(*Generic[T_co]*)

Deprecated alias to *collections.abc.AsyncIterable*.

Added in version 3.5.2.

Deprecated since version 3.9: *collections.abc.AsyncIterable* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

class typing.**AsyncIterator**(*AsyncIterable[T_co]*)

Deprecated alias to *collections.abc.AsyncIterator*.

Added in version 3.5.2.

Deprecated since version 3.9: `collections.abc.AsyncIterator` now supports subscripting (`[]`). See **PEP 585** and *Generic Alias Type*.

class typing.**Awaitable**(*Generic[T_co]*)

Deprecated alias to `collections.abc.Awaitable`.

Added in version 3.5.2.

Deprecated since version 3.9: `collections.abc.Awaitable` now supports subscripting (`[]`). See **PEP 585** and *Generic Alias Type*.

Aliases to other ABCs in `collections.abc`

class typing.**Iterable**(*Generic[T_co]*)

Deprecated alias to `collections.abc.Iterable`.

Deprecated since version 3.9: `collections.abc.Iterable` now supports subscripting (`[]`). See **PEP 585** and *Generic Alias Type*.

class typing.**Iterator**(*Iterable[T_co]*)

Deprecated alias to `collections.abc.Iterator`.

Deprecated since version 3.9: `collections.abc.Iterator` now supports subscripting (`[]`). See **PEP 585** and *Generic Alias Type*.

typing.**Callable**

Deprecated alias to `collections.abc.Callable`.

See *Annotating callable objects* for details on how to use `collections.abc.Callable` and `typing.Callable` in type annotations.

Deprecated since version 3.9: `collections.abc.Callable` now supports subscripting (`[]`). See **PEP 585** and *Generic Alias Type*.

Changed in version 3.10: `Callable` now supports `ParamSpec` and `Concatenate`. See **PEP 612** for more details.

class typing.**Generator**(*Iterator[YieldType], Generic[YieldType, SendType, ReturnType]*)

Deprecated alias to `collections.abc.Generator`.

A generator can be annotated by the generic type `Generator[YieldType, SendType, ReturnType]`. For example:

```
def echo_round() -> Generator[int, float, str]:
    sent = yield 0
    while sent >= 0:
        sent = yield round(sent)
    return 'Done'
```

Note that unlike many other generics in the typing module, the `SendType` of `Generator` behaves contravariantly, not covariantly or invariantly.

The `SendType` and `ReturnType` parameters default to `None`:

```
def infinite_stream(start: int) -> Generator[int]:
    while True:
        yield start
        start += 1
```

It is also possible to set these types explicitly:

```
def infinite_stream(start: int) -> Generator[int, None, None]:
    while True:
        yield start
        start += 1
```

Alternatively, annotate your generator as having a return type of either `Iterable[YieldType]` or `Iterator[YieldType]`:

```
def infinite_stream(start: int) -> Iterator[int]:
    while True:
        yield start
        start += 1
```

Deprecated since version 3.9: *collections.abc.Generator* now supports subscripting (`[]`). See **PEP 585** and *Generic Alias Type*.

Changed in version 3.13: Default values for the send and return types were added.

class typing.**Hashable**

Deprecated alias to *collections.abc.Hashable*.

Deprecated since version 3.12: Use *collections.abc.Hashable* directly instead.

class typing.**Reversible**(*Iterable[T_co]*)

Deprecated alias to *collections.abc.Reversible*.

Deprecated since version 3.9: *collections.abc.Reversible* now supports subscripting (`[]`). See **PEP 585** and *Generic Alias Type*.

class typing.**Sized**

Deprecated alias to *collections.abc.Sized*.

Deprecated since version 3.12: Use *collections.abc.Sized* directly instead.

Aliases to `contextlib` ABCs

class typing.**ContextManager**(*Generic[T_co, ExitT_co]*)

Deprecated alias to *contextlib.AbstractContextManager*.

The first type parameter, `T_co`, represents the type returned by the `__enter__()` method. The optional second type parameter, `ExitT_co`, which defaults to `bool | None`, represents the type returned by the `__exit__()` method.

Added in version 3.5.4.

Deprecated since version 3.9: *contextlib.AbstractContextManager* now supports subscripting (`[]`). See **PEP 585** and *Generic Alias Type*.

Changed in version 3.13: Added the optional second type parameter, `ExitT_co`.

class typing.**AsyncContextManager**(*Generic[T_co, AExitT_co]*)

Deprecated alias to *contextlib.AbstractAsyncContextManager*.

The first type parameter, `T_co`, represents the type returned by the `__aenter__()` method. The optional second type parameter, `AExitT_co`, which defaults to `bool | None`, represents the type returned by the `__aexit__()` method.

Added in version 3.6.2.

Deprecated since version 3.9: *contextlib.AbstractAsyncContextManager* now supports subscripting ([]). See **PEP 585** and *Generic Alias Type*.

Changed in version 3.13: Added the optional second type parameter, AExitT_co.

26.1.12 Deprecation Timeline of Major Features

Certain features in typing are deprecated and may be removed in a future version of Python. The following table summarizes major deprecations for your convenience. This is subject to change, and not all deprecations are listed.

Feature	Deprecated in	Projected removal	PEP/issue
typing versions of standard collections	3.9	Undecided (see *Deprecated aliases* for more information)	**PEP 585**
typing.ByteString	3.9	3.14	gh-91896
typing.Text	3.11	Undecided	gh-92332
typing.Hashable and *typing.Sized*	3.12	Undecided	gh-94309
typing.TypeAlias	3.12	Undecided	**PEP 695**
@typing.no_type_check_decorator	3.13	3.15	gh-106309
typing.AnyStr	3.13	3.18	gh-105578

26.2 pydoc — Documentation generator and online help system

Source code: Lib/pydoc.py

The pydoc module automatically generates documentation from Python modules. The documentation can be presented as pages of text on the console, served to a web browser, or saved to HTML files.

For modules, classes, functions and methods, the displayed documentation is derived from the docstring (i.e. the __doc__ attribute) of the object, and recursively of its documentable members. If there is no docstring, pydoc tries to obtain a description from the block of comment lines just above the definition of the class, function or method in the source file, or at the top of the module (see *inspect.getcomments()*).

The built-in function *help()* invokes the online help system in the interactive interpreter, which uses pydoc to generate its documentation as text on the console. The same text documentation can also be viewed from outside the Python interpreter by running **pydoc** as a script at the operating system's command prompt. For example, running

```
python -m pydoc sys
```

at a shell prompt will display documentation on the *sys* module, in a style similar to the manual pages shown by the Unix **man** command. The argument to **pydoc** can be the name of a function, module, or package, or a dotted reference to a class, method, or function within a module or module in a package. If the argument to **pydoc** looks like a path (that is, it contains the path separator for your operating system, such as a slash in Unix), and refers to an existing Python source file, then documentation is produced for that file.

Note

In order to find objects and their documentation, `pydoc` imports the module(s) to be documented. Therefore, any code on module level will be executed on that occasion. Use an `if __name__ == '__main__':` guard to only execute code when a file is invoked as a script and not just imported.

When printing output to the console, **pydoc** attempts to paginate the output for easier reading. If the `PAGER` environment variable is set, **pydoc** will use its value as a pagination program.

Specifying a `-w` flag before the argument will cause HTML documentation to be written out to a file in the current directory, instead of displaying text on the console.

Specifying a `-k` flag before the argument will search the synopsis lines of all available modules for the keyword given as the argument, again in a manner similar to the Unix **man** command. The synopsis line of a module is the first line of its documentation string.

You can also use **pydoc** to start an HTTP server on the local machine that will serve documentation to visiting web browsers. **python -m pydoc -p 1234** will start a HTTP server on port 1234, allowing you to browse the documentation at `http://localhost:1234/` in your preferred web browser. Specifying `0` as the port number will select an arbitrary unused port.

python -m pydoc -n <hostname> will start the server listening at the given hostname. By default the hostname is 'localhost' but if you want the server to be reached from other machines, you may want to change the host name that the server responds to. During development this is especially useful if you want to run pydoc from within a container.

python -m pydoc -b will start the server and additionally open a web browser to a module index page. Each served page has a navigation bar at the top where you can *Get* help on an individual item, *Search* all modules with a keyword in their synopsis line, and go to the *Module index*, *Topics* and *Keywords* pages.

When **pydoc** generates documentation, it uses the current environment and path to locate modules. Thus, invoking **pydoc spam** documents precisely the version of the module you would get if you started the Python interpreter and typed `import spam`.

Module docs for core modules are assumed to reside in `https://docs.python.org/X.Y/library/` where `X` and `Y` are the major and minor version numbers of the Python interpreter. This can be overridden by setting the `PYTHONDOCS` environment variable to a different URL or to a local directory containing the Library Reference Manual pages.

Changed in version 3.2: Added the `-b` option.

Changed in version 3.3: The `-g` command line option was removed.

Changed in version 3.4: `pydoc` now uses *inspect.signature()* rather than *inspect.getfullargspec()* to extract signature information from callables.

Changed in version 3.7: Added the `-n` option.

26.3 Python Development Mode

Added in version 3.7.

The Python Development Mode introduces additional runtime checks that are too expensive to be enabled by default. It should not be more verbose than the default if the code is correct; new warnings are only emitted when an issue is detected.

It can be enabled using the `-X dev` command line option or by setting the `PYTHONDEVMODE` environment variable to `1`.

See also Python debug build.

26.3.1 Effects of the Python Development Mode

Enabling the Python Development Mode is similar to the following command, but with additional effects described below:

```
PYTHONMALLOC=debug PYTHONASYNCIODEBUG=1 python -W default -X faulthandler
```

Effects of the Python Development Mode:

- Add `default` *warning filter*. The following warnings are shown:
  - *DeprecationWarning*
  - *ImportWarning*
  - *PendingDeprecationWarning*
  - *ResourceWarning*

  Normally, the above warnings are filtered by the default *warning filters*.

  It behaves as if the `-W default` command line option is used.

  Use the `-W error` command line option or set the `PYTHONWARNINGS` environment variable to `error` to treat warnings as errors.
- Install debug hooks on memory allocators to check for:
  - Buffer underflow
  - Buffer overflow
  - Memory allocator API violation
  - Unsafe usage of the GIL

  See the `PyMem_SetupDebugHooks()` C function.

  It behaves as if the `PYTHONMALLOC` environment variable is set to `debug`.

  To enable the Python Development Mode without installing debug hooks on memory allocators, set the `PYTHONMALLOC` environment variable to `default`.
- Call *faulthandler.enable()* at Python startup to install handlers for the *SIGSEGV*, *SIGFPE*, *SIGABRT*, *SIGBUS* and *SIGILL* signals to dump the Python traceback on a crash.

  It behaves as if the `-X faulthandler` command line option is used or if the `PYTHONFAULTHANDLER` environment variable is set to `1`.
- Enable *asyncio debug mode*. For example, *asyncio* checks for coroutines that were not awaited and logs them.

  It behaves as if the `PYTHONASYNCIODEBUG` environment variable is set to `1`.
- Check the *encoding* and *errors* arguments for string encoding and decoding operations. Examples: *open()*, *str.encode()* and *bytes.decode()*.

  By default, for best performance, the *errors* argument is only checked at the first encoding/decoding error and the *encoding* argument is sometimes ignored for empty strings.
- The *io.IOBase* destructor logs `close()` exceptions.
- Set the *dev_mode* attribute of *sys.flags* to `True`.

The Python Development Mode does not enable the *tracemalloc* module by default, because the overhead cost (to performance and memory) would be too large. Enabling the *tracemalloc* module provides additional information on the origin of some errors. For example, *ResourceWarning* logs the traceback where the resource was allocated, and a buffer overflow error logs the traceback where the memory block was allocated.

The Python Development Mode does not prevent the `-O` command line option from removing `assert` statements nor from setting *__debug__* to `False`.

The Python Development Mode can only be enabled at the Python startup. Its value can be read from *sys.flags.dev_mode*.

Changed in version 3.8: The *io.IOBase* destructor now logs `close()` exceptions.

Changed in version 3.9: The *encoding* and *errors* arguments are now checked for string encoding and decoding operations.

26.3.2 ResourceWarning Example

Example of a script counting the number of lines of the text file specified in the command line:

```
import sys

def main():
    fp = open(sys.argv[1])
    nlines = len(fp.readlines())
    print(nlines)
    # The file is closed implicitly

if __name__ == "__main__":
    main()
```

The script does not close the file explicitly. By default, Python does not emit any warning. Example using README.txt, which has 269 lines:

```
$ python script.py README.txt
269
```

Enabling the Python Development Mode displays a *ResourceWarning* warning:

```
$ python -X dev script.py README.txt
269
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='README.rst'␣
↪mode='r' encoding='UTF-8'>
  main()
ResourceWarning: Enable tracemalloc to get the object allocation traceback
```

In addition, enabling *tracemalloc* shows the line where the file was opened:

```
$ python -X dev -X tracemalloc=5 script.py README.rst
269
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='README.rst'␣
↪mode='r' encoding='UTF-8'>
  main()
Object allocated at (most recent call last):
  File "script.py", lineno 10
    main()
  File "script.py", lineno 4
    fp = open(sys.argv[1])
```

The fix is to close explicitly the file. Example using a context manager:

```
def main():
    # Close the file explicitly when exiting the with block
    with open(sys.argv[1]) as fp:
        nlines = len(fp.readlines())
    print(nlines)
```

Not closing a resource explicitly can leave a resource open for way longer than expected; it can cause severe issues upon exiting Python. It is bad in CPython, but it is even worse in PyPy. Closing resources explicitly makes an application more deterministic and more reliable.

26.3.3 Bad file descriptor error example

Script displaying the first line of itself:

```
import os

def main():
    fp = open(__file__)
    firstline = fp.readline()
    print(firstline.rstrip())
    os.close(fp.fileno())
    # The file is closed implicitly

main()
```

By default, Python does not emit any warning:

```
$ python script.py
import os
```

The Python Development Mode shows a *ResourceWarning* and logs a "Bad file descriptor" error when finalizing the file object:

```
$ python -X dev script.py
import os
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='script.py' mode=
↪'r' encoding='UTF-8'>
  main()
ResourceWarning: Enable tracemalloc to get the object allocation traceback
Exception ignored in: <_io.TextIOWrapper name='script.py' mode='r' encoding='UTF-8'>
Traceback (most recent call last):
  File "script.py", line 10, in <module>
    main()
OSError: [Errno 9] Bad file descriptor
```

`os.close(fp.fileno())` closes the file descriptor. When the file object finalizer tries to close the file descriptor again, it fails with the `Bad file descriptor` error. A file descriptor must be closed only once. In the worst case scenario, closing it twice can lead to a crash (see bpo-18748 for an example).

The fix is to remove the `os.close(fp.fileno())` line, or open the file with `closefd=False`.

26.4 doctest — Test interactive Python examples

Source code: Lib/doctest.py

The *doctest* module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown. There are several common ways to use doctest:

- To check that a module's docstrings are up-to-date by verifying that all interactive examples still work as documented.
- To perform regression testing by verifying that interactive examples from a test file or a test object work as expected.
- To write tutorial documentation for a package, liberally illustrated with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of "literate testing" or "executable documentation".

Here's a complete but small example module:

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
```

(continues on next page)

(continued from previous page)

```
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

If you run `example.py` directly from the command line, *doctest* works its magic:

```
$ python example.py
$
```

There's no output! That's normal, and it means all the examples worked. Pass `-v` to the script, and *doctest* prints a detailed log of what it's trying, and prints a summary at the end:

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

And so on, eventually ending with:

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
2 items passed all tests:
   1 test in __main__
   6 tests in __main__.factorial
7 tests in 2 items.
7 passed.
Test passed.
$
```

That's all you need to know to start making productive use of *doctest*! Jump in. The following sections provide full details. Note that there are many examples of doctests in the standard Python test suite and libraries. Especially useful examples can be found in the standard test file `Lib/test/test_doctest/test_doctest.py`.

26.4.1 Simple Usage: Checking Examples in Docstrings

The simplest way to start using doctest (but not necessarily the way you'll continue to do it) is to end each module `M` with:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

`doctest` then examines docstrings in module `M`.

Running the module as a script causes the examples in the docstrings to get executed and verified:

```
python M.py
```

This won't display anything unless an example fails, in which case the failing example(s) and the cause(s) of the failure(s) are printed to stdout, and the final line of output is `***Test Failed*** N failures.`, where *N* is the number of examples that failed.

Run it with the `-v` switch instead:

```
python M.py -v
```

and a detailed report of all examples tried is printed to standard output, along with assorted summaries at the end.

You can force verbose mode by passing `verbose=True` to *`testmod()`*, or prohibit it by passing `verbose=False`. In either of those cases, `sys.argv` is not examined by *`testmod()`* (so passing `-v` or not has no effect).

There is also a command line shortcut for running *`testmod()`*. You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the module name(s) on the command line:

```
python -m doctest -v example.py
```

This will import `example.py` as a standalone module and run *`testmod()`* on it. Note that this may not work correctly if the file is part of a package and imports other submodules from that package.

For more information on *`testmod()`*, see section *Basic API*.

26.4.2 Simple Usage: Checking Examples in a Text File

Another simple application of doctest is testing interactive examples in a text file. This can be done with the *`testfile()`* function:

```
import doctest
doctest.testfile("example.txt")
```

That short script executes and verifies any interactive Python examples contained in the file `example.txt`. The file content is treated as if it were a single giant docstring; the file doesn't need to contain a Python program! For example, perhaps `example.txt` contains this:

```
The ``example`` module
======================

Using ``factorial``
-------------------

This is an example text file in reStructuredText format.  First import
``factorial`` from the ``example`` module:
```

(continues on next page)

(continued from previous page)

```
    >>> from example import factorial

Now use it:

    >>> factorial(6)
    120
```

Running `doctest.testfile("example.txt")` then finds the error in this documentation:

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

As with *testmod()*, *testfile()* won't display anything unless an example fails. If an example does fail, then the failing example(s) and the cause(s) of the failure(s) are printed to stdout, using the same format as *testmod()*.

By default, *testfile()* looks for files in the calling module's directory. See section *Basic API* for a description of the optional arguments that can be used to tell it to look for files in other locations.

Like *testmod()*, *testfile()*'s verbosity can be set with the `-v` command-line switch or with the optional keyword argument *verbose*.

There is also a command line shortcut for running *testfile()*. You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the file name(s) on the command line:

```
python -m doctest -v example.txt
```

Because the file name does not end with `.py`, *doctest* infers that it must be run with *testfile()*, not *testmod()*.

For more information on *testfile()*, see section *Basic API*.

26.4.3 How It Works

This section examines in detail how doctest works: which docstrings it looks at, how it finds interactive examples, what execution context it uses, how it handles exceptions, and how option flags can be used to control its behavior. This is the information that you need to know to write doctest examples; for information about actually running doctest on these examples, see the following sections.

Which Docstrings Are Examined?

The module docstring, and all function, class and method docstrings are searched. Objects imported into the module are not searched.

In addition, there are cases when you want tests to be part of a module but not part of the help text, which requires that the tests not be included in the docstring. Doctest looks for a module-level variable called `__test__` and uses it to locate other tests. If `M.__test__` exists, it must be a dict, and each entry maps a (string) name to a function object, class object, or string. Function and class object docstrings found from `M.__test__` are searched, and strings are treated as if they were docstrings. In output, a key `K` in `M.__test__` appears with name `M.__test__.K`.

For example, place this block of code at the top of `example.py`:

```
__test__ = {
    'numbers': """
>>> factorial(6)
720

>>> [factorial(n) for n in range(6)]
[1, 1, 2, 6, 24, 120]
"""
}
```

The value of `example.__test__["numbers"]` will be treated as a docstring and all the tests inside it will be run. It is important to note that the value can be mapped to a function, class object, or module; if so, `doctest` searches them recursively for docstrings, which are then scanned for tests.

Any classes found are recursively searched similarly, to test docstrings in their contained methods and nested classes.

How are Docstring Examples Recognized?

In most cases a copy-and-paste of an interactive console session works fine, but doctest isn't trying to do an exact emulation of any specific Python shell.

```
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print("yes")
... else:
...     print("no")
...     print("NO")
...     print("NO!!!")
...
no
NO
NO!!!
>>>
```

Any expected output must immediately follow the final `'>>> '` or `'... '` line containing the code, and the expected output (if any) extends to the next `'>>> '` or all-whitespace line.

The fine print:

- Expected output cannot contain an all-whitespace line, since such a line is taken to signal the end of expected output. If expected output does contain a blank line, put `<BLANKLINE>` in your doctest example each place a blank line is expected.
- All hard tab characters are expanded to spaces, using 8-column tab stops. Tabs in output generated by the tested code are not modified. Because any hard tabs in the sample output *are* expanded, this means that if the code output includes hard tabs, the only way the doctest can pass is if the *NORMALIZE_WHITESPACE* option or *directive* is in effect. Alternatively, the test can be rewritten to capture the output and compare it to an expected value as part of the test. This handling of tabs in the source was arrived at through trial and error, and has proven to be the least error prone way of handling them. It is possible to use a different algorithm for handling tabs by writing a custom *DocTestParser* class.
- Output to stdout is captured, but not output to stderr (exception tracebacks are captured via a different means).
- If you continue a line via backslashing in an interactive session, or for any other reason use a backslash, you should use a raw docstring, which will preserve your backslashes exactly as you type them:

```
>>> def f(x):
...     r'''Backslashes in a raw docstring: m\n'''
...
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

Otherwise, the backslash will be interpreted as part of the string. For example, the `\n` above would be interpreted as a newline character. Alternatively, you can double each backslash in the doctest version (and not use a raw string):

```
>>> def f(x):
...     '''Backslashes in a raw docstring: m\\n'''
...
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

- The starting column doesn't matter:

```
>>> assert "Easy!"
      >>> import math
          >>> math.floor(1.9)
          1
```

and as many leading whitespace characters are stripped from the expected output as appeared in the initial `'>>> '` line that started the example.

What's the Execution Context?

By default, each time *doctest* finds a docstring to test, it uses a *shallow copy* of `M`'s globals, so that running tests doesn't change the module's real globals, and so that one test in `M` can't leave behind crumbs that accidentally allow another test to work. This means examples can freely use any names defined at top-level in `M`, and names defined earlier in the docstring being run. Examples cannot see names defined in other docstrings.

You can force use of your own dict as the execution context by passing `globs=your_dict` to *testmod()* or *testfile()* instead.

What About Exceptions?

No problem, provided that the traceback is the only output produced by the example: just paste in the traceback.[1] Since tracebacks contain details that are likely to change rapidly (for example, exact file paths and line numbers), this is one case where doctest works hard to be flexible in what it accepts.

Simple example:

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

That doctest succeeds if *ValueError* is raised, with the `list.remove(x): x not in list` detail as shown.

The expected output for an exception must start with a traceback header, which may be either of the following two lines, indented the same as the first line of the example:

[1] Examples containing both expected output and an exception are not supported. Trying to guess where one ends and the other begins is too error-prone, and that also makes for a confusing test.

```
Traceback (most recent call last):
Traceback (innermost last):
```

The traceback header is followed by an optional traceback stack, whose contents are ignored by doctest. The traceback stack is typically omitted, or copied verbatim from an interactive session.

The traceback stack is followed by the most interesting part: the line(s) containing the exception type and detail. This is usually the last line of a traceback, but can extend across multiple lines if the exception has a multi-line detail:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: multi
    line
detail
```

The last three lines (starting with *ValueError*) are compared against the exception's type and detail, and the rest are ignored.

Best practice is to omit the traceback stack, unless it adds significant documentation value to the example. So the last example is probably better as:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
    ...
ValueError: multi
    line
detail
```

Note that tracebacks are treated very specially. In particular, in the rewritten example, the use of `...` is independent of doctest's *ELLIPSIS* option. The ellipsis in that example could be left out, or could just as well be three (or three hundred) commas or digits, or an indented transcript of a Monty Python skit.

Some details you should read once, but won't need to remember:

- Doctest can't guess whether your expected output came from an exception traceback or from ordinary printing. So, e.g., an example that expects `ValueError: 42 is prime` will pass whether *ValueError* is actually raised or if the example merely prints that traceback text. In practice, ordinary output rarely begins with a traceback header line, so this doesn't create real problems.
- Each line of the traceback stack (if present) must be indented further than the first line of the example, *or* start with a non-alphanumeric character. The first line following the traceback header indented the same and starting with an alphanumeric is taken to be the start of the exception detail. Of course this does the right thing for genuine tracebacks.
- When the *IGNORE_EXCEPTION_DETAIL* doctest option is specified, everything following the leftmost colon and any module information in the exception name is ignored.
- The interactive shell omits the traceback header line for some *SyntaxError*s. But doctest uses the traceback header line to distinguish exceptions from non-exceptions. So in the rare case where you need to test a *SyntaxError* that omits the traceback header, you will need to manually add the traceback header line to your test example.
- For some exceptions, Python displays the position of the error using ^ markers and tildes:

```
>>> 1 + None
  File "<stdin>", line 1
    1 + None
```

(continues on next page)

(continued from previous page)

```
    ~~^~~~~
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

Since the lines showing the position of the error come before the exception type and detail, they are not checked by doctest. For example, the following test would pass, even though it puts the ^ marker in the wrong location:

```
>>> 1 + None
  File "<stdin>", line 1
    1 + None
    ^~~~~~~
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

Option Flags

A number of option flags control various aspects of doctest's behavior. Symbolic names for the flags are supplied as module constants, which can be bitwise ORed together and passed to various functions. The names can also be used in *doctest directives*, and may be passed to the doctest command line interface via the `-o` option.

Added in version 3.4: The `-o` command line option.

The first group of options define test semantics, controlling aspects of how doctest decides whether actual output matches an example's expected output:

`doctest.`**`DONT_ACCEPT_TRUE_FOR_1`**

By default, if an expected output block contains just `1`, an actual output block containing just `1` or just `True` is considered to be a match, and similarly for `0` versus `False`. When *`DONT_ACCEPT_TRUE_FOR_1`* is specified, neither substitution is allowed. The default behavior caters to that Python changed the return type of many functions from integer to boolean; doctests expecting "little integer" output still work in these cases. This option will probably go away, but not for several years.

`doctest.`**`DONT_ACCEPT_BLANKLINE`**

By default, if an expected output block contains a line containing only the string `<BLANKLINE>`, then that line will match a blank line in the actual output. Because a genuinely blank line delimits the expected output, this is the only way to communicate that a blank line is expected. When *`DONT_ACCEPT_BLANKLINE`* is specified, this substitution is not allowed.

`doctest.`**`NORMALIZE_WHITESPACE`**

When specified, all sequences of whitespace (blanks and newlines) are treated as equal. Any sequence of whitespace within the expected output will match any sequence of whitespace within the actual output. By default, whitespace must match exactly. *`NORMALIZE_WHITESPACE`* is especially useful when a line of expected output is very long, and you want to wrap it across multiple lines in your source.

`doctest.`**`ELLIPSIS`**

When specified, an ellipsis marker (`...`) in the expected output can match any substring in the actual output. This includes substrings that span line boundaries, and empty substrings, so it's best to keep usage of this simple. Complicated uses can lead to the same kinds of "oops, it matched too much!" surprises that `.*` is prone to in regular expressions.

`doctest.`**`IGNORE_EXCEPTION_DETAIL`**

When specified, doctests expecting exceptions pass so long as an exception of the expected type is raised, even if the details (message and fully qualified exception name) don't match.

For example, an example expecting `ValueError: 42` will pass if the actual exception raised is `ValueError: 3*14`, but will fail if, say, a *`TypeError`* is raised instead. It will also ignore any fully qualified name included

before the exception class, which can vary between implementations and versions of Python and the code/libraries in use. Hence, all three of these variations will work with the flag specified:

```
>>> raise Exception('message')
Traceback (most recent call last):
Exception: message

>>> raise Exception('message')
Traceback (most recent call last):
builtins.Exception: message

>>> raise Exception('message')
Traceback (most recent call last):
__main__.Exception: message
```

Note that *ELLIPSIS* can also be used to ignore the details of the exception message, but such a test may still fail based on whether the module name is present or matches exactly.

Changed in version 3.2: *IGNORE_EXCEPTION_DETAIL* now also ignores any information relating to the module containing the exception under test.

doctest.**SKIP**

When specified, do not run the example at all. This can be useful in contexts where doctest examples serve as both documentation and test cases, and an example should be included for documentation purposes, but should not be checked. E.g., the example's output might be random; or the example might depend on resources which would be unavailable to the test driver.

The SKIP flag can also be used for temporarily "commenting out" examples.

doctest.**COMPARISON_FLAGS**

A bitmask or'ing together all the comparison flags above.

The second group of options controls how test failures are reported:

doctest.**REPORT_UDIFF**

When specified, failures that involve multi-line expected and actual outputs are displayed using a unified diff.

doctest.**REPORT_CDIFF**

When specified, failures that involve multi-line expected and actual outputs will be displayed using a context diff.

doctest.**REPORT_NDIFF**

When specified, differences are computed by `difflib.Differ`, using the same algorithm as the popular `ndiff.py` utility. This is the only method that marks differences within lines as well as across lines. For example, if a line of expected output contains digit `1` where actual output contains letter `l`, a line is inserted with a caret marking the mismatching column positions.

doctest.**REPORT_ONLY_FIRST_FAILURE**

When specified, display the first failing example in each doctest, but suppress output for all remaining examples. This will prevent doctest from reporting correct examples that break because of earlier failures; but it might also hide incorrect examples that fail independently of the first failure. When *REPORT_ONLY_FIRST_FAILURE* is specified, the remaining examples are still run, and still count towards the total number of failures reported; only the output is suppressed.

doctest.**FAIL_FAST**

When specified, exit after the first failing example and don't attempt to run the remaining examples. Thus, the number of failures reported will be at most 1. This flag may be useful during debugging, since examples after the first failure won't even produce debugging output.

The doctest command line accepts the option `-f` as a shorthand for `-o FAIL_FAST`.

Added in version 3.4.

doctest.**REPORTING_FLAGS**

A bitmask or'ing together all the reporting flags above.

There is also a way to register new option flag names, though this isn't useful unless you intend to extend *doctest* internals via subclassing:

doctest.**register_optionflag**(*name*)

Create a new option flag with a given name, and return the new flag's integer value. *register_optionflag()* can be used when subclassing *OutputChecker* or *DocTestRunner* to create new options that are supported by your subclasses. *register_optionflag()* should always be called using the following idiom:

```
MY_FLAG = register_optionflag('MY_FLAG')
```

Directives

Doctest directives may be used to modify the *option flags* for an individual example. Doctest directives are special Python comments following an example's source code:

```
directive             ::=   "#" "doctest:" directive_options
directive_options     ::=   directive_option ("," directive_option)*
directive_option      ::=   on_or_off directive_option_name
on_or_off             ::=   "+" | "-"
directive_option_name ::=   "DONT_ACCEPT_BLANKLINE" | "NORMALIZE_WHITESPACE" | ...
```

Whitespace is not allowed between the + or − and the directive option name. The directive option name can be any of the option flag names explained above.

An example's doctest directives modify doctest's behavior for that single example. Use + to enable the named behavior, or − to disable it.

For example, this test passes:

```
>>> print(list(range(20)))   # doctest: +NORMALIZE_WHITESPACE
[0,   1,  2,  3,  4,  5,  6,  7,  8,  9,
10,  11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Without the directive it would fail, both because the actual output doesn't have two blanks before the single-digit list elements, and because the actual output is on a single line. This test also passes, and also requires a directive to do so:

```
>>> print(list(range(20)))   # doctest: +ELLIPSIS
[0, 1, ..., 18, 19]
```

Multiple directives can be used on a single physical line, separated by commas:

```
>>> print(list(range(20)))   # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

If multiple directive comments are used for a single example, then they are combined:

```
>>> print(list(range(20)))   # doctest: +ELLIPSIS
...                          # doctest: +NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

As the previous example shows, you can add `...` lines to your example containing only directives. This can be useful when an example is too long for a directive to comfortably fit on the same line:

```
>>> print(list(range(5)) + list(range(10, 20)) + list(range(30, 40)))
... # doctest: +ELLIPSIS
[0, ..., 4, 10, ..., 19, 30, ..., 39]
```

Note that since all options are disabled by default, and directives apply only to the example they appear in, enabling options (via + in a directive) is usually the only meaningful choice. However, option flags can also be passed to functions that run doctests, establishing different defaults. In such cases, disabling an option via − in a directive can be useful.

Warnings

doctest is serious about requiring exact matches in expected output. If even a single character doesn't match, the test fails. This will probably surprise you a few times, as you learn exactly what Python does and doesn't guarantee about output. For example, when printing a set, Python doesn't guarantee that the element is printed in any particular order, so a test like

```
>>> foo()
{"spam", "eggs"}
```

is vulnerable! One workaround is to do

```
>>> foo() == {"spam", "eggs"}
True
```

instead. Another is to do

```
>>> d = sorted(foo())
>>> d
['eggs', 'spam']
```

There are others, but you get the idea.

Another bad idea is to print things that embed an object address, like

```
>>> id(1.0)   # certain to fail some of the time
7948648
>>> class C: pass
>>> C()  # the default repr() for instances embeds an address
<C object at 0x00AC18F0>
```

The *ELLIPSIS* directive gives a nice approach for the last example:

```
>>> C()  # doctest: +ELLIPSIS
<C object at 0x...>
```

Floating-point numbers are also subject to small output variations across platforms, because Python defers to the platform C library for float formatting, and C libraries vary widely in quality here.

```
>>> 1./7  # risky
0.14285714285714285
>>> print(1./7) # safer
0.142857142857
>>> print(round(1./7, 6)) # much safer
0.142857
```

Numbers of the form `I/2.**J` are safe across all platforms, and I often contrive doctest examples to produce numbers of that form:

```
>>> 3./4  # utterly safe
0.75
```

Simple fractions are also easier for people to understand, and that makes for better documentation.

26.4.4 Basic API

The functions *testmod()* and *testfile()* provide a simple interface to doctest that should be sufficient for most basic uses. For a less formal introduction to these two functions, see sections *Simple Usage: Checking Examples in Docstrings* and *Simple Usage: Checking Examples in a Text File*.

doctest.**testfile**(*filename*, *module_relative=True*, *name=None*, *package=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *parser=DocTestParser()*, *encoding=None*)

All arguments except *filename* are optional, and should be specified in keyword form.

Test examples in the file named *filename*. Return `(failure_count, test_count)`.

Optional argument *module_relative* specifies how the filename should be interpreted:

- If *module_relative* is `True` (the default), then *filename* specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the *package* argument is specified, then it is relative to that package. To ensure OS-independence, *filename* should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If *module_relative* is `False`, then *filename* specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument *name* gives the name of the test; by default, or if `None`, `os.path.basename(filename)` is used.

Optional argument *package* is a Python package or the name of a Python package whose directory should be used as the base directory for a module-relative filename. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify *package* if *module_relative* is `False`.

Optional argument *globs* gives a dict to be used as the globals when executing examples. A new shallow copy of this dict is created for the doctest, so its examples start with a clean slate. By default, or if `None`, a new empty dict is used.

Optional argument *extraglobs* gives a dict merged into the globals used to execute examples. This works like *dict.update()*: if *globs* and *extraglobs* have a common key, the associated value in *extraglobs* appears in the combined dict. By default, or if `None`, no extra globals are used. This is an advanced feature that allows parameterization of doctests. For example, a doctest can be written for a base class, using a generic name for the class, then reused to test any number of subclasses by passing an *extraglobs* dict mapping the generic name to the subclass to be tested.

Optional argument *verbose* prints lots of stuff if true, and prints only failures if false; by default, or if `None`, it's true if and only if `'-v'` is in `sys.argv`.

Optional argument *report* prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else the summary is very brief (in fact, empty if all tests passed).

Optional argument *optionflags* (default value 0) takes the bitwise OR of option flags. See section *Option Flags*.

Optional argument *raise_on_error* defaults to false. If true, an exception is raised upon the first failure or unexpected exception in an example. This allows failures to be post-mortem debugged. Default behavior is to continue running examples.

Optional argument *parser* specifies a `DocTestParser` (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

`doctest.`**`testmod`**(*m=None*, *name=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *exclude_empty=False*)

All arguments are optional, and all except for *m* should be specified in keyword form.

Test examples in docstrings in functions and classes reachable from module *m* (or module `__main__` if *m* is not supplied or is `None`), starting with `m.__doc__`.

Also test examples reachable from dict `m.__test__`, if it exists. `m.__test__` maps names (strings) to functions, classes and strings; function and class docstrings are searched for examples; strings are searched directly, as if they were docstrings.

Only docstrings attached to objects belonging to module *m* are searched.

Return `(failure_count, test_count)`.

Optional argument *name* gives the name of the module; by default, or if `None`, `m.__name__` is used.

Optional argument *exclude_empty* defaults to false. If true, objects for which no doctests are found are excluded from consideration. The default is a backward compatibility hack, so that code still using `doctest.master.summarize` in conjunction with `testmod()` continues to get output for objects with no tests. The *exclude_empty* argument to the newer `DocTestFinder` constructor defaults to true.

Optional arguments *extraglobs*, *verbose*, *report*, *optionflags*, *raise_on_error*, and *globs* are the same as for function `testfile()` above, except that *globs* defaults to `m.__dict__`.

`doctest.`**`run_docstring_examples`**(*f*, *globs*, *verbose=False*, *name='NoName'*, *compileflags=None*, *optionflags=0*)

Test examples associated with object *f*; for example, *f* may be a string, a module, a function, or a class object.

A shallow copy of dictionary argument *globs* is used for the execution context.

Optional argument *name* is used in failure messages, and defaults to `"NoName"`.

If optional argument *verbose* is true, output is generated even if there are no failures. By default, output is generated only in case of an example failure.

Optional argument *compileflags* gives the set of flags that should be used by the Python compiler when running the examples. By default, or if `None`, flags are deduced corresponding to the set of future features found in *globs*.

Optional argument *optionflags* works as for function `testfile()` above.

26.4.5 Unittest API

As your collection of doctest'ed modules grows, you'll want a way to run all their doctests systematically. `doctest` provides two functions that can be used to create `unittest` test suites from modules and text files containing doctests. To integrate with `unittest` test discovery, include a *load_tests* function in your test module:

```
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

There are two main functions for creating *unittest.TestSuite* instances from text files and modules with doctests:

doctest.**DocFileSuite**(**paths*, *module_relative=True*, *package=None*, *setUp=None*, *tearDown=None*, *globs=None*, *optionflags=0*, *parser=DocTestParser()*, *encoding=None*)

Convert doctest tests from one or more text files to a *unittest.TestSuite*.

The returned *unittest.TestSuite* is to be run by the unittest framework and runs the interactive examples in each file. If an example in any file fails, then the synthesized unit test fails, and a *failureException* exception is raised showing the name of the file containing the test and a (sometimes approximate) line number. If all the examples in a file are skipped, then the synthesized unit test is also marked as skipped.

Pass one or more paths (as strings) to text files to be examined.

Options may be provided as keyword arguments:

Optional argument *module_relative* specifies how the filenames in *paths* should be interpreted:

- If *module_relative* is `True` (the default), then each filename in *paths* specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the *package* argument is specified, then it is relative to that package. To ensure OS-independence, each filename should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If *module_relative* is `False`, then each filename in *paths* specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument *package* is a Python package or the name of a Python package whose directory should be used as the base directory for module-relative filenames in *paths*. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify *package* if *module_relative* is `False`.

Optional argument *setUp* specifies a set-up function for the test suite. This is called before running the tests in each file. The *setUp* function will be passed a *DocTest* object. The setUp function can access the test globals as the *globs* attribute of the test passed.

Optional argument *tearDown* specifies a tear-down function for the test suite. This is called after running the tests in each file. The *tearDown* function will be passed a *DocTest* object. The setUp function can access the test globals as the *globs* attribute of the test passed.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is a new empty dictionary.

Optional argument *optionflags* specifies the default doctest options for the tests, created by or-ing together individual option flags. See section *Option Flags*. See function *set_unittest_reportflags()* below for a better way to set reporting options.

Optional argument *parser* specifies a *DocTestParser* (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

The global `__file__` is added to the globals provided to doctests loaded from a text file using *DocFileSuite()*.

doctest.**DocTestSuite**(*module=None*, *globs=None*, *extraglobs=None*, *test_finder=None*, *setUp=None*, *tearDown=None*, *optionflags=0*, *checker=None*)

Convert doctest tests for a module to a *unittest.TestSuite*.

The returned *unittest.TestSuite* is to be run by the unittest framework and runs each doctest in the module. If any of the doctests fail, then the synthesized unit test fails, and a *failureException* exception is raised showing the name of the file containing the test and a (sometimes approximate) line number. If all the examples in a docstring are skipped, then the synthesized unit test is also marked as skipped.

Optional argument *module* provides the module to be tested. It can be a module object or a (possibly dotted) module name. If not specified, the module calling this function is used.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is a new empty dictionary.

Optional argument *extraglobs* specifies an extra set of global variables, which is merged into *globs*. By default, no extra globals are used.

Optional argument *test_finder* is the *DocTestFinder* object (or a drop-in replacement) that is used to extract doctests from the module.

Optional arguments *setUp*, *tearDown*, and *optionflags* are the same as for function *DocFileSuite()* above.

This function uses the same search technique as *testmod()*.

Changed in version 3.5: *DocTestSuite()* returns an empty *unittest.TestSuite* if *module* contains no docstrings instead of raising *ValueError*.

exception doctest.**failureException**

When doctests which have been converted to unit tests by *DocFileSuite()* or *DocTestSuite()* fail, this exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Under the covers, *DocTestSuite()* creates a *unittest.TestSuite* out of doctest.DocTestCase instances, and DocTestCase is a subclass of *unittest.TestCase*. DocTestCase isn't documented here (it's an internal detail), but studying its code can answer questions about the exact details of *unittest* integration.

Similarly, *DocFileSuite()* creates a *unittest.TestSuite* out of doctest.DocFileCase instances, and DocFileCase is a subclass of DocTestCase.

So both ways of creating a *unittest.TestSuite* run instances of DocTestCase. This is important for a subtle reason: when you run *doctest* functions yourself, you can control the *doctest* options in use directly, by passing option flags to *doctest* functions. However, if you're writing a *unittest* framework, *unittest* ultimately controls when and how tests get run. The framework author typically wants to control *doctest* reporting options (perhaps, e.g., specified by command line options), but there's no way to pass options through *unittest* to *doctest* test runners.

For this reason, *doctest* also supports a notion of *doctest* reporting flags specific to *unittest* support, via this function:

doctest.**set_unittest_reportflags**(*flags*)

Set the *doctest* reporting flags to use.

Argument *flags* takes the bitwise OR of option flags. See section *Option Flags*. Only "reporting flags" can be used.

This is a module-global setting, and affects all future doctests run by module *unittest*: the runTest() method of DocTestCase looks at the option flags specified for the test case when the DocTestCase instance was constructed. If no reporting flags were specified (which is the typical and expected case), doctest's *unittest* reporting flags are bitwise ORed into the option flags, and the option flags so augmented are passed to the *DocTestRunner* instance created to run the doctest. If any reporting flags were specified when the DocTestCase instance was constructed, doctest's *unittest* reporting flags are ignored.

The value of the *unittest* reporting flags in effect before the function was called is returned by the function.

26.4.6 Advanced API

The basic API is a simple wrapper that's intended to make doctest easy to use. It is fairly flexible, and should meet most users' needs; however, if you require more fine-grained control over testing, or wish to extend doctest's capabilities, then you should use the advanced API.

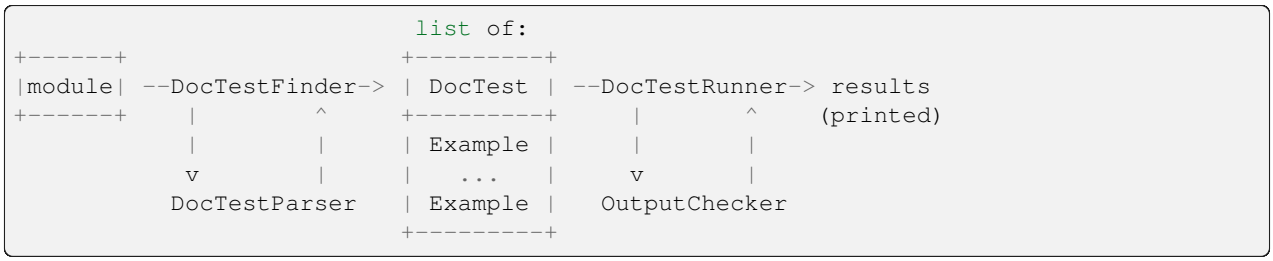
The advanced API revolves around two container classes, which are used to store the interactive examples extracted from doctest cases:

- `Example`: A single Python *statement*, paired with its expected output.
- `DocTest`: A collection of `Example`s, typically extracted from a single docstring or text file.

Additional processing classes are defined to find, parse, and run, and check doctest examples:

- `DocTestFinder`: Finds all docstrings in a given module, and uses a `DocTestParser` to create a `DocTest` from every docstring that contains interactive examples.
- `DocTestParser`: Creates a `DocTest` object from a string (such as an object's docstring).
- `DocTestRunner`: Executes the examples in a `DocTest`, and uses an `OutputChecker` to verify their output.
- `OutputChecker`: Compares the actual output from a doctest example with the expected output, and decides whether they match.

The relationships among these processing classes are summarized in the following diagram:



DocTest Objects

class doctest.**DocTest**(*examples*, *globs*, *name*, *filename*, *lineno*, *docstring*)

A collection of doctest examples that should be run in a single namespace. The constructor arguments are used to initialize the attributes of the same names.

`DocTest` defines the following attributes. They are initialized by the constructor, and should not be modified directly.

examples

A list of `Example` objects encoding the individual interactive Python examples that should be run by this test.

globs

The namespace (aka globals) that the examples should be run in. This is a dictionary mapping names to values. Any changes to the namespace made by the examples (such as binding new variables) will be reflected in `globs` after the test is run.

name

A string name identifying the `DocTest`. Typically, this is the name of the object or file that the test was extracted from.

filename

The name of the file that this *DocTest* was extracted from; or None if the filename is unknown, or if the *DocTest* was not extracted from a file.

lineno

The line number within *filename* where this *DocTest* begins, or None if the line number is unavailable. This line number is zero-based with respect to the beginning of the file.

docstring

The string that the test was extracted from, or None if the string is unavailable, or if the test was not extracted from a string.

Example Objects

class doctest.**Example**(*source*, *want*, *exc_msg=None*, *lineno=0*, *indent=0*, *options=None*)

A single interactive example, consisting of a Python statement and its expected output. The constructor arguments are used to initialize the attributes of the same names.

Example defines the following attributes. They are initialized by the constructor, and should not be modified directly.

source

A string containing the example's source code. This source code consists of a single Python statement, and always ends with a newline; the constructor adds a newline when necessary.

want

The expected output from running the example's source code (either from stdout, or a traceback in case of exception). *want* ends with a newline unless no output is expected, in which case it's an empty string. The constructor adds a newline when necessary.

exc_msg

The exception message generated by the example, if the example is expected to generate an exception; or None if it is not expected to generate an exception. This exception message is compared against the return value of *traceback.format_exception_only()*. *exc_msg* ends with a newline unless it's None. The constructor adds a newline if needed.

lineno

The line number within the string containing this example where the example begins. This line number is zero-based with respect to the beginning of the containing string.

indent

The example's indentation in the containing string, i.e., the number of space characters that precede the example's first prompt.

options

A dictionary mapping from option flags to True or False, which is used to override default options for this example. Any option flags not contained in this dictionary are left at their default value (as specified by the *DocTestRunner*'s *optionflags*). By default, no options are set.

DocTestFinder objects

class doctest.**DocTestFinder**(*verbose=False*, *parser=DocTestParser()*, *recurse=True*, *exclude_empty=True*)

A processing class used to extract the *DocTest*s that are relevant to a given object, from its docstring and the docstrings of its contained objects. *DocTest*s can be extracted from modules, classes, functions, methods, staticmethods, classmethods, and properties.

The optional argument *verbose* can be used to display the objects searched by the finder. It defaults to `False` (no output).

The optional argument *parser* specifies the *DocTestParser* object (or a drop-in replacement) that is used to extract doctests from docstrings.

If the optional argument *recurse* is false, then *DocTestFinder.find()* will only examine the given object, and not any contained objects.

If the optional argument *exclude_empty* is false, then *DocTestFinder.find()* will include tests for objects with empty docstrings.

DocTestFinder defines the following method:

find(*obj[, name][, module][, globs][, extraglobs]*)

Return a list of the *DocTest*s that are defined by *obj*'s docstring, or by any of its contained objects' docstrings.

The optional argument *name* specifies the object's name; this name will be used to construct names for the returned *DocTest*s. If *name* is not specified, then `obj.__name__` is used.

The optional parameter *module* is the module that contains the given object. If the module is not specified or is `None`, then the test finder will attempt to automatically determine the correct module. The object's module is used:

- As a default namespace, if *globs* is not specified.
- To prevent the DocTestFinder from extracting DocTests from objects that are imported from other modules. (Contained objects with modules other than *module* are ignored.)
- To find the name of the file containing the object.
- To help find the line number of the object within its file.

If *module* is `False`, no attempt to find the module will be made. This is obscure, of use mostly in testing doctest itself: if *module* is `False`, or is `None` but cannot be found automatically, then all objects are considered to belong to the (non-existent) module, so all contained objects will (recursively) be searched for doctests.

The globals for each *DocTest* is formed by combining *globs* and *extraglobs* (bindings in *extraglobs* override bindings in *globs*). A new shallow copy of the globals dictionary is created for each *DocTest*. If *globs* is not specified, then it defaults to the module's *__dict__*, if specified, or `{}` otherwise. If *extraglobs* is not specified, then it defaults to `{}`.

DocTestParser objects

class doctest.**DocTestParser**

A processing class used to extract interactive examples from a string, and use them to create a *DocTest* object.

DocTestParser defines the following methods:

get_doctest(*string*, *globs*, *name*, *filename*, *lineno*)

Extract all doctest examples from the given string, and collect them into a *DocTest* object.

globs, *name*, *filename*, and *lineno* are attributes for the new *DocTest* object. See the documentation for *DocTest* for more information.

get_examples(*string*, *name='<string>'*)

Extract all doctest examples from the given string, and return them as a list of *Example* objects. Line numbers are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

parse(*string*, *name='<string>'*)

Divide the given string into examples and intervening text, and return them as a list of alternating *Example*s and strings. Line numbers for the *Example*s are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

TestResults objects

class doctest.**TestResults**(*failed*, *attempted*)

failed

Number of failed tests.

attempted

Number of attempted tests.

skipped

Number of skipped tests.

Added in version 3.13.

DocTestRunner objects

class doctest.**DocTestRunner**(*checker=None*, *verbose=None*, *optionflags=0*)

A processing class used to execute and verify the interactive examples in a *DocTest*.

The comparison between expected outputs and actual outputs is done by an *OutputChecker*. This comparison may be customized with a number of option flags; see section *Option Flags* for more information. If the option flags are insufficient, then the comparison may also be customized by passing a subclass of *OutputChecker* to the constructor.

The test runner's display output can be controlled in two ways. First, an output function can be passed to *run()*; this function will be called with strings that should be displayed. It defaults to `sys.stdout.write`. If capturing the output is not sufficient, then the display output can be also customized by subclassing DocTestRunner, and overriding the methods *report_start()*, *report_success()*, *report_unexpected_exception()*, and *report_failure()*.

The optional keyword argument *checker* specifies the *OutputChecker* object (or drop-in replacement) that should be used to compare the expected outputs to the actual outputs of doctest examples.

The optional keyword argument *verbose* controls the `DocTestRunner`'s verbosity. If *verbose* is `True`, then information is printed about each example, as it is run. If *verbose* is `False`, then only failures are printed. If *verbose* is unspecified, or `None`, then verbose output is used iff the command-line switch `-v` is used.

The optional keyword argument *optionflags* can be used to control how the test runner compares expected output to actual output, and how it displays failures. For more information, see section *Option Flags*.

The test runner accumulates statistics. The aggregated number of attempted, failed and skipped examples is also available via the `tries`, `failures` and `skips` attributes. The `run()` and `summarize()` methods return a `TestResults` instance.

`DocTestRunner` defines the following methods:

report_start(*out*, *test*, *example*)

Report that the test runner is about to process the given example. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

example is the example about to be processed. *test* is the test *containing example*. *out* is the output function that was passed to `DocTestRunner.run()`.

report_success(*out*, *test*, *example*, *got*)

Report that the given example ran successfully. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run()`.

report_failure(*out*, *test*, *example*, *got*)

Report that the given example failed. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run()`.

report_unexpected_exception(*out*, *test*, *example*, *exc_info*)

Report that the given example raised an unexpected exception. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

example is the example about to be processed. *exc_info* is a tuple containing information about the unexpected exception (as returned by `sys.exc_info()`). *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run()`.

run(*test*, *compileflags=None*, *out=None*, *clear_globs=True*)

Run the examples in *test* (a `DocTest` object), and display the results using the writer function *out*. Return a `TestResults` instance.

The examples are run in the namespace `test.globs`. If *clear_globs* is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use *clear_globs=False*.

compileflags gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to *globs*.

The output of each example is checked using the `DocTestRunner`'s output checker, and the results are formatted by the `DocTestRunner.report_*()` methods.

summarize(*verbose=None*)

Print a summary of all the test cases that have been run by this DocTestRunner, and return a `TestResults` instance.

The optional *verbose* argument controls how detailed the summary is. If the verbosity is not specified, then the *DocTestRunner*'s verbosity is used.

DocTestParser has the following attributes:

tries

Number of attempted examples.

failures

Number of failed examples.

skips

Number of skipped examples.

Added in version 3.13.

OutputChecker objects

class doctest.**OutputChecker**

A class used to check the whether the actual output from a doctest example matches the expected output. *OutputChecker* defines two methods: *check_output()*, which compares a given pair of outputs, and returns `True` if they match; and *output_difference()*, which returns a string describing the differences between two outputs.

OutputChecker defines the following methods:

check_output(*want*, *got*, *optionflags*)

Return `True` iff the actual output from an example (*got*) matches the expected output (*want*). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See section *Option Flags* for more information about option flags.

output_difference(*example*, *got*, *optionflags*)

Return a string describing the differences between the expected output for a given example (*example*) and the actual output (*got*). *optionflags* is the set of option flags used to compare *want* and *got*.

26.4.7 Debugging

Doctest provides several mechanisms for debugging doctest examples:

- Several functions convert doctests to executable Python programs, which can be run under the Python debugger, *pdb*.
- The *DebugRunner* class is a subclass of *DocTestRunner* that raises an exception for the first failing example, containing information about that example. This information can be used to perform post-mortem debugging on the example.
- The *unittest* cases generated by *DocTestSuite()* support the *debug()* method defined by *unittest.TestCase*.
- You can add a call to *pdb.set_trace()* in a doctest example, and you'll drop into the Python debugger when that line is executed. Then you can inspect current values of variables, and so on. For example, suppose `a.py` contains just this module docstring:

```
"""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print(x+3)
...     import pdb; pdb.set_trace()
>>> f(3)
9
"""
```

Then an interactive Python session may look like this:

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
  1     def g(x):
  2         print(x+3)
  3  ->     import pdb; pdb.set_trace()
[EOF]
(Pdb) p x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
  1     def f(x):
  2  ->     g(x*2)
[EOF]
(Pdb) p x
3
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>
```

Functions that convert doctests to Python code, and possibly run the synthesized code under the debugger:

`doctest.`**`script_from_examples`**`(`*s*`)`

Convert text with examples to a script.

Argument *s* is a string containing doctest examples. The string is converted to a Python script, where doctest examples in *s* are converted to regular code, and everything else is converted to Python comments. The generated script is returned as a string. For example,

```
import doctest
print(doctest.script_from_examples(r"""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print(x+y)
```

(continues on next page)

(continued from previous page)

```
    3
"""))
```

displays:

```
# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print(x+y)
# Expected:
## 3
```

This function is used internally by other functions (see below), but can also be useful when you want to transform an interactive Python session into a Python script.

doctest.**testsource**(*module*, *name*)

Convert the doctest for an object to a script.

Argument *module* is a module object, or dotted name of a module, containing the object whose doctests are of interest. Argument *name* is the name (within the module) of the object with the doctests of interest. The result is a string, containing the object's docstring converted to a Python script, as described for *script_from_examples()* above. For example, if module `a.py` contains a top-level function `f()`, then

```
import a, doctest
print(doctest.testsource(a, "a.f"))
```

prints a script version of function `f()`'s docstring, with doctests converted to code, and the rest placed in comments.

doctest.**debug**(*module*, *name*, *pm=False*)

Debug the doctests for an object.

The *module* and *name* arguments are the same as for function *testsource()* above. The synthesized Python script for the named object's docstring is written to a temporary file, and then that file is run under the control of the Python debugger, *pdb*.

A shallow copy of `module.__dict__` is used for both local and global execution context.

Optional argument *pm* controls whether post-mortem debugging is used. If *pm* has a true value, the script file is run directly, and the debugger gets involved only if the script terminates via raising an unhandled exception. If it does, then post-mortem debugging is invoked, via *pdb.post_mortem()*, passing the traceback object from the unhandled exception. If *pm* is not specified, or is false, the script is run under the debugger from the start, via passing an appropriate *exec()* call to *pdb.run()*.

doctest.**debug_src**(*src*, *pm=False*, *globs=None*)

Debug the doctests in a string.

This is like function *debug()* above, except that a string containing doctest examples is specified directly, via the *src* argument.

Optional argument *pm* has the same meaning as in function *debug()* above.

Optional argument *globs* gives a dictionary to use as both local and global execution context. If not specified, or `None`, an empty dictionary is used. If specified, a shallow copy of the dictionary is used.

The *DebugRunner* class, and the special exceptions it may raise, are of most interest to testing framework authors, and will only be sketched here. See the source code, and especially *DebugRunner*'s docstring (which is a doctest!) for more details:

class doctest.**DebugRunner**(*checker=None*, *verbose=None*, *optionflags=0*)

A subclass of *DocTestRunner* that raises an exception as soon as a failure is encountered. If an unexpected exception occurs, an *UnexpectedException* exception is raised, containing the test, the example, and the original exception. If the output doesn't match, then a *DocTestFailure* exception is raised, containing the test, the example, and the actual output.

For information about the constructor parameters and methods, see the documentation for *DocTestRunner* in section *Advanced API*.

There are two exceptions that may be raised by *DebugRunner* instances:

exception doctest.**DocTestFailure**(*test*, *example*, *got*)

An exception raised by *DocTestRunner* to signal that a doctest example's actual output did not match its expected output. The constructor arguments are used to initialize the attributes of the same names.

DocTestFailure defines the following attributes:

DocTestFailure.**test**

The *DocTest* object that was being run when the example failed.

DocTestFailure.**example**

The *Example* that failed.

DocTestFailure.**got**

The example's actual output.

exception doctest.**UnexpectedException**(*test*, *example*, *exc_info*)

An exception raised by *DocTestRunner* to signal that a doctest example raised an unexpected exception. The constructor arguments are used to initialize the attributes of the same names.

UnexpectedException defines the following attributes:

UnexpectedException.**test**

The *DocTest* object that was being run when the example failed.

UnexpectedException.**example**

The *Example* that failed.

UnexpectedException.**exc_info**

A tuple containing information about the unexpected exception, as returned by *sys.exc_info()*.

26.4.8 Soapbox

As mentioned in the introduction, *doctest* has grown to have three primary uses:

1. Checking examples in docstrings.
2. Regression testing.
3. Executable documentation / literate testing.

These uses have different requirements, and it is important to distinguish them. In particular, filling your docstrings with obscure test cases makes for bad documentation.

When writing a docstring, choose docstring examples with care. There's an art to this that needs to be learned—it may not be natural at first. Examples should add genuine value to the documentation. A good example can often be worth many words. If done with care, the examples will be invaluable for your users, and will pay back the time it takes to collect them many times over as the years go by and things change. I'm still amazed at how often one of my *doctest* examples stops working after a "harmless" change.

Doctest also makes an excellent tool for regression testing, especially if you don't skimp on explanatory text. By interleaving prose and examples, it becomes much easier to keep track of what's actually being tested, and why. When a test fails, good prose can make it much easier to figure out what the problem is, and how it should be fixed. It's true that you could write extensive comments in code-based testing, but few programmers do. Many have found that using doctest approaches instead leads to much clearer tests. Perhaps this is simply because doctest makes writing prose a little easier than writing code, while writing comments in code is a little harder. I think it goes deeper than just that: the natural attitude when writing a doctest-based test is that you want to explain the fine points of your software, and illustrate them with examples. This in turn naturally leads to test files that start with the simplest features, and logically progress to complications and edge cases. A coherent narrative is the result, instead of a collection of isolated functions that test isolated bits of functionality seemingly at random. It's a different attitude, and produces different results, blurring the distinction between testing and explaining.

Regression testing is best confined to dedicated objects or files. There are several options for organizing tests:

- Write text files containing test cases as interactive examples, and test the files using *testfile()* or *DocFileSuite()*. This is recommended, although is easiest to do for new projects, designed from the start to use doctest.
- Define functions named `_regrtest_topic` that consist of single docstrings, containing test cases for the named topics. These functions can be included in the same file as the module, or separated out into a separate test file.
- Define a `__test__` dictionary mapping from regression test topics to docstrings containing test cases.

When you have placed your tests in a module, the module can itself be the test runner. When a test fails, you can arrange for your test runner to re-run only the failing doctest while you debug the problem. Here is a minimal example of such a test runner:

```
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.FAIL_FAST
    if len(sys.argv) > 1:
        name = sys.argv[1]
        if name in globals():
            obj = globals()[name]
        else:
            obj = __test__[name]
        doctest.run_docstring_examples(obj, globals(), name=name,
                                       optionflags=flags)
    else:
        fail, total = doctest.testmod(optionflags=flags)
        print(f"{fail} failures out of {total} tests")
```

26.5 `unittest` — Unit testing framework

Source code: Lib/unittest/__init__.py

(If you are already familiar with the basic concepts of testing, you might want to skip to *the list of assert methods*.)

The *unittest* unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

To achieve this, *unittest* supports some important concepts in an object-oriented way:

test fixture
: A *test fixture* represents the preparation needed to perform one or more tests, and any associated cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test case
: A *test case* is the individual unit of testing. It checks for a specific response to a particular set of inputs. *unittest* provides a base class, *TestCase*, which may be used to create new test cases.

test suite
: A *test suite* is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

test runner
: A *test runner* is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

See also

Module *doctest*
: Another test-support module with a very different flavor.

Simple Smalltalk Testing: With Patterns
: Kent Beck's original paper on testing frameworks using the pattern shared by *unittest*.

pytest
: Third-party unittest framework with a lighter-weight syntax for writing tests. For example, `assert func(10) == 42`.

The Python Testing Tools Taxonomy
: An extensive list of Python testing tools including functional testing frameworks and mock object libraries.

Testing in Python Mailing List
: A special-interest-group for discussion of testing, and testing tools, in Python.

The script `Tools/unittestgui/unittestgui.py` in the Python source distribution is a GUI tool for test discovery and execution. This is intended largely for ease of use for those new to unit testing. For production environments it is recommended that tests be driven by a continuous integration system such as Buildbot, Jenkins, GitHub Actions, or AppVeyor.

26.5.1 Basic example

The *unittest* module provides a rich set of tools for constructing and running tests. This section demonstrates that a small subset of the tools suffice to meet the needs of most users.

Here is a short script to test three string methods:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
```

(continues on next page)

(continued from previous page)

```
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

A testcase is created by subclassing *unittest.TestCase*. The three individual tests are defined with methods whose names start with the letters `test`. This naming convention informs the test runner about which methods represent tests.

The crux of each test is a call to *assertEqual()* to check for an expected result; *assertTrue()* or *assertFalse()* to verify a condition; or *assertRaises()* to verify that a specific exception gets raised. These methods are used instead of the `assert` statement so the test runner can accumulate all test results and produce a report.

The *setUp()* and *tearDown()* methods allow you to define instructions that will be executed before and after each test method. They are covered in more detail in the section *Organizing test code*.

The final block shows a simple way to run the tests. *unittest.main()* provides a command-line interface to the test script. When run from the command line, the above script produces an output that looks like this:

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Passing the `-v` option to your test script will instruct *unittest.main()* to enable a higher level of verbosity, and produce the following output:

```
test_isupper (__main__.TestStringMethods.test_isupper) ... ok
test_split (__main__.TestStringMethods.test_split) ... ok
test_upper (__main__.TestStringMethods.test_upper) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

The above examples show the most commonly used *unittest* features which are sufficient to meet many everyday testing needs. The remainder of the documentation explores the full feature set from first principles.

Changed in version 3.11: The behavior of returning a value from a test method (other than the default `None` value), is now deprecated.

26.5.2 Command-Line Interface

The unittest module can be used from the command line to run tests from modules, classes or even individual test methods:

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

You can pass in a list with any combination of module names, and fully qualified class or method names.

Test modules can be specified by file path as well:

```
python -m unittest tests/test_something.py
```

This allows you to use the shell filename completion to specify the test module. The file specified must still be importable as a module. The path is converted to a module name by removing the '.py' and converting path separators into '.'. If you want to execute a test file that isn't importable as a module you should execute the file directly instead.

You can run tests with more detail (higher verbosity) by passing in the -v flag:

```
python -m unittest -v test_module
```

When executed without arguments *Test Discovery* is started:

```
python -m unittest
```

For a list of all the command-line options:

```
python -m unittest -h
```

Changed in version 3.2: In earlier versions it was only possible to run individual test methods and not modules or classes.

Command-line options

unittest supports these command-line options:

-b, --buffer

The standard output and standard error streams are buffered during the test run. Output during a passing test is discarded. Output is echoed normally on test fail or error and is added to the failure messages.

-c, --catch

`Control-C` during the test run waits for the current test to end and then reports all the results so far. A second `Control-C` raises the normal *`KeyboardInterrupt`* exception.

See *Signal Handling* for the functions that provide this functionality.

-f, --failfast

Stop the test run on the first error or failure.

-k

Only run test methods and classes that match the pattern or substring. This option may be used multiple times, in which case all test cases that match any of the given patterns are included.

Patterns that contain a wildcard character (`*`) are matched against the test name using *`fnmatch.fnmatchcase()`*; otherwise simple case-sensitive substring matching is used.

Patterns are matched against the fully qualified test method name as imported by the test loader.

For example, `-k foo` matches `foo_tests.SomeTest.test_something`, `bar_tests.SomeTest.test_foo`, but not `bar_tests.FooTest.test_something`.

--locals

Show local variables in tracebacks.

--durations `N`

Show the N slowest test cases (N=0 for all).

Added in version 3.2: The command-line options `-b`, `-c` and `-f` were added.

Added in version 3.5: The command-line option `--locals`.

Added in version 3.7: The command-line option `-k`.

Added in version 3.12: The command-line option `--durations`.

The command line can also be used for test discovery, for running all of the tests in a project or just a subset.

26.5.3 Test Discovery

Added in version 3.2.

Unittest supports simple test discovery. In order to be compatible with test discovery, all of the test files must be modules or packages importable from the top-level directory of the project (this means that their filenames must be valid identifiers).

Test discovery is implemented in *TestLoader.discover()*, but can also be used from the command line. The basic command-line usage is:

```
cd project_directory
python -m unittest discover
```

Note

As a shortcut, `python -m unittest` is the equivalent of `python -m unittest discover`. If you want to pass arguments to test discovery the `discover` sub-command must be used explicitly.

The `discover` sub-command has the following options:

-v, --verbose
Verbose output

-s, --start-directory `directory`
Directory to start discovery (`.` default)

-p, --pattern `pattern`
Pattern to match test files (`test*.py` default)

-t, --top-level-directory `directory`
Top level directory of project (defaults to start directory)

The *-s*, *-p*, and *-t* options can be passed in as positional arguments in that order. The following two command lines are equivalent:

```
python -m unittest discover -s project_directory -p "*_test.py"
python -m unittest discover project_directory "*_test.py"
```

As well as being a path it is possible to pass a package name, for example `myproject.subpackage.test`, as the start directory. The package name you supply will then be imported and its location on the filesystem will be used as the start directory.

Caution

Test discovery loads tests by importing them. Once test discovery has found all the test files from the start directory you specify it turns the paths into package names to import. For example `foo/bar/baz.py` will be imported as `foo.bar.baz`.

If you have a package installed globally and attempt test discovery on a different copy of the package then the import *could* happen from the wrong place. If this happens test discovery will warn you and exit.

If you supply the start directory as a package name rather than a path to a directory then discover assumes that whichever location it imports from is the location you intended, so you will not get the warning.

Test modules and packages can customize test loading and discovery by through the *load_tests protocol*.

Changed in version 3.4: Test discovery supports *namespace packages* for the start directory. Note that you need to specify the top level directory too (e.g. `python -m unittest discover -s root/namespace -t root`).

Changed in version 3.11: `unittest` dropped the *namespace packages* support in Python 3.11. It has been broken since Python 3.7. Start directory and subdirectories containing tests must be regular package that have `__init__.py` file.

Directories containing start directory still can be a namespace package. In this case, you need to specify start directory as dotted package name, and target directory explicitly. For example:

```
# proj/  <-- current directory
#   namespace/
#     mypkg/
#       __init__.py
#       test_mypkg.py

python -m unittest discover -s namespace.mypkg -t .
```

26.5.4 Organizing test code

The basic building blocks of unit testing are *test cases* — single scenarios that must be set up and checked for correctness. In `unittest`, test cases are represented by `unittest.TestCase` instances. To make your own test cases you must write subclasses of `TestCase` or use `FunctionTestCase`.

The testing code of a `TestCase` instance should be entirely self contained, such that it can be run either in isolation or in arbitrary combination with any number of other test cases.

The simplest `TestCase` subclass will simply implement a test method (i.e. a method whose name starts with `test`) in order to perform specific testing code:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

Note that in order to test something, we use one of the *assert* methods* provided by the `TestCase` base class. If the test fails, an exception will be raised with an explanatory message, and `unittest` will identify the test case as a *failure*. Any other exceptions will be treated as *errors*.

Tests can be numerous, and their set-up can be repetitive. Luckily, we can factor out set-up code by implementing a method called `setUp()`, which the testing framework will automatically call for every single test we run:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
```

(continues on next page)

(continued from previous page)

```
                        'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

Note

The order in which the various tests will be run is determined by sorting the test method names with respect to the built-in ordering for strings.

If the *setUp()* method raises an exception while the test is running, the framework will consider the test to have suffered an error, and the test method will not be executed.

Similarly, we can provide a *tearDown()* method that tidies up after the test method has been run:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

If *setUp()* succeeded, *tearDown()* will be run whether the test method succeeded or not.

Such a working environment for the testing code is called a *test fixture*. A new TestCase instance is created as a unique test fixture used to execute each individual test method. Thus *setUp()*, *tearDown()*, and `__init__()` will be called once per test.

It is recommended that you use TestCase implementations to group tests together according to the features they test. *unittest* provides a mechanism for this: the *test suite*, represented by *unittest*'s *TestSuite* class. In most cases, calling *unittest.main()* will do the right thing and collect all the module's test cases for you and execute them.

However, should you want to customize the building of your test suite, you can do it yourself:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_widget_size'))
    suite.addTest(WidgetTestCase('test_widget_resize'))
    return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

You can place the definitions of test cases and test suites in the same modules as the code they are to test (such as `widget.py`), but there are several advantages to placing the test code in a separate module, such as `test_widget.py`:

- The test module can be run standalone from the command line.
- The test code can more easily be separated from shipped code.
- There is less temptation to change test code to fit the code it tests without a good reason.

- Test code should be modified much less frequently than the code it tests.
- Tested code can be refactored more easily.
- Tests for modules written in C must be in separate modules anyway, so why not be consistent?
- If the testing strategy changes, there is no need to change the source code.

26.5.5 Re-using old test code

Some users will find that they have existing test code that they would like to run from *unittest*, without converting every old test function to a *TestCase* subclass.

For this reason, *unittest* provides a *FunctionTestCase* class. This subclass of *TestCase* can be used to wrap an existing test function. Set-up and tear-down functions can also be provided.

Given the following test function:

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

one can create an equivalent test case instance as follows, with optional set-up and tear-down methods:

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

Note

Even though *FunctionTestCase* can be used to quickly convert an existing test base over to a *unittest*-based system, this approach is not recommended. Taking the time to set up proper *TestCase* subclasses will make future test refactorings infinitely easier.

In some cases, the existing tests may have been written using the *doctest* module. If so, *doctest* provides a `DocTestSuite` class that can automatically build *unittest.TestSuite* instances from the existing *doctest*-based tests.

26.5.6 Skipping tests and expected failures

Added in version 3.1.

Unittest supports skipping individual test methods and even whole classes of tests. In addition, it supports marking a test as an "expected failure," a test that is broken and will fail, but shouldn't be counted as a failure on a *TestResult*.

Skipping a test is simply a matter of using the *skip() decorator* or one of its conditional variants, calling *TestCase.skipTest()* within a *setUp()* or test method, or raising *SkipTest* directly.

Basic skipping looks like this:

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
```

(continues on next page)

(continued from previous page)

```
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass

    def test_maybe_skipped(self):
        if not external_resource_available():
            self.skipTest("external resource not available")
        # test code that depends on the external resource
        pass
```

This is the output of running the example above in verbose mode:

```
test_format (__main__.MyTestCase.test_format) ... skipped 'not supported in this␣
↪library version'
test_nothing (__main__.MyTestCase.test_nothing) ... skipped 'demonstrating skipping'
test_maybe_skipped (__main__.MyTestCase.test_maybe_skipped) ... skipped 'external␣
↪resource not available'
test_windows_support (__main__.MyTestCase.test_windows_support) ... skipped 'requires␣
↪Windows'

----------------------------------------------------------------------
Ran 4 tests in 0.005s

OK (skipped=4)
```

Classes can be skipped just like methods:

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

TestCase.setUp() can also skip the test. This is useful when a resource that needs to be set up is not available.

Expected failures use the *expectedFailure()* decorator.

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

It's easy to roll your own skipping decorators by making a decorator that calls *skip()* on the test when it wants it to be skipped. This decorator skips the test unless the passed object has a certain attribute:

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```

The following decorators and exception implement test skipping and expected failures:

`@unittest.skip(reason)`

Unconditionally skip the decorated test. *reason* should describe why the test is being skipped.

`@unittest.skipIf(condition, reason)`

Skip the decorated test if *condition* is true.

`@unittest.skipUnless(condition, reason)`

Skip the decorated test unless *condition* is true.

`@unittest.expectedFailure`

Mark the test as an expected failure or error. If the test fails or errors in the test function itself (rather than in one of the *test fixture* methods) then it will be considered a success. If the test passes, it will be considered a failure.

exception `unittest.SkipTest(reason)`

This exception is raised to skip a test.

Usually you can use *TestCase.skipTest()* or one of the skipping decorators instead of raising this directly.

Skipped tests will not have *setUp()* or *tearDown()* run around them. Skipped classes will not have *setUpClass()* or *tearDownClass()* run. Skipped modules will not have `setUpModule()` or `tearDownModule()` run.

26.5.7 Distinguishing test iterations using subtests

Added in version 3.4.

When there are very small differences among your tests, for instance some parameters, unittest allows you to distinguish them inside the body of a test method using the *subTest()* context manager.

For example, the following test:

```
class NumbersTest(unittest.TestCase):

    def test_even(self):
        """
        Test that numbers between 0 and 5 are all even.
        """
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

will produce the following output:

```
======================================================================
FAIL: test_even (__main__.NumbersTest.test_even) (i=1)
Test that numbers between 0 and 5 are all even.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest.test_even) (i=3)
Test that numbers between 0 and 5 are all even.
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest.test_even) (i=5)
Test that numbers between 0 and 5 are all even.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0
```

Without using a subtest, execution would stop after the first failure, and the error would be less easy to diagnose because the value of `i` wouldn't be displayed:

```
======================================================================
FAIL: test_even (__main__.NumbersTest.test_even)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

26.5.8 Classes and functions

This section describes in depth the API of *unittest*.

Test cases

class unittest.**TestCase**(*methodName='runTest'*)

Instances of the *TestCase* class represent the logical test units in the *unittest* universe. This class is intended to be used as a base class, with specific tests being implemented by concrete subclasses. This class implements the interface needed by the test runner to allow it to drive the tests, and methods that the test code can use to check for and report various kinds of failure.

Each instance of *TestCase* will run a single base method: the method named *methodName*. In most uses of *TestCase*, you will neither change the *methodName* nor reimplement the default `runTest()` method.

Changed in version 3.2: *TestCase* can be instantiated successfully without providing a *methodName*. This makes it easier to experiment with *TestCase* from the interactive interpreter.

TestCase instances provide three groups of methods: one group used to run the test, another used by the test implementation to check conditions and report failures, and some inquiry methods allowing information about the test itself to be gathered.

Methods in the first group (running the test) are:

setUp()

Method called to prepare the test fixture. This is called immediately before calling the test method; other than

AssertionError or *SkipTest*, any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

tearDown()

Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception, so the implementation in subclasses may need to be particularly careful about checking internal state. Any exception, other than *AssertionError* or *SkipTest*, raised by this method will be considered an additional error rather than a test failure (thus increasing the total number of reported errors). This method will only be called if the *setUp()* succeeds, regardless of the outcome of the test method. The default implementation does nothing.

setUpClass()

A class method called before tests in an individual class are run. `setUpClass` is called with the class as the only argument and must be decorated as a *classmethod()*:

```
@classmethod
def setUpClass(cls):
    ...
```

See *Class and Module Fixtures* for more details.

Added in version 3.2.

tearDownClass()

A class method called after tests in an individual class have run. `tearDownClass` is called with the class as the only argument and must be decorated as a *classmethod()*:

```
@classmethod
def tearDownClass(cls):
    ...
```

See *Class and Module Fixtures* for more details.

Added in version 3.2.

run(*result=None*)

Run the test, collecting the result into the *TestResult* object passed as *result*. If *result* is omitted or `None`, a temporary result object is created (by calling the *defaultTestResult()* method) and used. The result object is returned to *run()*'s caller.

The same effect may be had by simply calling the *TestCase* instance.

Changed in version 3.3: Previous versions of `run` did not return the result. Neither did calling an instance.

skipTest(*reason*)

Calling this during a test method or *setUp()* skips the current test. See *Skipping tests and expected failures* for more information.

Added in version 3.1.

subTest(*msg=None*, ***params*)

Return a context manager which executes the enclosed code block as a subtest. *msg* and *params* are optional, arbitrary values which are displayed whenever a subtest fails, allowing you to identify them clearly.

A test case can contain any number of subtest declarations, and they can be arbitrarily nested.

See *Distinguishing test iterations using subtests* for more information.

Added in version 3.4.

debug()

Run the test without collecting the result. This allows exceptions raised by the test to be propagated to the caller, and can be used to support running tests under a debugger.

The *TestCase* class provides several assert methods to check for and report failures. The following table lists the most commonly used methods (see the tables below for more assert methods):

Method	Checks that	New in
assertEqual(a, b)	`a == b`	
assertNotEqual(a, b)	`a != b`	
assertTrue(x)	`bool(x) is True`	
assertFalse(x)	`bool(x) is False`	
assertIs(a, b)	`a is b`	3.1
assertIsNot(a, b)	`a is not b`	3.1
assertIsNone(x)	`x is None`	3.1
assertIsNotNone(x)	`x is not None`	3.1
assertIn(a, b)	`a in b`	3.1
assertNotIn(a, b)	`a not in b`	3.1
assertIsInstance(a, b)	`isinstance(a, b)`	3.2
assertNotIsInstance(a, b)	`not isinstance(a, b)`	3.2

All the assert methods accept a *msg* argument that, if specified, is used as the error message on failure (see also *longMessage*). Note that the *msg* keyword argument can be passed to *assertRaises()*, *assertRaisesRegex()*, *assertWarns()*, *assertWarnsRegex()* only when they are used as a context manager.

assertEqual(*first*, *second*, *msg=None*)

Test that *first* and *second* are equal. If the values do not compare equal, the test will fail.

In addition, if *first* and *second* are the exact same type and one of list, tuple, dict, set, frozenset or str or any type that a subclass registers with *addTypeEqualityFunc()* the type-specific equality function will be called in order to generate a more useful default error message (see also the *list of type-specific methods*).

Changed in version 3.1: Added the automatic calling of type-specific equality function.

Changed in version 3.2: *assertMultiLineEqual()* added as the default type equality function for comparing strings.

assertNotEqual(*first*, *second*, *msg=None*)

Test that *first* and *second* are not equal. If the values do compare equal, the test will fail.

assertTrue(*expr*, *msg=None*)

assertFalse(*expr*, *msg=None*)

Test that *expr* is true (or false).

Note that this is equivalent to `bool(expr) is True` and not to `expr is True` (use `assertIs(expr, True)` for the latter). This method should also be avoided when more specific methods are available (e.g. `assertEqual(a, b)` instead of `assertTrue(a == b)`), because they provide a better error message in case of failure.

assertIs(*first*, *second*, *msg=None*)

assertIsNot(*first*, *second*, *msg=None*)

Test that *first* and *second* are (or are not) the same object.

Added in version 3.1.

assertIsNone(*expr*, *msg=None*)

assertIsNotNone(*expr*, *msg=None*)

Test that *expr* is (or is not) `None`.

Added in version 3.1.

assertIn(*member*, *container*, *msg=None*)

assertNotIn(*member*, *container*, *msg=None*)

Test that *member* is (or is not) in *container*.

Added in version 3.1.

assertIsInstance(*obj*, *cls*, *msg=None*)

assertNotIsInstance(*obj*, *cls*, *msg=None*)

Test that *obj* is (or is not) an instance of *cls* (which can be a class or a tuple of classes, as supported by *isinstance()*). To check for the exact type, use *assertIs(type(obj), cls)*.

Added in version 3.2.

It is also possible to check the production of exceptions, warnings, and log messages using the following methods:

Method	Checks that	New in
*assertRaises(exc, fun, *args, **kwds)*	`fun(*args, **kwds)` raises *exc*	
*assertRaisesRegex(exc, r, fun, *args, **kwds)*	`fun(*args, **kwds)` raises *exc* and the message matches regex *r*	3.1
*assertWarns(warn, fun, *args, **kwds)*	`fun(*args, **kwds)` raises *warn*	3.2
*assertWarnsRegex(warn, r, fun, *args, **kwds)*	`fun(*args, **kwds)` raises *warn* and the message matches regex *r*	3.2
assertLogs(logger, level)	The `with` block logs on *logger* with minimum *level*	3.4
assertNoLogs(logger, level)	**The `with` block does not log on** *logger* with minimum *level*	3.10

assertRaises(*exception*, *callable*, **args*, ***kwds*)

assertRaises(*exception*, *, *msg=None*)

Test that an exception is raised when *callable* is called with any positional or keyword arguments that are also passed to *assertRaises()*. The test passes if *exception* is raised, is an error if another exception is raised, or fails if no exception is raised. To catch any of a group of exceptions, a tuple containing the exception classes may be passed as *exception*.

If only the *exception* and possibly the *msg* arguments are given, return a context manager so that the code under test can be written inline rather than as a function:

```
with self.assertRaises(SomeException):
    do_something()
```

When used as a context manager, *assertRaises()* accepts the additional keyword argument *msg*.

The context manager will store the caught exception object in its `exception` attribute. This can be useful if the intention is to perform additional checks on the exception raised:

```
with self.assertRaises(SomeException) as cm:
    do_something()

the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

Changed in version 3.1: Added the ability to use *assertRaises()* as a context manager.

Changed in version 3.2: Added the `exception` attribute.

Changed in version 3.3: Added the *msg* keyword argument when used as a context manager.

assertRaisesRegex(*exception*, *regex*, *callable*, **args*, ***kwds*)

assertRaisesRegex(*exception*, *regex*, ***, *msg=None*)

Like *assertRaises()* but also tests that *regex* matches on the string representation of the raised exception. *regex* may be a regular expression object or a string containing a regular expression suitable for use by *re.search()*. Examples:

```
self.assertRaisesRegex(ValueError, "invalid literal for.*XYZ'$",
                       int, 'XYZ')
```

or:

```
with self.assertRaisesRegex(ValueError, 'literal'):
   int('XYZ')
```

Added in version 3.1: Added under the name `assertRaisesRegexp`.

Changed in version 3.2: Renamed to *assertRaisesRegex()*.

Changed in version 3.3: Added the *msg* keyword argument when used as a context manager.

assertWarns(*warning*, *callable*, **args*, ***kwds*)

assertWarns(*warning*, ***, *msg=None*)

Test that a warning is triggered when *callable* is called with any positional or keyword arguments that are also passed to *assertWarns()*. The test passes if *warning* is triggered and fails if it isn't. Any exception is an error. To catch any of a group of warnings, a tuple containing the warning classes may be passed as *warnings*.

If only the *warning* and possibly the *msg* arguments are given, return a context manager so that the code under test can be written inline rather than as a function:

```
with self.assertWarns(SomeWarning):
    do_something()
```

When used as a context manager, *assertWarns()* accepts the additional keyword argument *msg*.

The context manager will store the caught warning object in its `warning` attribute, and the source line which triggered the warnings in the `filename` and `lineno` attributes. This can be useful if the intention is to perform additional checks on the warning caught:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()

self.assertIn('myfile.py', cm.filename)
self.assertEqual(320, cm.lineno)
```

This method works regardless of the warning filters in place when it is called.

Added in version 3.2.

Changed in version 3.3: Added the *msg* keyword argument when used as a context manager.

assertWarnsRegex(*warning*, *regex*, *callable*, **args*, ***kwds*)

assertWarnsRegex(*warning*, *regex*, *, *msg=None*)

Like *assertWarns()* but also tests that *regex* matches on the message of the triggered warning. *regex* may be a regular expression object or a string containing a regular expression suitable for use by *re.search()*. Example:

```
self.assertWarnsRegex(DeprecationWarning,
                      r'legacy_function\(\) is deprecated',
                      legacy_function, 'XYZ')
```

or:

```
with self.assertWarnsRegex(RuntimeWarning, 'unsafe frobnicating'):
    frobnicate('/etc/passwd')
```

Added in version 3.2.

Changed in version 3.3: Added the *msg* keyword argument when used as a context manager.

assertLogs(*logger=None*, *level=None*)

A context manager to test that at least one message is logged on the *logger* or one of its children, with at least the given *level*.

If given, *logger* should be a *logging.Logger* object or a *str* giving the name of a logger. The default is the root logger, which will catch all messages that were not blocked by a non-propagating descendent logger.

If given, *level* should be either a numeric logging level or its string equivalent (for example either `"ERROR"` or *logging.ERROR*). The default is *logging.INFO*.

The test passes if at least one message emitted inside the `with` block matches the *logger* and *level* conditions, otherwise it fails.

The object returned by the context manager is a recording helper which keeps tracks of the matching log messages. It has two attributes:

records

A list of *logging.LogRecord* objects of the matching log messages.

output

A list of *str* objects with the formatted output of matching messages.

Example:

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

Added in version 3.4.

assertNoLogs(*logger=None*, *level=None*)

A context manager to test that no messages are logged on the *logger* or one of its children, with at least the given *level*.

If given, *logger* should be a *logging.Logger* object or a *str* giving the name of a logger. The default is the root logger, which will catch all messages.

If given, *level* should be either a numeric logging level or its string equivalent (for example either "ERROR" or *logging.ERROR*). The default is *logging.INFO*.

Unlike *assertLogs()*, nothing will be returned by the context manager.

Added in version 3.10.

There are also other methods used to perform more specific checks, such as:

Method	Checks that	New in
assertAlmostEqual(a, b)	`round(a-b, 7) == 0`	
assertNotAlmostEqual(a, b)	`round(a-b, 7) != 0`	
assertGreater(a, b)	`a > b`	3.1
assertGreaterEqual(a, b)	`a >= b`	3.1
assertLess(a, b)	`a < b`	3.1
assertLessEqual(a, b)	`a <= b`	3.1
assertRegex(s, r)	`r.search(s)`	3.1
assertNotRegex(s, r)	`not r.search(s)`	3.2
assertCountEqual(a, b)	*a* and *b* have the same elements in the same number, regardless of their order.	3.2

assertAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)

assertNotAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)

Test that *first* and *second* are approximately (or not approximately) equal by computing the difference, rounding to the given number of decimal *places* (default 7), and comparing to zero. Note that these methods round the values to the given number of *decimal places* (i.e. like the *round()* function) and not *significant digits*.

If *delta* is supplied instead of *places* then the difference between *first* and *second* must be less or equal to (or greater than) *delta*.

Supplying both *delta* and *places* raises a *TypeError*.

Changed in version 3.2: *assertAlmostEqual()* automatically considers almost equal objects that compare equal. *assertNotAlmostEqual()* automatically fails if the objects compare equal. Added the *delta* keyword argument.

assertGreater(*first*, *second*, *msg=None*)

assertGreaterEqual(*first*, *second*, *msg=None*)

assertLess(*first*, *second*, *msg=None*)

assertLessEqual(*first*, *second*, *msg=None*)

Test that *first* is respectively >, >=, < or <= than *second* depending on the method name. If not, the test will fail:

```
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

Added in version 3.1.

assertRegex(*text*, *regex*, *msg=None*)

assertNotRegex(*text*, *regex*, *msg=None*)

Test that a *regex* search matches (or does not match) *text*. In case of failure, the error message will include the pattern and the *text* (or the pattern and the part of *text* that unexpectedly matched). *regex* may be a regular expression object or a string containing a regular expression suitable for use by `re.search()`.

Added in version 3.1: Added under the name `assertRegexpMatches`.

Changed in version 3.2: The method `assertRegexpMatches()` has been renamed to `assertRegex()`.

Added in version 3.2: `assertNotRegex()`.

assertCountEqual(*first*, *second*, *msg=None*)

Test that sequence *first* contains the same elements as *second*, regardless of their order. When they don't, an error message listing the differences between the sequences will be generated.

Duplicate elements are *not* ignored when comparing *first* and *second*. It verifies whether each element has the same count in both sequences. Equivalent to: `assertEqual(Counter(list(first)), Counter(list(second)))` but works with sequences of unhashable objects as well.

Added in version 3.2.

The `assertEqual()` method dispatches the equality check for objects of the same type to different type-specific methods. These methods are already implemented for most of the built-in types, but it's also possible to register new methods using `addTypeEqualityFunc()`:

addTypeEqualityFunc(*typeobj*, *function*)

Registers a type-specific method called by `assertEqual()` to check if two objects of exactly the same *typeobj* (not subclasses) compare equal. *function* must take two positional arguments and a third msg=None keyword argument just as `assertEqual()` does. It must raise `self.failureException(msg)` when inequality between the first two parameters is detected – possibly providing useful information and explaining the inequalities in details in the error message.

Added in version 3.1.

The list of type-specific methods automatically used by `assertEqual()` are summarized in the following table. Note that it's usually not necessary to invoke these methods directly.

Method	Used to compare	New in
`assertMultiLineEqual(a, b)`	strings	3.1
`assertSequenceEqual(a, b)`	sequences	3.1
`assertListEqual(a, b)`	lists	3.1
`assertTupleEqual(a, b)`	tuples	3.1
`assertSetEqual(a, b)`	sets or frozensets	3.1
`assertDictEqual(a, b)`	dicts	3.1

assertMultiLineEqual(*first*, *second*, *msg=None*)

Test that the multiline string *first* is equal to the string *second*. When not equal a diff of the two strings highlighting the differences will be included in the error message. This method is used by default when comparing strings with `assertEqual()`.

Added in version 3.1.

assertSequenceEqual(*first*, *second*, *msg=None*, *seq_type=None*)

Tests that two sequences are equal. If a *seq_type* is supplied, both *first* and *second* must be instances of *seq_type* or a failure will be raised. If the sequences are different an error message is constructed that shows the difference between the two.

This method is not called directly by *assertEqual()*, but it's used to implement *assertListEqual()* and *assertTupleEqual()*.

Added in version 3.1.

assertListEqual(*first*, *second*, *msg=None*)

assertTupleEqual(*first*, *second*, *msg=None*)

Tests that two lists or tuples are equal. If not, an error message is constructed that shows only the differences between the two. An error is also raised if either of the parameters are of the wrong type. These methods are used by default when comparing lists or tuples with *assertEqual()*.

Added in version 3.1.

assertSetEqual(*first*, *second*, *msg=None*)

Tests that two sets are equal. If not, an error message is constructed that lists the differences between the sets. This method is used by default when comparing sets or frozensets with *assertEqual()*.

Fails if either of *first* or *second* does not have a `set.difference()` method.

Added in version 3.1.

assertDictEqual(*first*, *second*, *msg=None*)

Test that two dictionaries are equal. If not, an error message is constructed that shows the differences in the dictionaries. This method will be used by default to compare dictionaries in calls to *assertEqual()*.

Added in version 3.1.

Finally the *TestCase* provides the following methods and attributes:

fail(*msg=None*)

Signals a test failure unconditionally, with *msg* or `None` for the error message.

failureException

This class attribute gives the exception raised by the test method. If a test framework needs to use a specialized exception, possibly to carry additional information, it must subclass this exception in order to "play fair" with the framework. The initial value of this attribute is *AssertionError*.

longMessage

This class attribute determines what happens when a custom failure message is passed as the msg argument to an assertXYY call that fails. `True` is the default value. In this case, the custom message is appended to the end of the standard failure message. When set to `False`, the custom message replaces the standard message.

The class setting can be overridden in individual test methods by assigning an instance attribute, self.longMessage, to `True` or `False` before calling the assert methods.

The class setting gets reset before each test call.

Added in version 3.1.

maxDiff

This attribute controls the maximum length of diffs output by assert methods that report diffs on failure. It defaults to 80*8 characters. Assert methods affected by this attribute are *assertSequenceEqual()* (including all the sequence comparison methods that delegate to it), *assertDictEqual()* and *assertMultiLineEqual()*.

Setting `maxDiff` to `None` means that there is no maximum length of diffs.

Added in version 3.2.

Testing frameworks can use the following methods to collect information on the test:

countTestCases()

Return the number of tests represented by this test object. For *TestCase* instances, this will always be 1.

defaultTestResult()

Return an instance of the test result class that should be used for this test case class (if no other result instance is provided to the *run()* method).

For *TestCase* instances, this will always be an instance of *TestResult*; subclasses of *TestCase* should override this as necessary.

id()

Return a string identifying the specific test case. This is usually the full name of the test method, including the module and class name.

shortDescription()

Returns a description of the test, or None if no description has been provided. The default implementation of this method returns the first line of the test method's docstring, if available, or None.

Changed in version 3.1: In 3.1 this was changed to add the test name to the short description even in the presence of a docstring. This caused compatibility issues with unittest extensions and adding the test name was moved to the *TextTestResult* in Python 3.2.

addCleanup(*function*, /, **args*, ***kwargs*)

Add a function to be called after *tearDown()* to cleanup resources used during the test. Functions will be called in reverse order to the order they are added (LIFO). They are called with any arguments and keyword arguments passed into *addCleanup()* when they are added.

If *setUp()* fails, meaning that *tearDown()* is not called, then any cleanup functions added will still be called.

Added in version 3.1.

enterContext(*cm*)

Enter the supplied *context manager*. If successful, also add its __exit__() method as a cleanup function by *addCleanup()* and return the result of the __enter__() method.

Added in version 3.11.

doCleanups()

This method is called unconditionally after *tearDown()*, or after *setUp()* if *setUp()* raises an exception.

It is responsible for calling all the cleanup functions added by *addCleanup()*. If you need cleanup functions to be called *prior* to *tearDown()* then you can call *doCleanups()* yourself.

doCleanups() pops methods off the stack of cleanup functions one at a time, so it can be called at any time.

Added in version 3.1.

classmethod addClassCleanup(*function*, /, **args*, ***kwargs*)

Add a function to be called after *tearDownClass()* to cleanup resources used during the test class. Functions will be called in reverse order to the order they are added (LIFO). They are called with any arguments and keyword arguments passed into *addClassCleanup()* when they are added.

If *setUpClass()* fails, meaning that *tearDownClass()* is not called, then any cleanup functions added will still be called.

Added in version 3.8.

classmethod enterClassContext(*cm*)

Enter the supplied *context manager*. If successful, also add its `__exit__()` method as a cleanup function by `addClassCleanup()` and return the result of the `__enter__()` method.

Added in version 3.11.

classmethod doClassCleanups()

This method is called unconditionally after `tearDownClass()`, or after `setUpClass()` if `setUpClass()` raises an exception.

It is responsible for calling all the cleanup functions added by `addClassCleanup()`. If you need cleanup functions to be called *prior* to `tearDownClass()` then you can call `doClassCleanups()` yourself.

`doClassCleanups()` pops methods off the stack of cleanup functions one at a time, so it can be called at any time.

Added in version 3.8.

class unittest.**IsolatedAsyncioTestCase**(*methodName='runTest'*)

This class provides an API similar to `TestCase` and also accepts coroutines as test functions.

Added in version 3.8.

loop_factory

The *loop_factory* passed to `asyncio.Runner`. Override in subclasses with `asyncio.EventLoop` to avoid using the asyncio policy system.

Added in version 3.13.

coroutine asyncSetUp()

Method called to prepare the test fixture. This is called after `setUp()`. This is called immediately before calling the test method; other than `AssertionError` or `SkipTest`, any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

coroutine asyncTearDown()

Method called immediately after the test method has been called and the result recorded. This is called before `tearDown()`. This is called even if the test method raised an exception, so the implementation in subclasses may need to be particularly careful about checking internal state. Any exception, other than `AssertionError` or `SkipTest`, raised by this method will be considered an additional error rather than a test failure (thus increasing the total number of reported errors). This method will only be called if the `asyncSetUp()` succeeds, regardless of the outcome of the test method. The default implementation does nothing.

addAsyncCleanup(*function*, */*, **args*, ***kwargs*)

This method accepts a coroutine that can be used as a cleanup function.

coroutine enterAsyncContext(*cm*)

Enter the supplied *asynchronous context manager*. If successful, also add its `__aexit__()` method as a cleanup function by `addAsyncCleanup()` and return the result of the `__aenter__()` method.

Added in version 3.11.

run(*result=None*)

Sets up a new event loop to run the test, collecting the result into the `TestResult` object passed as *result*. If *result* is omitted or `None`, a temporary result object is created (by calling the `defaultTestResult()` method) and used. The result object is returned to `run()`'s caller. At the end of the test all the tasks in the event loop are cancelled.

An example illustrating the order:

```
from unittest import IsolatedAsyncioTestCase

events = []


class Test(IsolatedAsyncioTestCase):


    def setUp(self):
        events.append("setUp")

    async def asyncSetUp(self):
        self._async_connection = await AsyncConnection()
        events.append("asyncSetUp")

    async def test_response(self):
        events.append("test_response")
        response = await self._async_connection.get("https://example.com")
        self.assertEqual(response.status_code, 200)
        self.addAsyncCleanup(self.on_cleanup)

    def tearDown(self):
        events.append("tearDown")

    async def asyncTearDown(self):
        await self._async_connection.close()
        events.append("asyncTearDown")

    async def on_cleanup(self):
        events.append("cleanup")

if __name__ == "__main__":
    unittest.main()
```

After running the test, `events` would contain `["setUp", "asyncSetUp", "test_response", "asyncTearDown", "tearDown", "cleanup"]`.

class unittest.**FunctionTestCase**(*testFunc*, *setUp=None*, *tearDown=None*, *description=None*)

This class implements the portion of the *TestCase* interface which allows the test runner to drive the test, but does not provide the methods which test code can use to check and report errors. This is used to create test cases using legacy test code, allowing it to be integrated into a *unittest*-based test framework.

Grouping tests

class unittest.**TestSuite**(*tests=()*)

This class represents an aggregation of individual test cases and test suites. The class presents the interface needed by the test runner to allow it to be run as any other test case. Running a *TestSuite* instance is the same as iterating over the suite, running each test individually.

If *tests* is given, it must be an iterable of individual test cases or other test suites that will be used to build the suite initially. Additional methods are provided to add test cases and suites to the collection later on.

TestSuite objects behave much like *TestCase* objects, except they do not actually implement a test. Instead, they are used to aggregate tests into groups of tests that should be run together. Some additional methods are available to add tests to *TestSuite* instances:

addTest(*test*)

Add a *TestCase* or *TestSuite* to the suite.

addTests(*tests*)

Add all the tests from an iterable of *TestCase* and *TestSuite* instances to this test suite.

This is equivalent to iterating over *tests*, calling *addTest()* for each element.

TestSuite shares the following methods with *TestCase*:

run(*result*)

Run the tests associated with this suite, collecting the result into the test result object passed as *result*. Note that unlike *TestCase.run()*, *TestSuite.run()* requires the result object to be passed in.

debug()

Run the tests associated with this suite without collecting the result. This allows exceptions raised by the test to be propagated to the caller and can be used to support running tests under a debugger.

countTestCases()

Return the number of tests represented by this test object, including all individual tests and sub-suites.

__iter__()

Tests grouped by a *TestSuite* are always accessed by iteration. Subclasses can lazily provide tests by overriding `__iter__()`. Note that this method may be called several times on a single suite (for example when counting tests or comparing for equality) so the tests returned by repeated iterations before *TestSuite.run()* must be the same for each call iteration. After *TestSuite.run()*, callers should not rely on the tests returned by this method unless the caller uses a subclass that overrides `TestSuite._removeTestAtIndex()` to preserve test references.

Changed in version 3.2: In earlier versions the *TestSuite* accessed tests directly rather than through iteration, so overriding `__iter__()` wasn't sufficient for providing tests.

Changed in version 3.4: In earlier versions the *TestSuite* held references to each *TestCase* after *TestSuite.run()*. Subclasses can restore that behavior by overriding `TestSuite._removeTestAtIndex()`.

In the typical usage of a *TestSuite* object, the *run()* method is invoked by a `TestRunner` rather than by the end-user test harness.

Loading and running tests

class unittest.**TestLoader**

The *TestLoader* class is used to create test suites from classes and modules. Normally, there is no need to create an instance of this class; the *unittest* module provides an instance that can be shared as *unittest.defaultTestLoader*. Using a subclass or instance, however, allows customization of some configurable properties.

TestLoader objects have the following attributes:

errors

A list of the non-fatal errors encountered while loading tests. Not reset by the loader at any point. Fatal errors are signalled by the relevant method raising an exception to the caller. Non-fatal errors are also indicated by a synthetic test that will raise the original error when run.

Added in version 3.5.

TestLoader objects have the following methods:

loadTestsFromTestCase(*testCaseClass*)

Return a suite of all test cases contained in the *TestCase*-derived `testCaseClass`.

A test case instance is created for each method named by *getTestCaseNames()*. By default these are the method names beginning with `test`. If *getTestCaseNames()* returns no methods, but the `runTest()` method is implemented, a single test case is created for that method instead.

loadTestsFromModule(*module*, *, *pattern=None*)

Return a suite of all test cases contained in the given module. This method searches *module* for classes derived from *TestCase* and creates an instance of the class for each test method defined for the class.

> **Note**
>
> While using a hierarchy of *TestCase*-derived classes can be convenient in sharing fixtures and helper functions, defining test methods on base classes that are not intended to be instantiated directly does not play well with this method. Doing so, however, can be useful when the fixtures are different and defined in subclasses.

If a module provides a `load_tests` function it will be called to load the tests. This allows modules to customize test loading. This is the *load_tests protocol*. The *pattern* argument is passed as the third argument to `load_tests`.

Changed in version 3.2: Support for `load_tests` added.

Changed in version 3.5: Support for a keyword-only argument *pattern* has been added.

Changed in version 3.12: The undocumented and unofficial *use_load_tests* parameter has been removed.

loadTestsFromName(*name*, *module=None*)

Return a suite of all test cases given a string specifier.

The specifier *name* is a "dotted name" that may resolve either to a module, a test case class, a test method within a test case class, a *TestSuite* instance, or a callable object which returns a *TestCase* or *TestSuite* instance. These checks are applied in the order listed here; that is, a method on a possible test case class will be picked up as "a test method within a test case class", rather than "a callable object".

For example, if you have a module `SampleTests` containing a *TestCase*-derived class `SampleTestCase` with three test methods (`test_one()`, `test_two()`, and `test_three()`), the specifier `'SampleTests.SampleTestCase'` would cause this method to return a suite which will run all three test methods. Using the specifier `'SampleTests.SampleTestCase.test_two'` would cause it to return a test suite which will run only the `test_two()` test method. The specifier can refer to modules and packages which have not been imported; they will be imported as a side-effect.

The method optionally resolves *name* relative to the given *module*.

Changed in version 3.5: If an *ImportError* or *AttributeError* occurs while traversing *name* then a synthetic test that raises that error when run will be returned. These errors are included in the errors accumulated by self.errors.

loadTestsFromNames(*names*, *module=None*)

Similar to *loadTestsFromName()*, but takes a sequence of names rather than a single name. The return value is a test suite which supports all the tests defined for each name.

getTestCaseNames(*testCaseClass*)

Return a sorted sequence of method names found within *testCaseClass*; this should be a subclass of *TestCase*.

discover(*start_dir*, *pattern='test*.py'*, *top_level_dir=None*)

Find all the test modules by recursing into subdirectories from the specified start directory, and return a TestSuite object containing them. Only test files that match *pattern* will be loaded. (Using shell style pattern matching.) Only module names that are importable (i.e. are valid Python identifiers) will be loaded.

All test modules must be importable from the top level of the project. If the start directory is not the top level directory then *top_level_dir* must be specified separately.

If importing a module fails, for example due to a syntax error, then this will be recorded as a single error and discovery will continue. If the import failure is due to `SkipTest` being raised, it will be recorded as a skip instead of an error.

If a package (a directory containing a file named `__init__.py`) is found, the package will be checked for a `load_tests` function. If this exists then it will be called `package.load_tests(loader, tests, pattern)`. Test discovery takes care to ensure that a package is only checked for tests once during an invocation, even if the load_tests function itself calls `loader.discover`.

If `load_tests` exists then discovery does *not* recurse into the package, `load_tests` is responsible for loading all tests in the package.

The pattern is deliberately not stored as a loader attribute so that packages can continue discovery themselves.

top_level_dir is stored internally, and used as a default to any nested calls to `discover()`. That is, if a package's `load_tests` calls `loader.discover()`, it does not need to pass this argument.

start_dir can be a dotted module name as well as a directory.

Added in version 3.2.

Changed in version 3.4: Modules that raise `SkipTest` on import are recorded as skips, not errors.

Changed in version 3.4: *start_dir* can be a *namespace packages*.

Changed in version 3.4: Paths are sorted before being imported so that execution order is the same even if the underlying file system's ordering is not dependent on file name.

Changed in version 3.5: Found packages are now checked for `load_tests` regardless of whether their path matches *pattern*, because it is impossible for a package name to match the default pattern.

Changed in version 3.11: *start_dir* can not be a *namespace packages*. It has been broken since Python 3.7 and Python 3.11 officially remove it.

Changed in version 3.13: *top_level_dir* is only stored for the duration of *discover* call.

The following attributes of a `TestLoader` can be configured either by subclassing or assignment on an instance:

testMethodPrefix

String giving the prefix of method names which will be interpreted as test methods. The default value is `'test'`.

This affects `getTestCaseNames()` and all the `loadTestsFrom*` methods.

sortTestMethodsUsing

Function to be used to compare method names when sorting them in `getTestCaseNames()` and all the `loadTestsFrom*` methods.

suiteClass

Callable object that constructs a test suite from a list of tests. No methods on the resulting object are needed. The default value is the `TestSuite` class.

This affects all the `loadTestsFrom*` methods.

testNamePatterns

List of Unix shell-style wildcard test name patterns that test methods have to match to be included in test suites (see `-k` option).

If this attribute is not `None` (the default), all test methods to be included in test suites must match one of the patterns in this list. Note that matches are always performed using *fnmatch.fnmatchcase()*, so unlike patterns passed to the `-k` option, simple substring patterns will have to be converted using `*` wildcards.

This affects all the `loadTestsFrom*` methods.

Added in version 3.7.

class unittest.**TestResult**

This class is used to compile information about which tests have succeeded and which have failed.

A *TestResult* object stores the results of a set of tests. The *TestCase* and *TestSuite* classes ensure that results are properly recorded; test authors do not need to worry about recording the outcome of tests.

Testing frameworks built on top of *unittest* may want access to the *TestResult* object generated by running a set of tests for reporting purposes; a *TestResult* instance is returned by the `TestRunner.run()` method for this purpose.

TestResult instances have the following attributes that will be of interest when inspecting the results of running a set of tests:

errors

A list containing 2-tuples of *TestCase* instances and strings holding formatted tracebacks. Each tuple represents a test which raised an unexpected exception.

failures

A list containing 2-tuples of *TestCase* instances and strings holding formatted tracebacks. Each tuple represents a test where a failure was explicitly signalled using the *assert* methods*.

skipped

A list containing 2-tuples of *TestCase* instances and strings holding the reason for skipping the test.

Added in version 3.1.

expectedFailures

A list containing 2-tuples of *TestCase* instances and strings holding formatted tracebacks. Each tuple represents an expected failure or error of the test case.

unexpectedSuccesses

A list containing *TestCase* instances that were marked as expected failures, but succeeded.

collectedDurations

A list containing 2-tuples of test case names and floats representing the elapsed time of each test which was run.

Added in version 3.12.

shouldStop

Set to `True` when the execution of tests should stop by *stop()*.

testsRun

The total number of tests run so far.

buffer

If set to true, `sys.stdout` and `sys.stderr` will be buffered in between *startTest()* and *stopTest()* being called. Collected output will only be echoed onto the real `sys.stdout` and `sys.stderr` if the test fails or errors. Any output is also attached to the failure / error message.

Added in version 3.2.

failfast

If set to true *stop()* will be called on the first failure or error, halting the test run.

Added in version 3.2.

tb_locals

If set to true then local variables will be shown in tracebacks.

Added in version 3.5.

wasSuccessful()

Return `True` if all tests run so far have passed, otherwise returns `False`.

Changed in version 3.4: Returns `False` if there were any *unexpectedSuccesses* from tests marked with the *expectedFailure()* decorator.

stop()

This method can be called to signal that the set of tests being run should be aborted by setting the *shouldStop* attribute to `True`. `TestRunner` objects should respect this flag and return without running any additional tests.

For example, this feature is used by the *TextTestRunner* class to stop the test framework when the user signals an interrupt from the keyboard. Interactive tools which provide `TestRunner` implementations can use this in a similar manner.

The following methods of the *TestResult* class are used to maintain the internal data structures, and may be extended in subclasses to support additional reporting requirements. This is particularly useful in building tools which support interactive reporting while tests are being run.

startTest(*test*)

Called when the test case *test* is about to be run.

stopTest(*test*)

Called after the test case *test* has been executed, regardless of the outcome.

startTestRun()

Called once before any tests are executed.

Added in version 3.1.

stopTestRun()

Called once after all tests are executed.

Added in version 3.1.

addError(*test*, *err*)

Called when the test case *test* raises an unexpected exception. *err* is a tuple of the form returned by *sys.exc_info()*: `(type, value, traceback)`.

The default implementation appends a tuple `(test, formatted_err)` to the instance's *errors* attribute, where *formatted_err* is a formatted traceback derived from *err*.

addFailure(*test*, *err*)

Called when the test case *test* signals a failure. *err* is a tuple of the form returned by `sys.exc_info()`: `(type, value, traceback)`.

The default implementation appends a tuple `(test, formatted_err)` to the instance's `failures` attribute, where *formatted_err* is a formatted traceback derived from *err*.

addSuccess(*test*)

Called when the test case *test* succeeds.

The default implementation does nothing.

addSkip(*test*, *reason*)

Called when the test case *test* is skipped. *reason* is the reason the test gave for skipping.

The default implementation appends a tuple `(test, reason)` to the instance's `skipped` attribute.

addExpectedFailure(*test*, *err*)

Called when the test case *test* fails or errors, but was marked with the `expectedFailure()` decorator.

The default implementation appends a tuple `(test, formatted_err)` to the instance's `expectedFailures` attribute, where *formatted_err* is a formatted traceback derived from *err*.

addUnexpectedSuccess(*test*)

Called when the test case *test* was marked with the `expectedFailure()` decorator, but succeeded.

The default implementation appends the test to the instance's `unexpectedSuccesses` attribute.

addSubTest(*test*, *subtest*, *outcome*)

Called when a subtest finishes. *test* is the test case corresponding to the test method. *subtest* is a custom `TestCase` instance describing the subtest.

If *outcome* is `None`, the subtest succeeded. Otherwise, it failed with an exception where *outcome* is a tuple of the form returned by `sys.exc_info()`: `(type, value, traceback)`.

The default implementation does nothing when the outcome is a success, and records subtest failures as normal failures.

Added in version 3.4.

addDuration(*test*, *elapsed*)

Called when the test case finishes. *elapsed* is the time represented in seconds, and it includes the execution of cleanup functions.

Added in version 3.12.

class unittest.**TextTestResult**(*stream*, *descriptions*, *verbosity*, *, *durations=None*)

A concrete implementation of `TestResult` used by the `TextTestRunner`. Subclasses should accept `**kwargs` to ensure compatibility as the interface changes.

Added in version 3.2.

Changed in version 3.12: Added the *durations* keyword parameter.

unittest.**defaultTestLoader**

Instance of the `TestLoader` class intended to be shared. If no customization of the `TestLoader` is needed, this instance can be used instead of repeatedly creating new instances.

class unittest.**TextTestRunner**(*stream=None*, *descriptions=True*, *verbosity=1*, *failfast=False*, *buffer=False*, *resultclass=None*, *warnings=None*, *, *tb_locals=False*, *durations=None*)

A basic test runner implementation that outputs results to a stream. If *stream* is `None`, the default, `sys.stderr` is used as the output stream. This class has a few configurable parameters, but is essentially very simple. Graphical applications which run test suites should provide alternate implementations. Such implementations should accept `**kwargs` as the interface to construct runners changes when features are added to unittest.

By default this runner shows `DeprecationWarning`, `PendingDeprecationWarning`, `ResourceWarning` and `ImportWarning` even if they are *ignored by default*. This behavior can be overridden using Python's `-Wd` or `-Wa` options (see Warning control) and leaving *warnings* to `None`.

Changed in version 3.2: Added the *warnings* parameter.

Changed in version 3.2: The default stream is set to `sys.stderr` at instantiation time rather than import time.

Changed in version 3.5: Added the *tb_locals* parameter.

Changed in version 3.12: Added the *durations* parameter.

_makeResult()

This method returns the instance of `TestResult` used by `run()`. It is not intended to be called directly, but can be overridden in subclasses to provide a custom `TestResult`.

`_makeResult()` instantiates the class or callable passed in the `TextTestRunner` constructor as the `resultclass` argument. It defaults to `TextTestResult` if no `resultclass` is provided. The result class is instantiated with the following arguments:

```
stream, descriptions, verbosity
```

run(*test*)

This method is the main public interface to the `TextTestRunner`. This method takes a `TestSuite` or `TestCase` instance. A `TestResult` is created by calling `_makeResult()` and the test(s) are run and the results printed to stdout.

unittest.**main**(*module='__main__'*, *defaultTest=None*, *argv=None*, *testRunner=None*, *testLoader=unittest.defaultTestLoader*, *exit=True*, *verbosity=1*, *failfast=None*, *catchbreak=None*, *buffer=None*, *warnings=None*)

A command-line program that loads a set of tests from *module* and runs them; this is primarily for making test modules conveniently executable. The simplest use for this function is to include the following line at the end of a test script:

```
if __name__ == '__main__':
    unittest.main()
```

You can run tests with more detailed information by passing in the verbosity argument:

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

The *defaultTest* argument is either the name of a single test or an iterable of test names to run if no test names are specified via *argv*. If not specified or `None` and no test names are provided via *argv*, all tests found in *module* are run.

The *argv* argument can be a list of options passed to the program, with the first element being the program name. If not specified or `None`, the values of `sys.argv` are used.

The *testRunner* argument can either be a test runner class or an already created instance of it. By default `main` calls `sys.exit()` with an exit code indicating success (0) or failure (1) of the tests run. An exit code of 5 indicates that no tests were run or skipped.

The *testLoader* argument has to be a `TestLoader` instance, and defaults to `defaultTestLoader`.

`main` supports being used from the interactive interpreter by passing in the argument `exit=False`. This displays the result on standard output without calling *`sys.exit()`*:

```
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

The *failfast*, *catchbreak* and *buffer* parameters have the same effect as the same-name *command-line options*.

The *warnings* argument specifies the *warning filter* that should be used while running the tests. If it's not specified, it will remain `None` if a `-W` option is passed to **`python`** (see Warning control), otherwise it will be set to `'default'`.

Calling `main` returns an object with the `result` attribute that contains the result of the tests run as a *`unittest.TestResult`*.

Changed in version 3.1: The *exit* parameter was added.

Changed in version 3.2: The *verbosity*, *failfast*, *catchbreak*, *buffer* and *warnings* parameters were added.

Changed in version 3.4: The *defaultTest* parameter was changed to also accept an iterable of test names.

load_tests Protocol

Added in version 3.2.

Modules or packages can customize how tests are loaded from them during normal test runs or test discovery by implementing a function called `load_tests`.

If a test module defines `load_tests` it will be called by *`TestLoader.loadTestsFromModule()`* with the following arguments:

```
load_tests(loader, standard_tests, pattern)
```

where *pattern* is passed straight through from `loadTestsFromModule`. It defaults to `None`.

It should return a *`TestSuite`*.

loader is the instance of *`TestLoader`* doing the loading. *standard_tests* are the tests that would be loaded by default from the module. It is common for test modules to only want to add or remove tests from the standard set of tests. The third argument is used when loading packages as part of test discovery.

A typical `load_tests` function that loads tests from a specific set of *`TestCase`* classes may look like:

```
test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

If discovery is started in a directory containing a package, either from the command line or by calling *`TestLoader.discover()`*, then the package `__init__.py` will be checked for `load_tests`. If that function does not exist, discovery will recurse into the package as though it were just another directory. Otherwise, discovery of the package's tests will be left up to `load_tests` which is called with the following arguments:

```
load_tests(loader, standard_tests, pattern)
```

This should return a *TestSuite* representing all the tests from the package. (`standard_tests` will only contain tests collected from `__init__.py`.)

Because the pattern is passed into `load_tests` the package is free to continue (and potentially modify) test discovery. A 'do nothing' `load_tests` function for a test package would look like:

```
def load_tests(loader, standard_tests, pattern):
    # top level directory cached on loader instance
    this_dir = os.path.dirname(__file__)
    package_tests = loader.discover(start_dir=this_dir, pattern=pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```

Changed in version 3.5: Discovery no longer checks package names for matching *pattern* due to the impossibility of package names matching the default pattern.

26.5.9 Class and Module Fixtures

Class and module level fixtures are implemented in *TestSuite*. When the test suite encounters a test from a new class then `tearDownClass()` from the previous class (if there is one) is called, followed by `setUpClass()` from the new class.

Similarly if a test is from a different module from the previous test then `tearDownModule` from the previous module is run, followed by `setUpModule` from the new module.

After all the tests have run the final `tearDownClass` and `tearDownModule` are run.

Note that shared fixtures do not play well with [potential] features like test parallelization and they break test isolation. They should be used with care.

The default ordering of tests created by the unittest test loaders is to group all tests from the same modules and classes together. This will lead to `setUpClass` / `setUpModule` (etc) being called exactly once per class and module. If you randomize the order, so that tests from different modules and classes are adjacent to each other, then these shared fixture functions may be called multiple times in a single test run.

Shared fixtures are not intended to work with suites with non-standard ordering. A `BaseTestSuite` still exists for frameworks that don't want to support shared fixtures.

If there are any exceptions raised during one of the shared fixture functions the test is reported as an error. Because there is no corresponding test instance an `_ErrorHolder` object (that has the same interface as a *TestCase*) is created to represent the error. If you are just using the standard unittest test runner then this detail doesn't matter, but if you are a framework author it may be relevant.

setUpClass and tearDownClass

These must be implemented as class methods:

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

If you want the `setUpClass` and `tearDownClass` on base classes called then you must call up to them yourself. The implementations in *`TestCase`* are empty.

If an exception is raised during a `setUpClass` then the tests in the class are not run and the `tearDownClass` is not run. Skipped classes will not have `setUpClass` or `tearDownClass` run. If the exception is a *`SkipTest`* exception then the class will be reported as having been skipped instead of as an error.

setUpModule and tearDownModule

These should be implemented as functions:

```
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

If an exception is raised in a `setUpModule` then none of the tests in the module will be run and the `tearDownModule` will not be run. If the exception is a *`SkipTest`* exception then the module will be reported as having been skipped instead of as an error.

To add cleanup code that must be run even in the case of an exception, use `addModuleCleanup`:

`unittest.`**`addModuleCleanup`**(*function*, */*, **args*, ***kwargs*)

Add a function to be called after `tearDownModule()` to cleanup resources used during the test class. Functions will be called in reverse order to the order they are added (LIFO). They are called with any arguments and keyword arguments passed into *`addModuleCleanup()`* when they are added.

If `setUpModule()` fails, meaning that `tearDownModule()` is not called, then any cleanup functions added will still be called.

Added in version 3.8.

`classmethod` `unittest.`**`enterModuleContext`**(*cm*)

Enter the supplied *context manager*. If successful, also add its `__exit__()` method as a cleanup function by *`addModuleCleanup()`* and return the result of the `__enter__()` method.

Added in version 3.11.

`unittest.`**`doModuleCleanups`**()

This function is called unconditionally after `tearDownModule()`, or after `setUpModule()` if `setUpModule()` raises an exception.

It is responsible for calling all the cleanup functions added by *`addModuleCleanup()`*. If you need cleanup functions to be called *prior* to `tearDownModule()` then you can call *`doModuleCleanups()`* yourself.

`doModuleCleanups()` pops methods off the stack of cleanup functions one at a time, so it can be called at any time.

Added in version 3.8.

26.5.10 Signal Handling

Added in version 3.2.

The `-c/--catch` command-line option to unittest, along with the `catchbreak` parameter to `unittest.main()`, provide more friendly handling of control-C during a test run. With catch break behavior enabled control-C will allow the currently running test to complete, and the test run will then end and report all the results so far. A second control-c will raise a `KeyboardInterrupt` in the usual way.

The control-c handling signal handler attempts to remain compatible with code or tests that install their own `signal.SIGINT` handler. If the `unittest` handler is called but *isn't* the installed `signal.SIGINT` handler, i.e. it has been replaced by the system under test and delegated to, then it calls the default handler. This will normally be the expected behavior by code that replaces an installed handler and delegates to it. For individual tests that need `unittest` control-c handling disabled the `removeHandler()` decorator can be used.

There are a few utility functions for framework authors to enable control-c handling functionality within test frameworks.

`unittest.`**`installHandler`**`()`

Install the control-c handler. When a `signal.SIGINT` is received (usually in response to the user pressing control-c) all registered results have `stop()` called.

`unittest.`**`registerResult`**`(`*result*`)`

Register a `TestResult` object for control-c handling. Registering a result stores a weak reference to it, so it doesn't prevent the result from being garbage collected.

Registering a `TestResult` object has no side-effects if control-c handling is not enabled, so test frameworks can unconditionally register all results they create independently of whether or not handling is enabled.

`unittest.`**`removeResult`**`(`*result*`)`

Remove a registered result. Once a result has been removed then `stop()` will no longer be called on that result object in response to a control-c.

`unittest.`**`removeHandler`**`(`*function=None*`)`

When called without arguments this function removes the control-c handler if it has been installed. This function can also be used as a test decorator to temporarily remove the handler while the test is being executed:

```
@unittest.removeHandler
def test_signal_handling(self):
    ...
```

26.6 `unittest.mock` — mock object library

Added in version 3.3.

Source code: Lib/unittest/mock.py

`unittest.mock` is a library for testing in Python. It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used.

`unittest.mock` provides a core `Mock` class removing the need to create a host of stubs throughout your test suite. After performing an action, you can make assertions about which methods / attributes were used and arguments they were called with. You can also specify return values and set needed attributes in the normal way.

Additionally, mock provides a *patch()* decorator that handles patching module and class level attributes within the scope of a test, along with *sentinel* for creating unique objects. See the *quick guide* for some examples of how to use *Mock*, *MagicMock* and *patch()*.

Mock is designed for use with *unittest* and is based on the 'action -> assertion' pattern instead of 'record -> replay' used by many mocking frameworks.

There is a backport of *unittest.mock* for earlier versions of Python, available as mock on PyPI.

26.6.1 Quick Guide

Mock and *MagicMock* objects create all attributes and methods as you access them and store details of how they have been used. You can configure them, to specify return values or limit what attributes are available, and then make assertions about how they have been used:

```
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

`side_effect` allows you to perform side effects, including raising an exception when a mock is called:

```
>>> from unittest.mock import Mock
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
 ...
KeyError: 'foo'
```

```
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```

Mock has many other ways you can configure it and control its behaviour. For example the *spec* argument configures the mock to take its specification from another object. Attempting to access attributes or methods on the mock that don't exist on the spec will fail with an *AttributeError*.

The *patch()* decorator / context manager makes it easy to mock classes or objects in a module under test. The object you specify will be replaced with a mock (or other object) during the test and restored when the test ends:

```
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
...     module.ClassName2()
...     assert MockClass1 is module.ClassName1
```

(continues on next page)

(continued from previous page)

```
...     assert MockClass2 is module.ClassName2
...     assert MockClass1.called
...     assert MockClass2.called
...
>>> test()
```

Note

When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal *Python* order that decorators are applied). This means from the bottom up, so in the example above the mock for `module.ClassName1` is passed in first.

With *patch()* it matters that you patch objects in the namespace where they are looked up. This is normally straightforward, but for a quick guide read *where to patch*.

As well as a decorator *patch()* can be used as a context manager in a with statement:

```
>>> with patch.object(ProductionClass, 'method', return_value=None) as mock_method:
...     thing = ProductionClass()
...     thing.method(1, 2, 3)
...
>>> mock_method.assert_called_once_with(1, 2, 3)
```

There is also *patch.dict()* for setting values in a dictionary just during a scope and restoring the dictionary to its original state when the test ends:

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

Mock supports the mocking of Python *magic methods*. The easiest way of using magic methods is with the *MagicMock* class. It allows you to do things like:

```
>>> mock = MagicMock()
>>> mock.__str__.return_value = 'foobarbaz'
>>> str(mock)
'foobarbaz'
>>> mock.__str__.assert_called_with()
```

Mock allows you to assign functions (or other Mock instances) to magic methods and they will be called appropriately. The *MagicMock* class is just a Mock variant that has all of the magic methods pre-created for you (well, all the useful ones anyway).

The following is an example of using magic methods with the ordinary Mock class:

```
>>> mock = Mock()
>>> mock.__str__ = Mock(return_value='wheeeeee')
>>> str(mock)
'wheeeeee'
```

For ensuring that the mock objects in your tests have the same api as the objects they are replacing, you can use *auto-speccing*. Auto-speccing can be done through the *autospec* argument to patch, or the *create_autospec()* function.

Auto-speccing creates mock objects that have the same attributes and methods as the objects they are replacing, and any functions and methods (including constructors) have the same call signature as the real object.

This ensures that your mocks will fail in the same way as your production code if they are used incorrectly:

```
>>> from unittest.mock import create_autospec
>>> def function(a, b, c):
...     pass
...
>>> mock_function = create_autospec(function, return_value='fishy')
>>> mock_function(1, 2, 3)
'fishy'
>>> mock_function.assert_called_once_with(1, 2, 3)
>>> mock_function('wrong arguments')
Traceback (most recent call last):
 ...
TypeError: missing a required argument: 'b'
```

create_autospec() can also be used on classes, where it copies the signature of the __init__ method, and on callable objects where it copies the signature of the __call__ method.

26.6.2 The Mock Class

Mock is a flexible mock object intended to replace the use of stubs and test doubles throughout your code. Mocks are callable and create attributes as new mocks when you access them[1]. Accessing the same attribute will always return the same mock. Mocks record how you use them, allowing you to make assertions about what your code has done to them.

MagicMock is a subclass of *Mock* with all the magic methods pre-created and ready to use. There are also non-callable variants, useful when you are mocking out objects that aren't callable: *NonCallableMock* and *NonCallableMagicMock*

The *patch()* decorators makes it easy to temporarily replace classes in a particular module with a *Mock* object. By default *patch()* will create a *MagicMock* for you. You can specify an alternative class of *Mock* using the *new_callable* argument to *patch()*.

class unittest.mock.**Mock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***kwargs*)

Create a new *Mock* object. *Mock* takes several optional arguments that specify the behaviour of the Mock object:

- *spec*: This can be either a list of strings or an existing object (a class or instance) that acts as the specification for the mock object. If you pass in an object then a list of strings is formed by calling dir on the object (excluding unsupported magic attributes and methods). Accessing any attribute not in this list will raise an *AttributeError*.

  If *spec* is an object (rather than a list of strings) then *__class__* returns the class of the spec object. This allows mocks to pass *isinstance()* tests.

- *spec_set*: A stricter variant of *spec*. If used, attempting to *set* or get an attribute on the mock that isn't on the object passed as *spec_set* will raise an *AttributeError*.

- *side_effect*: A function to be called whenever the Mock is called. See the *side_effect* attribute. Useful for raising exceptions or dynamically changing return values. The function is called with the same arguments as the mock, and unless it returns *DEFAULT*, the return value of this function is used as the return value.

[1] The only exceptions are magic methods and attributes (those that have leading and trailing double underscores). Mock doesn't create these but instead raises an *AttributeError*. This is because the interpreter will often implicitly request these methods, and gets *very* confused to get a new Mock object when it expects a magic method. If you need magic method support see *magic methods*.

Alternatively *side_effect* can be an exception class or instance. In this case the exception will be raised when the mock is called.

If *side_effect* is an iterable then each call to the mock will return the next value from the iterable.

A *side_effect* can be cleared by setting it to `None`.

- *return_value*: The value returned when the mock is called. By default this is a new Mock (created on first access). See the *return_value* attribute.
- *unsafe*: By default, accessing any attribute whose name starts with *assert*, *assret*, *asert*, *aseert* or *assrt* will raise an *AttributeError*. Passing `unsafe=True` will allow access to these attributes.

  Added in version 3.5.
- *wraps*: Item for the mock object to wrap. If *wraps* is not `None` then calling the Mock will pass the call through to the wrapped object (returning the real result). Attribute access on the mock will return a Mock object that wraps the corresponding attribute of the wrapped object (so attempting to access an attribute that doesn't exist will raise an *AttributeError*).

  If the mock has an explicit *return_value* set then calls are not passed to the wrapped object and the *return_value* is returned instead.
- *name*: If the mock has a name then it will be used in the repr of the mock. This can be useful for debugging. The name is propagated to child mocks.

Mocks can also be called with arbitrary keyword arguments. These will be used to set attributes on the mock after it is created. See the *configure_mock()* method for details.

assert_called()

Assert that the mock was called at least once.

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called()
```

Added in version 3.6.

assert_called_once()

Assert that the mock was called exactly once.

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
Traceback (most recent call last):
...
AssertionError: Expected 'method' to have been called once. Called 2 times.
Calls: [call(), call()].
```

Added in version 3.6.

assert_called_with(**args*, ***kwargs*)

This method is a convenient way of asserting that the last call has been made in a particular way:

```
>>> mock = Mock()
>>> mock.method(1, 2, 3, test='wow')
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_with(1, 2, 3, test='wow')
```

assert_called_once_with(**args*, ***kwargs*)

Assert that the mock was called exactly once and that call was with the specified arguments.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
>>> mock('other', bar='values')
>>> mock.assert_called_once_with('other', bar='values')
Traceback (most recent call last):
  ...
AssertionError: Expected 'mock' to be called once. Called 2 times.
Calls: [call('foo', bar='baz'), call('other', bar='values')].
```

assert_any_call(**args*, ***kwargs*)

assert the mock has been called with the specified arguments.

The assert passes if the mock has *ever* been called, unlike *assert_called_with()* and *assert_called_once_with()* that only pass if the call is the most recent one, and in the case of *assert_called_once_with()* it must also be the only call.

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, arg='thing')
>>> mock('some', 'thing', 'else')
>>> mock.assert_any_call(1, 2, arg='thing')
```

assert_has_calls(*calls*, *any_order=False*)

assert the mock has been called with the specified calls. The *mock_calls* list is checked for the calls.

If *any_order* is false then the calls must be sequential. There can be extra calls before or after the specified calls.

If *any_order* is true then the calls can be in any order, but they must all appear in *mock_calls*.

```
>>> mock = Mock(return_value=None)
>>> mock(1)
>>> mock(2)
>>> mock(3)
>>> mock(4)
>>> calls = [call(2), call(3)]
>>> mock.assert_has_calls(calls)
>>> calls = [call(4), call(2), call(3)]
>>> mock.assert_has_calls(calls, any_order=True)
```

assert_not_called()

Assert the mock was never called.

```
>>> m = Mock()
>>> m.hello.assert_not_called()
>>> obj = m.hello()
>>> m.hello.assert_not_called()
Traceback (most recent call last):
  ...
```

(continues on next page)

(continued from previous page)

```
AssertionError: Expected 'hello' to not have been called. Called 1 times.
Calls: [call()].
```

Added in version 3.5.

reset_mock(*, *return_value=False*, *side_effect=False*)

The reset_mock method resets all the call attributes on a mock object:

```
>>> mock = Mock(return_value=None)
>>> mock('hello')
>>> mock.called
True
>>> mock.reset_mock()
>>> mock.called
False
```

Changed in version 3.6: Added two keyword-only arguments to the reset_mock function.

This can be useful where you want to make a series of assertions that reuse the same object. Note that *reset_mock() doesn't* clear the *return_value*, *side_effect* or any child attributes you have set using normal assignment by default. In case you want to reset *return_value* or *side_effect*, then pass the corresponding parameter as `True`. Child mocks and the return value mock (if any) are reset as well.

Note

return_value, and *side_effect* are keyword-only arguments.

mock_add_spec(*spec*, *spec_set=False*)

Add a spec to a mock. *spec* can either be an object or a list of strings. Only attributes on the *spec* can be fetched as attributes from the mock.

If *spec_set* is true then only attributes on the spec can be set.

attach_mock(*mock*, *attribute*)

Attach a mock as an attribute of this one, replacing its name and parent. Calls to the attached mock will be recorded in the *method_calls* and *mock_calls* attributes of this one.

configure_mock(***kwargs*)

Set attributes on the mock through keyword arguments.

Attributes plus return values and side effects can be set on child mocks using standard dot notation and unpacking a dictionary in the method call:

```
>>> mock = Mock()
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock.configure_mock(**attrs)
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

The same thing can be achieved in the constructor call to mocks:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

configure_mock() exists to make it easier to do configuration after the mock has been created.

__dir__()

Mock objects limit the results of `dir(some_mock)` to useful results. For mocks with a *spec* this includes all the permitted attributes for the mock.

See *FILTER_DIR* for what this filtering does, and how to switch it off.

_get_child_mock(**kw*)

Create the child mocks for attributes and return value. By default child mocks will be the same type as the parent. Subclasses of Mock may want to override this to customize the way child mocks are made.

For non-callable mocks the callable variant will be used (rather than any custom subclass).

called

A boolean representing whether or not the mock object has been called:

```
>>> mock = Mock(return_value=None)
>>> mock.called
False
>>> mock()
>>> mock.called
True
```

call_count

An integer telling you how many times the mock object has been called:

```
>>> mock = Mock(return_value=None)
>>> mock.call_count
0
>>> mock()
>>> mock()
>>> mock.call_count
2
```

return_value

Set this to configure the value returned by calling the mock:

```
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish'
```

The default return value is a mock object and you can configure it in the normal way:

```
>>> mock = Mock()
>>> mock.return_value.attribute = sentinel.Attribute
>>> mock.return_value()
<Mock name='mock()()' id='...'>
>>> mock.return_value.assert_called_with()
```

return_value can also be set in the constructor:

```
>>> mock = Mock(return_value=3)
>>> mock.return_value
3
>>> mock()
3
```

side_effect

This can either be a function to be called when the mock is called, an iterable or an exception (class or instance) to be raised.

If you pass in a function it will be called with same arguments as the mock and unless the function returns the *DEFAULT* singleton the call to the mock will then return whatever the function returns. If the function returns *DEFAULT* then the mock will return its normal value (from the *return_value*).

If you pass in an iterable, it is used to retrieve an iterator which must yield a value on every call. This value can either be an exception instance to be raised, or a value to be returned from the call to the mock (*DEFAULT* handling is identical to the function case).

An example of a mock that raises an exception (to test exception handling of an API):

```
>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

Using *side_effect* to return a sequence of values:

```
>>> mock = Mock()
>>> mock.side_effect = [3, 2, 1]
>>> mock(), mock(), mock()
(3, 2, 1)
```

Using a callable:

```
>>> mock = Mock(return_value=3)
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> mock.side_effect = side_effect
>>> mock()
3
```

side_effect can be set in the constructor. Here's an example that adds one to the value the mock is called with and returns it:

```
>>> side_effect = lambda value: value + 1
>>> mock = Mock(side_effect=side_effect)
```

(continues on next page)

(continued from previous page)

```
>>> mock(3)
4
>>> mock(-8)
-7
```

Setting *side_effect* to `None` clears it:

```
>>> m = Mock(side_effect=KeyError, return_value=3)
>>> m()
Traceback (most recent call last):
 ...
KeyError
>>> m.side_effect = None
>>> m()
3
```

call_args

This is either `None` (if the mock hasn't been called), or the arguments that the mock was last called with. This will be in the form of a tuple: the first member, which can also be accessed through the `args` property, is any ordered arguments the mock was called with (or an empty tuple) and the second member, which can also be accessed through the `kwargs` property, is any keyword arguments (or an empty dictionary).

```
>>> mock = Mock(return_value=None)
>>> print(mock.call_args)
None
>>> mock()
>>> mock.call_args
call()
>>> mock.call_args == ()
True
>>> mock(3, 4)
>>> mock.call_args
call(3, 4)
>>> mock.call_args == ((3, 4),)
True
>>> mock.call_args.args
(3, 4)
>>> mock.call_args.kwargs
{}
>>> mock(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args
call(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args.args
(3, 4, 5)
>>> mock.call_args.kwargs
{'key': 'fish', 'next': 'w00t!'}
```

call_args, along with members of the lists *call_args_list*, *method_calls* and *mock_calls* are *call* objects. These are tuples, so they can be unpacked to get at the individual arguments and make more complex assertions. See *calls as tuples*.

Changed in version 3.8: Added `args` and `kwargs` properties.

call_args_list

This is a list of all the calls made to the mock object in sequence (so the length of the list is the number of times it has been called). Before any calls have been made it is an empty list. The *call* object can be used for conveniently constructing lists of calls to compare with *call_args_list*.

```
>>> mock = Mock(return_value=None)
>>> mock()
>>> mock(3, 4)
>>> mock(key='fish', next='w00t!')
>>> mock.call_args_list
[call(), call(3, 4), call(key='fish', next='w00t!')]
>>> expected = [(), ((3, 4),), ({'key': 'fish', 'next': 'w00t!'},)]
>>> mock.call_args_list == expected
True
```

Members of *call_args_list* are *call* objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

method_calls

As well as tracking calls to themselves, mocks also track calls to methods and attributes, and *their* methods and attributes:

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.property.method.attribute()
<Mock name='mock.property.method.attribute()' id='...'>
>>> mock.method_calls
[call.method(), call.property.method.attribute()]
```

Members of *method_calls* are *call* objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

mock_calls

mock_calls records *all* calls to the mock object, its methods, magic methods *and* return value mocks.

```
>>> mock = MagicMock()
>>> result = mock(1, 2, 3)
>>> mock.first(a=3)
<MagicMock name='mock.first()' id='...'>
>>> mock.second()
<MagicMock name='mock.second()' id='...'>
>>> int(mock)
1
>>> result(1)
<MagicMock name='mock()()' id='...'>
>>> expected = [call(1, 2, 3), call.first(a=3), call.second(),
... call.__int__(), call()(1)]
>>> mock.mock_calls == expected
True
```

Members of *mock_calls* are *call* objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

Note

The way *mock_calls* are recorded means that where nested calls are made, the parameters of ancestor calls are not recorded and so will always compare equal:

```
>>> mock = MagicMock()
>>> mock.top(a=3).bottom()
<MagicMock name='mock.top().bottom()' id='...'>
```

```
>>> mock.mock_calls
[call.top(a=3), call.top().bottom()]
>>> mock.mock_calls[-1] == call.top(a=-1).bottom()
True
```

__class__

Normally the *__class__* attribute of an object will return its type. For a mock object with a `spec`, `__class__` returns the spec class instead. This allows mock objects to pass *isinstance()* tests for the object they are replacing / masquerading as:

```
>>> mock = Mock(spec=3)
>>> isinstance(mock, int)
True
```

__class__ is assignable to, this allows a mock to pass an *isinstance()* check without forcing you to use a spec:

```
>>> mock = Mock()
>>> mock.__class__ = dict
>>> isinstance(mock, dict)
True
```

class unittest.mock.**NonCallableMock**(*spec=None*, *wraps=None*, *name=None*, *spec_set=None*, ***kwargs*)

A non-callable version of *Mock*. The constructor parameters have the same meaning of *Mock*, with the exception of *return_value* and *side_effect* which have no meaning on a non-callable mock.

Mock objects that use a class or an instance as a `spec` or `spec_set` are able to pass *isinstance()* tests:

```
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
>>> isinstance(mock, SomeClass)
True
```

The *Mock* classes have support for mocking magic methods. See *magic methods* for the full details.

The mock classes and the *patch()* decorators all take arbitrary keyword arguments for configuration. For the *patch()* decorators the keywords are passed to the constructor of the mock being created. The keyword arguments are for configuring attributes of the mock:

```
>>> m = MagicMock(attribute=3, other='fish')
>>> m.attribute
3
>>> m.other
'fish'
```

The return value and side effect of child mocks can be set in the same way, using dotted notation. As you can't use dotted names directly in a call you have to create a dictionary and unpack it using `**`:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
```

(continues on next page)

(continued from previous page)

```
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

A callable mock which was created with a *spec* (or a *spec_set*) will introspect the specification object's signature when matching calls to the mock. Therefore, it can match the actual call's arguments regardless of whether they were passed positionally or by name:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, c=3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(1, 2, 3)
>>> mock.assert_called_with(a=1, b=2, c=3)
```

This applies to *assert_called_with()*, *assert_called_once_with()*, *assert_has_calls()* and *assert_any_call()*. When *Autospeccing*, it will also apply to method calls on the mock object.

Changed in version 3.4: Added signature introspection on specced and autospecced mock objects.

class unittest.mock.**PropertyMock**(**args*, ***kwargs*)

A mock intended to be used as a *property*, or other *descriptor*, on a class. *PropertyMock* provides `__get__()` and `__set__()` methods so you can specify a return value when it is fetched.

Fetching a *PropertyMock* instance from an object calls the mock, with no args. Setting it calls the mock with the value being set.

```
>>> class Foo:
...     @property
...     def foo(self):
...         return 'something'
...     @foo.setter
...     def foo(self, value):
...         pass
...
>>> with patch('__main__.Foo.foo', new_callable=PropertyMock) as mock_foo:
...     mock_foo.return_value = 'mockity-mock'
...     this_foo = Foo()
...     print(this_foo.foo)
...     this_foo.foo = 6
...
mockity-mock
>>> mock_foo.mock_calls
[call(), call(6)]
```

Because of the way mock attributes are stored you can't directly attach a *PropertyMock* to a mock object. Instead you can attach it to the mock type object:

```
>>> m = MagicMock()
>>> p = PropertyMock(return_value=3)
>>> type(m).foo = p
>>> m.foo
```

(continues on next page)

(continued from previous page)

```
3
>>> p.assert_called_once_with()
```

Caution

If an *AttributeError* is raised by *PropertyMock*, it will be interpreted as a missing descriptor and `__getattr__()` will be called on the parent mock:

```
>>> m = MagicMock()
>>> no_attribute = PropertyMock(side_effect=AttributeError)
>>> type(m).my_property = no_attribute
>>> m.my_property
<MagicMock name='mock.my_property' id='140165240345424'>
```

See `__getattr__()` for details.

class unittest.mock.**AsyncMock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***kwargs*)

An asynchronous version of *MagicMock*. The *AsyncMock* object will behave so the object is recognized as an async function, and the result of a call is an awaitable.

```
>>> mock = AsyncMock()
>>> asyncio.iscoroutinefunction(mock)
True
>>> inspect.isawaitable(mock())
True
```

The result of `mock()` is an async function which will have the outcome of `side_effect` or `return_value` after it has been awaited:

- if `side_effect` is a function, the async function will return the result of that function,
- if `side_effect` is an exception, the async function will raise the exception,
- if `side_effect` is an iterable, the async function will return the next value of the iterable, however, if the sequence of result is exhausted, `StopAsyncIteration` is raised immediately,
- if `side_effect` is not defined, the async function will return the value defined by `return_value`, hence, by default, the async function returns a new *AsyncMock* object.

Setting the *spec* of a *Mock* or *MagicMock* to an async function will result in a coroutine object being returned after calling.

```
>>> async def async_func(): pass
...
>>> mock = MagicMock(async_func)
>>> mock
<MagicMock spec='function' id='...'>
>>> mock()
<coroutine object AsyncMockMixin._mock_call at ...>
```

Setting the *spec* of a *Mock*, *MagicMock*, or *AsyncMock* to a class with asynchronous and synchronous functions will automatically detect the synchronous functions and set them as *MagicMock* (if the parent mock is *AsyncMock* or *MagicMock*) or *Mock* (if the parent mock is *Mock*). All asynchronous functions will be *AsyncMock*.

```
>>> class ExampleClass:
...     def sync_foo():
...         pass
...     async def async_foo():
...         pass
...
>>> a_mock = AsyncMock(ExampleClass)
>>> a_mock.sync_foo
<MagicMock name='mock.sync_foo' id='...'>
>>> a_mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
>>> mock = Mock(ExampleClass)
>>> mock.sync_foo
<Mock name='mock.sync_foo' id='...'>
>>> mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
```

Added in version 3.8.

assert_awaited()

Assert that the mock was awaited at least once. Note that this is separate from the object having been called, the `await` keyword must be used:

```
>>> mock = AsyncMock()
>>> async def main(coroutine_mock):
...     await coroutine_mock
...
>>> coroutine_mock = mock()
>>> mock.called
True
>>> mock.assert_awaited()
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited.
>>> asyncio.run(main(coroutine_mock))
>>> mock.assert_awaited()
```

assert_awaited_once()

Assert that the mock was awaited exactly once.

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.assert_awaited_once()
>>> asyncio.run(main())
>>> mock.assert_awaited_once()
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

assert_awaited_with(**args*, ***kwargs*)

Assert that the last await was with the specified arguments.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_with('foo', bar='bar')
>>> mock.assert_awaited_with('other')
Traceback (most recent call last):
...
AssertionError: expected await not found.
Expected: mock('other')
Actual: mock('foo', bar='bar')
```

assert_awaited_once_with(**args*, ***kwargs*)

Assert that the mock was awaited exactly once and with the specified arguments.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_once_with('foo', bar='bar')
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_once_with('foo', bar='bar')
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

assert_any_await(**args*, ***kwargs*)

Assert the mock has ever been awaited with the specified arguments.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> asyncio.run(main('hello'))
>>> mock.assert_any_await('foo', bar='bar')
>>> mock.assert_any_await('other')
Traceback (most recent call last):
...
AssertionError: mock('other') await not found
```

assert_has_awaits(*calls*, *any_order=False*)

Assert the mock has been awaited with the specified calls. The *await_args_list* list is checked for the awaits.

If *any_order* is false then the awaits must be sequential. There can be extra calls before or after the specified awaits.

If *any_order* is true then the awaits can be in any order, but they must all appear in *await_args_list*.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> calls = [call("foo"), call("bar")]
>>> mock.assert_has_awaits(calls)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
AssertionError: Awaits not found.
Expected: [call('foo'), call('bar')]
Actual: []
>>> asyncio.run(main('foo'))
>>> asyncio.run(main('bar'))
>>> mock.assert_has_awaits(calls)
```

assert_not_awaited()

Assert that the mock was never awaited.

```
>>> mock = AsyncMock()
>>> mock.assert_not_awaited()
```

reset_mock(**args*, ***kwargs*)

See *Mock.reset_mock()*. Also sets *await_count* to 0, *await_args* to None, and clears the *await_args_list*.

await_count

An integer keeping track of how many times the mock object has been awaited.

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.await_count
1
>>> asyncio.run(main())
>>> mock.await_count
2
```

await_args

This is either `None` (if the mock hasn't been awaited), or the arguments that the mock was last awaited with. Functions the same as *Mock.call_args*.

```
>>> mock = AsyncMock()
>>> async def main(*args):
...     await mock(*args)
...
>>> mock.await_args
>>> asyncio.run(main('foo'))
>>> mock.await_args
call('foo')
>>> asyncio.run(main('bar'))
>>> mock.await_args
call('bar')
```

await_args_list

This is a list of all the awaits made to the mock object in sequence (so the length of the list is the number of times it has been awaited). Before any awaits have been made it is an empty list.

```
>>> mock = AsyncMock()
>>> async def main(*args):
```

(continues on next page)

(continued from previous page)

```
...     await mock(*args)
...
>>> mock.await_args_list
[]
>>> asyncio.run(main('foo'))
>>> mock.await_args_list
[call('foo')]
>>> asyncio.run(main('bar'))
>>> mock.await_args_list
[call('foo'), call('bar')]
```

class unittest.mock.**ThreadingMock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***, *timeout=UNSET*, ***kwargs*)

A version of *MagicMock* for multithreading tests. The *ThreadingMock* object provides extra methods to wait for a call to be invoked, rather than assert on it immediately.

The default timeout is specified by the `timeout` argument, or if unset by the *ThreadingMock.DEFAULT_TIMEOUT* attribute, which defaults to blocking (`None`).

You can configure the global default timeout by setting *ThreadingMock.DEFAULT_TIMEOUT*.

wait_until_called(***, *timeout=UNSET*)

Waits until the mock is called.

If a timeout was passed at the creation of the mock or if a timeout argument is passed to this function, the function raises an *AssertionError* if the call is not performed in time.

```
>>> mock = ThreadingMock()
>>> thread = threading.Thread(target=mock)
>>> thread.start()
>>> mock.wait_until_called(timeout=1)
>>> thread.join()
```

wait_until_any_call_with(**args*, ***kwargs*)

Waits until the mock is called with the specified arguments.

If a timeout was passed at the creation of the mock the function raises an *AssertionError* if the call is not performed in time.

```
>>> mock = ThreadingMock()
>>> thread = threading.Thread(target=mock, args=("arg1", "arg2",), kwargs={
↪"arg": "thing"})
>>> thread.start()
>>> mock.wait_until_any_call_with("arg1", "arg2", arg="thing")
>>> thread.join()
```

DEFAULT_TIMEOUT

Global default timeout in seconds to create instances of *ThreadingMock*.

Added in version 3.13.

Calling

Mock objects are callable. The call will return the value set as the *return_value* attribute. The default return value is a new Mock object; it is created the first time the return value is accessed (either explicitly or by calling the Mock) - but it is stored and the same one returned each time.

Calls made to the object will be recorded in the attributes like *call_args* and *call_args_list*.

If *side_effect* is set then it will be called after the call has been recorded, so if `side_effect` raises an exception the call is still recorded.

The simplest way to make a mock raise an exception when called is to make *side_effect* an exception class or instance:

```
>>> m = MagicMock(side_effect=IndexError)
>>> m(1, 2, 3)
Traceback (most recent call last):
  ...
IndexError
>>> m.mock_calls
[call(1, 2, 3)]
>>> m.side_effect = KeyError('Bang!')
>>> m('two', 'three', 'four')
Traceback (most recent call last):
  ...
KeyError: 'Bang!'
>>> m.mock_calls
[call(1, 2, 3), call('two', 'three', 'four')]
```

If `side_effect` is a function then whatever that function returns is what calls to the mock return. The `side_effect` function is called with the same arguments as the mock. This allows you to vary the return value of the call dynamically, based on the input:

```
>>> def side_effect(value):
...     return value + 1
...
>>> m = MagicMock(side_effect=side_effect)
>>> m(1)
2
>>> m(2)
3
>>> m.mock_calls
[call(1), call(2)]
```

If you want the mock to still return the default return value (a new mock), or any set return value, then there are two ways of doing this. Either return `mock.return_value` from inside `side_effect`, or return *DEFAULT*:

```
>>> m = MagicMock()
>>> def side_effect(*args, **kwargs):
...     return m.return_value
...
>>> m.side_effect = side_effect
>>> m.return_value = 3
>>> m()
3
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
```

(continues on next page)

(continued from previous page)

```
>>> m.side_effect = side_effect
>>> m()
3
```

To remove a `side_effect`, and return to the default behaviour, set the `side_effect` to `None`:

```
>>> m = MagicMock(return_value=6)
>>> def side_effect(*args, **kwargs):
...     return 3
...
>>> m.side_effect = side_effect
>>> m()
3
>>> m.side_effect = None
>>> m()
6
```

The `side_effect` can also be any iterable object. Repeated calls to the mock will return values from the iterable (until the iterable is exhausted and a *`StopIteration`* is raised):

```
>>> m = MagicMock(side_effect=[1, 2, 3])
>>> m()
1
>>> m()
2
>>> m()
3
>>> m()
Traceback (most recent call last):
  ...
StopIteration
```

If any members of the iterable are exceptions they will be raised instead of returned:

```
>>> iterable = (33, ValueError, 66)
>>> m = MagicMock(side_effect=iterable)
>>> m()
33
>>> m()
Traceback (most recent call last):
 ...
ValueError
>>> m()
66
```

Deleting Attributes

Mock objects create attributes on demand. This allows them to pretend to be objects of any type.

You may want a mock object to return `False` to a *`hasattr()`* call, or raise an *`AttributeError`* when an attribute is fetched. You can do this by providing an object as a `spec` for a mock, but that isn't always convenient.

You "block" attributes by deleting them. Once deleted, accessing an attribute will raise an *`AttributeError`*.

```
>>> mock = MagicMock()
>>> hasattr(mock, 'm')
True
>>> del mock.m
>>> hasattr(mock, 'm')
False
>>> del mock.f
>>> mock.f
Traceback (most recent call last):
    ...
AttributeError: f
```

Mock names and the name attribute

Since "name" is an argument to the *Mock* constructor, if you want your mock object to have a "name" attribute you can't just pass it in at creation time. There are two alternatives. One option is to use *configure_mock()*:

```
>>> mock = MagicMock()
>>> mock.configure_mock(name='my_name')
>>> mock.name
'my_name'
```

A simpler option is to simply set the "name" attribute after mock creation:

```
>>> mock = MagicMock()
>>> mock.name = "foo"
```

Attaching Mocks as Attributes

When you attach a mock as an attribute of another mock (or as the return value) it becomes a "child" of that mock. Calls to the child are recorded in the *method_calls* and *mock_calls* attributes of the parent. This is useful for configuring child mocks and then attaching them to the parent, or for attaching mocks to a parent that records all calls to the children and allows you to make assertions about the order of calls between mocks:

```
>>> parent = MagicMock()
>>> child1 = MagicMock(return_value=None)
>>> child2 = MagicMock(return_value=None)
>>> parent.child1 = child1
>>> parent.child2 = child2
>>> child1(1)
>>> child2(2)
>>> parent.mock_calls
[call.child1(1), call.child2(2)]
```

The exception to this is if the mock has a name. This allows you to prevent the "parenting" if for some reason you don't want it to happen.

```
>>> mock = MagicMock()
>>> not_a_child = MagicMock(name='not-a-child')
>>> mock.attribute = not_a_child
>>> mock.attribute()
<MagicMock name='not-a-child()' id='...'>
>>> mock.mock_calls
[]
```

Mocks created for you by *patch()* are automatically given names. To attach mocks that have names to a parent you use the *attach_mock()* method:

```
>>> thing1 = object()
>>> thing2 = object()
>>> parent = MagicMock()
>>> with patch('__main__.thing1', return_value=None) as child1:
...     with patch('__main__.thing2', return_value=None) as child2:
...         parent.attach_mock(child1, 'child1')
...         parent.attach_mock(child2, 'child2')
...         child1('one')
...         child2('two')
...
>>> parent.mock_calls
[call.child1('one'), call.child2('two')]
```

26.6.3 The patchers

The patch decorators are used for patching objects only within the scope of the function they decorate. They automatically handle the unpatching for you, even if exceptions are raised. All of these functions can also be used in with statements or as class decorators.

patch

Note

The key is to do the patching in the right namespace. See the section *where to patch*.

unittest.mock.**patch**(*target*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

patch() acts as a function decorator, class decorator or a context manager. Inside the body of the function or with statement, the *target* is patched with a *new* object. When the function/with statement exits the patch is undone.

If *new* is omitted, then the target is replaced with an *AsyncMock* if the patched object is an async function or a *MagicMock* otherwise. If *patch()* is used as a decorator and *new* is omitted, the created mock is passed in as an extra argument to the decorated function. If *patch()* is used as a context manager the created mock is returned by the context manager.

target should be a string in the form `'package.module.ClassName'`. The *target* is imported and the specified object replaced with the *new* object, so the *target* must be importable from the environment you are calling *patch()* from. The target is imported when the decorated function is executed, not at decoration time.

The *spec* and *spec_set* keyword arguments are passed to the *MagicMock* if patch is creating one for you.

In addition you can pass `spec=True` or `spec_set=True`, which causes patch to pass in the object being mocked as the spec/spec_set object.

new_callable allows you to specify a different class, or callable object, that will be called to create the *new* object. By default *AsyncMock* is used for async functions and *MagicMock* for the rest.

A more powerful form of *spec* is *autospec*. If you set `autospec=True` then the mock will be created with a spec from the object being replaced. All attributes of the mock will also have the spec of the corresponding attribute of the object being replaced. Methods and functions being mocked will have their arguments checked and will raise

a *TypeError* if they are called with the wrong signature. For mocks replacing a class, their return value (the 'instance') will have the same spec as the class. See the *create_autospec()* function and *Autospeccing*.

Instead of `autospec=True` you can pass `autospec=some_object` to use an arbitrary object as the spec instead of the one being replaced.

By default *patch()* will fail to replace attributes that don't exist. If you pass in `create=True`, and the attribute doesn't exist, patch will create the attribute for you when the patched function is called, and delete it again after the patched function has exited. This is useful for writing tests against attributes that your production code creates at runtime. It is off by default because it can be dangerous. With it switched on you can write passing tests against APIs that don't actually exist!

Note

Changed in version 3.5: If you are patching builtins in a module then you don't need to pass `create=True`, it will be added by default.

Patch can be used as a `TestCase` class decorator. It works by decorating each test method in the class. This reduces the boilerplate code when your test methods share a common patchings set. *patch()* finds tests by looking for method names that start with `patch.TEST_PREFIX`. By default this is `'test'`, which matches the way *unittest* finds tests. You can specify an alternative prefix by setting `patch.TEST_PREFIX`.

Patch can be used as a context manager, with the with statement. Here the patching applies to the indented block after the with statement. If you use "as" then the patched object will be bound to the name after the "as"; very useful if *patch()* is creating a mock object for you.

patch() takes arbitrary keyword arguments. These will be passed to *AsyncMock* if the patched object is asynchronous, to *MagicMock* otherwise or to *new_callable* if specified.

`patch.dict(...)`, `patch.multiple(...)` and `patch.object(...)` are available for alternate use-cases.

patch() as function decorator, creating the mock for you and passing it into the decorated function:

```
>>> @patch('__main__.SomeClass')
... def function(normal_argument, mock_class):
...     print(mock_class is SomeClass)
...
>>> function(None)
True
```

Patching a class replaces the class with a *MagicMock* *instance*. If the class is instantiated in the code under test then it will be the *return_value* of the mock that will be used.

If the class is instantiated multiple times you could use *side_effect* to return a new mock each time. Alternatively you can set the *return_value* to be anything you want.

To configure return values on methods of *instances* on the patched class you must do this on the `return_value`. For example:

```
>>> class Class:
...     def method(self):
...         pass
...
>>> with patch('__main__.Class') as MockClass:
...     instance = MockClass.return_value
...     instance.method.return_value = 'foo'
...     assert Class() is instance
```

(continues on next page)

(continued from previous page)

```
...     assert Class().method() == 'foo'
...
```

If you use *spec* or *spec_set* and `patch()` is replacing a *class*, then the return value of the created mock will have the same spec.

```
>>> Original = Class
>>> patcher = patch('__main__.Class', spec=True)
>>> MockClass = patcher.start()
>>> instance = MockClass()
>>> assert isinstance(instance, Original)
>>> patcher.stop()
```

The *new_callable* argument is useful where you want to use an alternative class to the default `MagicMock` for the created mock. For example, if you wanted a `NonCallableMock` to be used:

```
>>> thing = object()
>>> with patch('__main__.thing', new_callable=NonCallableMock) as mock_thing:
...     assert thing is mock_thing
...     thing()
...
Traceback (most recent call last):
  ...
TypeError: 'NonCallableMock' object is not callable
```

Another use case might be to replace an object with an `io.StringIO` instance:

```
>>> from io import StringIO
>>> def foo():
...     print('Something')
...
>>> @patch('sys.stdout', new_callable=StringIO)
... def test(mock_stdout):
...     foo()
...     assert mock_stdout.getvalue() == 'Something\n'
...
>>> test()
```

When `patch()` is creating a mock for you, it is common that the first thing you need to do is to configure the mock. Some of that configuration can be done in the call to patch. Any arbitrary keywords you pass into the call will be used to set attributes on the created mock:

```
>>> patcher = patch('__main__.thing', first='one', second='two')
>>> mock_thing = patcher.start()
>>> mock_thing.first
'one'
>>> mock_thing.second
'two'
```

As well as attributes on the created mock attributes, like the `return_value` and `side_effect`, of child mocks can also be configured. These aren't syntactically valid to pass in directly as keyword arguments, but a dictionary with these as keys can still be expanded into a `patch()` call using `**`:

```
>>> config = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> patcher = patch('__main__.thing', **config)
>>> mock_thing = patcher.start()
```

(continues on next page)

(continued from previous page)

```
>>> mock_thing.method()
3
>>> mock_thing.other()
Traceback (most recent call last):
  ...
KeyError
```

By default, attempting to patch a function in a module (or a method or an attribute in a class) that does not exist will fail with *AttributeError*:

```
>>> @patch('sys.non_existing_attribute', 42)
... def test():
...     assert sys.non_existing_attribute == 42
...
>>> test()
Traceback (most recent call last):
  ...
AttributeError: <module 'sys' (built-in)> does not have the attribute 'non_existing_
↪attribute'
```

but adding `create=True` in the call to *patch()* will make the previous example work as expected:

```
>>> @patch('sys.non_existing_attribute', 42, create=True)
... def test(mock_stdout):
...     assert sys.non_existing_attribute == 42
...
>>> test()
```

Changed in version 3.8: *patch()* now returns an *AsyncMock* if the target is an async function.

patch.object

patch.**object**(*target*, *attribute*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

patch the named member (*attribute*) on an object (*target*) with a mock object.

patch.object() can be used as a decorator, class decorator or a context manager. Arguments *new*, *spec*, *create*, *spec_set*, *autospec* and *new_callable* have the same meaning as for *patch()*. Like *patch()*, *patch.object()* takes arbitrary keyword arguments for configuring the mock object it creates.

When used as a class decorator *patch.object()* honours `patch.TEST_PREFIX` for choosing which methods to wrap.

You can either call *patch.object()* with three arguments or two arguments. The three argument form takes the object to be patched, the attribute name and the object to replace the attribute with.

When calling with the two argument form you omit the replacement object, and a mock is created for you and passed in as an extra argument to the decorated function:

```
>>> @patch.object(SomeClass, 'class_method')
... def test(mock_method):
...     SomeClass.class_method(3)
...     mock_method.assert_called_with(3)
...
>>> test()
```

spec, *create* and the other arguments to *patch.object()* have the same meaning as they do for *patch()*.

patch.dict

patch.**dict**(*in_dict*, *values=()*, *clear=False*, ***kwargs*)

Patch a dictionary, or dictionary like object, and restore the dictionary to its original state after the test.

in_dict can be a dictionary or a mapping like container. If it is a mapping then it must at least support getting, setting and deleting items plus iterating over keys.

in_dict can also be a string specifying the name of the dictionary, which will then be fetched by importing it.

values can be a dictionary of values to set in the dictionary. *values* can also be an iterable of `(key, value)` pairs.

If *clear* is true then the dictionary will be cleared before the new values are set.

patch.dict() can also be called with arbitrary keyword arguments to set values in the dictionary.

Changed in version 3.8: *patch.dict()* now returns the patched dictionary when used as a context manager.

patch.dict() can be used as a context manager, decorator or class decorator:

```
>>> foo = {}
>>> @patch.dict(foo, {'newkey': 'newvalue'})
... def test():
...     assert foo == {'newkey': 'newvalue'}
...
>>> test()
>>> assert foo == {}
```

When used as a class decorator *patch.dict()* honours `patch.TEST_PREFIX` (default to `'test'`) for choosing which methods to wrap:

```
>>> import os
>>> import unittest
>>> from unittest.mock import patch
>>> @patch.dict('os.environ', {'newkey': 'newvalue'})
... class TestSample(unittest.TestCase):
...     def test_sample(self):
...         self.assertEqual(os.environ['newkey'], 'newvalue')
```

If you want to use a different prefix for your test, you can inform the patchers of the different prefix by setting `patch.TEST_PREFIX`. For more details about how to change the value of see *TEST_PREFIX*.

patch.dict() can be used to add members to a dictionary, or simply let a test change a dictionary, and ensure the dictionary is restored when the test ends.

```
>>> foo = {}
>>> with patch.dict(foo, {'newkey': 'newvalue'}) as patched_foo:
...     assert foo == {'newkey': 'newvalue'}
...     assert patched_foo == {'newkey': 'newvalue'}
...     # You can add, update or delete keys of foo (or patched_foo, it's the same␣
↪dict)
...     patched_foo['spam'] = 'eggs'
...
>>> assert foo == {}
>>> assert patched_foo == {}
```

```
>>> import os
>>> with patch.dict('os.environ', {'newkey': 'newvalue'}):
```

(continues on next page)

(continued from previous page)

```
...     print(os.environ['newkey'])
...
newvalue
>>> assert 'newkey' not in os.environ
```

Keywords can be used in the *patch.dict()* call to set values in the dictionary:

```
>>> mymodule = MagicMock()
>>> mymodule.function.return_value = 'fish'
>>> with patch.dict('sys.modules', mymodule=mymodule):
...     import mymodule
...     mymodule.function('some', 'args')
...
'fish'
```

patch.dict() can be used with dictionary like objects that aren't actually dictionaries. At the very minimum they must support item getting, setting, deleting and either iteration or membership test. This corresponds to the magic methods `__getitem__()`, `__setitem__()`, `__delitem__()` and either *__iter__()* or `__contains__()`.

```
>>> class Container:
...     def __init__(self):
...         self.values = {}
...     def __getitem__(self, name):
...         return self.values[name]
...     def __setitem__(self, name, value):
...         self.values[name] = value
...     def __delitem__(self, name):
...         del self.values[name]
...     def __iter__(self):
...         return iter(self.values)
...
>>> thing = Container()
>>> thing['one'] = 1
>>> with patch.dict(thing, one=2, two=3):
...     assert thing['one'] == 2
...     assert thing['two'] == 3
...
>>> assert thing['one'] == 1
>>> assert list(thing) == ['one']
```

patch.multiple

patch.**multiple**(*target*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

Perform multiple patches in a single call. It takes the object to be patched (either as an object or a string to fetch the object by importing) and keyword arguments for the patches:

```
with patch.multiple(settings, FIRST_PATCH='one', SECOND_PATCH='two'):
    ...
```

Use *DEFAULT* as the value if you want *patch.multiple()* to create mocks for you. In this case the created mocks are passed into a decorated function by keyword, and a dictionary is returned when *patch.multiple()* is used as a context manager.

patch.multiple() can be used as a decorator, class decorator or a context manager. The arguments *spec*, *spec_set*, *create*, *autospec* and *new_callable* have the same meaning as for *patch()*. These arguments will be applied to *all* patches done by *patch.multiple()*.

When used as a class decorator *patch.multiple()* honours `patch.TEST_PREFIX` for choosing which methods to wrap.

If you want *patch.multiple()* to create mocks for you, then you can use *DEFAULT* as the value. If you use *patch.multiple()* as a decorator then the created mocks are passed into the decorated function by keyword.

```
>>> thing = object()
>>> other = object()

>>> @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(thing, other):
...     assert isinstance(thing, MagicMock)
...     assert isinstance(other, MagicMock)
...
>>> test_function()
```

patch.multiple() can be nested with other `patch` decorators, but put arguments passed by keyword *after* any of the standard arguments created by *patch()*:

```
>>> @patch('sys.exit')
... @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(mock_exit, other, thing):
...     assert 'other' in repr(other)
...     assert 'thing' in repr(thing)
...     assert 'exit' in repr(mock_exit)
...
>>> test_function()
```

If *patch.multiple()* is used as a context manager, the value returned by the context manager is a dictionary where created mocks are keyed by name:

```
>>> with patch.multiple('__main__', thing=DEFAULT, other=DEFAULT) as values:
...     assert 'other' in repr(values['other'])
...     assert 'thing' in repr(values['thing'])
...     assert values['thing'] is thing
...     assert values['other'] is other
...
```

patch methods: start and stop

All the patchers have `start()` and `stop()` methods. These make it simpler to do patching in `setUp` methods or where you want to do multiple patches without nesting decorators or with statements.

To use them call *patch()*, *patch.object()* or *patch.dict()* as normal and keep a reference to the returned `patcher` object. You can then call `start()` to put the patch in place and `stop()` to undo it.

If you are using *patch()* to create a mock for you then it will be returned by the call to `patcher.start`.

```
>>> patcher = patch('package.module.ClassName')
>>> from package import module
>>> original = module.ClassName
>>> new_mock = patcher.start()
>>> assert module.ClassName is not original
```

(continues on next page)

(continued from previous page)

```
>>> assert module.ClassName is new_mock
>>> patcher.stop()
>>> assert module.ClassName is original
>>> assert module.ClassName is not new_mock
```

A typical use case for this might be for doing multiple patches in the `setUp` method of a `TestCase`:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher1 = patch('package.module.Class1')
...         self.patcher2 = patch('package.module.Class2')
...         self.MockClass1 = self.patcher1.start()
...         self.MockClass2 = self.patcher2.start()
...
...     def tearDown(self):
...         self.patcher1.stop()
...         self.patcher2.stop()
...
...     def test_something(self):
...         assert package.module.Class1 is self.MockClass1
...         assert package.module.Class2 is self.MockClass2
...
>>> MyTest('test_something').run()
```

Caution

If you use this technique you must ensure that the patching is "undone" by calling `stop`. This can be fiddlier than you might think, because if an exception is raised in the `setUp` then `tearDown` is not called. *`unittest.TestCase.addCleanup()`* makes this easier:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('package.module.Class')
...         self.MockClass = patcher.start()
...         self.addCleanup(patcher.stop)
...
...     def test_something(self):
...         assert package.module.Class is self.MockClass
...
```

As an added bonus you no longer need to keep a reference to the `patcher` object.

It is also possible to stop all patches which have been started by using *`patch.stopall()`*.

`patch.`**`stopall`**`()`

Stop all active patches. Only stops patches started with `start`.

patch builtins

You can patch any builtins within a module. The following example patches builtin *ord()*:

```
>>> @patch('__main__.ord')
... def test(mock_ord):
...     mock_ord.return_value = 101
...     print(ord('c'))
...
>>> test()
101
```

TEST_PREFIX

All of the patchers can be used as class decorators. When used in this way they wrap every test method on the class. The patchers recognise methods that start with `'test'` as being test methods. This is the same way that the *unittest.TestLoader* finds test methods by default.

It is possible that you want to use a different prefix for your tests. You can inform the patchers of the different prefix by setting `patch.TEST_PREFIX`:

```
>>> patch.TEST_PREFIX = 'foo'
>>> value = 3
>>>
>>> @patch('__main__.value', 'not three')
... class Thing:
...     def foo_one(self):
...         print(value)
...     def foo_two(self):
...         print(value)
...
>>>
>>> Thing().foo_one()
not three
>>> Thing().foo_two()
not three
>>> value
3
```

Nesting Patch Decorators

If you want to perform multiple patches then you can simply stack up the decorators.

You can stack up multiple patch decorators using this pattern:

```
>>> @patch.object(SomeClass, 'class_method')
... @patch.object(SomeClass, 'static_method')
... def test(mock1, mock2):
...     assert SomeClass.static_method is mock1
...     assert SomeClass.class_method is mock2
...     SomeClass.static_method('foo')
...     SomeClass.class_method('bar')
...     return mock1, mock2
...
>>> mock1, mock2 = test()
```

(continues on next page)

(continued from previous page)

```
>>> mock1.assert_called_once_with('foo')
>>> mock2.assert_called_once_with('bar')
```

Note that the decorators are applied from the bottom upwards. This is the standard way that Python applies decorators. The order of the created mocks passed into your test function matches this order.

Where to patch

patch() works by (temporarily) changing the object that a *name* points to with another one. There can be many names pointing to any individual object, so for patching to work you must ensure that you patch the name used by the system under test.

The basic principle is that you patch where an object is *looked up*, which is not necessarily the same place as where it is defined. A couple of examples will help to clarify this.

Imagine we have a project that we want to test with the following structure:

```
a.py
    -> Defines SomeClass

b.py
    -> from a import SomeClass
    -> some_function instantiates SomeClass
```

Now we want to test `some_function` but we want to mock out `SomeClass` using *patch()*. The problem is that when we import module b, which we will have to do then it imports `SomeClass` from module a. If we use *patch()* to mock out `a.SomeClass` then it will have no effect on our test; module b already has a reference to the *real* `SomeClass` and it looks like our patching had no effect.

The key is to patch out `SomeClass` where it is used (or where it is looked up). In this case `some_function` will actually look up `SomeClass` in module b, where we have imported it. The patching should look like:

```
@patch('b.SomeClass')
```

However, consider the alternative scenario where instead of `from a import SomeClass` module b does `import a` and `some_function` uses `a.SomeClass`. Both of these import forms are common. In this case the class we want to patch is being looked up in the module and so we have to patch `a.SomeClass` instead:

```
@patch('a.SomeClass')
```

Patching Descriptors and Proxy Objects

Both *patch* and *patch.object* correctly patch and restore descriptors: class methods, static methods and properties. You should patch these on the *class* rather than an instance. They also work with *some* objects that proxy attribute access, like the django settings object.

26.6.4 MagicMock and magic method support

Mocking Magic Methods

Mock supports mocking the Python protocol methods, also known as *"magic methods"*. This allows mock objects to replace containers or other objects that implement Python protocols.

Because magic methods are looked up differently from normal methods[2], this support has been specially implemented. This means that only specific magic methods are supported. The supported list includes *almost* all of them. If there are any missing that you need please let us know.

You mock magic methods by setting the method you are interested in to a function or a mock instance. If you are using a function then it *must* take `self` as the first argument[3].

```
>>> def __str__(self):
...     return 'fooble'
...
>>> mock = Mock()
>>> mock.__str__ = __str__
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__str__ = Mock()
>>> mock.__str__.return_value = 'fooble'
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__iter__ = Mock(return_value=iter([]))
>>> list(mock)
[]
```

One use case for this is for mocking objects used as context managers in a `with` statement:

```
>>> mock = Mock()
>>> mock.__enter__ = Mock(return_value='foo')
>>> mock.__exit__ = Mock(return_value=False)
>>> with mock as m:
...     assert m == 'foo'
...
>>> mock.__enter__.assert_called_with()
>>> mock.__exit__.assert_called_with(None, None, None)
```

Calls to magic methods do not appear in *method_calls*, but they are recorded in *mock_calls*.

> **Note**
>
> If you use the *spec* keyword argument to create a mock then attempting to set a magic method that isn't in the spec will raise an *AttributeError*.

The full list of supported magic methods is:

[2] Magic methods *should* be looked up on the class rather than the instance. Different versions of Python are inconsistent about applying this rule. The supported protocol methods should work with all supported versions of Python.

[3] The function is basically hooked up to the class, but each `Mock` instance is kept isolated from the others.

- __hash__, __sizeof__, __repr__ and __str__
- __dir__, __format__ and __subclasses__
- __round__, __floor__, __trunc__ and __ceil__
- Comparisons: __lt__, __gt__, __le__, __ge__, __eq__ and __ne__
- Container methods: __getitem__, __setitem__, __delitem__, __contains__, __len__, __iter__, __reversed__ and __missing__
- Context manager: __enter__, __exit__, __aenter__ and __aexit__
- Unary numeric methods: __neg__, __pos__ and __invert__
- The numeric methods (including right hand and in-place variants): __add__, __sub__, __mul__, __matmul__, __truediv__, __floordiv__, __mod__, __divmod__, __lshift__, __rshift__, __and__, __xor__, __or__, and __pow__
- Numeric conversion methods: __complex__, __int__, __float__ and __index__
- Descriptor methods: __get__, __set__ and __delete__
- Pickling: __reduce__, __reduce_ex__, __getinitargs__, __getnewargs__, __getstate__ and __setstate__
- File system path representation: __fspath__
- Asynchronous iteration methods: __aiter__ and __anext__

Changed in version 3.8: Added support for *os.PathLike.__fspath__()*.

Changed in version 3.8: Added support for __aenter__, __aexit__, __aiter__ and __anext__.

The following methods exist but are *not* supported as they are either in use by mock, can't be set dynamically, or can cause problems:

- __getattr__, __setattr__, __init__ and __new__
- __prepare__, __instancecheck__, __subclasscheck__, __del__

Magic Mock

There are two MagicMock variants: *MagicMock* and *NonCallableMagicMock*.

class unittest.mock.**MagicMock**(**args*, ***kw*)

MagicMock is a subclass of *Mock* with default implementations of most of the *magic methods*. You can use MagicMock without having to configure the magic methods yourself.

The constructor parameters have the same meaning as for *Mock*.

If you use the *spec* or *spec_set* arguments then *only* magic methods that exist in the spec will be created.

class unittest.mock.**NonCallableMagicMock**(**args*, ***kw*)

A non-callable version of *MagicMock*.

The constructor parameters have the same meaning as for *MagicMock*, with the exception of *return_value* and *side_effect* which have no meaning on a non-callable mock.

The magic methods are setup with *MagicMock* objects, so you can configure them and use them in the usual way:

```
>>> mock = MagicMock()
>>> mock[3] = 'fish'
>>> mock.__setitem__.assert_called_with(3, 'fish')
>>> mock.__getitem__.return_value = 'result'
>>> mock[2]
'result'
```

By default many of the protocol methods are required to return objects of a specific type. These methods are preconfigured with a default return value, so that they can be used without you having to do anything if you aren't interested in the return value. You can still *set* the return value manually if you want to change the default.

Methods and their defaults:

- `__lt__`: *NotImplemented*
- `__gt__`: NotImplemented
- `__le__`: NotImplemented
- `__ge__`: NotImplemented
- `__int__`: 1
- `__contains__`: False
- `__len__`: 0
- `__iter__`: iter([])
- `__exit__`: False
- `__aexit__`: False
- `__complex__`: 1j
- `__float__`: 1.0
- `__bool__`: True
- `__index__`: 1
- `__hash__`: default hash for the mock
- `__str__`: default str for the mock
- `__sizeof__`: default sizeof for the mock

For example:

```
>>> mock = MagicMock()
>>> int(mock)
1
>>> len(mock)
0
>>> list(mock)
[]
>>> object() in mock
False
```

The two equality methods, `__eq__()` and `__ne__()`, are special. They do the default equality comparison on identity, using the *side_effect* attribute, unless you change their return value to return something else:

```
>>> MagicMock() == 3
False
>>> MagicMock() != 3
True
>>> mock = MagicMock()
>>> mock.__eq__.return_value = True
>>> mock == 3
True
```

The return value of `MagicMock.__iter__()` can be any iterable object and isn't required to be an iterator:

```
>>> mock = MagicMock()
>>> mock.__iter__.return_value = ['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
```

If the return value *is* an iterator, then iterating over it once will consume it and subsequent iterations will result in an empty list:

```
>>> mock.__iter__.return_value = iter(['a', 'b', 'c'])
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
[]
```

`MagicMock` has all of the supported magic methods configured except for some of the obscure and obsolete ones. You can still set these up if you want.

Magic methods that are supported but not setup by default in `MagicMock` are:

- `__subclasses__`
- `__dir__`
- `__format__`
- `__get__`, `__set__` and `__delete__`
- `__reversed__` and `__missing__`
- `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` and `__setstate__`
- `__getformat__`

26.6.5 Helpers

sentinel

`unittest.mock.`**`sentinel`**

The `sentinel` object provides a convenient way of providing unique objects for your tests.

Attributes are created on demand when you access them by name. Accessing the same attribute will always return the same object. The objects returned have a sensible repr so that test failure messages are readable.

Changed in version 3.7: The `sentinel` attributes now preserve their identity when they are *copied* or *pickled*.

Sometimes when testing you need to test that a specific object is passed as an argument to another method, or returned. It can be common to create named sentinel objects to test this. *sentinel* provides a convenient way of creating and testing the identity of objects like this.

In this example we monkey patch `method` to return `sentinel.some_object`:

```
>>> real = ProductionClass()
>>> real.method = Mock(name="method")
>>> real.method.return_value = sentinel.some_object
>>> result = real.method()
>>> assert result is sentinel.some_object
>>> result
sentinel.some_object
```

DEFAULT

unittest.mock.**DEFAULT**

The *DEFAULT* object is a pre-created sentinel (actually `sentinel.DEFAULT`). It can be used by *side_effect* functions to indicate that the normal return value should be used.

call

unittest.mock.**call**(**args*, ***kwargs*)

call() is a helper object for making simpler assertions, for comparing with *call_args*, *call_args_list*, *mock_calls* and *method_calls*. *call()* can also be used with *assert_has_calls()*.

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, a='foo', b='bar')
>>> m()
>>> m.call_args_list == [call(1, 2, a='foo', b='bar'), call()]
True
```

call.**call_list**()

For a call object that represents multiple calls, *call_list()* returns a list of all the intermediate calls as well as the final call.

`call_list` is particularly useful for making assertions on "chained calls". A chained call is multiple calls on a single line of code. This results in multiple entries in *mock_calls* on a mock. Manually constructing the sequence of calls can be tedious.

call_list() can construct the sequence of calls from the same chained call:

```
>>> m = MagicMock()
>>> m(1).method(arg='foo').other('bar')(2.0)
<MagicMock name='mock().method().other()()' id='...'>
>>> kall = call(1).method(arg='foo').other('bar')(2.0)
>>> kall.call_list()
[call(1),
 call().method(arg='foo'),
 call().method().other('bar'),
 call().method().other()(2.0)]
>>> m.mock_calls == kall.call_list()
True
```

A `call` object is either a tuple of (positional args, keyword args) or (name, positional args, keyword args) depending on how it was constructed. When you construct them yourself this isn't particularly interesting, but the `call` objects that are in the *Mock.call_args*, *Mock.call_args_list* and *Mock.mock_calls* attributes can be introspected to get at the individual arguments they contain.

The `call` objects in *Mock.call_args* and *Mock.call_args_list* are two-tuples of (positional args, keyword args) whereas the `call` objects in *Mock.mock_calls*, along with ones you construct yourself, are three-tuples of (name, positional args, keyword args).

You can use their "tupleness" to pull out the individual arguments for more complex introspection and assertions. The positional arguments are a tuple (an empty tuple if there are no positional arguments) and the keyword arguments are a dictionary:

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, 3, arg='one', arg2='two')
>>> kall = m.call_args
>>> kall.args
(1, 2, 3)
>>> kall.kwargs
{'arg': 'one', 'arg2': 'two'}
>>> kall.args is kall[0]
True
>>> kall.kwargs is kall[1]
True
```

```
>>> m = MagicMock()
>>> m.foo(4, 5, 6, arg='two', arg2='three')
<MagicMock name='mock.foo()' id='...'>
>>> kall = m.mock_calls[0]
>>> name, args, kwargs = kall
>>> name
'foo'
>>> args
(4, 5, 6)
>>> kwargs
{'arg': 'two', 'arg2': 'three'}
>>> name is m.mock_calls[0][0]
True
```

create_autospec

unittest.mock.**create_autospec**(*spec*, *spec_set=False*, *instance=False*, ***kwargs*)

Create a mock object using another object as a spec. Attributes on the mock will use the corresponding attribute on the *spec* object as their spec.

Functions or methods being mocked will have their arguments checked to ensure that they are called with the correct signature.

If *spec_set* is `True` then attempting to set attributes that don't exist on the spec object will raise an *AttributeError*.

If a class is used as a spec then the return value of the mock (the instance of the class) will have the same spec. You can use a class as the spec for an instance object by passing `instance=True`. The returned mock will only be callable if instances of the mock are callable.

create_autospec() also takes arbitrary keyword arguments that are passed to the constructor of the created mock.

See *Autospeccing* for examples of how to use auto-speccing with *create_autospec()* and the *autospec* argument to *patch()*.

Changed in version 3.8: *create_autospec()* now returns an *AsyncMock* if the target is an async function.

ANY

unittest.mock.**ANY**

Sometimes you may need to make assertions about *some* of the arguments in a call to mock, but either not care about some of the arguments or want to pull them individually out of *call_args* and make more complex assertions on them.

To ignore certain arguments you can pass in objects that compare equal to *everything*. Calls to *assert_called_with()* and *assert_called_once_with()* will then succeed no matter what was passed in.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar=object())
>>> mock.assert_called_once_with('foo', bar=ANY)
```

ANY can also be used in comparisons with call lists like *mock_calls*:

```
>>> m = MagicMock(return_value=None)
>>> m(1)
>>> m(1, 2)
>>> m(object())
>>> m.mock_calls == [call(1), call(1, 2), ANY]
True
```

ANY is not limited to comparisons with call objects and so can also be used in test assertions:

```
class TestStringMethods(unittest.TestCase):

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', ANY])
```

FILTER_DIR

unittest.mock.**FILTER_DIR**

FILTER_DIR is a module level variable that controls the way mock objects respond to *dir()*. The default is `True`, which uses the filtering described below, to only show useful members. If you dislike this filtering, or need to switch it off for diagnostic purposes, then set `mock.FILTER_DIR = False`.

With filtering on, `dir(some_mock)` shows only useful attributes and will include any dynamically created attributes that wouldn't normally be shown. If the mock was created with a *spec* (or *autospec* of course) then all the attributes from the original are shown, even if they haven't been accessed yet:

```
>>> dir(Mock())
['assert_any_call',
 'assert_called',
 'assert_called_once',
 'assert_called_once_with',
 'assert_called_with',
 'assert_has_calls',
 'assert_not_called',
```

(continues on next page)

(continued from previous page)

```
 'attach_mock',
 ...
>>> from urllib import request
>>> dir(Mock(spec=request))
['AbstractBasicAuthHandler',
 'AbstractDigestAuthHandler',
 'AbstractHTTPHandler',
 'BaseHandler',
 ...
```

Many of the not-very-useful (private to *Mock* rather than the thing being mocked) underscore and double underscore prefixed attributes have been filtered from the result of calling *dir()* on a *Mock*. If you dislike this behaviour you can switch it off by setting the module level switch *FILTER_DIR*:

```
>>> from unittest import mock
>>> mock.FILTER_DIR = False
>>> dir(mock.Mock())
['_NonCallableMock__get_return_value',
 '_NonCallableMock__get_side_effect',
 '_NonCallableMock__return_value_doc',
 '_NonCallableMock__set_return_value',
 '_NonCallableMock__set_side_effect',
 '__call__',
 '__class__',
 ...
```

Alternatively you can just use `vars(my_mock)` (instance members) and `dir(type(my_mock))` (type members) to bypass the filtering irrespective of `mock.FILTER_DIR`.

mock_open

`unittest.mock.`**`mock_open`**(*mock=None*, *read_data=None*)

A helper function to create a mock to replace the use of *open()*. It works for *open()* called directly or used as a context manager.

The *mock* argument is the mock object to configure. If `None` (the default) then a *MagicMock* will be created for you, with the API limited to methods or attributes available on standard file handles.

read_data is a string for the `read()`, *readline()*, and *readlines()* methods of the file handle to return. Calls to those methods will take data from *read_data* until it is depleted. The mock of these methods is pretty simplistic: every time the *mock* is called, the *read_data* is rewound to the start. If you need more control over the data that you are feeding to the tested code you will need to customize this mock for yourself. When that is insufficient, one of the in-memory filesystem packages on PyPI can offer a realistic filesystem for testing.

Changed in version 3.4: Added *readline()* and *readlines()* support. The mock of `read()` changed to consume *read_data* rather than returning it on each call.

Changed in version 3.5: *read_data* is now reset on each call to the *mock*.

Changed in version 3.8: Added *__iter__()* to implementation so that iteration (such as in for loops) correctly consumes *read_data*.

Using *open()* as a context manager is a great way to ensure your file handles are closed properly and is becoming common:

```
with open('/some/path', 'w') as f:
    f.write('something')
```

The issue is that even if you mock out the call to *open()* it is the *returned object* that is used as a context manager (and has __enter__() and __exit__() called).

Mocking context managers with a *MagicMock* is common enough and fiddly enough that a helper function is useful.

```
>>> m = mock_open()
>>> with patch('__main__.open', m):
...     with open('foo', 'w') as h:
...         h.write('some stuff')
...
>>> m.mock_calls
[call('foo', 'w'),
 call().__enter__(),
 call().write('some stuff'),
 call().__exit__(None, None, None)]
>>> m.assert_called_once_with('foo', 'w')
>>> handle = m()
>>> handle.write.assert_called_once_with('some stuff')
```

And for reading files:

```
>>> with patch('__main__.open', mock_open(read_data='bibble')) as m:
...     with open('foo') as h:
...         result = h.read()
...
>>> m.assert_called_once_with('foo')
>>> assert result == 'bibble'
```

Autospeccing

Autospeccing is based on the existing spec feature of mock. It limits the api of mocks to the api of an original object (the spec), but it is recursive (implemented lazily) so that attributes of mocks only have the same api as the attributes of the spec. In addition mocked functions / methods have the same call signature as the original so they raise a *TypeError* if they are called incorrectly.

Before I explain how auto-speccing works, here's why it is needed.

Mock is a very powerful and flexible object, but it suffers from a flaw which is general to mocking. If you refactor some of your code, rename members and so on, any tests for code that is still using the *old api* but uses mocks instead of the real objects will still pass. This means your tests can all pass even though your code is broken.

Changed in version 3.5: Before 3.5, tests with a typo in the word assert would silently pass when they should raise an error. You can still achieve this behavior by passing unsafe=True to Mock.

Note that this is another reason why you need integration tests as well as unit tests. Testing everything in isolation is all fine and dandy, but if you don't test how your units are "wired together" there is still lots of room for bugs that tests might have caught.

mock already provides a feature to help with this, called speccing. If you use a class or instance as the spec for a mock then you can only access attributes on the mock that exist on the real class:

```
>>> from urllib import request
>>> mock = Mock(spec=request.Request)
>>> mock.assret_called_with  # Intentional typo!
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret_called_with'
```

The spec only applies to the mock itself, so we still have the same issue with any methods on the mock:

```
>>> mock.has_data()
<mock.Mock object at 0x...>
>>> mock.has_data.assret_called_with()  # Intentional typo!
```

Auto-speccing solves this problem. You can either pass `autospec=True` to *patch()* / *patch.object()* or use the *create_autospec()* function to create a mock with a spec. If you use the `autospec=True` argument to *patch()* then the object that is being replaced will be used as the spec object. Because the speccing is done "lazily" (the spec is created as attributes on the mock are accessed) you can use it with very complex or deeply nested objects (like modules that import modules that import modules) without a big performance hit.

Here's an example of it in use:

```
>>> from urllib import request
>>> patcher = patch('__main__.request', autospec=True)
>>> mock_request = patcher.start()
>>> request is mock_request
True
>>> mock_request.Request
<MagicMock name='request.Request' spec='Request' id='...'>
```

You can see that `request.Request` has a spec. `request.Request` takes two arguments in the constructor (one of which is *self*). Here's what happens if we try to call it incorrectly:

```
>>> req = request.Request()
Traceback (most recent call last):
  ...
TypeError: <lambda>() takes at least 2 arguments (1 given)
```

The spec also applies to instantiated classes (i.e. the return value of specced mocks):

```
>>> req = request.Request('foo')
>>> req
<NonCallableMagicMock name='request.Request()' spec='Request' id='...'>
```

`Request` objects are not callable, so the return value of instantiating our mocked out `request.Request` is a non-callable mock. With the spec in place any typos in our asserts will raise the correct error:

```
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='...'>
>>> req.add_header.assret_called_with  # Intentional typo!
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret_called_with'
>>> req.add_header.assert_called_with('spam', 'eggs')
```

In many cases you will just be able to add `autospec=True` to your existing *patch()* calls and then be protected against bugs due to typos and api changes.

As well as using *autospec* through *patch()* there is a *create_autospec()* for creating autospecced mocks directly:

```
>>> from urllib import request
>>> mock_request = create_autospec(request)
>>> mock_request.Request('foo', 'bar')
<NonCallableMagicMock name='mock.Request()' spec='Request' id='...'>
```

This isn't without caveats and limitations however, which is why it is not the default behaviour. In order to know what attributes are available on the spec object, autospec has to introspect (access attributes) the spec. As you traverse attributes

on the mock a corresponding traversal of the original object is happening under the hood. If any of your specced objects have properties or descriptors that can trigger code execution then you may not be able to use autospec. On the other hand it is much better to design your objects so that introspection is safe[4].

A more serious problem is that it is common for instance attributes to be created in the `__init__()` method and not to exist on the class at all. *autospec* can't know about any dynamically created attributes and restricts the api to visible attributes.

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

There are a few different ways of resolving this problem. The easiest, but not necessarily the least annoying, way is to simply set the required attributes on the mock after creation. Just because *autospec* doesn't allow you to fetch attributes that don't exist on the spec it doesn't prevent you setting them:

```
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a = 33
...
```

There is a more aggressive version of both *spec* and *autospec* that *does* prevent you setting non-existent attributes. This is useful if you want to ensure your code only *sets* valid attributes too, but obviously it prevents this particular scenario:

```
>>> with patch('__main__.Something', autospec=True, spec_set=True):
...   thing = Something()
...   thing.a = 33
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

Probably the best way of solving the problem is to add class attributes as default values for instance members initialised in `__init__()`. Note that if you are only setting default attributes in `__init__()` then providing them via class attributes (shared between instances of course) is faster too. e.g.

```
class Something:
    a = 33
```

This brings up another issue. It is relatively common to provide a default value of `None` for members that will later be an object of a different type. `None` would be useless as a spec because it wouldn't let you access *any* attributes or methods on it. As `None` is *never* going to be useful as a spec, and probably indicates a member that will normally of some other type, autospec doesn't use a spec for members that are set to `None`. These will just be ordinary mocks (well - MagicMocks):

```
>>> class Something:
...     member = None
...
```

(continues on next page)

[4] This only applies to classes or already instantiated objects. Calling a mocked class to create a mock instance *does not* create a real instance. It is only attribute lookups - along with calls to *dir()* - that are done.

(continued from previous page)

```
>>> mock = create_autospec(Something)
>>> mock.member.foo.bar.baz()
<MagicMock name='mock.member.foo.bar.baz()' id='...'>
```

If modifying your production classes to add defaults isn't to your liking then there are more options. One of these is simply to use an instance as the spec rather than the class. The other is to create a subclass of the production class and add the defaults to the subclass without affecting the production class. Both of these require you to use an alternative object as the spec. Thankfully *patch()* supports this - you can simply pass the alternative object as the *autospec* argument:

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> class SomethingForTest(Something):
...   a = 33
...
>>> p = patch('__main__.Something', autospec=SomethingForTest)
>>> mock = p.start()
>>> mock.a
<NonCallableMagicMock name='Something.a' spec='int' id='...'>
```

Sealing mocks

unittest.mock.**seal**(*mock*)

Seal will disable the automatic creation of mocks when accessing an attribute of the mock being sealed or any of its attributes that are already mocks recursively.

If a mock instance with a name or a spec is assigned to an attribute it won't be considered in the sealing chain. This allows one to prevent seal from fixing part of the mock object.

```
>>> mock = Mock()
>>> mock.submock.attribute1 = 2
>>> mock.not_submock = mock.Mock(name="sample_name")
>>> seal(mock)
>>> mock.new_attribute  # This will raise AttributeError.
>>> mock.submock.attribute2  # This will raise AttributeError.
>>> mock.not_submock.attribute2  # This won't raise.
```

Added in version 3.7.

26.6.6 Order of precedence of `side_effect`, `return_value` and *wraps*

The order of their precedence is:

1. *side_effect*
2. *return_value*
3. *wraps*

If all three are set, mock will return the value from *side_effect*, ignoring *return_value* and the wrapped object altogether. If any two are set, the one with the higher precedence will return the value. Regardless of the order of which was set first, the order of precedence remains unchanged.

```
>>> from unittest.mock import Mock
>>> class Order:
...     @staticmethod
...     def get_value():
...         return "third"
...
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.get_value.side_effect = ["first"]
>>> order_mock.get_value.return_value = "second"
>>> order_mock.get_value()
'first'
```

As `None` is the default value of *side_effect*, if you reassign its value back to `None`, the order of precedence will be checked between *return_value* and the wrapped object, ignoring *side_effect*.

```
>>> order_mock.get_value.side_effect = None
>>> order_mock.get_value()
'second'
```

If the value being returned by *side_effect* is *DEFAULT*, it is ignored and the order of precedence moves to the successor to obtain the value to return.

```
>>> from unittest.mock import DEFAULT
>>> order_mock.get_value.side_effect = [DEFAULT]
>>> order_mock.get_value()
'second'
```

When *Mock* wraps an object, the default value of *return_value* will be *DEFAULT*.

```
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.return_value
sentinel.DEFAULT
>>> order_mock.get_value.return_value
sentinel.DEFAULT
```

The order of precedence will ignore this value and it will move to the last successor which is the wrapped object.

As the real call is being made to the wrapped object, creating an instance of this mock will return the real instance of the class. The positional arguments, if any, required by the wrapped object must be passed.

```
>>> order_mock_instance = order_mock()
>>> isinstance(order_mock_instance, Order)
True
>>> order_mock_instance.get_value()
'third'
```

```
>>> order_mock.get_value.return_value = DEFAULT
>>> order_mock.get_value()
'third'
```

```
>>> order_mock.get_value.return_value = "second"
>>> order_mock.get_value()
'second'
```

But if you assign `None` to it, this will not be ignored as it is an explicit assignment. So, the order of precedence will not move to the wrapped object.

```
>>> order_mock.get_value.return_value = None
>>> order_mock.get_value() is None
True
```

Even if you set all three at once when initializing the mock, the order of precedence remains the same:

```
>>> order_mock = Mock(spec=Order, wraps=Order,
...                   **{"get_value.side_effect": ["first"],
...                      "get_value.return_value": "second"}
...                   )
...
>>> order_mock.get_value()
'first'
>>> order_mock.get_value.side_effect = None
>>> order_mock.get_value()
'second'
>>> order_mock.get_value.return_value = DEFAULT
>>> order_mock.get_value()
'third'
```

If *side_effect* is exhausted, the order of precedence will not cause a value to be obtained from the successors. Instead, `StopIteration` exception is raised.

```
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.get_value.side_effect = ["first side effect value",
...                                     "another side effect value"]
>>> order_mock.get_value.return_value = "second"
```

```
>>> order_mock.get_value()
'first side effect value'
>>> order_mock.get_value()
'another side effect value'
```

```
>>> order_mock.get_value()
Traceback (most recent call last):
  ...
StopIteration
```

26.7 `unittest.mock` — getting started

Added in version 3.3.

26.7.1 Using Mock

Mock Patching Methods

Common uses for *Mock* objects include:

- Patching methods
- Recording method calls on objects

You might want to replace a method on an object to check that it is called with the correct arguments by another part of the system:

```
>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
>>> real.method(3, 4, 5, key='value')
<MagicMock name='method()' id='...'>
```

Once our mock has been used (`real.method` in this example) it has methods and attributes that allow you to make assertions about how it has been used.

Note

In most of these examples the *Mock* and *MagicMock* classes are interchangeable. As the `MagicMock` is the more capable class it makes a sensible one to use by default.

Once the mock has been called its *called* attribute is set to `True`. More importantly we can use the *assert_called_with()* or *assert_called_once_with()* method to check that it was called with the correct arguments.

This example tests that calling `ProductionClass().method` results in a call to the `something` method:

```
>>> class ProductionClass:
...     def method(self):
...         self.something(1, 2, 3)
...     def something(self, a, b, c):
...         pass
...
>>> real = ProductionClass()
>>> real.something = MagicMock()
>>> real.method()
>>> real.something.assert_called_once_with(1, 2, 3)
```

Mock for Method Calls on an Object

In the last example we patched a method directly on an object to check that it was called correctly. Another common use case is to pass an object into a method (or some part of the system under test) and then check that it is used in the correct way.

The simple `ProductionClass` below has a `closer` method. If it is called with an object then it calls `close` on it.

```
>>> class ProductionClass:
...     def closer(self, something):
...         something.close()
...
```

So to test it we need to pass in an object with a `close` method and check that it was called correctly.

```
>>> real = ProductionClass()
>>> mock = Mock()
>>> real.closer(mock)
>>> mock.close.assert_called_with()
```

We don't have to do any work to provide the 'close' method on our mock. Accessing close creates it. So, if 'close' hasn't already been called then accessing it in the test will create it, but *assert_called_with()* will raise a failure exception.

Mocking Classes

A common use case is to mock out classes instantiated by your code under test. When you patch a class, then that class is replaced with a mock. Instances are created by *calling the class*. This means you access the "mock instance" by looking at the return value of the mocked class.

In the example below we have a function `some_function` that instantiates `Foo` and calls a method on it. The call to *`patch()`* replaces the class `Foo` with a mock. The `Foo` instance is the result of calling the mock, so it is configured by modifying the mock *`return_value`*.

```
>>> def some_function():
...     instance = module.Foo()
...     return instance.method()
...
>>> with patch('module.Foo') as mock:
...     instance = mock.return_value
...     instance.method.return_value = 'the result'
...     result = some_function()
...     assert result == 'the result'
```

Naming your mocks

It can be useful to give your mocks a name. The name is shown in the repr of the mock and can be helpful when the mock appears in test failure messages. The name is also propagated to attributes or methods of the mock:

```
>>> mock = MagicMock(name='foo')
>>> mock
<MagicMock name='foo' id='...'>
>>> mock.method
<MagicMock name='foo.method' id='...'>
```

Tracking all Calls

Often you want to track more than a single call to a method. The *`mock_calls`* attribute records all calls to child attributes of the mock - and also to their children.

```
>>> mock = MagicMock()
>>> mock.method()
<MagicMock name='mock.method()' id='...'>
>>> mock.attribute.method(10, x=53)
<MagicMock name='mock.attribute.method()' id='...'>
>>> mock.mock_calls
[call.method(), call.attribute.method(10, x=53)]
```

If you make an assertion about `mock_calls` and any unexpected methods have been called, then the assertion will fail. This is useful because as well as asserting that the calls you expected have been made, you are also checking that they were made in the right order and with no additional calls:

You use the *`call`* object to construct lists for comparing with `mock_calls`:

```
>>> expected = [call.method(), call.attribute.method(10, x=53)]
>>> mock.mock_calls == expected
True
```

However, parameters to calls that return mocks are not recorded, which means it is not possible to track nested calls where the parameters used to create ancestors are important:

```
>>> m = Mock()
>>> m.factory(important=True).deliver()
<Mock name='mock.factory().deliver()' id='...'>
>>> m.mock_calls[-1] == call.factory(important=False).deliver()
True
```

Setting Return Values and Attributes

Setting the return values on a mock object is trivially easy:

```
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

Of course you can do the same for methods on the mock:

```
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3
```

The return value can also be set in the constructor:

```
>>> mock = Mock(return_value=3)
>>> mock()
3
```

If you need an attribute setting on your mock, just do it:

```
>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3
```

Sometimes you want to mock up a more complex situation, like for example `mock.connection.cursor().execute("SELECT 1")`. If we wanted this call to return a list, then we have to configure the result of the nested call.

We can use *call* to construct the set of calls in a "chained call" like this for easy assertion afterwards:

```
>>> mock = Mock()
>>> cursor = mock.connection.cursor.return_value
>>> cursor.execute.return_value = ['foo']
>>> mock.connection.cursor().execute("SELECT 1")
['foo']
>>> expected = call.connection.cursor().execute("SELECT 1").call_list()
>>> mock.mock_calls
[call.connection.cursor(), call.connection.cursor().execute('SELECT 1')]
>>> mock.mock_calls == expected
True
```

It is the call to `.call_list()` that turns our call object into a list of calls representing the chained calls.

Raising exceptions with mocks

A useful attribute is *side_effect*. If you set this to an exception class or instance then the exception will be raised when the mock is called.

```
>>> mock = Mock(side_effect=Exception('Boom!'))
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

Side effect functions and iterables

`side_effect` can also be set to a function or an iterable. The use case for `side_effect` as an iterable is where your mock is going to be called several times, and you want each call to return a different value. When you set `side_effect` to an iterable every call to the mock returns the next value from the iterable:

```
>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
```

For more advanced use cases, like dynamically varying the return values depending on what the mock is called with, `side_effect` can be a function. The function will be called with the same arguments as the mock. Whatever the function returns is what the call returns:

```
>>> vals = {(1, 2): 1, (2, 3): 2}
>>> def side_effect(*args):
...     return vals[args]
...
>>> mock = MagicMock(side_effect=side_effect)
>>> mock(1, 2)
1
>>> mock(2, 3)
2
```

Mocking asynchronous iterators

Since Python 3.8, `AsyncMock` and `MagicMock` have support to mock async-iterators through `__aiter__`. The *return_value* attribute of `__aiter__` can be used to set the return values to be used for iteration.

```
>>> mock = MagicMock()  # AsyncMock also works here
>>> mock.__aiter__.return_value = [1, 2, 3]
>>> async def main():
...     return [i async for i in mock]
...
>>> asyncio.run(main())
[1, 2, 3]
```

Mocking asynchronous context manager

Since Python 3.8, `AsyncMock` and `MagicMock` have support to mock async-context-managers through `__aenter__` and `__aexit__`. By default, `__aenter__` and `__aexit__` are `AsyncMock` instances that return an async function.

```
>>> class AsyncContextManager:
...     async def __aenter__(self):
...         return self
...     async def __aexit__(self, exc_type, exc, tb):
...         pass
...
>>> mock_instance = MagicMock(AsyncContextManager())  # AsyncMock also works here
>>> async def main():
...     async with mock_instance as result:
...         pass
...
>>> asyncio.run(main())
>>> mock_instance.__aenter__.assert_awaited_once()
>>> mock_instance.__aexit__.assert_awaited_once()
```

Creating a Mock from an Existing Object

One problem with over use of mocking is that it couples your tests to the implementation of your mocks rather than your real code. Suppose you have a class that implements `some_method`. In a test for another class, you provide a mock of this object that *also* provides `some_method`. If later you refactor the first class, so that it no longer has `some_method` - then your tests will continue to pass even though your code is now broken!

Mock allows you to provide an object as a specification for the mock, using the *spec* keyword argument. Accessing methods / attributes on the mock that don't exist on your specification object will immediately raise an attribute error. If you change the implementation of your specification, then tests that use that class will start failing immediately without you having to instantiate the class in those tests.

```
>>> mock = Mock(spec=SomeClass)
>>> mock.old_method()
Traceback (most recent call last):
   ...
AttributeError: Mock object has no attribute 'old_method'. Did you mean: 'class_method
↪'?
```

Using a specification also enables a smarter matching of calls made to the mock, regardless of whether some parameters were passed as positional or named arguments:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, 3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(a=1, b=2, c=3)
```

If you want this smarter matching to also work with method calls on the mock, you can use *auto-speccing*.

If you want a stronger form of specification that prevents the setting of arbitrary attributes as well as the getting of them then you can use *spec_set* instead of *spec*.

Using side_effect to return per file content

mock_open() is used to patch *open()* method. *side_effect* can be used to return a new Mock object per call. This can be used to return different contents per file stored in a dictionary:

```
DEFAULT = "default"
data_dict = {"file1": "data1",
             "file2": "data2"}

def open_side_effect(name):
    return mock_open(read_data=data_dict.get(name, DEFAULT))()

with patch("builtins.open", side_effect=open_side_effect):
    with open("file1") as file1:
        assert file1.read() == "data1"

    with open("file2") as file2:
        assert file2.read() == "data2"

    with open("file3") as file2:
        assert file2.read() == "default"
```

26.7.2 Patch Decorators

Note

With *patch()* it matters that you patch objects in the namespace where they are looked up. This is normally straightforward, but for a quick guide read *where to patch*.

A common need in tests is to patch a class attribute or a module attribute, for example patching a builtin or patching a class in a module to test that it is instantiated. Modules and classes are effectively global, so patching on them has to be undone after the test or the patch will persist into other tests and cause hard to diagnose problems.

mock provides three convenient decorators for this: *patch()*, *patch.object()* and *patch.dict()*. `patch` takes a single string, of the form `package.module.Class.attribute` to specify the attribute you are patching. It also optionally takes a value that you want the attribute (or class or whatever) to be replaced with. 'patch.object' takes an object and the name of the attribute you would like patched, plus optionally the value to patch it with.

`patch.object`:

```
>>> original = SomeClass.attribute
>>> @patch.object(SomeClass, 'attribute', sentinel.attribute)
... def test():
...     assert SomeClass.attribute == sentinel.attribute
...
>>> test()
>>> assert SomeClass.attribute == original

>>> @patch('package.module.attribute', sentinel.attribute)
... def test():
...     from package.module import attribute
...     assert attribute is sentinel.attribute
...
>>> test()
```

If you are patching a module (including *builtins*) then use *patch()* instead of *patch.object()*:

```
>>> mock = MagicMock(return_value=sentinel.file_handle)
>>> with patch('builtins.open', mock):
...     handle = open('filename', 'r')
...
>>> mock.assert_called_with('filename', 'r')
>>> assert handle == sentinel.file_handle, "incorrect file handle returned"
```

The module name can be 'dotted', in the form `package.module` if needed:

```
>>> @patch('package.module.ClassName.attribute', sentinel.attribute)
... def test():
...     from package.module import ClassName
...     assert ClassName.attribute == sentinel.attribute
...
>>> test()
```

A nice pattern is to actually decorate test methods themselves:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'attribute', sentinel.attribute)
...     def test_something(self):
...         self.assertEqual(SomeClass.attribute, sentinel.attribute)
...
>>> original = SomeClass.attribute
>>> MyTest('test_something').test_something()
>>> assert SomeClass.attribute == original
```

If you want to patch with a Mock, you can use *patch()* with only one argument (or *patch.object()* with two arguments). The mock will be created for you and passed into the test function / method:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'static_method')
...     def test_something(self, mock_method):
...         SomeClass.static_method()
...         mock_method.assert_called_with()
...
>>> MyTest('test_something').test_something()
```

You can stack up multiple patch decorators using this pattern:

```
>>> class MyTest(unittest.TestCase):
...     @patch('package.module.ClassName1')
...     @patch('package.module.ClassName2')
...     def test_something(self, MockClass2, MockClass1):
...         self.assertIs(package.module.ClassName1, MockClass1)
...         self.assertIs(package.module.ClassName2, MockClass2)
...
>>> MyTest('test_something').test_something()
```

When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal *Python* order that decorators are applied). This means from the bottom up, so in the example above the mock for `test_module.ClassName2` is passed in first.

There is also *patch.dict()* for setting values in a dictionary just during a scope and restoring the dictionary to its original state when the test ends:

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

`patch`, `patch.object` and `patch.dict` can all be used as context managers.

Where you use *`patch()`* to create a mock for you, you can get a reference to the mock using the "as" form of the with statement:

```
>>> class ProductionClass:
...     def method(self):
...         pass
...
>>> with patch.object(ProductionClass, 'method') as mock_method:
...     mock_method.return_value = None
...     real = ProductionClass()
...     real.method(1, 2, 3)
...
>>> mock_method.assert_called_with(1, 2, 3)
```

As an alternative `patch`, `patch.object` and `patch.dict` can be used as class decorators. When used in this way it is the same as applying the decorator individually to every method whose name starts with "test".

26.7.3 Further Examples

Here are some more examples for some slightly more advanced scenarios.

Mocking chained calls

Mocking chained calls is actually straightforward with mock once you understand the *`return_value`* attribute. When a mock is called for the first time, or you fetch its `return_value` before it has been called, a new *`Mock`* is created.

This means that you can see how the object returned from a call to a mocked object has been used by interrogating the `return_value` mock:

```
>>> mock = Mock()
>>> mock().foo(a=2, b=3)
<Mock name='mock().foo()' id='...'>
>>> mock.return_value.foo.assert_called_with(a=2, b=3)
```

From here it is a simple step to configure and then make assertions about chained calls. Of course another alternative is writing your code in a more testable way in the first place…

So, suppose we have some code that looks a little bit like this:

```
>>> class Something:
...     def __init__(self):
...         self.backend = BackendProvider()
...     def method(self):
...         response = self.backend.get_endpoint('foobar').create_call('spam', 'eggs
↪').start_call()
...         # more code
```

Assuming that `BackendProvider` is already well tested, how do we test `method()`? Specifically, we want to test that the code section `# more code` uses the response object in the correct way.

As this chain of calls is made from an instance attribute we can monkey patch the `backend` attribute on a `Something` instance. In this particular case we are only interested in the return value from the final call to `start_call` so we don't have much configuration to do. Let's assume the object it returns is 'file-like', so we'll ensure that our response object uses the builtin *`open()`* as its `spec`.

To do this we create a mock instance as our mock backend and create a mock response object for it. To set the response as the return value for that final `start_call` we could do this:

```
mock_backend.get_endpoint.return_value.create_call.return_value.start_call.return_
↪value = mock_response
```

We can do that in a slightly nicer way using the *`configure_mock()`* method to directly set the return value for us:

```
>>> something = Something()
>>> mock_response = Mock(spec=open)
>>> mock_backend = Mock()
>>> config = {'get_endpoint.return_value.create_call.return_value.start_call.return_
↪value': mock_response}
>>> mock_backend.configure_mock(**config)
```

With these we monkey patch the "mock backend" in place and can make the real call:

```
>>> something.backend = mock_backend
>>> something.method()
```

Using *`mock_calls`* we can check the chained call with a single assert. A chained call is several calls in one line of code, so there will be several entries in `mock_calls`. We can use *`call.call_list()`* to create this list of calls for us:

```
>>> chained = call.get_endpoint('foobar').create_call('spam', 'eggs').start_call()
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

Partial mocking

In some tests I wanted to mock out a call to *`datetime.date.today()`* to return a known date, but I didn't want to prevent the code under test from creating new date objects. Unfortunately *`datetime.date`* is written in C, and so I couldn't just monkey-patch out the static *`datetime.date.today()`* method.

I found a simple way of doing this that involved effectively wrapping the date class with a mock, but passing through calls to the constructor to the real class (and returning real instances).

The *`patch decorator`* is used here to mock out the `date` class in the module under test. The *`side_effect`* attribute on the mock date class is then set to a lambda function that returns a real date. When the mock date class is called a real date will be constructed and returned by `side_effect`.

```
>>> from datetime import date
>>> with patch('mymodule.date') as mock_date:
...     mock_date.today.return_value = date(2010, 10, 8)
...     mock_date.side_effect = lambda *args, **kw: date(*args, **kw)
...
...     assert mymodule.date.today() == date(2010, 10, 8)
...     assert mymodule.date(2009, 6, 8) == date(2009, 6, 8)
```

Note that we don't patch *datetime.date* globally, we patch `date` in the module that *uses* it. See *where to patch*.

When `date.today()` is called a known date is returned, but calls to the `date(...)` constructor still return normal dates. Without this you can find yourself having to calculate an expected result using exactly the same algorithm as the code under test, which is a classic testing anti-pattern.

Calls to the date constructor are recorded in the `mock_date` attributes (`call_count` and friends) which may also be useful for your tests.

An alternative way of dealing with mocking dates, or other builtin classes, is discussed in this blog entry.

Mocking a Generator Method

A Python generator is a function or method that uses the `yield` statement to return a series of values when iterated over[1].

A generator method / function is called to return the generator object. It is the generator object that is then iterated over. The protocol method for iteration is *__iter__()*, so we can mock this using a *MagicMock*.

Here's an example class with an "iter" method implemented as a generator:

```
>>> class Foo:
...     def iter(self):
...         for i in [1, 2, 3]:
...             yield i
...
>>> foo = Foo()
>>> list(foo.iter())
[1, 2, 3]
```

How would we mock this class, and in particular its "iter" method?

To configure the values returned from the iteration (implicit in the call to *list*), we need to configure the object returned by the call to `foo.iter()`.

```
>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]
```

Applying the same patch to every test method

If you want several patches in place for multiple test methods the obvious way is to apply the patch decorators to every method. This can feel like unnecessary repetition. Instead, you can use *patch()* (in all its various forms) as a class decorator. This applies the patches to all test methods on the class. A test method is identified by methods whose names start with `test`:

```
>>> @patch('mymodule.SomeClass')
... class MyTest(unittest.TestCase):
...
...     def test_one(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def test_two(self, MockSomeClass):
```

(continues on next page)

[1] There are also generator expressions and more advanced uses of generators, but we aren't concerned about them here. A very good introduction to generators and how powerful they are is: Generator Tricks for Systems Programmers.

(continued from previous page)

```
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def not_a_test(self):
...         return 'something'
...
>>> MyTest('test_one').test_one()
>>> MyTest('test_two').test_two()
>>> MyTest('test_two').not_a_test()
'something'
```

An alternative way of managing patches is to use the *patch methods: start and stop*. These allow you to move the patching into your `setUp` and `tearDown` methods.

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher = patch('mymodule.foo')
...         self.mock_foo = self.patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
...     def tearDown(self):
...         self.patcher.stop()
...
>>> MyTest('test_foo').run()
```

If you use this technique you must ensure that the patching is "undone" by calling `stop`. This can be fiddlier than you might think, because if an exception is raised in the setUp then tearDown is not called. *`unittest.TestCase.addCleanup()`* makes this easier:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('mymodule.foo')
...         self.addCleanup(patcher.stop)
...         self.mock_foo = patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
>>> MyTest('test_foo').run()
```

Mocking Unbound Methods

Whilst writing tests today I needed to patch an *unbound method* (patching the method on the class rather than on the instance). I needed self to be passed in as the first argument because I want to make asserts about which objects were calling this particular method. The issue is that you can't patch with a mock for this, because if you replace an unbound method with a mock it doesn't become a bound method when fetched from the instance, and so it doesn't get self passed in. The workaround is to patch the unbound method with a real function instead. The *`patch()`* decorator makes it so simple to patch out methods with a mock that having to create a real function becomes a nuisance.

If you pass `autospec=True` to patch then it does the patching with a *real* function object. This function object has the same signature as the one it is replacing, but delegates to a mock under the hood. You still get your mock auto-created in exactly the same way as before. What it means though, is that if you use it to patch out an unbound method on a class the mocked function will be turned into a bound method if it is fetched from an instance. It will have `self` passed in as the first argument, which is exactly what I wanted:

```
>>> class Foo:
...   def foo(self):
...     pass
...
>>> with patch.object(Foo, 'foo', autospec=True) as mock_foo:
...   mock_foo.return_value = 'foo'
...   foo = Foo()
...   foo.foo()
...
'foo'
>>> mock_foo.assert_called_once_with(foo)
```

If we don't use `autospec=True` then the unbound method is patched out with a Mock instance instead, and isn't called with `self`.

Checking multiple calls with mock

mock has a nice API for making assertions about how your mock objects are used.

```
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
```

If your mock is only being called once you can use the *assert_called_once_with()* method that also asserts that the *call_count* is one.

```
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
>>> mock.foo_bar()
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
Traceback (most recent call last):
    ...
AssertionError: Expected 'foo_bar' to be called once. Called 2 times.
Calls: [call('baz', spam='eggs'), call()].
```

Both `assert_called_with` and `assert_called_once_with` make assertions about the *most recent* call. If your mock is going to be called several times, and you want to make assertions about *all* those calls you can use *call_args_list*:

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock(4, 5, 6)
>>> mock()
>>> mock.call_args_list
[call(1, 2, 3), call(4, 5, 6), call()]
```

The *call* helper makes it easy to make assertions about these calls. You can build up a list of expected calls and compare it to `call_args_list`. This looks remarkably similar to the repr of the `call_args_list`:

```
>>> expected = [call(1, 2, 3), call(4, 5, 6), call()]
>>> mock.call_args_list == expected
True
```

Coping with mutable arguments

Another situation is rare, but can bite you, is when your mock is called with mutable arguments. `call_args` and `call_args_list` store *references* to the arguments. If the arguments are mutated by the code under test then you can no longer make assertions about what the values were when the mock was called.

Here's some example code that shows the problem. Imagine the following functions defined in 'mymodule':

```
def frob(val):
    pass

def grob(val):
    "First frob and then clear val"
    frob(val)
    val.clear()
```

When we try to test that `grob` calls `frob` with the correct argument look what happens:

```
>>> with patch('mymodule.frob') as mock_frob:
...     val = {6}
...     mymodule.grob(val)
...
>>> val
set()
>>> mock_frob.assert_called_with({6})
Traceback (most recent call last):
    ...
AssertionError: Expected: (({6},), {})
Called with: ((set(),), {})
```

One possibility would be for mock to copy the arguments you pass in. This could then cause problems if you do assertions that rely on object identity for equality.

Here's one solution that uses the *side_effect* functionality. If you provide a `side_effect` function for a mock then `side_effect` will be called with the same args as the mock. This gives us an opportunity to copy the arguments and store them for later assertions. In this example I'm using *another* mock to store the arguments so that I can use the mock methods for doing the assertion. Again a helper function sets this up for me.

```
>>> from copy import deepcopy
>>> from unittest.mock import Mock, patch, DEFAULT
>>> def copy_call_args(mock):
...     new_mock = Mock()
...     def side_effect(*args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         new_mock(*args, **kwargs)
...         return DEFAULT
...     mock.side_effect = side_effect
...     return new_mock
...
>>> with patch('mymodule.frob') as mock_frob:
...     new_mock = copy_call_args(mock_frob)
...     val = {6}
...     mymodule.grob(val)
...
>>> new_mock.assert_called_with({6})
>>> new_mock.call_args
call({6})
```

`copy_call_args` is called with the mock that will be called. It returns a new mock that we do the assertion on. The `side_effect` function makes a copy of the args and calls our `new_mock` with the copy.

Note

If your mock is only going to be used once there is an easier way of checking arguments at the point they are called. You can simply do the checking inside a `side_effect` function.

```
>>> def side_effect(arg):
...     assert arg == {6}
...
>>> mock = Mock(side_effect=side_effect)
>>> mock({6})
>>> mock(set())
Traceback (most recent call last):
    ...
AssertionError
```

An alternative approach is to create a subclass of *Mock* or *MagicMock* that copies (using *copy.deepcopy()*) the arguments. Here's an example implementation:

```
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...     def __call__(self, /, *args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         return super().__call__(*args, **kwargs)
...
>>> c = CopyingMock(return_value=None)
>>> arg = set()
>>> c(arg)
>>> arg.add(1)
>>> c.assert_called_with(set())
>>> c.assert_called_with(arg)
Traceback (most recent call last):
    ...
AssertionError: expected call not found.
Expected: mock({1})
Actual: mock(set())
>>> c.foo
<CopyingMock name='mock.foo' id='...'>
```

When you subclass `Mock` or `MagicMock` all dynamically created attributes, and the `return_value` will use your subclass automatically. That means all children of a `CopyingMock` will also have the type `CopyingMock`.

Nesting Patches

Using patch as a context manager is nice, but if you do multiple patches you can end up with nested with statements indenting further and further to the right:

```
>>> class MyTest(unittest.TestCase):
...
...     def test_foo(self):
...         with patch('mymodule.Foo') as mock_foo:
...             with patch('mymodule.Bar') as mock_bar:
...                 with patch('mymodule.Spam') as mock_spam:
...                     assert mymodule.Foo is mock_foo
...                     assert mymodule.Bar is mock_bar
...                     assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').test_foo()
>>> assert mymodule.Foo is original
```

With unittest `cleanup` functions and the *patch methods: start and stop* we can achieve the same effect without the nested indentation. A simple helper method, `create_patch`, puts the patch in place and returns the created mock for us:

```
>>> class MyTest(unittest.TestCase):
...
...     def create_patch(self, name):
...         patcher = patch(name)
...         thing = patcher.start()
...         self.addCleanup(patcher.stop)
...         return thing
...
...     def test_foo(self):
...         mock_foo = self.create_patch('mymodule.Foo')
...         mock_bar = self.create_patch('mymodule.Bar')
...         mock_spam = self.create_patch('mymodule.Spam')
...
...         assert mymodule.Foo is mock_foo
...         assert mymodule.Bar is mock_bar
...         assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').run()
>>> assert mymodule.Foo is original
```

Mocking a dictionary with MagicMock

You may want to mock a dictionary, or other container object, recording all access to it whilst having it still behave like a dictionary.

We can do this with *MagicMock*, which will behave like a dictionary, and using *side_effect* to delegate dictionary access to a real underlying dictionary that is under our control.

When the `__getitem__()` and `__setitem__()` methods of our `MagicMock` are called (normal dictionary access) then `side_effect` is called with the key (and in the case of `__setitem__` the value too). We can also control what is returned.

After the `MagicMock` has been used we can use attributes like *call_args_list* to assert about how the dictionary was used:

```
>>> my_dict = {'a': 1, 'b': 2, 'c': 3}
>>> def getitem(name):
...      return my_dict[name]
...
>>> def setitem(name, val):
...     my_dict[name] = val
...
>>> mock = MagicMock()
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

Note

An alternative to using `MagicMock` is to use `Mock` and *only* provide the magic methods you specifically want:

```
>>> mock = Mock()
>>> mock.__getitem__ = Mock(side_effect=getitem)
>>> mock.__setitem__ = Mock(side_effect=setitem)
```

A *third* option is to use `MagicMock` but passing in `dict` as the *spec* (or *spec_set*) argument so that the `MagicMock` created only has dictionary magic methods available:

```
>>> mock = MagicMock(spec_set=dict)
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

With these side effect functions in place, the `mock` will behave like a normal dictionary but recording the access. It even raises a *KeyError* if you try to access a key that doesn't exist.

```
>>> mock['a']
1
>>> mock['c']
3
>>> mock['d']
Traceback (most recent call last):
    ...
KeyError: 'd'
>>> mock['b'] = 'fish'
>>> mock['d'] = 'eggs'
>>> mock['b']
'fish'
>>> mock['d']
'eggs'
```

After it has been used you can make assertions about the access using the normal mock methods and attributes:

```
>>> mock.__getitem__.call_args_list
[call('a'), call('c'), call('d'), call('b'), call('d')]
>>> mock.__setitem__.call_args_list
[call('b', 'fish'), call('d', 'eggs')]
>>> my_dict
{'a': 1, 'b': 'fish', 'c': 3, 'd': 'eggs'}
```

Mock subclasses and their attributes

There are various reasons why you might want to subclass *Mock*. One reason might be to add helper methods. Here's a silly example:

```
>>> class MyMock(MagicMock):
...     def has_been_called(self):
...         return self.called
...
>>> mymock = MyMock(return_value=None)
>>> mymock
<MyMock id='...'>
>>> mymock.has_been_called()
False
>>> mymock()
>>> mymock.has_been_called()
True
```

The standard behaviour for `Mock` instances is that attributes and the return value mocks are of the same type as the mock they are accessed on. This ensures that `Mock` attributes are `Mocks` and `MagicMock` attributes are `MagicMocks`[2]. So if you're subclassing to add helper methods then they'll also be available on the attributes and return value mock of instances of your subclass.

```
>>> mymock.foo
<MyMock name='mock.foo' id='...'>
>>> mymock.foo.has_been_called()
False
>>> mymock.foo()
<MyMock name='mock.foo()' id='...'>
>>> mymock.foo.has_been_called()
True
```

Sometimes this is inconvenient. For example, one user is subclassing mock to created a Twisted adaptor. Having this applied to attributes too actually causes errors.

`Mock` (in all its flavours) uses a method called `_get_child_mock` to create these "sub-mocks" for attributes and return values. You can prevent your subclass being used for attributes by overriding this method. The signature is that it takes arbitrary keyword arguments (`**kwargs`) which are then passed onto the mock constructor:

```
>>> class Subclass(MagicMock):
...     def _get_child_mock(self, /, **kwargs):
...         return MagicMock(**kwargs)
...
>>> mymock = Subclass()
>>> mymock.foo
<MagicMock name='mock.foo' id='...'>
>>> assert isinstance(mymock, Subclass)
>>> assert not isinstance(mymock.foo, Subclass)
>>> assert not isinstance(mymock(), Subclass)
```

[2] An exception to this rule are the non-callable mocks. Attributes use the callable variant because otherwise non-callable mocks couldn't have callable methods.

Mocking imports with patch.dict

One situation where mocking can be hard is where you have a local import inside a function. These are harder to mock because they aren't using an object from the module namespace that we can patch out.

Generally local imports are to be avoided. They are sometimes done to prevent circular dependencies, for which there is *usually* a much better way to solve the problem (refactor the code) or to prevent "up front costs" by delaying the import. This can also be solved in better ways than an unconditional local import (store the module as a class or module attribute and only do the import on first use).

That aside there is a way to use `mock` to affect the results of an import. Importing fetches an *object* from the *sys.modules* dictionary. Note that it fetches an *object*, which need not be a module. Importing a module for the first time results in a module object being put in `sys.modules`, so usually when you import something you get a module back. This need not be the case however.

This means you can use *patch.dict()* to *temporarily* put a mock in place in *sys.modules*. Any imports whilst this patch is active will fetch the mock. When the patch is complete (the decorated function exits, the with statement body is complete or `patcher.stop()` is called) then whatever was there previously will be restored safely.

Here's an example that mocks out the 'fooble' module.

```
>>> import sys
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     import fooble
...     fooble.blob()
...
<Mock name='mock.blob()' id='...'>
>>> assert 'fooble' not in sys.modules
>>> mock.blob.assert_called_once_with()
```

As you can see the `import fooble` succeeds, but on exit there is no 'fooble' left in *sys.modules*.

This also works for the `from module import name` form:

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     from fooble import blob
...     blob.blip()
...
<Mock name='mock.blob.blip()' id='...'>
>>> mock.blob.blip.assert_called_once_with()
```

With slightly more work you can also mock package imports:

```
>>> mock = Mock()
>>> modules = {'package': mock, 'package.module': mock.module}
>>> with patch.dict('sys.modules', modules):
...     from package.module import fooble
...     fooble()
...
<Mock name='mock.module.fooble()' id='...'>
>>> mock.module.fooble.assert_called_once_with()
```

Tracking order of calls and less verbose call assertions

The *Mock* class allows you to track the *order* of method calls on your mock objects through the *method_calls* attribute. This doesn't allow you to track the order of calls between separate mock objects, however we can use *mock_calls* to achieve the same effect.

Because mocks track calls to child mocks in `mock_calls`, and accessing an arbitrary attribute of a mock creates a child mock, we can create our separate mocks from a parent one. Calls to those child mock will then all be recorded, in order, in the `mock_calls` of the parent:

```
>>> manager = Mock()
>>> mock_foo = manager.foo
>>> mock_bar = manager.bar
```

```
>>> mock_foo.something()
<Mock name='mock.foo.something()' id='...'>
>>> mock_bar.other.thing()
<Mock name='mock.bar.other.thing()' id='...'>
```

```
>>> manager.mock_calls
[call.foo.something(), call.bar.other.thing()]
```

We can then assert about the calls, including the order, by comparing with the `mock_calls` attribute on the manager mock:

```
>>> expected_calls = [call.foo.something(), call.bar.other.thing()]
>>> manager.mock_calls == expected_calls
True
```

If `patch` is creating, and putting in place, your mocks then you can attach them to a manager mock using the *attach_mock()* method. After attaching calls will be recorded in `mock_calls` of the manager.

```
>>> manager = MagicMock()
>>> with patch('mymodule.Class1') as MockClass1:
...     with patch('mymodule.Class2') as MockClass2:
...         manager.attach_mock(MockClass1, 'MockClass1')
...         manager.attach_mock(MockClass2, 'MockClass2')
...         MockClass1().foo()
...         MockClass2().bar()
<MagicMock name='mock.MockClass1().foo()' id='...'>
<MagicMock name='mock.MockClass2().bar()' id='...'>
>>> manager.mock_calls
[call.MockClass1(),
call.MockClass1().foo(),
call.MockClass2(),
call.MockClass2().bar()]
```

If many calls have been made, but you're only interested in a particular sequence of them then an alternative is to use the *assert_has_calls()* method. This takes a list of calls (constructed with the *call* object). If that sequence of calls are in *mock_calls* then the assert succeeds.

```
>>> m = MagicMock()
>>> m().foo().bar().baz()
<MagicMock name='mock().foo().bar().baz()' id='...'>
>>> m.one().two().three()
<MagicMock name='mock.one().two().three()' id='...'>
```

(continues on next page)

(continued from previous page)

```
>>> calls = call.one().two().three().call_list()
>>> m.assert_has_calls(calls)
```

Even though the chained call `m.one().two().three()` aren't the only calls that have been made to the mock, the assert still succeeds.

Sometimes a mock may have several calls made to it, and you are only interested in asserting about *some* of those calls. You may not even care about the order. In this case you can pass `any_order=True` to `assert_has_calls`:

```
>>> m = MagicMock()
>>> m(1), m.two(2, 3), m.seven(7), m.fifty('50')
(...)
>>> calls = [call.fifty('50'), call(1), call.seven(7)]
>>> m.assert_has_calls(calls, any_order=True)
```

More complex argument matching

Using the same basic concept as *ANY* we can implement matchers to do more complex assertions on objects used as arguments to mocks.

Suppose we expect some object to be passed to a mock that by default compares equal based on object identity (which is the Python default for user defined classes). To use *assert_called_with()* we would need to pass in the exact same object. If we are only interested in some of the attributes of this object then we can create a matcher that will check these attributes for us.

You can see in this example how a 'standard' call to `assert_called_with` isn't sufficient:

```
>>> class Foo:
...     def __init__(self, a, b):
...         self.a, self.b = a, b
...
>>> mock = Mock(return_value=None)
>>> mock(Foo(1, 2))
>>> mock.assert_called_with(Foo(1, 2))
Traceback (most recent call last):
    ...
AssertionError: expected call not found.
Expected: mock(<__main__.Foo object at 0x...>)
Actual: mock(<__main__.Foo object at 0x...>)
```

A comparison function for our `Foo` class might look something like this:

```
>>> def compare(self, other):
...     if not type(self) == type(other):
...         return False
...     if self.a != other.a:
...         return False
...     if self.b != other.b:
...         return False
...     return True
...
```

And a matcher object that can use comparison functions like this for its equality operation would look something like this:

```
>>> class Matcher:
...     def __init__(self, compare, some_obj):
...         self.compare = compare
...         self.some_obj = some_obj
...     def __eq__(self, other):
...         return self.compare(self.some_obj, other)
...
```

Putting all this together:

```
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

The `Matcher` is instantiated with our compare function and the `Foo` object we want to compare against. In `assert_called_with` the `Matcher` equality method will be called, which compares the object the mock was called with against the one we created our matcher with. If they match then `assert_called_with` passes, and if they don't an *AssertionError* is raised:

```
>>> match_wrong = Matcher(compare, Foo(3, 4))
>>> mock.assert_called_with(match_wrong)
Traceback (most recent call last):
    ...
AssertionError: Expected: ((<Matcher object at 0x...>,), {})
Called with: ((<Foo object at 0x...>,), {})
```

With a bit of tweaking you could have the comparison function raise the *AssertionError* directly and provide a more useful failure message.

As of version 1.5, the Python testing library PyHamcrest provides similar functionality, that may be useful here, in the form of its equality matcher (hamcrest.library.integration.match_equality).

26.8 `test` — Regression tests package for Python

Note

The *test* package is meant for internal use by Python only. It is documented for the benefit of the core developers of Python. Any use of this package outside of Python's standard library is discouraged as code mentioned here can change or be removed without notice between releases of Python.

The *test* package contains all regression tests for Python as well as the modules *test.support* and *test.regrtest*. *test.support* is used to enhance your tests while *test.regrtest* drives the testing suite.

Each module in the *test* package whose name starts with `test_` is a testing suite for a specific module or feature. All new tests should be written using the *unittest* or *doctest* module. Some older tests are written using a "traditional" testing style that compares output printed to `sys.stdout`; this style of test is considered deprecated.

See also

Module *unittest*
Writing PyUnit regression tests.

Module *doctest*
Tests embedded in documentation strings.

26.8.1 Writing Unit Tests for the `test` package

It is preferred that tests that use the *unittest* module follow a few guidelines. One is to name the test module by starting it with `test_` and end it with the name of the module being tested. The test methods in the test module should start with `test_` and end with a description of what the method is testing. This is needed so that the methods are recognized by the test driver as test methods. Also, no documentation string for the method should be included. A comment (such as `# Tests function returns only True or False`) should be used to provide documentation for test methods. This is done because documentation strings get printed out if they exist and thus what test is being run is not stated.

A basic boilerplate is often used:

```
import unittest
from test import support

class MyTestCase1(unittest.TestCase):

    # Only use setUp() and tearDown() if necessary

    def setUp(self):
        ... code to execute in preparation for tests ...

    def tearDown(self):
        ... code to execute to clean up after tests ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...

    def test_feature_two(self):
        # Test feature two.
        ... testing code ...

    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...

... more test classes ...

if __name__ == '__main__':
    unittest.main()
```

This code pattern allows the testing suite to be run by *test.regrtest*, on its own as a script that supports the *unittest* CLI, or via the `python -m unittest` CLI.

The goal for regression testing is to try to break code. This leads to a few guidelines to be followed:

- The testing suite should exercise all classes, functions, and constants. This includes not just the external API that is to be presented to the outside world but also "private" code.
- Whitebox testing (examining the code being tested when the tests are being written) is preferred. Blackbox testing (testing only the published user interface) is not complete enough to make sure all boundary and edge cases are

tested.

- Make sure all possible values are tested including invalid ones. This makes sure that not only all valid values are acceptable but also that improper values are handled correctly.
- Exhaust as many code paths as possible. Test where branching occurs and thus tailor input to make sure as many different paths through the code are taken.
- Add an explicit test for any bugs discovered for the tested code. This will make sure that the error does not crop up again if the code is changed in the future.
- Make sure to clean up after your tests (such as close and remove all temporary files).
- If a test is dependent on a specific condition of the operating system then verify the condition already exists before attempting the test.
- Import as few modules as possible and do it as soon as possible. This minimizes external dependencies of tests and also minimizes possible anomalous behavior from side-effects of importing a module.
- Try to maximize code reuse. On occasion, tests will vary by something as small as what type of input is used. Minimize code duplication by subclassing a basic test class with a class that specifies the input:

```
class TestFuncAcceptsSequencesMixin:

    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = [1, 2, 3]

class AcceptStrings(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = (1, 2, 3)
```

When using this pattern, remember that all classes that inherit from *unittest.TestCase* are run as tests. The `TestFuncAcceptsSequencesMixin` class in the example above does not have any data and so can't be run by itself, thus it does not inherit from *unittest.TestCase*.

See also

Test Driven Development
A book by Kent Beck on writing tests before code.

26.8.2 Running tests using the command-line interface

The *test* package can be run as a script to drive Python's regression test suite, thanks to the `-m` option: **python -m test**. Under the hood, it uses *test.regrtest*; the call **python -m test.regrtest** used in previous Python versions still works. Running the script by itself automatically starts running all regression tests in the *test* package. It does this by finding all modules in the package whose name starts with `test_`, importing them, and executing the function `test_main()` if present or loading the tests via unittest.TestLoader.loadTestsFromModule if `test_main` does not exist. The names of tests to execute may also be passed to the script. Specifying a single regression test (**python -m test test_spam**) will minimize output and only print whether the test passed or failed.

Running *test* directly allows what resources are available for tests to use to be set. You do this by using the `-u` command-line option. Specifying `all` as the value for the `-u` option enables all possible resources: **python -m test -uall**. If all but one resource is desired (a more common case), a comma-separated list of resources that are not desired may be listed after `all`. The command **python -m test -uall,-audio,-largefile** will run *test* with all resources except the `audio` and `largefile` resources. For a list of all resources and more command-line options, run **python -m test -h**.

Some other ways to execute the regression tests depend on what platform the tests are being executed on. On Unix, you can run **make test** at the top-level directory where Python was built. On Windows, executing **rt.bat** from your `PCbuild` directory will run all regression tests.

26.9 test.support — Utilities for the Python test suite

The *test.support* module provides support for Python's regression test suite.

Note

test.support is not a public module. It is documented here to help Python developers write tests. The API of this module is subject to change without backwards compatibility concerns between releases.

This module defines the following exceptions:

exception test.support.**TestFailed**

Exception to be raised when a test fails. This is deprecated in favor of *unittest*-based tests and *unittest.TestCase*'s assertion methods.

exception test.support.**ResourceDenied**

Subclass of *unittest.SkipTest*. Raised when a resource (such as a network connection) is not available. Raised by the *requires()* function.

The *test.support* module defines the following constants:

test.support.**verbose**

`True` when verbose output is enabled. Should be checked when more detailed information is desired about a running test. *verbose* is set by *test.regrtest*.

test.support.**is_jython**

`True` if the running interpreter is Jython.

test.support.**is_android**

`True` if the system is Android.

test.support.**unix_shell**

Path for shell if not on Windows; otherwise `None`.

test.support.**LOOPBACK_TIMEOUT**

Timeout in seconds for tests using a network server listening on the network local loopback interface like `127.0.0.1`.

The timeout is long enough to prevent test failure: it takes into account that the client and the server can run in different threads or even different processes.

The timeout should be long enough for *connect()*, *recv()* and *send()* methods of *socket.socket*.

Its default value is 5 seconds.

See also *INTERNET_TIMEOUT*.

test.support.**INTERNET_TIMEOUT**

Timeout in seconds for network requests going to the internet.

The timeout is short enough to prevent a test to wait for too long if the internet request is blocked for whatever reason.

Usually, a timeout using *INTERNET_TIMEOUT* should not mark a test as failed, but skip the test instead: see *transient_internet()*.

Its default value is 1 minute.

See also *LOOPBACK_TIMEOUT*.

test.support.**SHORT_TIMEOUT**

Timeout in seconds to mark a test as failed if the test takes "too long".

The timeout value depends on the regrtest `--timeout` command line option.

If a test using *SHORT_TIMEOUT* starts to fail randomly on slow buildbots, use *LONG_TIMEOUT* instead.

Its default value is 30 seconds.

test.support.**LONG_TIMEOUT**

Timeout in seconds to detect when a test hangs.

It is long enough to reduce the risk of test failure on the slowest Python buildbots. It should not be used to mark a test as failed if the test takes "too long". The timeout value depends on the regrtest `--timeout` command line option.

Its default value is 5 minutes.

See also *LOOPBACK_TIMEOUT*, *INTERNET_TIMEOUT* and *SHORT_TIMEOUT*.

test.support.**PGO**

Set when tests can be skipped when they are not useful for PGO.

test.support.**PIPE_MAX_SIZE**

A constant that is likely larger than the underlying OS pipe buffer size, to make writes blocking.

test.support.**Py_DEBUG**

`True` if Python was built with the `Py_DEBUG` macro defined, that is, if Python was built in debug mode.

Added in version 3.12.

test.support.**SOCK_MAX_SIZE**

A constant that is likely larger than the underlying OS socket buffer size, to make writes blocking.

test.support.**TEST_SUPPORT_DIR**

Set to the top level directory that contains *test.support*.

test.support.**TEST_HOME_DIR**

Set to the top level directory for the test package.

test.support.**TEST_DATA_DIR**

Set to the `data` directory within the test package.

test.support.**MAX_Py_ssize_t**

Set to *sys.maxsize* for big memory tests.

test.support.**max_memuse**

Set by *set_memlimit()* as the memory limit for big memory tests. Limited by *MAX_Py_ssize_t*.

test.support.**real_max_memuse**

Set by *set_memlimit()* as the memory limit for big memory tests. Not limited by *MAX_Py_ssize_t*.

test.support.**MISSING_C_DOCSTRINGS**

Set to `True` if Python is built without docstrings (the `WITH_DOC_STRINGS` macro is not defined). See the `configure --without-doc-strings` option.

See also the *HAVE_DOCSTRINGS* variable.

test.support.**HAVE_DOCSTRINGS**

Set to `True` if function docstrings are available. See the `python -OO` option, which strips docstrings of functions implemented in Python.

See also the *MISSING_C_DOCSTRINGS* variable.

test.support.**TEST_HTTP_URL**

Define the URL of a dedicated HTTP server for the network tests.

test.support.**ALWAYS_EQ**

Object that is equal to anything. Used to test mixed type comparison.

test.support.**NEVER_EQ**

Object that is not equal to anything (even to *ALWAYS_EQ*). Used to test mixed type comparison.

test.support.**LARGEST**

Object that is greater than anything (except itself). Used to test mixed type comparison.

test.support.**SMALLEST**

Object that is less than anything (except itself). Used to test mixed type comparison.

The *test.support* module defines the following functions:

test.support.**busy_retry**(*timeout*, *err_msg=None*, */*, ***, *error=True*)

Run the loop body until `break` stops the loop.

After *timeout* seconds, raise an *AssertionError* if *error* is true, or just stop the loop if *error* is false.

Example:

```
for _ in support.busy_retry(support.SHORT_TIMEOUT):
    if check():
        break
```

Example of error=False usage:

```
for _ in support.busy_retry(support.SHORT_TIMEOUT, error=False):
    if check():
        break
else:
    raise RuntimeError('my custom error')
```

test.support.**sleeping_retry**(*timeout*, *err_msg=None*, */*, ***, *init_delay=0.010*, *max_delay=1.0*, *error=True*)

Wait strategy that applies exponential backoff.

Run the loop body until `break` stops the loop. Sleep at each loop iteration, but not at the first iteration. The sleep delay is doubled at each iteration (up to *max_delay* seconds).

See *busy_retry()* documentation for the parameters usage.

Example raising an exception after SHORT_TIMEOUT seconds:

```
for _ in support.sleeping_retry(support.SHORT_TIMEOUT):
    if check():
        break
```

Example of error=False usage:

```
for _ in support.sleeping_retry(support.SHORT_TIMEOUT, error=False):
    if check():
        break
else:
    raise RuntimeError('my custom error')
```

test.support.**is_resource_enabled**(*resource*)

Return `True` if *resource* is enabled and available. The list of available resources is only set when *test.regrtest* is executing the tests.

test.support.**python_is_optimized**()

Return `True` if Python was not built with `-O0` or `-Og`.

test.support.**with_pymalloc**()

Return `_testcapi.WITH_PYMALLOC`.

test.support.**requires**(*resource*, *msg=None*)

Raise *ResourceDenied* if *resource* is not available. *msg* is the argument to *ResourceDenied* if it is raised. Always returns `True` if called by a function whose `__name__` is `'__main__'`. Used when tests are executed by *test.regrtest*.

test.support.**sortdict**(*dict*)

Return a repr of *dict* with keys sorted.

test.support.**findfile**(*filename*, *subdir=None*)

Return the path to the file named *filename*. If no match is found *filename* is returned. This does not equal a failure since it could be the path to the file.

Setting *subdir* indicates a relative path to use to find the file rather than looking directly in the path directories.

test.support.**get_pagesize**()

Get size of a page in bytes.

Added in version 3.12.

test.support.**setswitchinterval**(*interval*)

Set the *sys.setswitchinterval()* to the given *interval*. Defines a minimum interval for Android systems to prevent the system from hanging.

test.support.**check_impl_detail**(***guards*)

Use this check to guard CPython's implementation-specific tests or to run them only on the implementations guarded by the arguments. This function returns `True` or `False` depending on the host platform. Example usage:

```
check_impl_detail()               # Only on CPython (default).
check_impl_detail(jython=True)    # Only on Jython.
check_impl_detail(cpython=False)  # Everywhere except CPython.
```

test.support.**set_memlimit**(*limit*)

Set the values for *max_memuse* and *real_max_memuse* for big memory tests.

test.support.**record_original_stdout**(*stdout*)

Store the value from *stdout*. It is meant to hold the stdout at the time the regrtest began.

test.support.**get_original_stdout**()

Return the original stdout set by *record_original_stdout()* or `sys.stdout` if it's not set.

test.support.**args_from_interpreter_flags**()

Return a list of command line arguments reproducing the current settings in `sys.flags` and `sys.warnoptions`.

test.support.**optim_args_from_interpreter_flags**()

Return a list of command line arguments reproducing the current optimization settings in `sys.flags`.

test.support.**captured_stdin**()

test.support.**captured_stdout**()

test.support.**captured_stderr**()

A context managers that temporarily replaces the named stream with *io.StringIO* object.

Example use with output streams:

```
with captured_stdout() as stdout, captured_stderr() as stderr:
    print("hello")
    print("error", file=sys.stderr)
assert stdout.getvalue() == "hello\n"
assert stderr.getvalue() == "error\n"
```

Example use with input stream:

```
with captured_stdin() as stdin:
    stdin.write('hello\n')
    stdin.seek(0)
    # call test code that consumes from sys.stdin
    captured = input()
self.assertEqual(captured, "hello")
```

test.support.**disable_faulthandler**()

A context manager that temporary disables *faulthandler*.

test.support.**gc_collect**()

Force as many objects as possible to be collected. This is needed because timely deallocation is not guaranteed by the garbage collector. This means that `__del__` methods may be called later than expected and weakrefs may remain alive for longer than expected.

test.support.**disable_gc**()

A context manager that disables the garbage collector on entry. On exit, the garbage collector is restored to its prior state.

test.support.**swap_attr**(*obj*, *attr*, *new_val*)

Context manager to swap out an attribute with a new object.

Usage:

```
with swap_attr(obj, "attr", 5):
    ...
```

This will set `obj.attr` to 5 for the duration of the `with` block, restoring the old value at the end of the block. If `attr` doesn't exist on `obj`, it will be created and then deleted at the end of the block.

The old value (or `None` if it doesn't exist) will be assigned to the target of the "as" clause, if there is one.

`test.support.`**`swap_item`**(*obj*, *attr*, *new_val*)

Context manager to swap out an item with a new object.

Usage:

```
with swap_item(obj, "item", 5):
    ...
```

This will set `obj["item"]` to 5 for the duration of the `with` block, restoring the old value at the end of the block. If `item` doesn't exist on `obj`, it will be created and then deleted at the end of the block.

The old value (or `None` if it doesn't exist) will be assigned to the target of the "as" clause, if there is one.

`test.support.`**`flush_std_streams`**()

Call the `flush()` method on *sys.stdout* and then on *sys.stderr*. It can be used to make sure that the logs order is consistent before writing into stderr.

Added in version 3.11.

`test.support.`**`print_warning`**(*msg*)

Print a warning into *sys.__stderr__*. Format the message as: `f"Warning -- {msg}"`. If *msg* is made of multiple lines, add `"Warning -- "` prefix to each line.

Added in version 3.9.

`test.support.`**`wait_process`**(*pid*, *, *exitcode*, *timeout=None*)

Wait until process *pid* completes and check that the process exit code is *exitcode*.

Raise an *AssertionError* if the process exit code is not equal to *exitcode*.

If the process runs longer than *timeout* seconds (*SHORT_TIMEOUT* by default), kill the process and raise an *AssertionError*. The timeout feature is not available on Windows.

Added in version 3.9.

`test.support.`**`calcobjsize`**(*fmt*)

Return the size of the `PyObject` whose structure members are defined by *fmt*. The returned value includes the size of the Python object header and alignment.

`test.support.`**`calcvobjsize`**(*fmt*)

Return the size of the `PyVarObject` whose structure members are defined by *fmt*. The returned value includes the size of the Python object header and alignment.

`test.support.`**`checksizeof`**(*test*, *o*, *size*)

For testcase *test*, assert that the `sys.getsizeof` for *o* plus the GC header size equals *size*.

`@test.support.`**`anticipate_failure`**(*condition*)

A decorator to conditionally mark tests with *unittest.expectedFailure()*. Any use of this decorator should have an associated comment identifying the relevant tracker issue.

`test.support.`**`system_must_validate_cert`**(*f*)

A decorator that skips the decorated test on TLS certification validation failures.

`@test.support.`**`run_with_locale`**(*catstr*, **locales*)

A decorator for running a function in a different locale, correctly resetting it after it has finished. *catstr* is the locale category as a string (for example `"LC_ALL"`). The *locales* passed will be tried sequentially, and the first valid locale will be used.

`@test.support.`**`run_with_tz`**(*tz*)

A decorator for running a function in a specific timezone, correctly resetting it after it has finished.

`@test.support.`**`requires_freebsd_version`**(**min_version*)

Decorator for the minimum version when running test on FreeBSD. If the FreeBSD version is less than the minimum, the test is skipped.

`@test.support.`**`requires_linux_version`**(**min_version*)

Decorator for the minimum version when running test on Linux. If the Linux version is less than the minimum, the test is skipped.

`@test.support.`**`requires_mac_version`**(**min_version*)

Decorator for the minimum version when running test on macOS. If the macOS version is less than the minimum, the test is skipped.

`@test.support.`**`requires_gil_enabled`**

Decorator for skipping tests on the free-threaded build. If the *GIL* is disabled, the test is skipped.

`@test.support.`**`requires_IEEE_754`**

Decorator for skipping tests on non-IEEE 754 platforms.

`@test.support.`**`requires_zlib`**

Decorator for skipping tests if *zlib* doesn't exist.

`@test.support.`**`requires_gzip`**

Decorator for skipping tests if *gzip* doesn't exist.

`@test.support.`**`requires_bz2`**

Decorator for skipping tests if *bz2* doesn't exist.

`@test.support.`**`requires_lzma`**

Decorator for skipping tests if *lzma* doesn't exist.

`@test.support.`**`requires_resource`**(*resource*)

Decorator for skipping tests if *resource* is not available.

`@test.support.`**`requires_docstrings`**

Decorator for only running the test if *HAVE_DOCSTRINGS*.

`@test.support.`**`requires_limited_api`**

Decorator for only running the test if Limited C API is available.

`@test.support.`**`cpython_only`**

Decorator for tests only applicable to CPython.

`@test.support.`**`impl_detail`**(*msg=None*, ***guards*)

Decorator for invoking *check_impl_detail()* on *guards*. If that returns `False`, then uses *msg* as the reason for skipping the test.

`@test.support.`**`no_tracing`**

Decorator to temporarily turn off tracing for the duration of the test.

@test.support.**refcount_test**

Decorator for tests which involve reference counting. The decorator does not run the test if it is not run by CPython. Any trace function is unset for the duration of the test to prevent unexpected refcounts caused by the trace function.

@test.support.**bigmemtest**(*size*, *memuse*, *dry_run=True*)

Decorator for bigmem tests.

size is a requested size for the test (in arbitrary, test-interpreted units.) *memuse* is the number of bytes per unit for the test, or a good estimate of it. For example, a test that needs two byte buffers, of 4 GiB each, could be decorated with `@bigmemtest(size=_4G, memuse=2)`.

The *size* argument is normally passed to the decorated test method as an extra argument. If *dry_run* is `True`, the value passed to the test method may be less than the requested value. If *dry_run* is `False`, it means the test doesn't support dummy runs when `-M` is not specified.

@test.support.**bigaddrspacetest**

Decorator for tests that fill the address space.

test.support.**check_syntax_error**(*testcase*, *statement*, *errtext=''*, ***, *lineno=None*, *offset=None*)

Test for syntax errors in *statement* by attempting to compile *statement*. *testcase* is the *unittest* instance for the test. *errtext* is the regular expression which should match the string representation of the raised *SyntaxError*. If *lineno* is not `None`, compares to the line of the exception. If *offset* is not `None`, compares to the offset of the exception.

test.support.**open_urlresource**(*url*, **args*, ***kw*)

Open *url*. If open fails, raises *TestFailed*.

test.support.**reap_children**()

Use this at the end of `test_main` whenever sub-processes are started. This will help ensure that no extra children (zombies) stick around to hog resources and create problems when looking for refleaks.

test.support.**get_attribute**(*obj*, *name*)

Get an attribute, raising *unittest.SkipTest* if *AttributeError* is raised.

test.support.**catch_unraisable_exception**()

Context manager catching unraisable exception using *sys.unraisablehook()*.

Storing the exception value (`cm.unraisable.exc_value`) creates a reference cycle. The reference cycle is broken explicitly when the context manager exits.

Storing the object (`cm.unraisable.object`) can resurrect it if it is set to an object which is being finalized. Exiting the context manager clears the stored object.

Usage:

```
with support.catch_unraisable_exception() as cm:
    # code creating an "unraisable exception"
    ...

    # check the unraisable exception: use cm.unraisable
    ...

# cm.unraisable attribute no longer exists at this point
# (to break a reference cycle)
```

Added in version 3.8.

test.support.**load_package_tests**(*pkg_dir*, *loader*, *standard_tests*, *pattern*)

Generic implementation of the *unittest* `load_tests` protocol for use in test packages. *pkg_dir* is the root directory of the package; *loader*, *standard_tests*, and *pattern* are the arguments expected by `load_tests`. In simple cases, the test package's `__init__.py` can be the following:

```
import os
from test.support import load_package_tests

def load_tests(*args):
    return load_package_tests(os.path.dirname(__file__), *args)
```

test.support.**detect_api_mismatch**(*ref_api*, *other_api*, *, *ignore=()*)

Returns the set of attributes, functions or methods of *ref_api* not found on *other_api*, except for a defined list of items to be ignored in this check specified in *ignore*.

By default this skips private attributes beginning with '_' but includes all magic methods, i.e. those starting and ending in '__'.

Added in version 3.5.

test.support.**patch**(*test_instance*, *object_to_patch*, *attr_name*, *new_value*)

Override *object_to_patch.attr_name* with *new_value*. Also add cleanup procedure to *test_instance* to restore *object_to_patch* for *attr_name*. The *attr_name* should be a valid attribute for *object_to_patch*.

test.support.**run_in_subinterp**(*code*)

Run *code* in subinterpreter. Raise *unittest.SkipTest* if *tracemalloc* is enabled.

test.support.**check_free_after_iterating**(*test*, *iter*, *cls*, *args=()*)

Assert instances of *cls* are deallocated after iterating.

test.support.**missing_compiler_executable**(*cmd_names=[]*)

Check for the existence of the compiler executables whose names are listed in *cmd_names* or all the compiler executables when *cmd_names* is empty and return the first missing executable or `None` when none is found missing.

test.support.**check__all__**(*test_case*, *module*, *name_of_module=None*, *extra=()*, *not_exported=()*)

Assert that the `__all__` variable of *module* contains all public names.

The module's public names (its API) are detected automatically based on whether they match the public name convention and were defined in *module*.

The *name_of_module* argument can specify (as a string or tuple thereof) what module(s) an API could be defined in order to be detected as a public API. One case for this is when *module* imports part of its public API from other modules, possibly a C backend (like `csv` and its `_csv`).

The *extra* argument can be a set of names that wouldn't otherwise be automatically detected as "public", like objects without a proper `__module__` attribute. If provided, it will be added to the automatically detected ones.

The *not_exported* argument can be a set of names that must not be treated as part of the public API even though their names indicate otherwise.

Example use:

```
import bar
import foo
import unittest
from test import support

class MiscTestCase(unittest.TestCase):
    def test__all__(self):
```

(continues on next page)

(continued from previous page)

```
        support.check__all__(self, foo)

class OtherTestCase(unittest.TestCase):
    def test__all__(self):
        extra = {'BAR_CONST', 'FOO_CONST'}
        not_exported = {'baz'}  # Undocumented name.
        # bar imports part of its API from _bar.
        support.check__all__(self, bar, ('bar', '_bar'),
                             extra=extra, not_exported=not_exported)
```

Added in version 3.6.

test.support.**skip_if_broken_multiprocessing_synchronize**()

Skip tests if the `multiprocessing.synchronize` module is missing, if there is no available semaphore implementation, or if creating a lock raises an *OSError*.

Added in version 3.10.

test.support.**check_disallow_instantiation**(*test_case*, *tp*, **args*, ***kwds*)

Assert that type *tp* cannot be instantiated using *args* and *kwds*.

Added in version 3.10.

test.support.**adjust_int_max_str_digits**(*max_digits*)

This function returns a context manager that will change the global *sys.set_int_max_str_digits()* setting for the duration of the context to allow execution of test code that needs a different limit on the number of digits when converting between an integer and string.

Added in version 3.11.

The *test.support* module defines the following classes:

class test.support.**SuppressCrashReport**

A context manager used to try to prevent crash dialog popups on tests that are expected to crash a subprocess.

On Windows, it disables Windows Error Reporting dialogs using SetErrorMode.

On UNIX, *resource.setrlimit()* is used to set *resource.RLIMIT_CORE*'s soft limit to 0 to prevent coredump file creation.

On both platforms, the old value is restored by `__exit__()`.

class test.support.**SaveSignals**

Class to save and restore signal handlers registered by the Python signal handler.

save(*self*)

Save the signal handlers to a dictionary mapping signal numbers to the current signal handler.

restore(*self*)

Set the signal numbers from the *save()* dictionary to the saved handler.

class test.support.**Matcher**

matches(*self*, *d*, ***kwargs*)

Try to match a single dict with the supplied arguments.

match_value(*self*, *k*, *dv*, *v*)

Try to match a single stored value (*dv*) with a supplied value (*v*).

26.10 test.support.socket_helper — Utilities for socket tests

The *test.support.socket_helper* module provides support for socket tests.

Added in version 3.9.

test.support.socket_helper.**IPV6_ENABLED**

Set to `True` if IPv6 is enabled on this host, `False` otherwise.

test.support.socket_helper.**find_unused_port**(*family=socket.AF_INET*, *socktype=socket.SOCK_STREAM*)

Returns an unused port that should be suitable for binding. This is achieved by creating a temporary socket with the same family and type as the `sock` parameter (default is *AF_INET*, *SOCK_STREAM*), and binding it to the specified host address (defaults to `0.0.0.0`) with the port set to 0, eliciting an unused ephemeral port from the OS. The temporary socket is then closed and deleted, and the ephemeral port is returned.

Either this method or *bind_port()* should be used for any tests where a server socket needs to be bound to a particular port for the duration of the test. Which one to use depends on whether the calling code is creating a Python socket, or if an unused port needs to be provided in a constructor or passed to an external program (i.e. the `-accept` argument to openssl's s_server mode). Always prefer *bind_port()* over *find_unused_port()* where possible. Using a hard coded port is discouraged since it can make multiple instances of the test impossible to run simultaneously, which is a problem for buildbots.

test.support.socket_helper.**bind_port**(*sock*, *host=HOST*)

Bind the socket to a free port and return the port number. Relies on ephemeral ports in order to ensure we are using an unbound port. This is important as many tests may be running simultaneously, especially in a buildbot environment. This method raises an exception if the `sock.family` is *AF_INET* and `sock.type` is *SOCK_STREAM*, and the socket has `SO_REUSEADDR` or `SO_REUSEPORT` set on it. Tests should never set these socket options for TCP/IP sockets. The only case for setting these options is testing multicasting via multiple UDP sockets.

Additionally, if the `SO_EXCLUSIVEADDRUSE` socket option is available (i.e. on Windows), it will be set on the socket. This will prevent anyone else from binding to our host/port for the duration of the test.

test.support.socket_helper.**bind_unix_socket**(*sock*, *addr*)

Bind a Unix socket, raising *unittest.SkipTest* if *PermissionError* is raised.

@test.support.socket_helper.**skip_unless_bind_unix_socket**

A decorator for running tests that require a functional `bind()` for Unix sockets.

test.support.socket_helper.**transient_internet**(*resource_name*, ***, *timeout=30.0*, *errnos=()*)

A context manager that raises *ResourceDenied* when various issues with the internet connection manifest themselves as exceptions.

26.11 test.support.script_helper — Utilities for the Python execution tests

The *test.support.script_helper* module provides support for Python's script execution tests.

test.support.script_helper.**interpreter_requires_environment**()

Return `True` if `sys.executable interpreter` requires environment variables in order to be able to run at all.

This is designed to be used with `@unittest.skipIf()` to annotate tests that need to use an `assert_python*()` function to launch an isolated mode (`-I`) or no environment mode (`-E`) sub-interpreter process.

A normal build & test does not run into this situation but it can happen when trying to run the standard library test suite from an interpreter that doesn't have an obvious home with Python's current home finding logic.

Setting `PYTHONHOME` is one way to get most of the testsuite to run in that situation. `PYTHONPATH` or `PYTHONUSERSITE` are other common environment variables that might impact whether or not the interpreter can start.

`test.support.script_helper.`**`run_python_until_end`**(**args*, ***env_vars*)

Set up the environment based on *env_vars* for running the interpreter in a subprocess. The values can include `__isolated`, `__cleanenv`, `__cwd`, and `TERM`.

Changed in version 3.9: The function no longer strips whitespaces from *stderr*.

`test.support.script_helper.`**`assert_python_ok`**(**args*, ***env_vars*)

Assert that running the interpreter with *args* and optional environment variables *env_vars* succeeds (`rc == 0`) and return a `(return code, stdout, stderr)` tuple.

If the *__cleanenv* keyword-only parameter is set, *env_vars* is used as a fresh environment.

Python is started in isolated mode (command line option `-I`), except if the *__isolated* keyword-only parameter is set to `False`.

Changed in version 3.9: The function no longer strips whitespaces from *stderr*.

`test.support.script_helper.`**`assert_python_failure`**(**args*, ***env_vars*)

Assert that running the interpreter with *args* and optional environment variables *env_vars* fails (`rc != 0`) and return a `(return code, stdout, stderr)` tuple.

See *assert_python_ok()* for more options.

Changed in version 3.9: The function no longer strips whitespaces from *stderr*.

`test.support.script_helper.`**`spawn_python`**(**args*, *stdout=subprocess.PIPE*, *stderr=subprocess.STDOUT*, ***kw*)

Run a Python subprocess with the given arguments.

kw is extra keyword args to pass to *subprocess.Popen()*. Returns a *subprocess.Popen* object.

`test.support.script_helper.`**`kill_python`**(*p*)

Run the given *subprocess.Popen* process until completion and return stdout.

`test.support.script_helper.`**`make_script`**(*script_dir*, *script_basename*, *source*, *omit_suffix=False*)

Create script containing *source* in path *script_dir* and *script_basename*. If *omit_suffix* is `False`, append `.py` to the name. Return the full script path.

`test.support.script_helper.`**`make_zip_script`**(*zip_dir*, *zip_basename*, *script_name*, *name_in_zip=None*)

Create zip file at *zip_dir* and *zip_basename* with extension `zip` which contains the files in *script_name*. *name_in_zip* is the archive name. Return a tuple containing `(full path, full path of archive name)`.

`test.support.script_helper.`**`make_pkg`**(*pkg_dir*, *init_source=''*)

Create a directory named *pkg_dir* containing an `__init__` file with *init_source* as its contents.

`test.support.script_helper.`**`make_zip_pkg`**(*zip_dir*, *zip_basename*, *pkg_name*, *script_basename*, *source*, *depth=1*, *compiled=False*)

Create a zip package directory with a path of *zip_dir* and *zip_basename* containing an empty `__init__` file and a file *script_basename* containing the *source*. If *compiled* is `True`, both source files will be compiled and added to the zip package. Return a tuple of the full zip path and the archive name for the zip file.

26.12 test.support.bytecode_helper — Support tools for testing correct bytecode generation

The *test.support.bytecode_helper* module provides support for testing and inspecting bytecode generation.

Added in version 3.9.

The module defines the following class:

class test.support.bytecode_helper.**BytecodeTestCase**(*unittest.TestCase*)

This class has custom assertion methods for inspecting bytecode.

BytecodeTestCase.**get_disassembly_as_string**(*co*)

Return the disassembly of *co* as string.

BytecodeTestCase.**assertInBytecode**(*x*, *opname*, *argval=_UNSPECIFIED*)

Return instr if *opname* is found, otherwise throws *AssertionError*.

BytecodeTestCase.**assertNotInBytecode**(*x*, *opname*, *argval=_UNSPECIFIED*)

Throws *AssertionError* if *opname* is found.

26.13 test.support.threading_helper — Utilities for threading tests

The *test.support.threading_helper* module provides support for threading tests.

Added in version 3.10.

test.support.threading_helper.**join_thread**(*thread*, *timeout=None*)

Join a *thread* within *timeout*. Raise an *AssertionError* if thread is still alive after *timeout* seconds.

@test.support.threading_helper.**reap_threads**

Decorator to ensure the threads are cleaned up even if the test fails.

test.support.threading_helper.**start_threads**(*threads*, *unlock=None*)

Context manager to start *threads*, which is a sequence of threads. *unlock* is a function called after the threads are started, even if an exception was raised; an example would be *threading.Event.set()*. start_threads will attempt to join the started threads upon exit.

test.support.threading_helper.**threading_cleanup**(**original_values*)

Cleanup up threads not specified in *original_values*. Designed to emit a warning if a test leaves running threads in the background.

test.support.threading_helper.**threading_setup**()

Return current thread count and copy of dangling threads.

test.support.threading_helper.**wait_threads_exit**(*timeout=None*)

Context manager to wait until all threads created in the with statement exit.

test.support.threading_helper.**catch_threading_exception**()

Context manager catching *threading.Thread* exception using *threading.excepthook()*.

Attributes set when an exception is caught:

- exc_type

- exc_value
- exc_traceback
- thread

See *threading.excepthook()* documentation.

These attributes are deleted at the context manager exit.

Usage:

```
with threading_helper.catch_threading_exception() as cm:
    # code spawning a thread which raises an exception
    ...

    # check the thread exception, use cm attributes:
    # exc_type, exc_value, exc_traceback, thread
    ...

# exc_type, exc_value, exc_traceback, thread attributes of cm no longer
# exists at this point
# (to avoid reference cycles)
```

Added in version 3.8.

26.14 test.support.os_helper — Utilities for os tests

The *test.support.os_helper* module provides support for os tests.

Added in version 3.10.

test.support.os_helper.**FS_NONASCII**

A non-ASCII character encodable by *os.fsencode()*.

test.support.os_helper.**SAVEDCWD**

Set to *os.getcwd()*.

test.support.os_helper.**TESTFN**

Set to a name that is safe to use as the name of a temporary file. Any temporary file that is created should be closed and unlinked (removed).

test.support.os_helper.**TESTFN_NONASCII**

Set to a filename containing the *FS_NONASCII* character, if it exists. This guarantees that if the filename exists, it can be encoded and decoded with the default filesystem encoding. This allows tests that require a non-ASCII filename to be easily skipped on platforms where they can't work.

test.support.os_helper.**TESTFN_UNENCODABLE**

Set to a filename (str type) that should not be able to be encoded by file system encoding in strict mode. It may be None if it's not possible to generate such a filename.

test.support.os_helper.**TESTFN_UNDECODABLE**

Set to a filename (bytes type) that should not be able to be decoded by file system encoding in strict mode. It may be None if it's not possible to generate such a filename.

test.support.os_helper.**TESTFN_UNICODE**

Set to a non-ASCII name for a temporary file.

class test.support.os_helper.**EnvironmentVarGuard**

Class used to temporarily set or unset environment variables. Instances can be used as a context manager and have a complete dictionary interface for querying/modifying the underlying `os.environ`. After exit from the context manager all changes to environment variables done through this instance will be rolled back.

Changed in version 3.1: Added dictionary interface.

class test.support.os_helper.**FakePath**(*path*)

Simple *path-like object*. It implements the *__fspath__()* method which just returns the *path* argument. If *path* is an exception, it will be raised in `__fspath__()`.

EnvironmentVarGuard.**set**(*envvar*, *value*)

Temporarily set the environment variable `envvar` to the value of `value`.

EnvironmentVarGuard.**unset**(*envvar*)

Temporarily unset the environment variable `envvar`.

test.support.os_helper.**can_symlink**()

Return `True` if the OS supports symbolic links, `False` otherwise.

test.support.os_helper.**can_xattr**()

Return `True` if the OS supports xattr, `False` otherwise.

test.support.os_helper.**change_cwd**(*path*, *quiet=False*)

A context manager that temporarily changes the current working directory to *path* and yields the directory.

If *quiet* is `False`, the context manager raises an exception on error. Otherwise, it issues only a warning and keeps the current working directory the same.

test.support.os_helper.**create_empty_file**(*filename*)

Create an empty file with *filename*. If it already exists, truncate it.

test.support.os_helper.**fd_count**()

Count the number of open file descriptors.

test.support.os_helper.**fs_is_case_insensitive**(*directory*)

Return `True` if the file system for *directory* is case-insensitive.

test.support.os_helper.**make_bad_fd**()

Create an invalid file descriptor by opening and closing a temporary file, and returning its descriptor.

test.support.os_helper.**rmdir**(*filename*)

Call *os.rmdir()* on *filename*. On Windows platforms, this is wrapped with a wait loop that checks for the existence of the file, which is needed due to antivirus programs that can hold files open and prevent deletion.

test.support.os_helper.**rmtree**(*path*)

Call *shutil.rmtree()* on *path* or call *os.lstat()* and *os.rmdir()* to remove a path and its contents. As with *rmdir()*, on Windows platforms this is wrapped with a wait loop that checks for the existence of the files.

@test.support.os_helper.**skip_unless_symlink**

A decorator for running tests that require support for symbolic links.

@test.support.os_helper.**skip_unless_xattr**

A decorator for running tests that require support for xattr.

test.support.os_helper.**temp_cwd**(*name='tempcwd'*, *quiet=False*)

A context manager that temporarily creates a new directory and changes the current working directory (CWD).

The context manager creates a temporary directory in the current directory with name *name* before temporarily changing the current working directory. If *name* is None, the temporary directory is created using *tempfile.mkdtemp()*.

If *quiet* is False and it is not possible to create or change the CWD, an error is raised. Otherwise, only a warning is raised and the original CWD is used.

test.support.os_helper.**temp_dir**(*path=None*, *quiet=False*)

A context manager that creates a temporary directory at *path* and yields the directory.

If *path* is None, the temporary directory is created using *tempfile.mkdtemp()*. If *quiet* is False, the context manager raises an exception on error. Otherwise, if *path* is specified and cannot be created, only a warning is issued.

test.support.os_helper.**temp_umask**(*umask*)

A context manager that temporarily sets the process umask.

test.support.os_helper.**unlink**(*filename*)

Call *os.unlink()* on *filename*. As with *rmdir()*, on Windows platforms, this is wrapped with a wait loop that checks for the existence of the file.

26.15 test.support.import_helper — Utilities for import tests

The *test.support.import_helper* module provides support for import tests.

Added in version 3.10.

test.support.import_helper.**forget**(*module_name*)

Remove the module named *module_name* from sys.modules and delete any byte-compiled files of the module.

test.support.import_helper.**import_fresh_module**(*name*, *fresh=()*, *blocked=()*, *deprecated=False*)

This function imports and returns a fresh copy of the named Python module by removing the named module from sys.modules before doing the import. Note that unlike reload(), the original module is not affected by this operation.

fresh is an iterable of additional module names that are also removed from the sys.modules cache before doing the import.

blocked is an iterable of module names that are replaced with None in the module cache during the import to ensure that attempts to import them raise *ImportError*.

The named module and any modules named in the *fresh* and *blocked* parameters are saved before starting the import and then reinserted into sys.modules when the fresh import is complete.

Module and package deprecation messages are suppressed during this import if *deprecated* is True.

This function will raise *ImportError* if the named module cannot be imported.

Example use:

```
# Get copies of the warnings module for testing without affecting the
# version being used by the rest of the test suite. One copy uses the
# C implementation, the other is forced to use the pure Python fallback
# implementation
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

Added in version 3.1.

test.support.import_helper.**import_module**(*name*, *deprecated=False*, ***, *required_on=()*)

This function imports and returns the named module. Unlike a normal import, this function raises *unittest.SkipTest* if the module cannot be imported.

Module and package deprecation messages are suppressed during this import if *deprecated* is `True`. If a module is required on a platform but optional for others, set *required_on* to an iterable of platform prefixes which will be compared against *sys.platform*.

Added in version 3.1.

test.support.import_helper.**modules_setup**()

Return a copy of *sys.modules*.

test.support.import_helper.**modules_cleanup**(*oldmodules*)

Remove modules except for *oldmodules* and `encodings` in order to preserve internal cache.

test.support.import_helper.**unload**(*name*)

Delete *name* from `sys.modules`.

test.support.import_helper.**make_legacy_pyc**(*source*)

Move a **PEP 3147/PEP 488** pyc file to its legacy pyc location and return the file system path to the legacy pyc file. The *source* value is the file system path to the source file. It does not need to exist, however the PEP 3147/488 pyc file must exist.

class test.support.import_helper.**CleanImport**(**module_names*)

A context manager to force import to return a new module reference. This is useful for testing module-level behaviors, such as the emission of a *DeprecationWarning* on import. Example usage:

```
with CleanImport('foo'):
    importlib.import_module('foo')  # New reference.
```

class test.support.import_helper.**DirsOnSysPath**(**paths*)

A context manager to temporarily add directories to *sys.path*.

This makes a copy of *sys.path*, appends any directories given as positional arguments, then reverts *sys.path* to the copied settings when the context ends.

Note that *all* *sys.path* modifications in the body of the context manager, including replacement of the object, will be reverted at the end of the block.

26.16 test.support.warnings_helper — Utilities for warnings tests

The *test.support.warnings_helper* module provides support for warnings tests.

Added in version 3.10.

test.support.warnings_helper.**ignore_warnings**(***, *category*)

Suppress warnings that are instances of *category*, which must be *Warning* or a subclass. Roughly equivalent to *warnings.catch_warnings()* with *warnings.simplefilter('ignore', category=category)*. For example:

```
@warning_helper.ignore_warnings(category=DeprecationWarning)
def test_suppress_warning():
    # do something
```

Added in version 3.8.

test.support.warnings_helper.**check_no_resource_warning**(*testcase*)

Context manager to check that no *ResourceWarning* was raised. You must remove the object which may emit *ResourceWarning* before the end of the context manager.

test.support.warnings_helper.**check_syntax_warning**(*testcase*, *statement*, *errtext=''*, ***, *lineno=1*, *offset=None*)

Test for syntax warning in *statement* by attempting to compile *statement*. Test also that the *SyntaxWarning* is emitted only once, and that it will be converted to a *SyntaxError* when turned into error. *testcase* is the *unittest* instance for the test. *errtext* is the regular expression which should match the string representation of the emitted *SyntaxWarning* and raised *SyntaxError*. If *lineno* is not `None`, compares to the line of the warning and exception. If *offset* is not `None`, compares to the offset of the exception.

Added in version 3.8.

test.support.warnings_helper.**check_warnings**(**filters*, *quiet=True*)

A convenience wrapper for *warnings.catch_warnings()* that makes it easier to test that a warning was correctly raised. It is approximately equivalent to calling `warnings.catch_warnings(record=True)` with *warnings.simplefilter()* set to `always` and with the option to automatically validate the results that are recorded.

`check_warnings` accepts 2-tuples of the form `("message regexp", WarningCategory)` as positional arguments. If one or more *filters* are provided, or if the optional keyword argument *quiet* is `False`, it checks to make sure the warnings are as expected: each specified filter must match at least one of the warnings raised by the enclosed code or the test fails, and if any warnings are raised that do not match any of the specified filters the test fails. To disable the first of these checks, set *quiet* to `True`.

If no arguments are specified, it defaults to:

```
check_warnings(("", Warning), quiet=True)
```

In this case all warnings are caught and no errors are raised.

On entry to the context manager, a `WarningRecorder` instance is returned. The underlying warnings list from *catch_warnings()* is available via the recorder object's *warnings* attribute. As a convenience, the attributes of the object representing the most recent warning can also be accessed directly through the recorder object (see example below). If no warning has been raised, then any of the attributes that would otherwise be expected on an object representing a warning will return `None`.

The recorder object also has a `reset()` method, which clears the warnings list.

The context manager is designed to be used like this:

```
with check_warnings(("assertion is always true", SyntaxWarning),
                    ("", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

In this case if either warning was not raised, or some other warning was raised, *check_warnings()* would raise an error.

When a test needs to look more deeply into the warnings, rather than just checking whether or not they occurred, code like this can be used:

```
with check_warnings(quiet=True) as w:
    warnings.warn("foo")
    assert str(w.args[0]) == "foo"
    warnings.warn("bar")
```

(continues on next page)

(continued from previous page)

```
    assert str(w.args[0]) == "bar"
    assert str(w.warnings[0].args[0]) == "foo"
    assert str(w.warnings[1].args[0]) == "bar"
    w.reset()
    assert len(w.warnings) == 0
```

Here all warnings will be caught, and the test code tests the captured warnings directly.

Changed in version 3.2: New optional arguments *filters* and *quiet*.

class test.support.warnings_helper.**WarningsRecorder**

Class used to record warnings for unit tests. See documentation of *check_warnings()* above for more details.

CHAPTER

TWENTYSEVEN

DEBUGGING AND PROFILING

These libraries help you with Python development: the debugger enables you to step through code, analyze stack frames and set breakpoints etc., and the profilers run code and give you a detailed breakdown of execution times, allowing you to identify bottlenecks in your programs. Auditing events provide visibility into runtime behaviors that would otherwise require intrusive debugging or patching.

27.1 Audit events table

This table contains all events raised by *sys.audit()* or PySys_Audit() calls throughout the CPython runtime and the standard library. These calls were added in 3.8 or later (see **PEP 578**).

See *sys.addaudithook()* and PySys_AddAuditHook() for information on handling these events.

CPython implementation detail: This table is generated from the CPython documentation, and may not represent events raised by other implementations. See your runtime specific documentation for actual events raised.

Audit event	Arguments
_thread.start_new_thread	function, args, kwargs
array.__new__	typecode, initializer
builtins.breakpoint	breakpointhook
builtins.id	id
builtins.input	prompt
builtins.input/result	result
code.__new__	code, filename, name, argcount, posonlyargcount, kwonlyargcount, nlocals
compile	source, filename
cpython.PyInterpreterState_Clear	
cpython.PyInterpreterState_New	
cpython._PySys_ClearAuditHooks	
cpython.run_command	command
cpython.run_file	filename
cpython.run_interactivehook	hook
cpython.run_module	module-name
cpython.run_startup	filename
cpython.run_stdin	
ctypes.addressof	obj
ctypes.call_function	func_pointer, arguments
ctypes.cdata	address
ctypes.cdata/buffer	pointer, size, offset
ctypes.create_string_buffer	init, size

Table 1 – continued from previous page

Audit event	Arguments
ctypes.create_unicode_buffer	init, size
ctypes.dlopen	name
ctypes.dlsym	library, name
ctypes.dlsym/handle	handle, name
ctypes.get_errno	
ctypes.get_last_error	
ctypes.set_errno	errno
ctypes.set_exception	code
ctypes.set_last_error	error
ctypes.string_at	ptr, size
ctypes.wstring_at	ptr, size
ensurepip.bootstrap	root
exec	code_object
fcntl.fcntl	fd, cmd, arg
fcntl.flock	fd, operation
fcntl.ioctl	fd, request, arg
fcntl.lockf	fd, cmd, len, start, whence
ftplib.connect	self, host, port
ftplib.sendcmd	self, cmd
function.__new__	code
gc.get_objects	generation
gc.get_referents	objs
gc.get_referrers	objs
glob.glob	pathname, recursive
glob.glob/2	pathname, recursive, root_dir, dir_fd
http.client.connect	self, host, port
http.client.send	self, data
imaplib.open	self, host, port
imaplib.send	self, data
import	module, filename, sys.path, sys.meta_path, sys.path_hooks
marshal.dumps	value, version
marshal.load	
marshal.loads	bytes
mmap.__new__	fileno, length, access, offset
msvcrt.get_osfhandle	fd
msvcrt.locking	fd, mode, nbytes
msvcrt.open_osfhandle	handle, flags
object.__delattr__	obj, name
object.__getattr__	obj, name
object.__setattr__	obj, name, value
open	path, mode, flags
os.add_dll_directory	path
os.chdir	path
os.chflags	path, flags
os.chmod	path, mode, dir_fd
os.chown	path, uid, gid, dir_fd
os.exec	path, args, env
os.fork	
os.forkpty	
os.fwalk	top, topdown, onerror, follow_symlinks, dir_fd

Table 1 – continued from previous page

Audit event	Arguments
os.getxattr	path, attribute
os.kill	pid, sig
os.killpg	pgid, sig
os.link	src, dst, src_dir_fd, dst_dir_fd
os.listdir	path
os.listdrives	
os.listmounts	volume
os.listvolumes	
os.listxattr	path
os.lockf	fd, cmd, len
os.mkdir	path, mode, dir_fd
os.posix_spawn	path, argv, env
os.putenv	key, value
os.remove	path, dir_fd
os.removexattr	path, attribute
os.rename	src, dst, src_dir_fd, dst_dir_fd
os.rmdir	path, dir_fd
os.scandir	path
os.setxattr	path, attribute, value, flags
os.spawn	mode, path, args, env
os.startfile	path, operation
os.startfile/2	path, operation, arguments, cwd, show_cmd
os.symlink	src, dst, dir_fd
os.system	command
os.truncate	fd, length
os.unsetenv	key
os.utime	path, times, ns, dir_fd
os.walk	top, topdown, onerror, followlinks
pathlib.Path.glob	self, pattern
pathlib.Path.rglob	self, pattern
pdb.Pdb	
pickle.find_class	module, name
poplib.connect	self, host, port
poplib.putline	self, line
pty.spawn	argv
resource.prlimit	pid, resource, limits
resource.setrlimit	resource, limits
setopencodehook	
shutil.chown	path, user, group
shutil.copyfile	src, dst
shutil.copymode	src, dst
shutil.copystat	src, dst
shutil.copytree	src, dst
shutil.make_archive	base_name, format, root_dir, base_dir
shutil.move	src, dst
shutil.rmtree	path, dir_fd
shutil.unpack_archive	filename, extract_dir, format
signal.pthread_kill	thread_id, signalnum
smtplib.connect	self, host, port
smtplib.send	self, data

Table 1 – continued from previous page

Audit event	Arguments
socket.__new__	self, family, type, protocol
socket.bind	self, address
socket.connect	self, address
socket.getaddrinfo	host, port, family, type, protocol
socket.gethostbyaddr	ip_address
socket.gethostbyname	hostname
socket.gethostname	
socket.getnameinfo	sockaddr
socket.getservbyname	servicename, protocolname
socket.getservbyport	port, protocolname
socket.sendmsg	self, address
socket.sendto	self, address
socket.sethostname	name
sqlite3.connect	database
sqlite3.connect/handle	connection_handle
sqlite3.enable_load_extension	connection, enabled
sqlite3.load_extension	connection, path
subprocess.Popen	executable, args, cwd, env
sys._current_exceptions	
sys._current_frames	
sys._getframe	frame
sys._getframemodulename	depth
sys.addaudithook	
sys.excepthook	hook, type, value, traceback
sys.set_asyncgen_hooks_finalizer	
sys.set_asyncgen_hooks_firstiter	
sys.setprofile	
sys.settrace	
sys.unraisablehook	hook, unraisable
syslog.closelog	
syslog.openlog	ident, logoption, facility
syslog.setlogmask	maskpri
syslog.syslog	priority, message
tempfile.mkdtemp	fullpath
tempfile.mkstemp	fullpath
time.sleep	secs
urllib.Request	fullurl, data, headers, method
webbrowser.open	url
winreg.ConnectRegistry	computer_name, key
winreg.CreateKey	key, sub_key, access
winreg.DeleteKey	key, sub_key, access
winreg.DeleteValue	key, value
winreg.DisableReflectionKey	key
winreg.EnableReflectionKey	key
winreg.EnumKey	key, index
winreg.EnumValue	key, index
winreg.ExpandEnvironmentStrings	str
winreg.LoadKey	key, sub_key, file_name
winreg.OpenKey	key, sub_key, access
winreg.OpenKey/result	key

Table 1 – continued from previous page

Audit event	Arguments
winreg.PyHKEY.Detach	`key`
winreg.QueryInfoKey	`key`
winreg.QueryReflectionKey	`key`
winreg.QueryValue	`key`, `sub_key`, `value_name`
winreg.SaveKey	`key`, `file_name`
winreg.SetValue	`key`, `sub_key`, `type`, `value`

The following events are raised internally and do not correspond to any public API of CPython:

Audit event	Arguments
_winapi.CreateFile	`file_name`, `desired_access`, `share_mode`, `creation_disposition`, `flags_and_attributes`
_winapi.CreateJunctio	`src_path`, `dst_path`
_winapi.CreateNamedI	`name`, `open_mode`, `pipe_mode`
_winapi.CreatePipe	
_winapi.CreateProcess	`application_name`, `command_line`, `current_directory`
_winapi.OpenProcess	`process_id`, `desired_access`
_winapi.TerminatePro	`handle`, `exit_code`
ctypes.PyObj_FromPt	`obj`

27.2 bdb — Debugger framework

Source code: Lib/bdb.py

The *bdb* module handles basic debugger functions, like setting breakpoints or managing execution via the debugger.

The following exception is defined:

exception bdb.**BdbQuit**

Exception raised by the *Bdb* class for quitting the debugger.

The *bdb* module also defines two classes:

class bdb.**Breakpoint**(*self*, *file*, *line*, *temporary=False*, *cond=None*, *funcname=None*)

This class implements temporary breakpoints, ignore counts, disabling and (re-)enabling, and conditionals.

Breakpoints are indexed by number through a list called *bpbynumber* and by `(file, line)` pairs through *bplist*. The former points to a single instance of class *Breakpoint*. The latter points to a list of such instances since there may be more than one breakpoint per line.

When creating a breakpoint, its associated *file name* should be in canonical form. If a *funcname* is defined, a breakpoint *hit* will be counted when the first line of that function is executed. A *conditional* breakpoint always counts a *hit*.

Breakpoint instances have the following methods:

deleteMe()

Delete the breakpoint from the list associated to a file/line. If it is the last breakpoint in that position, it also deletes the entry for the file/line.

enable()
Mark the breakpoint as enabled.

disable()
Mark the breakpoint as disabled.

bpformat()
Return a string with all the information about the breakpoint, nicely formatted:

- Breakpoint number.
- Temporary status (del or keep).
- File/line position.
- Break condition.
- Number of times to ignore.
- Number of times hit.

Added in version 3.2.

bpprint(*out=None*)
Print the output of *bpformat()* to the file *out*, or if it is None, to standard output.

Breakpoint instances have the following attributes:

file
File name of the *Breakpoint*.

line
Line number of the *Breakpoint* within *file*.

temporary
True if a *Breakpoint* at (file, line) is temporary.

cond
Condition for evaluating a *Breakpoint* at (file, line).

funcname
Function name that defines whether a *Breakpoint* is hit upon entering the function.

enabled
True if *Breakpoint* is enabled.

bpbynumber
Numeric index for a single instance of a *Breakpoint*.

bplist
Dictionary of *Breakpoint* instances indexed by (*file*, *line*) tuples.

ignore
Number of times to ignore a *Breakpoint*.

hits
Count of the number of times a *Breakpoint* has been hit.

class bdb.**Bdb**(*skip=None*)

The *Bdb* class acts as a generic Python debugger base class.

This class takes care of the details of the trace facility; a derived class should implement user interaction. The standard debugger class (*pdb.Pdb*) is an example.

The *skip* argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. Whether a frame is considered to originate in a certain module is determined by the `__name__` in the frame globals.

Changed in version 3.1: Added the *skip* parameter.

The following methods of *Bdb* normally don't need to be overridden.

canonic(*filename*)

Return canonical form of *filename*.

For real file names, the canonical form is an operating-system-dependent, *case-normalized absolute path*. A *filename* with angle brackets, such as `"<stdin>"` generated in interactive mode, is returned unchanged.

reset()

Set the `botframe`, `stopframe`, `returnframe` and *quitting* attributes with values ready to start debugging.

trace_dispatch(*frame*, *event*, *arg*)

This function is installed as the trace function of debugged frames. Its return value is the new trace function (in most cases, that is, itself).

The default implementation decides how to dispatch a frame, depending on the type of event (passed as a string) that is about to be executed. *event* can be one of the following:

- `"line"`: A new line of code is going to be executed.
- `"call"`: A function is about to be called, or another code block entered.
- `"return"`: A function or other code block is about to return.
- `"exception"`: An exception has occurred.
- `"c_call"`: A C function is about to be called.
- `"c_return"`: A C function has returned.
- `"c_exception"`: A C function has raised an exception.

For the Python events, specialized functions (see below) are called. For the C events, no action is taken.

The *arg* parameter depends on the previous event.

See the documentation for *sys.settrace()* for more information on the trace function. For more information on code and frame objects, refer to types.

dispatch_line(*frame*)

If the debugger should stop on the current line, invoke the *user_line()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the *quitting* flag is set (which can be set from *user_line()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_call(*frame*, *arg*)

If the debugger should stop on this function call, invoke the *user_call()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the *quitting* flag is set (which can be set from *user_call()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_return(*frame*, *arg*)

If the debugger should stop on this function return, invoke the *user_return()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the *quitting* flag is set (which can be set from *user_return()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_exception(*frame*, *arg*)

If the debugger should stop at this exception, invokes the *user_exception()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the *quitting* flag is set (which can be set from *user_exception()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

Normally derived classes don't override the following methods, but they may if they want to redefine the definition of stopping and breakpoints.

is_skipped_line(*module_name*)

Return `True` if *module_name* matches any skip pattern.

stop_here(*frame*)

Return `True` if *frame* is below the starting frame in the stack.

break_here(*frame*)

Return `True` if there is an effective breakpoint for this line.

Check whether a line or function breakpoint exists and is in effect. Delete temporary breakpoints based on information from *effective()*.

break_anywhere(*frame*)

Return `True` if any breakpoint exists for *frame*'s filename.

Derived classes should override these methods to gain control over debugger operation.

user_call(*frame*, *argument_list*)

Called from *dispatch_call()* if a break might stop inside the called function.

argument_list is not used anymore and will always be `None`. The argument is kept for backwards compatibility.

user_line(*frame*)

Called from *dispatch_line()* when either *stop_here()* or *break_here()* returns `True`.

user_return(*frame*, *return_value*)

Called from *dispatch_return()* when *stop_here()* returns `True`.

user_exception(*frame*, *exc_info*)

Called from *dispatch_exception()* when *stop_here()* returns `True`.

do_clear(*arg*)

Handle how a breakpoint must be removed when it is a temporary one.

This method must be implemented by derived classes.

Derived classes and clients can call the following methods to affect the stepping state.

set_step()

Stop after one line of code.

set_next(*frame*)

Stop on the next line in or below the given frame.

set_return(*frame*)

Stop when returning from the given frame.

set_until(*frame*, *lineno=None*)

Stop when the line with the *lineno* greater than the current one is reached or when returning from current frame.

set_trace([*frame*])

Start debugging from *frame*. If *frame* is not specified, debugging starts from caller's frame.

Changed in version 3.13: `set_trace()` will enter the debugger immediately, rather than on the next line of code to be executed.

set_continue()

Stop only at breakpoints or when finished. If there are no breakpoints, set the system trace function to `None`.

set_quit()

Set the `quitting` attribute to `True`. This raises `BdbQuit` in the next call to one of the `dispatch_*()` methods.

Derived classes and clients can call the following methods to manipulate breakpoints. These methods return a string containing an error message if something went wrong, or `None` if all is well.

set_break(*filename*, *lineno*, *temporary=False*, *cond=None*, *funcname=None*)

Set a new breakpoint. If the *lineno* line doesn't exist for the *filename* passed as argument, return an error message. The *filename* should be in canonical form, as described in the `canonic()` method.

clear_break(*filename*, *lineno*)

Delete the breakpoints in *filename* and *lineno*. If none were set, return an error message.

clear_bpbynumber(*arg*)

Delete the breakpoint which has the index *arg* in the `Breakpoint.bpbynumber`. If *arg* is not numeric or out of range, return an error message.

clear_all_file_breaks(*filename*)

Delete all breakpoints in *filename*. If none were set, return an error message.

clear_all_breaks()

Delete all existing breakpoints. If none were set, return an error message.

get_bpbynumber(*arg*)

Return a breakpoint specified by the given number. If *arg* is a string, it will be converted to a number. If *arg* is a non-numeric string, if the given breakpoint never existed or has been deleted, a `ValueError` is raised.

Added in version 3.2.

get_break(*filename*, *lineno*)

Return `True` if there is a breakpoint for *lineno* in *filename*.

get_breaks(*filename*, *lineno*)

Return all breakpoints for *lineno* in *filename*, or an empty list if none are set.

get_file_breaks(*filename*)

Return all breakpoints in *filename*, or an empty list if none are set.

get_all_breaks()

Return all breakpoints that are set.

Derived classes and clients can call the following methods to get a data structure representing a stack trace.

get_stack(*f*, *t*)

Return a list of (frame, lineno) tuples in a stack trace, and a size.

The most recently called frame is last in the list. The size is the number of frames below the frame where the debugger was invoked.

format_stack_entry(*frame_lineno*, *lprefix=': '*)

Return a string with information about a stack entry, which is a `(frame, lineno)` tuple. The return string contains:

- The canonical filename which contains the frame.
- The function name or `"<lambda>"`.
- The input arguments.
- The return value.
- The line of code (if it exists).

The following two methods can be called by clients to use a debugger to debug a *statement*, given as a string.

run(*cmd*, *globals=None*, *locals=None*)

Debug a statement executed via the *exec()* function. *globals* defaults to `__main__.__dict__`, *locals* defaults to *globals*.

runeval(*expr*, *globals=None*, *locals=None*)

Debug an expression executed via the *eval()* function. *globals* and *locals* have the same meaning as in *run()*.

runctx(*cmd*, *globals*, *locals*)

For backwards compatibility. Calls the *run()* method.

runcall(*func*, */*, **args*, ***kwds*)

Debug a single function call, and return its result.

Finally, the module defines the following functions:

bdb.**checkfuncname**(*b*, *frame*)

Return `True` if we should break here, depending on the way the *Breakpoint* *b* was set.

If it was set via line number, it checks if *b.line* is the same as the one in *frame*. If the breakpoint was set via *function name*, we have to check we are in the right *frame* (the right function) and if we are on its first executable line.

bdb.**effective**(*file*, *line*, *frame*)

Return `(active breakpoint, delete temporary flag)` or `(None, None)` as the breakpoint to act upon.

The *active breakpoint* is the first entry in *bplist* for the (*file*, *line*) (which must exist) that is *enabled*, for which *checkfuncname()* is true, and that has neither a false *condition* nor positive *ignore* count. The *flag*, meaning that a temporary breakpoint should be deleted, is `False` only when the *cond* cannot be evaluated (in which case, *ignore* count is ignored).

If no such entry exists, then `(None, None)` is returned.

bdb.**set_trace**()

Start debugging with a *Bdb* instance from caller's frame.

27.3 faulthandler — Dump the Python traceback

Added in version 3.3.

This module contains functions to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal. Call *faulthandler.enable()* to install fault handlers for the *SIGSEGV*, *SIGFPE*, *SIGABRT*, *SIGBUS*, and *SIGILL* signals. You can also enable them at startup by setting the PYTHONFAULTHANDLER environment variable or by using the -X faulthandler command line option.

The fault handler is compatible with system fault handlers like Apport or the Windows fault handler. The module uses an alternative stack for signal handlers if the sigaltstack() function is available. This allows it to dump the traceback even on a stack overflow.

The fault handler is called on catastrophic cases and therefore can only use signal-safe functions (e.g. it cannot allocate memory on the heap). Because of this limitation traceback dumping is minimal compared to normal Python tracebacks:

- Only ASCII is supported. The backslashreplace error handler is used on encoding.
- Each string is limited to 500 characters.
- Only the filename, the function name and the line number are displayed. (no source code)
- It is limited to 100 frames and 100 threads.
- The order is reversed: the most recent call is shown first.

By default, the Python traceback is written to *sys.stderr*. To see tracebacks, applications must be run in the terminal. A log file can alternatively be passed to *faulthandler.enable()*.

The module is implemented in C, so tracebacks can be dumped on a crash or when Python is deadlocked.

The *Python Development Mode* calls *faulthandler.enable()* at Python startup.

See also

Module *pdb*
Interactive source code debugger for Python programs.

Module *traceback*
Standard interface to extract, format and print stack traces of Python programs.

27.3.1 Dumping the traceback

faulthandler.**dump_traceback**(*file=sys.stderr*, *all_threads=True*)
Dump the tracebacks of all threads into *file*. If *all_threads* is False, dump only the current thread.

See also

traceback.print_tb(), which can be used to print a traceback object.

Changed in version 3.5: Added support for passing file descriptor to this function.

27.3.2 Fault handler state

faulthandler.**enable**(*file=sys.stderr*, *all_threads=True*)

Enable the fault handler: install handlers for the *SIGSEGV*, *SIGFPE*, *SIGABRT*, *SIGBUS* and *SIGILL* signals to dump the Python traceback. If *all_threads* is `True`, produce tracebacks for every running thread. Otherwise, dump only the current thread.

The *file* must be kept open until the fault handler is disabled: see *issue with file descriptors*.

Changed in version 3.5: Added support for passing file descriptor to this function.

Changed in version 3.6: On Windows, a handler for Windows exception is also installed.

Changed in version 3.10: The dump now mentions if a garbage collector collection is running if *all_threads* is true.

faulthandler.**disable**()

Disable the fault handler: uninstall the signal handlers installed by *enable()*.

faulthandler.**is_enabled**()

Check if the fault handler is enabled.

27.3.3 Dumping the tracebacks after a timeout

faulthandler.**dump_traceback_later**(*timeout*, *repeat=False*, *file=sys.stderr*, *exit=False*)

Dump the tracebacks of all threads, after a timeout of *timeout* seconds, or every *timeout* seconds if *repeat* is `True`. If *exit* is `True`, call `_exit()` with status=1 after dumping the tracebacks. (Note `_exit()` exits the process immediately, which means it doesn't do any cleanup like flushing file buffers.) If the function is called twice, the new call replaces previous parameters and resets the timeout. The timer has a sub-second resolution.

The *file* must be kept open until the traceback is dumped or *cancel_dump_traceback_later()* is called: see *issue with file descriptors*.

This function is implemented using a watchdog thread.

Changed in version 3.5: Added support for passing file descriptor to this function.

Changed in version 3.7: This function is now always available.

faulthandler.**cancel_dump_traceback_later**()

Cancel the last call to *dump_traceback_later()*.

27.3.4 Dumping the traceback on a user signal

faulthandler.**register**(*signum*, *file=sys.stderr*, *all_threads=True*, *chain=False*)

Register a user signal: install a handler for the *signum* signal to dump the traceback of all threads, or of the current thread if *all_threads* is `False`, into *file*. Call the previous handler if chain is `True`.

The *file* must be kept open until the signal is unregistered by *unregister()*: see *issue with file descriptors*.

Not available on Windows.

Changed in version 3.5: Added support for passing file descriptor to this function.

faulthandler.**unregister**(*signum*)

Unregister a user signal: uninstall the handler of the *signum* signal installed by *register()*. Return `True` if the signal was registered, `False` otherwise.

Not available on Windows.

27.3.5 Issue with file descriptors

enable(), *dump_traceback_later()* and *register()* keep the file descriptor of their *file* argument. If the file is closed and its file descriptor is reused by a new file, or if *os.dup2()* is used to replace the file descriptor, the traceback will be written into a different file. Call these functions again each time that the file is replaced.

27.3.6 Example

Example of a segmentation fault on Linux with and without enabling the fault handler:

```
$ python -c "import ctypes; ctypes.string_at(0)"
Segmentation fault

$ python -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault

Current thread 0x00007fb899f39700 (most recent call first):
  File "/home/python/cpython/Lib/ctypes/__init__.py", line 486 in string_at
  File "<stdin>", line 1 in <module>
Segmentation fault
```

27.4 pdb — The Python Debugger

Source code: Lib/pdb.py

The module *pdb* defines an interactive source code debugger for Python programs. It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame. It also supports post-mortem debugging and can be called under program control.

The debugger is extensible – it is actually defined as the class *Pdb*. This is currently undocumented but easily understood by reading the source. The extension interface uses the modules *bdb* and *cmd*.

See also

Module *faulthandler*
Used to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal.

Module *traceback*
Standard interface to extract, format and print stack traces of Python programs.

The typical usage to break into the debugger is to insert:

```
import pdb; pdb.set_trace()
```

Or:

```
breakpoint()
```

at the location you want to break into the debugger, and then run the program. You can then step through the code following this statement, and continue running without the debugger using the *continue* command.

Changed in version 3.7: The built-in *breakpoint()*, when called with defaults, can be used instead of `import pdb; pdb.set_trace()`.

```
def double(x):
   breakpoint()
   return x * 2
val = 3
print(f"{val} * 2 is {double(val)}")
```

The debugger's prompt is `(Pdb)`, which is the indicator that you are in debug mode:

```
> ...(2)double()
-> breakpoint()
(Pdb) p x
3
(Pdb) continue
3 * 2 is 6
```

Changed in version 3.3: Tab-completion via the *readline* module is available for commands and command arguments, e.g. the current global and local names are offered as arguments of the `p` command.

You can also invoke *pdb* from the command line to debug other scripts. For example:

```
python -m pdb myscript.py
```

When invoked as a module, pdb will automatically enter post-mortem debugging if the program being debugged exits abnormally. After post-mortem debugging (or after normal exit of the program), pdb will restart the program. Automatic restarting preserves pdb's state (such as breakpoints) and in most cases is more useful than quitting the debugger upon program's exit.

Changed in version 3.2: Added the `-c` option to execute commands as if given in a `.pdbrc` file; see *Debugger Commands*.

Changed in version 3.7: Added the `-m` option to execute modules similar to the way `python -m` does. As with a script, the debugger will pause execution just before the first line of the module.

Typical usage to execute a statement under control of the debugger is:

```
>>> import pdb
>>> def f(x):
...     print(1 / x)
>>> pdb.run("f(2)")
> <string>(1)<module>()
(Pdb) continue
0.5
>>>
```

The typical usage to inspect a crashed program is:

```
>>> import pdb
>>> def f(x):
...     print(1 / x)
...
>>> f(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
```

(continues on next page)

(continued from previous page)

```
ZeroDivisionError: division by zero
>>> pdb.pm()
> <stdin>(2)f()
(Pdb) p x
0
(Pdb)
```

Changed in version 3.13: The implementation of **PEP 667** means that name assignments made via `pdb` will immediately affect the active scope, even when running inside an *optimized scope*.

The module defines the following functions; each enters the debugger in a slightly different way:

`pdb.`**`run`**(*statement*, *globals=None*, *locals=None*)

Execute the *statement* (given as a string or a code object) under debugger control. The debugger prompt appears before any code is executed; you can set breakpoints and type `continue`, or you can step through the statement using `step` or `next` (all these commands are explained below). The optional *globals* and *locals* arguments specify the environment in which the code is executed; by default the dictionary of the module `__main__` is used. (See the explanation of the built-in `exec()` or `eval()` functions.)

`pdb.`**`runeval`**(*expression*, *globals=None*, *locals=None*)

Evaluate the *expression* (given as a string or a code object) under debugger control. When `runeval()` returns, it returns the value of the *expression*. Otherwise this function is similar to `run()`.

`pdb.`**`runcall`**(*function*, **args*, ***kwds*)

Call the *function* (a function or method object, not a string) with the given arguments. When `runcall()` returns, it returns whatever the function call returned. The debugger prompt appears as soon as the function is entered.

`pdb.`**`set_trace`**(*, *header=None*)

Enter the debugger at the calling stack frame. This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails). If given, *header* is printed to the console just before debugging begins.

Changed in version 3.7: The keyword-only argument *header*.

Changed in version 3.13: `set_trace()` will enter the debugger immediately, rather than on the next line of code to be executed.

`pdb.`**`post_mortem`**(*traceback=None*)

Enter post-mortem debugging of the given *traceback* object. If no *traceback* is given, it uses the one of the exception that is currently being handled (an exception must be being handled if the default is to be used).

`pdb.`**`pm`**()

Enter post-mortem debugging of the exception found in `sys.last_exc`.

The `run*` functions and `set_trace()` are aliases for instantiating the `Pdb` class and calling the method of the same name. If you want to access further features, you have to do this yourself:

`class` `pdb.`**`Pdb`**(*completekey='tab'*, *stdin=None*, *stdout=None*, *skip=None*, *nosigint=False*, *readrc=True*)

`Pdb` is the debugger class.

The *completekey*, *stdin* and *stdout* arguments are passed to the underlying `cmd.Cmd` class; see the description there.

The *skip* argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns.[1]

[1] Whether a frame is considered to originate in a certain module is determined by the `__name__` in the frame globals.

By default, Pdb sets a handler for the SIGINT signal (which is sent when the user presses `Ctrl-C` on the console) when you give a *continue* command. This allows you to break into the debugger again by pressing `Ctrl-C`. If you want Pdb not to touch the SIGINT handler, set *nosigint* to true.

The *readrc* argument defaults to true and controls whether Pdb will load .pdbrc files from the filesystem.

Example call to enable tracing with *skip*:

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

Raises an *auditing event* `pdb.Pdb` with no arguments.

Changed in version 3.1: Added the *skip* parameter.

Changed in version 3.2: Added the *nosigint* parameter. Previously, a SIGINT handler was never set by Pdb.

Changed in version 3.6: The *readrc* argument.

run(*statement*, *globals=None*, *locals=None*)

runeval(*expression*, *globals=None*, *locals=None*)

runcall(*function*, **args*, ***kwds*)

set_trace()

See the documentation for the functions explained above.

27.4.1 Debugger Commands

The commands recognized by the debugger are listed below. Most commands can be abbreviated to one or two letters as indicated; e.g. `h(elp)` means that either `h` or `help` can be used to enter the help command (but not `he` or `hel`, nor `H` or `Help` or `HELP`). Arguments to commands must be separated by whitespace (spaces or tabs). Optional arguments are enclosed in square brackets (`[]`) in the command syntax; the square brackets must not be typed. Alternatives in the command syntax are separated by a vertical bar (`|`).

Entering a blank line repeats the last command entered. Exception: if the last command was a *list* command, the next 11 lines are listed.

Commands that the debugger doesn't recognize are assumed to be Python statements and are executed in the context of the program being debugged. Python statements can also be prefixed with an exclamation point (`!`). This is a powerful way to inspect the program being debugged; it is even possible to change a variable or call a function. When an exception occurs in such a statement, the exception name is printed but the debugger's state is not changed.

Changed in version 3.13: Expressions/Statements whose prefix is a pdb command are now correctly identified and executed.

The debugger supports *aliases*. Aliases can have parameters which allows one a certain level of adaptability to the context under examination.

Multiple commands may be entered on a single line, separated by `;;`. (A single `;` is not used as it is the separator for multiple commands in a line that is passed to the Python parser.) No intelligence is applied to separating the commands; the input is split at the first `;;` pair, even if it is in the middle of a quoted string. A workaround for strings with double semicolons is to use implicit string concatenation `';'';'` or `";"";"`.

To set a temporary global variable, use a *convenience variable*. A *convenience variable* is a variable whose name starts with `$`. For example, `$foo = 1` sets a global variable `$foo` which you can use in the debugger session. The *convenience variables* are cleared when the program resumes execution so it's less likely to interfere with your program compared to using normal variables like `foo = 1`.

There are three preset *convenience variables*:

- `$_frame`: the current frame you are debugging

- `$_retval`: the return value if the frame is returning
- `$_exception`: the exception if the frame is raising an exception

Added in version 3.12: Added the *convenience variable* feature.

If a file `.pdbrc` exists in the user's home directory or in the current directory, it is read with `'utf-8'` encoding and executed as if it had been typed at the debugger prompt, with the exception that empty lines and lines starting with `#` are ignored. This is particularly useful for aliases. If both files exist, the one in the home directory is read first and aliases defined there can be overridden by the local file.

Changed in version 3.2: `.pdbrc` can now contain commands that continue debugging, such as *continue* or *next*. Previously, these commands had no effect.

Changed in version 3.11: `.pdbrc` is now read with `'utf-8'` encoding. Previously, it was read with the system locale encoding.

h(elp) `[command]`

Without argument, print the list of available commands. With a *command* as argument, print help about that command. `help pdb` displays the full documentation (the docstring of the *pdb* module). Since the *command* argument must be an identifier, `help exec` must be entered to get help on the `!` command.

w(here)

Print a stack trace, with the most recent frame at the bottom. An arrow (>) indicates the current frame, which determines the context of most commands.

d(own) `[count]`

Move the current frame *count* (default one) levels down in the stack trace (to a newer frame).

u(p) `[count]`

Move the current frame *count* (default one) levels up in the stack trace (to an older frame).

b(reak) `[([filename:]lineno | function) [, condition]]`

With a *lineno* argument, set a break at line *lineno* in the current file. The line number may be prefixed with a *filename* and a colon, to specify a breakpoint in another file (possibly one that hasn't been loaded yet). The file is searched on *sys.path*. Accepatable forms of *filename* are `/abspath/to/file.py`, `relpath/file.py`, `module` and `package.module`.

With a *function* argument, set a break at the first executable statement within that function. *function* can be any expression that evaluates to a function in the current namespace.

If a second argument is present, it is an expression which must evaluate to true before the breakpoint is honored.

Without argument, list all breaks, including for each breakpoint, the number of times that breakpoint has been hit, the current ignore count, and the associated condition if any.

Each breakpoint is assigned a number to which all the other breakpoint commands refer.

tbreak `[([filename:]lineno | function) [, condition]]`

Temporary breakpoint, which is removed automatically when it is first hit. The arguments are the same as for *break*.

cl(ear) `[filename:lineno | bpnumber ...]`

With a *filename:lineno* argument, clear all the breakpoints at this line. With a space separated list of breakpoint numbers, clear those breakpoints. Without argument, clear all breaks (but first ask confirmation).

disable `bpnumber [bpnumber ...]`

Disable the breakpoints given as a space separated list of breakpoint numbers. Disabling a breakpoint means it cannot cause the program to stop execution, but unlike clearing a breakpoint, it remains in the list of breakpoints and can be (re-)enabled.

enable bpnumber [bpnumber ...]

Enable the breakpoints specified.

ignore bpnumber [count]

Set the ignore count for the given breakpoint number. If *count* is omitted, the ignore count is set to 0. A breakpoint becomes active when the ignore count is zero. When non-zero, the *count* is decremented each time the breakpoint is reached and the breakpoint is not disabled and any associated condition evaluates to true.

condition bpnumber [condition]

Set a new *condition* for the breakpoint, an expression which must evaluate to true before the breakpoint is honored. If *condition* is absent, any existing condition is removed; i.e., the breakpoint is made unconditional.

commands [bpnumber]

Specify a list of commands for breakpoint number *bpnumber*. The commands themselves appear on the following lines. Type a line containing just `end` to terminate the commands. An example:

```
(Pdb) commands 1
(com) p some_variable
(com) end
(Pdb)
```

To remove all commands from a breakpoint, type `commands` and follow it immediately with `end`; that is, give no commands.

With no *bpnumber* argument, `commands` refers to the last breakpoint set.

You can use breakpoint commands to start your program up again. Simply use the *continue* command, or *step*, or any other command that resumes execution.

Specifying any command resuming execution (currently *continue*, *step*, *next*, *return*, *jump*, *quit* and their abbreviations) terminates the command list (as if that command was immediately followed by end). This is because any time you resume execution (even with a simple next or step), you may encounter another breakpoint—which could have its own command list, leading to ambiguities about which list to execute.

If you use the `silent` command in the command list, the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If none of the other commands print anything, you see no sign that the breakpoint was reached.

s(tep)

Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).

n(ext)

Continue execution until the next line in the current function is reached or it returns. (The difference between *next* and *step* is that *step* stops inside a called function, while *next* executes called functions at (nearly) full speed, only stopping at the next line in the current function.)

unt(il) [lineno]

Without argument, continue execution until the line with a number greater than the current one is reached.

With *lineno*, continue execution until a line with a number greater or equal to *lineno* is reached. In both cases, also stop when the current frame returns.

Changed in version 3.2: Allow giving an explicit line number.

r(eturn)

Continue execution until the current function returns.

c(ont(inue))

Continue execution, only stop when a breakpoint is encountered.

j(ump) lineno

Set the next line that will be executed. Only available in the bottom-most frame. This lets you jump back and execute code again, or jump forward to skip code that you don't want to run.

It should be noted that not all jumps are allowed – for instance it is not possible to jump into the middle of a `for` loop or out of a `finally` clause.

l(ist) [first[, last]]

List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing. With `.` as argument, list 11 lines around the current line. With one argument, list 11 lines around at that line. With two arguments, list the given range; if the second argument is less than the first, it is interpreted as a count.

The current line in the current frame is indicated by `->`. If an exception is being debugged, the line where the exception was originally raised or propagated is indicated by `>>`, if it differs from the current line.

Changed in version 3.2: Added the `>>` marker.

ll | longlist

List all source code for the current function or frame. Interesting lines are marked as for *list*.

Added in version 3.2.

a(rgs)

Print the arguments of the current function and their current values.

p expression

Evaluate *expression* in the current context and print its value.

> **Note**
>
> `print()` can also be used, but is not a debugger command — this executes the Python *print()* function.

pp expression

Like the *p* command, except the value of *expression* is pretty-printed using the *pprint* module.

whatis expression

Print the type of *expression*.

source expression

Try to get source code of *expression* and display it.

Added in version 3.2.

display [expression]

Display the value of *expression* if it changed, each time execution stops in the current frame.

Without *expression*, list all display expressions for the current frame.

> **Note**
>
> Display evaluates *expression* and compares to the result of the previous evaluation of *expression*, so when the result is mutable, display may not be able to pick up the changes.

Example:

```
lst = []
breakpoint()
pass
lst.append(1)
print(lst)
```

Display won't realize `lst` has been changed because the result of evaluation is modified in place by `lst.append(1)` before being compared:

```
> example.py(3)<module>()
-> pass
(Pdb) display lst
display lst: []
(Pdb) n
> example.py(4)<module>()
-> lst.append(1)
(Pdb) n
> example.py(5)<module>()
-> print(lst)
(Pdb)
```

You can do some tricks with copy mechanism to make it work:

```
> example.py(3)<module>()
-> pass
(Pdb) display lst[:]
display lst[:]: []
(Pdb) n
> example.py(4)<module>()
-> lst.append(1)
(Pdb) n
> example.py(5)<module>()
-> print(lst)
display lst[:]: [1]  [old: []]
(Pdb)
```

Added in version 3.2.

undisplay [expression]

Do not display *expression* anymore in the current frame. Without *expression*, clear all display expressions for the current frame.

Added in version 3.2.

interact

Start an interactive interpreter (using the *code* module) in a new global namespace initialised from the local and global namespaces for the current scope. Use `exit()` or `quit()` to exit the interpreter and return to the debugger.

> **Note**
>
> As `interact` creates a new dedicated namespace for code execution, assignments to variables will not affect the original namespaces. However, modifications to any referenced mutable objects will be reflected in the original namespaces as usual.

Added in version 3.2.

Changed in version 3.13: `exit()` and `quit()` can be used to exit the *interact* command.

Changed in version 3.13: *interact* directs its output to the debugger's output channel rather than *sys.stderr*.

alias [name [command]]

Create an alias called *name* that executes *command*. The *command* must *not* be enclosed in quotes. Replaceable parameters can be indicated by `%1`, `%2`, … and `%9`, while `%*` is replaced by all the parameters. If *command* is omitted, the current alias for *name* is shown. If no arguments are given, all aliases are listed.

Aliases may be nested and can contain anything that can be legally typed at the pdb prompt. Note that internal pdb commands *can* be overridden by aliases. Such a command is then hidden until the alias is removed. Aliasing is recursively applied to the first word of the command line; all other words in the line are left alone.

As an example, here are two useful aliases (especially when placed in the `.pdbrc` file):

```
# Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print(f"%1.{k} = {%1.__dict__[k]}")
# Print instance variables in self
alias ps pi self
```

unalias name

Delete the specified alias *name*.

! statement

Execute the (one-line) *statement* in the context of the current stack frame. The exclamation point can be omitted unless the first word of the statement resembles a debugger command, e.g.:

```
(Pdb) ! n=42
(Pdb)
```

To set a global variable, you can prefix the assignment command with a `global` statement on the same line, e.g.:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

run [args ...]

restart [args ...]

Restart the debugged Python program. If *args* is supplied, it is split with *shlex* and the result is used as the new *sys.argv*. History, breakpoints, actions and debugger options are preserved. *restart* is an alias for *run*.

q(uit)

Quit from the debugger. The program being executed is aborted.

debug code

Enter a recursive debugger that steps through *code* (which is an arbitrary expression or statement to be executed in the current environment).

retval

Print the return value for the last return of the current function.

exceptions [excnumber]

List or jump between chained exceptions.

When using `pdb.pm()` or `Pdb.post_mortem(...)` with a chained exception instead of a traceback, it allows the user to move between the chained exceptions using `exceptions` command to list exceptions, and `exception <number>` to switch to that exception.

Example:

```
def out():
    try:
        middle()
    except Exception as e:
        raise ValueError("reraise middle() error") from e

def middle():
    try:
        return inner(0)
    except Exception as e:
        raise ValueError("Middle fail")

def inner(x):
    1 / x

 out()
```

calling `pdb.pm()` will allow to move between exceptions:

```
> example.py(5)out()
-> raise ValueError("reraise middle() error") from e

(Pdb) exceptions
  0 ZeroDivisionError('division by zero')
  1 ValueError('Middle fail')
> 2 ValueError('reraise middle() error')

(Pdb) exceptions 0
> example.py(16)inner()
-> 1 / x

(Pdb) up
> example.py(10)middle()
-> return inner(0)
```

Added in version 3.13.

27.5 The Python Profilers

Source code: Lib/profile.py and Lib/pstats.py

27.5.1 Introduction to the profilers

cProfile and *profile* provide *deterministic profiling* of Python programs. A *profile* is a set of statistics that describes how often and for how long various parts of the program executed. These statistics can be formatted into reports via the *pstats* module.

The Python standard library provides two different implementations of the same profiling interface:

1. *cProfile* is recommended for most users; it's a C extension with reasonable overhead that makes it suitable for profiling long-running programs. Based on `lsprof`, contributed by Brett Rosen and Ted Czotter.

2. *profile*, a pure Python module whose interface is imitated by *cProfile*, but which adds significant overhead to profiled programs. If you're trying to extend the profiler in some way, the task might be easier with this module. Originally designed and written by Jim Roskind.

> **Note**
>
> The profiler modules are designed to provide an execution profile for a given program, not for benchmarking purposes (for that, there is *timeit* for reasonably accurate results). This particularly applies to benchmarking Python code against C code: the profilers introduce overhead for Python code, but not for C-level functions, and so the C code would seem faster than any Python one.

27.5.2 Instant User's Manual

This section is provided for users that "don't want to read the manual." It provides a very brief overview, and allows a user to rapidly perform profiling on an existing application.

To profile a function that takes a single argument, you can do:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

(Use *profile* instead of *cProfile* if the latter is not available on your system.)

The above action would run *re.compile()* and print profile results like the following:

```
      214 function calls (207 primitive calls) in 0.002 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.002    0.002 {built-in method builtins.exec}
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 __init__.py:250(compile)
     1    0.000    0.000    0.001    0.001 __init__.py:289(_compile)
     1    0.000    0.000    0.000    0.000 _compiler.py:759(compile)
     1    0.000    0.000    0.000    0.000 _parser.py:937(parse)
     1    0.000    0.000    0.000    0.000 _compiler.py:598(_code)
     1    0.000    0.000    0.000    0.000 _parser.py:435(_parse_sub)
```

The first line indicates that 214 calls were monitored. Of those calls, 207 were *primitive*, meaning that the call was not induced via recursion. The next line: `Ordered by: cumulative time` indicates the output is sorted by the `cumtime` values. The column headings include:

ncalls
: for the number of calls.

tottime
: for the total time spent in the given function (and excluding time made in calls to sub-functions)

percall
: is the quotient of `tottime` divided by `ncalls`

cumtime
: is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate *even* for recursive functions.

percall
: is the quotient of `cumtime` divided by primitive calls

filename:lineno(function)
: provides the respective data of each function

When there are two numbers in the first column (for example `3/1`), it means that the function recursed. The second value is the number of primitive calls and the former is the total number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

Instead of printing the output at the end of the profile run, you can save the results to a file by specifying a filename to the `run()` function:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

The *pstats.Stats* class reads profile results from a file and formats them in various ways.

The files *cProfile* and *profile* can also be invoked as a script to profile another script. For example:

```
python -m cProfile [-o output_file] [-s sort_order] (-m module | myscript.py)
```

`-o` writes the profile results to a file instead of to stdout

`-s` specifies one of the *sort_stats()* sort values to sort the output by. This only applies when `-o` is not supplied.

`-m` specifies that a module is being profiled instead of a script.

Added in version 3.7: Added the `-m` option to *cProfile*.

Added in version 3.8: Added the `-m` option to *profile*.

The *pstats* module's *Stats* class has a variety of methods for manipulating and printing the data saved into a profile results file:

```
import pstats
from pstats import SortKey
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

The *strip_dirs()* method removed the extraneous path from all the module names. The *sort_stats()* method sorted all the entries according to the standard module/line/name string that is printed. The *print_stats()* method printed out all the statistics. You might try the following sort calls:

```
p.sort_stats(SortKey.NAME)
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

```
p.sort_stats(SortKey.CUMULATIVE).print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

```
p.sort_stats(SortKey.TIME).print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

```
p.sort_stats(SortKey.FILENAME).print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods (since they are spelled with `__init__` in them). As one final example, you could try:

```
p.sort_stats(SortKey.TIME, SortKey.CUMULATIVE).print_stats(.5, 'init')
```

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: `.5`) of its original size, then only lines containing `init` are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (`p` is still sorted according to the last criteria) do:

```
p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If you want more functionality, you're going to have to read the manual, or guess what the following functions do:

```
p.print_callees()
p.add('restats')
```

Invoked as a script, the *pstats* module is a statistics browser for reading and examining profile dumps. It has a simple line-oriented interface (implemented using *cmd*) and interactive help.

27.5.3 `profile` and `cProfile` Module Reference

Both the *profile* and *cProfile* modules provide the following functions:

`profile.`**`run`**(*command*, *filename=None*, *sort=-1*)

This function takes a single argument that can be passed to the *exec()* function, and an optional file name. In all cases this routine executes:

```
exec(command, __main__.__dict__, __main__.__dict__)
```

and gathers profiling statistics from the execution. If no file name is present, then this function automatically creates a *Stats* instance and prints a simple profiling report. If the sort value is specified, it is passed to this *Stats* instance to control how the results are sorted.

`profile.`**`runctx`**(*command*, *globals*, *locals*, *filename=None*, *sort=-1*)

This function is similar to *run()*, with added arguments to supply the globals and locals mappings for the *command* string. This routine executes:

```
exec(command, globals, locals)
```

and gathers profiling statistics as in the *run()* function above.

`class` `profile.`**`Profile`**(*timer=None*, *timeunit=0.0*, *subcalls=True*, *builtins=True*)

This class is normally only used if more precise control over profiling is needed than what the `cProfile.run()` function provides.

A custom timer can be supplied for measuring how long code takes to run via the *timer* argument. This must be a function that returns a single number representing the current time. If the number is an integer, the *timeunit* specifies

a multiplier that specifies the duration of each unit of time. For example, if the timer returns times measured in thousands of seconds, the time unit would be `.001`.

Directly using the *Profile* class allows formatting profile results without writing the profile data to a file:

```
import cProfile, pstats, io
from pstats import SortKey
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
s = io.StringIO()
sortby = SortKey.CUMULATIVE
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print(s.getvalue())
```

The *Profile* class can also be used as a context manager (supported only in *cProfile* module. see *Context Manager Types*):

```
import cProfile

with cProfile.Profile() as pr:
    # ... do something ...

    pr.print_stats()
```

Changed in version 3.8: Added context manager support.

enable()

Start collecting profiling data. Only in *cProfile*.

disable()

Stop collecting profiling data. Only in *cProfile*.

create_stats()

Stop collecting profiling data and record the results internally as the current profile.

print_stats(*sort=-1*)

Create a *Stats* object based on the current profile and print the results to stdout.

The *sort* parameter specifies the sorting order of the displayed statistics. It accepts a single key or a tuple of keys to enable multi-level sorting, as in *Stats.sort_stats*.

Added in version 3.13: *print_stats()* now accepts a tuple of keys.

dump_stats(*filename*)

Write the results of the current profile to *filename*.

run(*cmd*)

Profile the cmd via *exec()*.

runctx(*cmd*, *globals*, *locals*)

Profile the cmd via *exec()* with the specified global and local environment.

runcall(*func*, */*, **args*, ***kwargs*)

Profile `func(*args, **kwargs)`

Note that profiling will only work if the called command/function actually returns. If the interpreter is terminated (e.g. via a *sys.exit()* call during the called command/function execution) no profiling results will be printed.

27.5.4 The `Stats` Class

Analysis of the profiler data is done using the *Stats* class.

class pstats.**Stats**(**filenames or profile*, *stream=sys.stdout*)

This class constructor creates an instance of a "statistics object" from a *filename* (or list of filenames) or from a `Profile` instance. Output will be printed to the stream specified by *stream*.

The file selected by the above constructor must have been created by the corresponding version of *profile* or *cProfile*. To be specific, there is *no* file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers, or the same profiler run on a different operating system. If several files are provided, all the statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing *Stats* object, the *add()* method can be used.

Instead of reading the profile data from a file, a `cProfile.Profile` or *profile.Profile* object can be used as the profile data source.

Stats objects have the following methods:

strip_dirs()

This method for the *Stats* class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the stripped information is lost. After performing a strip operation, the object is considered to have its entries in a "random" order, as it was just after object initialization and loading. If *strip_dirs()* causes two function names to be indistinguishable (they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

add(**filenames*)

This method of the *Stats* class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of *profile.run()* or `cProfile.run()`. Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

dump_stats(*filename*)

Save the data loaded into the *Stats* object to a file named *filename*. The file is created if it does not exist, and is overwritten if it already exists. This is equivalent to the method of the same name on the *profile.Profile* and `cProfile.Profile` classes.

sort_stats(**keys*)

This method modifies the *Stats* object by sorting it according to the supplied criteria. The argument can be either a string or a SortKey enum identifying the basis of a sort (example: `'time'`, `'name'`, `SortKey.TIME` or `SortKey.NAME`). The SortKey enums argument have advantage over the string argument in that it is more robust and less error prone.

When more than one key is provided, then additional keys are used as secondary criteria when there is equality in all keys selected before them. For example, `sort_stats(SortKey.NAME, SortKey.FILE)` will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

For the string argument, abbreviations can be used for any key names, as long as the abbreviation is unambiguous.

The following are the valid string and SortKey:

Valid String Arg	Valid enum Arg	Meaning
`'calls'`	SortKey.CALLS	call count
`'cumulative'`	SortKey.CUMULATIVE	cumulative time
`'cumtime'`	N/A	cumulative time
`'file'`	N/A	file name
`'filename'`	SortKey.FILENAME	file name
`'module'`	N/A	file name
`'ncalls'`	N/A	call count
`'pcalls'`	SortKey.PCALLS	primitive call count
`'line'`	SortKey.LINE	line number
`'name'`	SortKey.NAME	function name
`'nfl'`	SortKey.NFL	name/file/line
`'stdname'`	SortKey.STDNAME	standard name
`'time'`	SortKey.TIME	internal time
`'tottime'`	N/A	internal time

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (alphabetical). The subtle distinction between `SortKey.NFL` and `SortKey.STDNAME` is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, `SortKey.NFL` does a numeric compare of the line numbers. In fact, `sort_stats(SortKey.NFL)` is the same as `sort_stats(SortKey.NAME, SortKey.FILENAME, SortKey.LINE)`.

For backward-compatibility reasons, the numeric arguments `-1`, `0`, `1`, and `2` are permitted. They are interpreted as `'stdname'`, `'calls'`, `'time'`, and `'cumulative'` respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

Added in version 3.7: Added the SortKey enum.

reverse_order()

This method for the *Stats* class reverses the ordering of the basic list within the object. Note that by default ascending vs descending order is properly selected based on the sort key of choice.

print_stats(**restrictions*)

This method for the *Stats* class prints out a report as described in the *profile.run()* definition.

The order of the printing is based on the last *sort_stats()* operation done on the object (subject to caveats in *add()* and *strip_dirs()*).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a string that will interpreted as a regular expression (to pattern match the standard name that is printed). If several restrictions are provided, then they are applied sequentially. For example:

```
print_stats(.1, 'foo:')
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename `.*foo:`. In contrast, the command:

```
print_stats('foo:', .1)
```

would limit the list to all functions having file names `.*foo:`, and then proceed to only print the first 10% of them.

print_callers(**restrictions*)

This method for the *Stats* class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by *print_stats()*, and the definition of the restricting argument is also identical. Each caller is reported on its own line. The format differs slightly depending on the profiler that produced the stats:

- With *profile*, a number is shown in parentheses after each caller to show how many times this specific call was made. For convenience, a second non-parenthesized number repeats the cumulative time spent in the function at the right.
- With *cProfile*, each caller is preceded by three numbers: the number of times this specific call was made, and the total and cumulative times spent in the current function while it was invoked by this specific caller.

print_callees(**restrictions*)

This method for the *Stats* class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to the *print_callers()* method.

get_stats_profile()

This method returns an instance of StatsProfile, which contains a mapping of function names to instances of FunctionProfile. Each FunctionProfile instance holds information related to the function's profile such as how long the function took to run, how many times it was called, etc...

Added in version 3.9: Added the following dataclasses: StatsProfile, FunctionProfile. Added the following function: get_stats_profile.

27.5.5 What Is Deterministic Profiling?

Deterministic profiling is meant to reflect the fact that all *function call*, *function return*, and *exception* events are monitored, and precise timings are made for the intervals between these events (during which time the user's code is executing). In contrast, *statistical profiling* (which is not done by this module) randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required in order to do deterministic profiling. Python automatically provides a *hook* (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run time statistics about the execution of a Python program.

Call count statistics can be used to identify bugs in code (surprising counts), and to identify possible inline-expansion points (high call counts). Internal time statistics can be used to identify "hot loops" that should be carefully optimized. Cumulative time statistics should be used to identify high level errors in the selection of algorithms. Note that the unusual handling of cumulative times in this profiler allows statistics for recursive implementations of algorithms to be directly compared to iterative implementations.

27.5.6 Limitations

One limitation has to do with accuracy of timing information. There is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying "clock" is only ticking at a rate (typically) of about .001 seconds. Hence no measurements will be more accurate than the underlying clock. If enough measurements are taken, then the "error" will tend to average out. Unfortunately, removing this first error induces a second source of error.

The second problem is that it "takes a while" from when an event is dispatched until the profiler's call to get the time actually *gets* the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the time that the clock's value was obtained (and then squirreled away), until the user's code is once again executing. As a result, functions that are called many times, or call many functions, will typically accumulate this error. The error that accumulates in this fashion is typically less than the accuracy of the clock (less than one clock tick), but it *can* accumulate and become very significant.

The problem is more important with `profile` than with the lower-overhead `cProfile`. For this reason, `profile` provides a means of calibrating itself for a given platform so that this error can be probabilistically (on the average) removed. After the profiler is calibrated, it will be more accurate (in a least square sense), but it will sometimes produce negative numbers (when call counts are exceptionally low, and the gods of probability work against you :-).) Do *not* be alarmed by negative numbers in the profile. They should *only* appear if you have calibrated your profiler, and the results are actually better than without calibration.

27.5.7 Calibration

The profiler of the `profile` module subtracts a constant from each event handling time to compensate for the overhead of calling the time function, and socking away the results. By default, the constant is 0. The following procedure can be used to obtain a better constant for a given platform (see *Limitations*).

```
import profile
pr = profile.Profile()
for i in range(5):
    print(pr.calibrate(10000))
```

The method executes the number of Python calls given by the argument, directly and again under the profiler, measuring the time for both. It then computes the hidden overhead per profiler event, and returns that as a float. For example, on a 1.8Ghz Intel Core i5 running macOS, and using Python's time.process_time() as the timer, the magical number is about 4.04e-6.

The object of this exercise is to get a fairly consistent result. If your computer is *very* fast, or your timer function has poor resolution, you might have to pass 100000, or even 1000000, to get consistent results.

When you have a consistent answer, there are three ways you can use it:

```
import profile

# 1. Apply computed bias to all Profile instances created hereafter.
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

If you have a choice, you are better off choosing a smaller constant, and then your results will "less often" show up as negative in profile statistics.

27.5.8 Using a custom timer

If you want to change how current time is determined (for example, to force use of wall-clock time or elapsed process time), pass the timing function you want to the `Profile` class constructor:

```
pr = profile.Profile(your_time_func)
```

The resulting profiler will then call `your_time_func`. Depending on whether you are using *profile.Profile* or `cProfile.Profile`, `your_time_func`'s return value will be interpreted differently:

profile.Profile
: `your_time_func` should return a single number, or a list of numbers whose sum is the current time (like what *os.times()* returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

  Be warned that you should calibrate the profiler class for the timer function that you choose (see *Calibration*). For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (*os.times()* is *pretty* bad, as it returns a tuple of floating-point values). If you want to substitute a better timer in the cleanest fashion, derive a class and hardwire a replacement dispatch method that best handles your timer call, along with the appropriate calibration constant.

cProfile.Profile
: `your_time_func` should return a single number. If it returns integers, you can also invoke the class constructor with a second argument specifying the real duration of one unit of time. For example, if `your_integer_time_func` returns times measured in thousands of seconds, you would construct the `Profile` instance as follows:

  ```
  pr = cProfile.Profile(your_integer_time_func, 0.001)
  ```

  As the `cProfile.Profile` class cannot be calibrated, custom timer functions should be used with care and should be as fast as possible. For the best results with a custom timer, it might be necessary to hard-code it in the C source of the internal `_lsprof` module.

Python 3.3 adds several new functions in *time* that can be used to make precise measurements of process or wall-clock time. For example, see *time.perf_counter()*.

27.6 `timeit` — Measure execution time of small code snippets

Source code: Lib/timeit.py

This module provides a simple way to time small bits of Python code. It has both a *Command-Line Interface* as well as a *callable* one. It avoids a number of common traps for measuring execution times. See also Tim Peters' introduction to the "Algorithms" chapter in the second edition of *Python Cookbook*, published by O'Reilly.

27.6.1 Basic Examples

The following example shows how the *Command-Line Interface* can be used to compare three different expressions:

```
$ python -m timeit "'-'.join(str(n) for n in range(100))"
10000 loops, best of 5: 30.2 usec per loop
$ python -m timeit "'-'.join([str(n) for n in range(100)])"
10000 loops, best of 5: 27.5 usec per loop
$ python -m timeit "'-'.join(map(str, range(100)))"
10000 loops, best of 5: 23.2 usec per loop
```

This can be achieved from the *Python Interface* with:

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.3018611848820001
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.2727368790656328
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.23702679807320237
```

A callable can also be passed from the *Python Interface*:

```
>>> timeit.timeit(lambda: "-".join(map(str, range(100))), number=10000)
0.19665591977536678
```

Note however that `timeit()` will automatically determine the number of repetitions only when the command-line interface is used. In the *Examples* section you can find more advanced examples.

27.6.2 Python Interface

The module defines three convenience functions and a public class:

timeit.**timeit**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *number=1000000*, *globals=None*)

Create a `Timer` instance with the given statement, *setup* code and *timer* function and run its `timeit()` method with *number* executions. The optional *globals* argument specifies a namespace in which to execute the code.

Changed in version 3.5: The optional *globals* parameter was added.

timeit.**repeat**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *repeat=5*, *number=1000000*, *globals=None*)

Create a `Timer` instance with the given statement, *setup* code and *timer* function and run its `repeat()` method with the given *repeat* count and *number* executions. The optional *globals* argument specifies a namespace in which to execute the code.

Changed in version 3.5: The optional *globals* parameter was added.

Changed in version 3.7: Default value of *repeat* changed from 3 to 5.

timeit.**default_timer**()

The default timer, which is always time.perf_counter(), returns float seconds. An alternative, time.perf_counter_ns, returns integer nanoseconds.

Changed in version 3.3: `time.perf_counter()` is now the default timer.

class timeit.**Timer**(*stmt='pass'*, *setup='pass'*, *timer=<timer function>*, *globals=None*)

Class for timing execution speed of small code snippets.

The constructor takes a statement to be timed, an additional statement used for setup, and a timer function. Both statements default to `'pass'`; the timer function is platform-dependent (see the module doc string). *stmt* and

setup may also contain multiple statements separated by `;` or newlines, as long as they don't contain multi-line string literals. The statement will by default be executed within timeit's namespace; this behavior can be controlled by passing a namespace to *globals*.

To measure the execution time of the first statement, use the `timeit()` method. The `repeat()` and `autorange()` methods are convenience methods to call `timeit()` multiple times.

The execution time of *setup* is excluded from the overall timed execution run.

The *stmt* and *setup* parameters can also take objects that are callable without arguments. This will embed calls to them in a timer function that will then be executed by `timeit()`. Note that the timing overhead is a little larger in this case because of the extra function calls.

Changed in version 3.5: The optional *globals* parameter was added.

timeit(*number=1000000*)

Time *number* executions of the main statement. This executes the setup statement once, and then returns the time it takes to execute the main statement a number of times. The default timer returns seconds as a float. The argument is the number of times through the loop, defaulting to one million. The main statement, the setup statement and the timer function to be used are passed to the constructor.

> **Note**
>
> By default, `timeit()` temporarily turns off *garbage collection* during the timing. The advantage of this approach is that it makes independent timings more comparable. The disadvantage is that GC may be an important component of the performance of the function being measured. If so, GC can be re-enabled as the first statement in the *setup* string. For example:
>
> ```
> timeit.Timer('for i in range(10): oct(i)', 'gc.enable()').timeit()
> ```

autorange(*callback=None*)

Automatically determine how many times to call `timeit()`.

This is a convenience function that calls `timeit()` repeatedly so that the total time >= 0.2 second, returning the eventual (number of loops, time taken for that number of loops). It calls `timeit()` with increasing numbers from the sequence 1, 2, 5, 10, 20, 50, … until the time taken is at least 0.2 seconds.

If *callback* is given and is not `None`, it will be called after each trial with two arguments: `callback(number, time_taken)`.

Added in version 3.6.

repeat(*repeat=5*, *number=1000000*)

Call `timeit()` a few times.

This is a convenience function that calls the `timeit()` repeatedly, returning a list of results. The first argument specifies how many times to call `timeit()`. The second argument specifies the *number* argument for `timeit()`.

> **Note**
>
> It's tempting to calculate mean and standard deviation from the result vector and report these. However, this is not very useful. In a typical case, the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values in the result vector are typically not caused by variability in Python's speed, but by other processes interfering with your timing accuracy. So the `min()` of the result

is probably the only number you should be interested in. After that, you should look at the entire vector and apply common sense rather than statistics.

Changed in version 3.7: Default value of *repeat* changed from 3 to 5.

print_exc(*file=None*)

Helper to print a traceback from the timed code.

Typical use:

```
t = Timer(...)       # outside the try/except
try:
    t.timeit(...)    # or t.repeat(...)
except Exception:
    t.print_exc()
```

The advantage over the standard traceback is that source lines in the compiled template will be displayed. The optional *file* argument directs where the traceback is sent; it defaults to `sys.stderr`.

27.6.3 Command-Line Interface

When called as a program from the command line, the following form is used:

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-p] [-v] [-h] [statement ...]
```

Where the following options are understood:

-n N, **--number**=N

how many times to execute 'statement'

-r N, **--repeat**=N

how many times to repeat the timer (default 5)

-s S, **--setup**=S

statement to be executed once initially (default `pass`)

-p, **--process**

measure process time, not wallclock time, using `time.process_time()` instead of `time.perf_counter()`, which is the default

Added in version 3.3.

-u, **--unit**=U

specify a time unit for timer output; can select `nsec`, `usec`, `msec`, or `sec`

Added in version 3.5.

-v, **--verbose**

print raw timing results; repeat for more digits precision

-h, **--help**

print a short usage message and exit

A multi-line statement may be given by specifying each line as a separate statement argument; indented lines are possible by enclosing an argument in quotes and using leading spaces. Multiple `-s` options are treated similarly.

If `-n` is not given, a suitable number of loops is calculated by trying increasing numbers from the sequence 1, 2, 5, 10, 20, 50, … until the total time is at least 0.2 seconds.

default_timer() measurements can be affected by other programs running on the same machine, so the best thing to do when accurate timing is necessary is to repeat the timing a few times and use the best time. The *-r* option is good for this; the default of 5 repetitions is probably enough in most cases. You can use *time.process_time()* to measure CPU time.

Note

There is a certain baseline overhead associated with executing a pass statement. The code here doesn't try to hide it, but you should be aware of it. The baseline overhead can be measured by invoking the program without arguments, and it might differ between Python versions.

27.6.4 Examples

It is possible to provide a setup statement that is executed only once at the beginning:

```
$ python -m timeit -s "text = 'sample string'; char = 'g'" "char in text"
5000000 loops, best of 5: 0.0877 usec per loop
$ python -m timeit -s "text = 'sample string'; char = 'g'" "text.find(char)"
1000000 loops, best of 5: 0.342 usec per loop
```

In the output, there are three fields. The loop count, which tells you how many times the statement body was run per timing loop repetition. The repetition count ('best of 5') which tells you how many times the timing loop was repeated, and finally the time the statement body took on average within the best repetition of the timing loop. That is, the time the fastest repetition took divided by the loop count.

```
>>> import timeit
>>> timeit.timeit('char in text', setup='text = "sample string"; char = "g"')
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample string"; char = "g"')
1.7246671520006203
```

The same can be done using the *Timer* class and its methods:

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.3955516149999312
>>> t.repeat()
[0.40183617287970225, 0.37027556854118704, 0.38344867356679524, 0.3712595970846668, 0.
↪37866875250654886]
```

The following examples show how to time expressions that contain multiple lines. Here we compare the cost of using *hasattr()* vs. `try`/`except` to test for missing and present object attributes:

```
$ python -m timeit "try:" "  str.__bool__" "except AttributeError:" "  pass"
20000 loops, best of 5: 15.7 usec per loop
$ python -m timeit "if hasattr(str, '__bool__'): pass"
50000 loops, best of 5: 4.26 usec per loop

$ python -m timeit "try:" "  int.__bool__" "except AttributeError:" "  pass"
200000 loops, best of 5: 1.43 usec per loop
$ python -m timeit "if hasattr(int, '__bool__'): pass"
100000 loops, best of 5: 2.23 usec per loop
```

```
>>> import timeit
>>> # attribute is missing
>>> s = """\
... try:
...     str.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.5829014980008651
>>>
>>> # attribute is present
>>> s = """\
... try:
...     int.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

To give the *timeit* module access to functions you define, you can pass a *setup* parameter which contains an import statement:

```
def test():
    """Stupid test function"""
    L = [i for i in range(100)]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

Another option is to pass *globals()* to the *globals* parameter, which will cause the code to be executed within your current global namespace. This can be more convenient than individually specifying imports:

```
def f(x):
    return x**2
def g(x):
    return x**4
def h(x):
    return x**8

import timeit
print(timeit.timeit('[func(42) for func in (f,g,h)]', globals=globals()))
```

27.7 `trace` — Trace or track Python statement execution

Source code: Lib/trace.py

The *trace* module allows you to trace program execution, generate annotated statement coverage listings, print caller/callee relationships and list functions executed during a program run. It can be used in another program or from the command line.

See also

Coverage.py
A popular third-party coverage tool that provides HTML output along with advanced features such as branch coverage.

27.7.1 Command-Line Usage

The *trace* module can be invoked from the command line. It can be as simple as

```
python -m trace --count -C . somefile.py ...
```

The above will execute `somefile.py` and generate annotated listings of all Python modules imported during the execution into the current directory.

--help
Display usage and exit.

--version
Display the version of the module and exit.

Added in version 3.8: Added `--module` option that allows to run an executable module.

Main options

At least one of the following options must be specified when invoking *trace*. The *--listfuncs* option is mutually exclusive with the *--trace* and *--count* options. When *--listfuncs* is provided, neither *--count* nor *--trace* are accepted, and vice versa.

-c, --count
Produce a set of annotated listing files upon program completion that shows how many times each statement was executed. See also *--coverdir*, *--file* and *--no-report* below.

-t, --trace
Display lines as they are executed.

-l, --listfuncs
Display the functions executed by running the program.

-r, --report
Produce an annotated list from an earlier program run that used the *--count* and *--file* option. This does not execute any code.

-T, --trackcalls
Display the calling relationships exposed by running the program.

Modifiers

-f, --file=<file>

Name of a file to accumulate counts over several tracing runs. Should be used with the *--count* option.

-C, --coverdir=<dir>

Directory where the report files go. The coverage report for `package.module` is written to file *dir/package/module*`.cover`.

-m, --missing

When generating annotated listings, mark lines which were not executed with >>>>>>.

-s, --summary

When using *--count* or *--report*, write a brief summary to stdout for each file processed.

-R, --no-report

Do not generate annotated listings. This is useful if you intend to make several runs with *--count*, and then produce a single set of annotated listings at the end.

-g, --timing

Prefix each line with the time since the program started. Only used while tracing.

Filters

These options may be repeated multiple times.

--ignore-module=<mod>

Ignore each of the given module names and its submodules (if it is a package). The argument can be a list of names separated by a comma.

--ignore-dir=<dir>

Ignore all modules and packages in the named directory and subdirectories. The argument can be a list of directories separated by *os.pathsep*.

27.7.2 Programmatic Interface

class trace.**Trace**(*count=1*, *trace=1*, *countfuncs=0*, *countcallers=0*, *ignoremods=()*, *ignoredirs=()*, *infile=None*, *outfile=None*, *timing=False*)

Create an object to trace execution of a single statement or expression. All parameters are optional. *count* enables counting of line numbers. *trace* enables line execution tracing. *countfuncs* enables listing of the functions called during the run. *countcallers* enables call relationship tracking. *ignoremods* is a list of modules or packages to ignore. *ignoredirs* is a list of directories whose modules or packages should be ignored. *infile* is the name of the file from which to read stored count information. *outfile* is the name of the file in which to write updated count information. *timing* enables a timestamp relative to when tracing was started to be displayed.

run(*cmd*)

Execute the command and gather statistics from the execution with the current tracing parameters. *cmd* must be a string or code object, suitable for passing into *exec()*.

runctx(*cmd*, *globals=None*, *locals=None*)

Execute the command and gather statistics from the execution with the current tracing parameters, in the defined global and local environments. If not defined, *globals* and *locals* default to empty dictionaries.

runfunc(*func*, */*, **args*, ***kwds*)

Call *func* with the given arguments under control of the `Trace` object with the current tracing parameters.

results()

Return a `CoverageResults` object that contains the cumulative results of all previous calls to `run`, `runctx` and `runfunc` for the given `Trace` instance. Does not reset the accumulated trace results.

class trace.**CoverageResults**

A container for coverage results, created by `Trace.results()`. Should not be created directly by the user.

update(*other*)

Merge in data from another `CoverageResults` object.

write_results(*show_missing=True*, *summary=False*, *coverdir=None*, ***, *ignore_missing_files=False*)

Write coverage results. Set *show_missing* to show lines that had no hits. Set *summary* to include in the output the coverage summary per module. *coverdir* specifies the directory into which the coverage result files will be output. If `None`, the results for each source file are placed in its directory.

If *ignore_missing_files* is `True`, coverage counts for files that no longer exist are silently ignored. Otherwise, a missing file will raise a `FileNotFoundError`.

Changed in version 3.13: Added *ignore_missing_files* parameter.

A simple example demonstrating the use of the programmatic interface:

```
import sys
import trace

# create a Trace object, telling it what to ignore, and whether to
# do tracing or line-counting or both.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# run the new command using the given tracer
tracer.run('main()')

# make a report, placing output in the current directory
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```

27.8 `tracemalloc` — Trace memory allocations

Added in version 3.4.

Source code: Lib/tracemalloc.py

The tracemalloc module is a debug tool to trace memory blocks allocated by Python. It provides the following information:

- Traceback where an object was allocated
- Statistics on allocated memory blocks per filename and per line number: total size, number and average size of allocated memory blocks
- Compute the differences between two snapshots to detect memory leaks

To trace most memory blocks allocated by Python, the module should be started as early as possible by setting the `PYTHONTRACEMALLOC` environment variable to `1`, or by using `-X tracemalloc` command line option. The *`tracemalloc.start()`* function can be called at runtime to start tracing Python memory allocations.

By default, a trace of an allocated memory block only stores the most recent frame (1 frame). To store 25 frames at startup: set the `PYTHONTRACEMALLOC` environment variable to `25`, or use the `-X tracemalloc=25` command line option.

27.8.1 Examples

Display the top 10

Display the 10 files allocating the most memory:

```
import tracemalloc

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output of the Python test suite:

```
[ Top 10 ]
<frozen importlib._bootstrap>:716: size=4855 KiB, count=39328, average=126 B
<frozen importlib._bootstrap>:284: size=521 KiB, count=3199, average=167 B
/usr/lib/python3.4/collections/__init__.py:368: size=244 KiB, count=2315, average=108␣
↪B
/usr/lib/python3.4/unittest/case.py:381: size=185 KiB, count=779, average=243 B
/usr/lib/python3.4/unittest/case.py:402: size=154 KiB, count=378, average=416 B
/usr/lib/python3.4/abc.py:133: size=88.7 KiB, count=347, average=262 B
<frozen importlib._bootstrap>:1446: size=70.4 KiB, count=911, average=79 B
<frozen importlib._bootstrap>:1454: size=52.0 KiB, count=25, average=2131 B
<string>:5: size=49.7 KiB, count=148, average=344 B
/usr/lib/python3.4/sysconfig.py:411: size=48.0 KiB, count=1, average=48.0 KiB
```

We can see that Python loaded `4855 KiB` data (bytecode and constants) from modules and that the *`collections`* module allocated `244 KiB` to build *`namedtuple`* types.

See *`Snapshot.statistics()`* for more options.

Compute differences

Take two snapshots and display the differences:

```
import tracemalloc
tracemalloc.start()
# ... start your application ...

snapshot1 = tracemalloc.take_snapshot()
# ... call the function leaking memory ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output before/after running some tests of the Python test suite:

```
[ Top 10 differences ]
<frozen importlib._bootstrap>:716: size=8173 KiB (+4428 KiB), count=71332 (+39369),␣
↪average=117 B
/usr/lib/python3.4/linecache.py:127: size=940 KiB (+940 KiB), count=8106 (+8106),␣
↪average=119 B
/usr/lib/python3.4/unittest/case.py:571: size=298 KiB (+298 KiB), count=589 (+589),␣
↪average=519 B
<frozen importlib._bootstrap>:284: size=1005 KiB (+166 KiB), count=7423 (+1526),␣
↪average=139 B
/usr/lib/python3.4/mimetypes.py:217: size=112 KiB (+112 KiB), count=1334 (+1334),␣
↪average=86 B
/usr/lib/python3.4/http/server.py:848: size=96.0 KiB (+96.0 KiB), count=1 (+1),␣
↪average=96.0 KiB
/usr/lib/python3.4/inspect.py:1465: size=83.5 KiB (+83.5 KiB), count=109 (+109),␣
↪average=784 B
/usr/lib/python3.4/unittest/mock.py:491: size=77.7 KiB (+77.7 KiB), count=143 (+143),␣
↪average=557 B
/usr/lib/python3.4/urllib/parse.py:476: size=71.8 KiB (+71.8 KiB), count=969 (+969),␣
↪average=76 B
/usr/lib/python3.4/contextlib.py:38: size=67.2 KiB (+67.2 KiB), count=126 (+126),␣
↪average=546 B
```

We can see that Python has loaded `8173 KiB` of module data (bytecode and constants), and that this is `4428 KiB` more than had been loaded before the tests, when the previous snapshot was taken. Similarly, the *linecache* module has cached `940 KiB` of Python source code to format tracebacks, all of it since the previous snapshot.

If the system has little free memory, snapshots can be written on disk using the *Snapshot.dump()* method to analyze the snapshot offline. Then use the *Snapshot.load()* method reload the snapshot.

Get the traceback of a memory block

Code to display the traceback of the biggest memory block:

```
import tracemalloc

# Store 25 frames
tracemalloc.start(25)

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')

# pick the biggest memory block
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))
for line in stat.traceback.format():
    print(line)
```

Example of output of the Python test suite (traceback limited to 25 frames):

```
903 memory blocks: 870.1 KiB
  File "<frozen importlib._bootstrap>", line 716
  File "<frozen importlib._bootstrap>", line 1036
  File "<frozen importlib._bootstrap>", line 934
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/doctest.py", line 101
    import pdb
  File "<frozen importlib._bootstrap>", line 284
  File "<frozen importlib._bootstrap>", line 938
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/test/support/__init__.py", line 1728
    import doctest
  File "/usr/lib/python3.4/test/test_pickletools.py", line 21
    support.run_doctest(pickletools)
  File "/usr/lib/python3.4/test/regrtest.py", line 1276
    test_runner()
  File "/usr/lib/python3.4/test/regrtest.py", line 976
    display_failure=not verbose)
  File "/usr/lib/python3.4/test/regrtest.py", line 761
    match_tests=ns.match_tests)
  File "/usr/lib/python3.4/test/regrtest.py", line 1563
    main()
  File "/usr/lib/python3.4/test/__main__.py", line 3
    regrtest.main_in_temp_cwd()
  File "/usr/lib/python3.4/runpy.py", line 73
    exec(code, run_globals)
  File "/usr/lib/python3.4/runpy.py", line 160
    "__main__", fname, loader, pkg_name)
```

We can see that the most memory was allocated in the *importlib* module to load data (bytecode and constants) from

modules: 870.1 KiB. The traceback is where the *importlib* loaded data most recently: on the import pdb line of the *doctest* module. The traceback may change if a new module is loaded.

Pretty top

Code to display the 10 lines allocating the most memory with a pretty output, ignoring <frozen importlib._bootstrap> and <unknown> files:

```
import linecache
import os
import tracemalloc

def display_top(snapshot, key_type='lineno', limit=10):
    snapshot = snapshot.filter_traces((
        tracemalloc.Filter(False, "<frozen importlib._bootstrap>"),
        tracemalloc.Filter(False, "<unknown>"),
    ))
    top_stats = snapshot.statistics(key_type)

    print("Top %s lines" % limit)
    for index, stat in enumerate(top_stats[:limit], 1):
        frame = stat.traceback[0]
        print("#%s: %s:%s: %.1f KiB"
              % (index, frame.filename, frame.lineno, stat.size / 1024))
        line = linecache.getline(frame.filename, frame.lineno).strip()
        if line:
            print('    %s' % line)

    other = top_stats[limit:]
    if other:
        size = sum(stat.size for stat in other)
        print("%s other: %.1f KiB" % (len(other), size / 1024))
    total = sum(stat.size for stat in top_stats)
    print("Total allocated size: %.1f KiB" % (total / 1024))

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
display_top(snapshot)
```

Example of output of the Python test suite:

```
Top 10 lines
#1: Lib/base64.py:414: 419.8 KiB
    _b85chars2 = [(a + b) for a in _b85chars for b in _b85chars]
#2: Lib/base64.py:306: 419.8 KiB
    _a85chars2 = [(a + b) for a in _a85chars for b in _a85chars]
#3: collections/__init__.py:368: 293.6 KiB
    exec(class_definition, namespace)
#4: Lib/abc.py:133: 115.2 KiB
    cls = super().__new__(mcls, name, bases, namespace)
#5: unittest/case.py:574: 103.1 KiB
    testMethod()
#6: Lib/linecache.py:127: 95.4 KiB
    lines = fp.readlines()
```

(continues on next page)

(continued from previous page)

```
#7: urllib/parse.py:476: 71.8 KiB
    for a in _hexdig for b in _hexdig}
#8: <string>:5: 62.0 KiB
#9: Lib/_weakrefset.py:37: 60.0 KiB
    self.data = set()
#10: Lib/base64.py:142: 59.8 KiB
    _b32tab2 = [a + b for a in _b32tab for b in _b32tab]
6220 other: 3602.8 KiB
Total allocated size: 5303.1 KiB
```

See *Snapshot.statistics()* for more options.

Record the current and peak size of all traced memory blocks

The following code computes two sums like `0 + 1 + 2 + ...` inefficiently, by creating a list of those numbers. This list consumes a lot of memory temporarily. We can use *get_traced_memory()* and *reset_peak()* to observe the small memory usage after the sum is computed as well as the peak memory usage during the computations:

```
import tracemalloc

tracemalloc.start()

# Example code: compute a sum with a large temporary list
large_sum = sum(list(range(100000)))

first_size, first_peak = tracemalloc.get_traced_memory()

tracemalloc.reset_peak()

# Example code: compute a sum with a small temporary list
small_sum = sum(list(range(1000)))

second_size, second_peak = tracemalloc.get_traced_memory()

print(f"{first_size=}, {first_peak=}")
print(f"{second_size=}, {second_peak=}")
```

Output:

```
first_size=664, first_peak=3592984
second_size=804, second_peak=29704
```

Using *reset_peak()* ensured we could accurately record the peak during the computation of `small_sum`, even though it is much smaller than the overall peak size of memory blocks since the *start()* call. Without the call to *reset_peak()*, `second_peak` would still be the peak from the computation `large_sum` (that is, equal to `first_peak`). In this case, both peaks are much higher than the final memory usage, and which suggests we could optimise (by removing the unnecessary call to *list*, and writing `sum(range(...))`).

27.8.2 API

Functions

tracemalloc.**clear_traces**()

Clear traces of memory blocks allocated by Python.

See also *stop()*.

tracemalloc.**get_object_traceback**(*obj*)

Get the traceback where the Python object *obj* was allocated. Return a *Traceback* instance, or `None` if the *tracemalloc* module is not tracing memory allocations or did not trace the allocation of the object.

See also *gc.get_referrers()* and *sys.getsizeof()* functions.

tracemalloc.**get_traceback_limit**()

Get the maximum number of frames stored in the traceback of a trace.

The *tracemalloc* module must be tracing memory allocations to get the limit, otherwise an exception is raised.

The limit is set by the *start()* function.

tracemalloc.**get_traced_memory**()

Get the current size and peak size of memory blocks traced by the *tracemalloc* module as a tuple: `(current: int, peak: int)`.

tracemalloc.**reset_peak**()

Set the peak size of memory blocks traced by the *tracemalloc* module to the current size.

Do nothing if the *tracemalloc* module is not tracing memory allocations.

This function only modifies the recorded peak size, and does not modify or clear any traces, unlike *clear_traces()*. Snapshots taken with *take_snapshot()* before a call to *reset_peak()* can be meaningfully compared to snapshots taken after the call.

See also *get_traced_memory()*.

Added in version 3.9.

tracemalloc.**get_tracemalloc_memory**()

Get the memory usage in bytes of the *tracemalloc* module used to store traces of memory blocks. Return an *int*.

tracemalloc.**is_tracing**()

`True` if the *tracemalloc* module is tracing Python memory allocations, `False` otherwise.

See also *start()* and *stop()* functions.

tracemalloc.**start**(*nframe:* int = *1*)

Start tracing Python memory allocations: install hooks on Python memory allocators. Collected tracebacks of traces will be limited to *nframe* frames. By default, a trace of a memory block only stores the most recent frame: the limit is `1`. *nframe* must be greater or equal to `1`.

You can still read the original number of total frames that composed the traceback by looking at the *Traceback.total_nframe* attribute.

Storing more than `1` frame is only useful to compute statistics grouped by `'traceback'` or to compute cumulative statistics: see the *Snapshot.compare_to()* and *Snapshot.statistics()* methods.

Storing more frames increases the memory and CPU overhead of the *tracemalloc* module. Use the *get_tracemalloc_memory()* function to measure how much memory is used by the *tracemalloc* module.

The PYTHONTRACEMALLOC environment variable (PYTHONTRACEMALLOC=NFRAME) and the -X tracemalloc=NFRAME command line option can be used to start tracing at startup.

See also *stop()*, *is_tracing()* and *get_traceback_limit()* functions.

tracemalloc.**stop**()

Stop tracing Python memory allocations: uninstall hooks on Python memory allocators. Also clears all previously collected traces of memory blocks allocated by Python.

Call *take_snapshot()* function to take a snapshot of traces before clearing them.

See also *start()*, *is_tracing()* and *clear_traces()* functions.

tracemalloc.**take_snapshot**()

Take a snapshot of traces of memory blocks allocated by Python. Return a new *Snapshot* instance.

The snapshot does not include memory blocks allocated before the *tracemalloc* module started to trace memory allocations.

Tracebacks of traces are limited to *get_traceback_limit()* frames. Use the *nframe* parameter of the *start()* function to store more frames.

The *tracemalloc* module must be tracing memory allocations to take a snapshot, see the *start()* function.

See also the *get_object_traceback()* function.

DomainFilter

class tracemalloc.**DomainFilter**(*inclusive:* bool, *domain:* int)

Filter traces of memory blocks by their address space (domain).

Added in version 3.6.

inclusive

If *inclusive* is True (include), match memory blocks allocated in the address space *domain*.

If *inclusive* is False (exclude), match memory blocks not allocated in the address space *domain*.

domain

Address space of a memory block (int). Read-only property.

Filter

class tracemalloc.**Filter**(*inclusive:* bool, *filename_pattern:* str, *lineno:* int = *None*, *all_frames:* bool = *False*, *domain:* int = *None*)

Filter on traces of memory blocks.

See the *fnmatch.fnmatch()* function for the syntax of *filename_pattern*. The '.pyc' file extension is replaced with '.py'.

Examples:

- Filter(True, subprocess.__file__) only includes traces of the *subprocess* module
- Filter(False, tracemalloc.__file__) excludes traces of the *tracemalloc* module
- Filter(False, "<unknown>") excludes empty tracebacks

Changed in version 3.5: The '.pyo' file extension is no longer replaced with '.py'.

Changed in version 3.6: Added the *domain* attribute.

domain

Address space of a memory block (`int` or `None`).

tracemalloc uses the domain `0` to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

inclusive

If *inclusive* is `True` (include), only match memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

If *inclusive* is `False` (exclude), ignore memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

lineno

Line number (`int`) of the filter. If *lineno* is `None`, the filter matches any line number.

filename_pattern

Filename pattern of the filter (`str`). Read-only property.

all_frames

If *all_frames* is `True`, all frames of the traceback are checked. If *all_frames* is `False`, only the most recent frame is checked.

This attribute has no effect if the traceback limit is `1`. See the *get_traceback_limit()* function and *Snapshot.traceback_limit* attribute.

Frame

class tracemalloc.**Frame**

Frame of a traceback.

The *Traceback* class is a sequence of *Frame* instances.

filename

Filename (`str`).

lineno

Line number (`int`).

Snapshot

class tracemalloc.**Snapshot**

Snapshot of traces of memory blocks allocated by Python.

The *take_snapshot()* function creates a snapshot instance.

compare_to(*old_snapshot:* Snapshot, *key_type:* str, *cumulative:* bool = *False*)

Compute the differences with an old snapshot. Get statistics as a sorted list of *StatisticDiff* instances grouped by *key_type*.

See the *Snapshot.statistics()* method for *key_type* and *cumulative* parameters.

The result is sorted from the biggest to the smallest by: absolute value of *StatisticDiff.size_diff*, *StatisticDiff.size*, absolute value of *StatisticDiff.count_diff*, *Statistic.count* and then by *StatisticDiff.traceback*.

dump(*filename*)

Write the snapshot into a file.

Use *load()* to reload the snapshot.

filter_traces(*filters*)

Create a new *Snapshot* instance with a filtered *traces* sequence, *filters* is a list of *DomainFilter* and *Filter* instances. If *filters* is an empty list, return a new *Snapshot* instance with a copy of the traces.

All inclusive filters are applied at once, a trace is ignored if no inclusive filters match it. A trace is ignored if at least one exclusive filter matches it.

Changed in version 3.6: *DomainFilter* instances are now also accepted in *filters*.

classmethod load(*filename*)

Load a snapshot from a file.

See also *dump()*.

statistics(*key_type:* str, *cumulative:* bool = *False*)

Get statistics as a sorted list of *Statistic* instances grouped by *key_type*:

key_type	description
`'filename'`	filename
`'lineno'`	filename and line number
`'traceback'`	traceback

If *cumulative* is `True`, cumulate size and count of memory blocks of all frames of the traceback of a trace, not only the most recent frame. The cumulative mode can only be used with *key_type* equals to `'filename'` and `'lineno'`.

The result is sorted from the biggest to the smallest by: *Statistic.size*, *Statistic.count* and then by *Statistic.traceback*.

traceback_limit

Maximum number of frames stored in the traceback of *traces*: result of the *get_traceback_limit()* when the snapshot was taken.

traces

Traces of all memory blocks allocated by Python: sequence of *Trace* instances.

The sequence has an undefined order. Use the *Snapshot.statistics()* method to get a sorted list of statistics.

Statistic

class tracemalloc.**Statistic**

Statistic on memory allocations.

Snapshot.statistics() returns a list of *Statistic* instances.

See also the *StatisticDiff* class.

count

Number of memory blocks (`int`).

size

Total size of memory blocks in bytes (`int`).

traceback

Traceback where the memory block was allocated, *`Traceback`* instance.

StatisticDiff

class tracemalloc.**StatisticDiff**

Statistic difference on memory allocations between an old and a new *`Snapshot`* instance.

`Snapshot.compare_to()` returns a list of *`StatisticDiff`* instances. See also the *`Statistic`* class.

count

Number of memory blocks in the new snapshot (`int`): `0` if the memory blocks have been released in the new snapshot.

count_diff

Difference of number of memory blocks between the old and the new snapshots (`int`): `0` if the memory blocks have been allocated in the new snapshot.

size

Total size of memory blocks in bytes in the new snapshot (`int`): `0` if the memory blocks have been released in the new snapshot.

size_diff

Difference of total size of memory blocks in bytes between the old and the new snapshots (`int`): `0` if the memory blocks have been allocated in the new snapshot.

traceback

Traceback where the memory blocks were allocated, *`Traceback`* instance.

Trace

class tracemalloc.**Trace**

Trace of a memory block.

The *`Snapshot.traces`* attribute is a sequence of *`Trace`* instances.

Changed in version 3.6: Added the *`domain`* attribute.

domain

Address space of a memory block (`int`). Read-only property.

tracemalloc uses the domain `0` to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

size

Size of the memory block in bytes (`int`).

traceback

Traceback where the memory block was allocated, *`Traceback`* instance.

Traceback

class tracemalloc.**Traceback**

Sequence of *Frame* instances sorted from the oldest frame to the most recent frame.

A traceback contains at least `1` frame. If the `tracemalloc` module failed to get a frame, the filename `"<unknown>"` at line number `0` is used.

When a snapshot is taken, tracebacks of traces are limited to *get_traceback_limit()* frames. See the *take_snapshot()* function. The original number of frames of the traceback is stored in the *Traceback.total_nframe* attribute. That allows to know if a traceback has been truncated by the traceback limit.

The *Trace.traceback* attribute is an instance of *Traceback* instance.

Changed in version 3.7: Frames are now sorted from the oldest to the most recent, instead of most recent to oldest.

total_nframe

Total number of frames that composed the traceback before truncation. This attribute can be set to `None` if the information is not available.

Changed in version 3.9: The *Traceback.total_nframe* attribute was added.

format(*limit=None*, *most_recent_first=False*)

Format the traceback as a list of lines. Use the *linecache* module to retrieve lines from the source code. If *limit* is set, format the *limit* most recent frames if *limit* is positive. Otherwise, format the `abs(limit)` oldest frames. If *most_recent_first* is `True`, the order of the formatted frames is reversed, returning the most recent frame first instead of last.

Similar to the *traceback.format_tb()* function, except that *format()* does not include newlines.

Example:

```
print("Traceback (most recent call first):")
for line in traceback:
    print(line)
```

Output:

```
Traceback (most recent call first):
  File "test.py", line 9
    obj = Object()
  File "test.py", line 12
    tb = tracemalloc.get_object_traceback(f())
```

CHAPTER

TWENTYEIGHT

SOFTWARE PACKAGING AND DISTRIBUTION

These libraries help you with publishing and installing Python software. While these modules are designed to work in conjunction with the Python Package Index, they can also be used with a local index server, or without any index server at all.

28.1 `ensurepip` — Bootstrapping the `pip` installer

Added in version 3.4.

Source code: Lib/ensurepip

The *`ensurepip`* package provides support for bootstrapping the `pip` installer into an existing Python installation or virtual environment. This bootstrapping approach reflects the fact that `pip` is an independent project with its own release cycle, and the latest available stable version is bundled with maintenance and feature releases of the CPython reference interpreter.

In most cases, end users of Python shouldn't need to invoke this module directly (as `pip` should be bootstrapped by default), but it may be needed if installing `pip` was skipped when installing Python (or when creating a virtual environment) or after explicitly uninstalling `pip`.

Note

This module *does not* access the internet. All of the components needed to bootstrap `pip` are included as internal parts of the package.

See also

installing-index
The end user guide for installing Python packages

PEP 453: Explicit bootstrapping of pip in Python installations
The original rationale and specification for this module.

Availability: not WASI, not iOS.

This module does not work or is not available on WebAssembly platforms, or on iOS. See *WebAssembly platforms* for more information on WASM availability; see *iOS* for more information on iOS availability.

28.1.1 Command line interface

The command line interface is invoked using the interpreter's `-m` switch.

The simplest possible invocation is:

```
python -m ensurepip
```

This invocation will install `pip` if it is not already installed, but otherwise does nothing. To ensure the installed version of `pip` is at least as recent as the one available in `ensurepip`, pass the `--upgrade` option:

```
python -m ensurepip --upgrade
```

By default, `pip` is installed into the current virtual environment (if one is active) or into the system site packages (if there is no active virtual environment). The installation location can be controlled through two additional command line options:

- `--root` *dir*: Installs `pip` relative to the given root directory rather than the root of the currently active virtual environment (if any) or the default root for the current Python installation.
- `--user`: Installs `pip` into the user site packages directory rather than globally for the current Python installation (this option is not permitted inside an active virtual environment).

By default, the scripts `pipX` and `pipX.Y` will be installed (where X.Y stands for the version of Python used to invoke `ensurepip`). The scripts installed can be controlled through two additional command line options:

- `--altinstall`: if an alternate installation is requested, the `pipX` script will *not* be installed.
- `--default-pip`: if a "default pip" installation is requested, the `pip` script will be installed in addition to the two regular scripts.

Providing both of the script selection options will trigger an exception.

28.1.2 Module API

ensurepip exposes two functions for programmatic use:

ensurepip.**version**()

Returns a string specifying the available version of pip that will be installed when bootstrapping an environment.

ensurepip.**bootstrap**(*root=None*, *upgrade=False*, *user=False*, *altinstall=False*, *default_pip=False*, *verbosity=0*)

Bootstraps `pip` into the current or designated environment.

root specifies an alternative root directory to install relative to. If *root* is `None`, then installation uses the default install location for the current environment.

upgrade indicates whether or not to upgrade an existing installation of an earlier version of `pip` to the available version.

user indicates whether to use the user scheme rather than installing globally.

By default, the scripts `pipX` and `pipX.Y` will be installed (where X.Y stands for the current version of Python).

If *altinstall* is set, then `pipX` will *not* be installed.

If *default_pip* is set, then `pip` will be installed in addition to the two regular scripts.

Setting both *altinstall* and *default_pip* will trigger *ValueError*.

verbosity controls the level of output to *sys.stdout* from the bootstrapping operation.

Raises an *auditing event* `ensurepip.bootstrap` with argument `root`.

Note

The bootstrapping process has side effects on both `sys.path` and `os.environ`. Invoking the command line interface in a subprocess instead allows these side effects to be avoided.

Note

The bootstrapping process may install additional modules required by `pip`, but other software should not assume those dependencies will always be present by default (as the dependencies may be removed in a future version of `pip`).

28.2 `venv` — Creation of virtual environments

Added in version 3.3.

Source code: Lib/venv/

The `venv` module supports creating lightweight "virtual environments", each with their own independent set of Python packages installed in their `site` directories. A virtual environment is created on top of an existing Python installation, known as the virtual environment's "base" Python, and may optionally be isolated from the packages in the base environment, so only those explicitly installed in the virtual environment are available.

When used from within a virtual environment, common installation tools such as pip will install Python packages into a virtual environment without needing to be told to do so explicitly.

A virtual environment is (amongst other things):

- Used to contain a specific Python interpreter and software libraries and binaries which are needed to support a project (library or application). These are by default isolated from software in other virtual environments and Python interpreters and libraries installed in the operating system.
- Contained in a directory, conventionally either named `venv` or `.venv` in the project directory, or under a container directory for lots of virtual environments, such as `~/.virtualenvs`.
- Not checked into source control systems such as Git.
- Considered as disposable – it should be simple to delete and recreate it from scratch. You don't place any project code in the environment
- Not considered as movable or copyable – you just recreate the same environment in the target location.

See **PEP 405** for more background on Python virtual environments.

See also

Python Packaging User Guide: Creating and using virtual environments

Availability: not WASI, not iOS.

This module does not work or is not available on WebAssembly platforms, or on iOS. See *WebAssembly platforms* for more information on WASM availability; see *iOS* for more information on iOS availability.

28.2.1 Creating virtual environments

Creation of *virtual environments* is done by executing the command `venv`:

```
python -m venv /path/to/new/virtual/environment
```

Running this command creates the target directory (creating any parent directories that don't exist already) and places a `pyvenv.cfg` file in it with a `home` key pointing to the Python installation from which the command was run (a common name for the target directory is `.venv`). It also creates a `bin` (or `Scripts` on Windows) subdirectory containing a copy/symlink of the Python binary/binaries (as appropriate for the platform or arguments used at environment creation time). It also creates an (initially empty) `lib/pythonX.Y/site-packages` subdirectory (on Windows, this is `Lib\site-packages`). If an existing directory is specified, it will be re-used.

Changed in version 3.5: The use of `venv` is now recommended for creating virtual environments.

Deprecated since version 3.6: `pyvenv` was the recommended tool for creating virtual environments for Python 3.3 and 3.4, and is deprecated in Python 3.6.

On Windows, invoke the `venv` command as follows:

```
c:\>Python35\python -m venv c:\path\to\myenv
```

Alternatively, if you configured the `PATH` and `PATHEXT` variables for your Python installation:

```
c:\>python -m venv c:\path\to\myenv
```

The command, if run with `-h`, will show the available options:

```
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]
            [--upgrade] [--without-pip] [--prompt PROMPT] [--upgrade-deps]
            [--without-scm-ignore-file]
            ENV_DIR [ENV_DIR ...]

Creates virtual Python environments in one or more target directories.

positional arguments:
ENV_DIR               A directory to create the environment in.

options:
-h, --help            show this help message and exit
--system-site-packages
                        Give the virtual environment access to the system
                        site-packages dir.
--symlinks            Try to use symlinks rather than copies, when
                        symlinks are not the default for the platform.
--copies              Try to use copies rather than symlinks, even when
                        symlinks are the default for the platform.
--clear               Delete the contents of the environment directory if
                        it already exists, before environment creation.
--upgrade             Upgrade the environment directory to use this
                        version of Python, assuming Python has been upgraded
                        in-place.
--without-pip         Skips installing or upgrading pip in the virtual
                        environment (pip is bootstrapped by default)
--prompt PROMPT       Provides an alternative prompt prefix for this
                        environment.
--upgrade-deps        Upgrade core dependencies (pip) to the latest
                        version in PyPI
```

(continues on next page)

(continued from previous page)

```
--without-scm-ignore-file
                        Skips adding the default SCM ignore file to the
                        environment directory (the default is a .gitignore
                        file).

Once an environment has been created, you may wish to activate it, e.g. by
sourcing an activate script in its bin directory.
```

Changed in version 3.13: `--without-scm-ignore-file` was added along with creating an ignore file for `git` by default.

Changed in version 3.12: `setuptools` is no longer a core venv dependency.

Changed in version 3.9: Add `--upgrade-deps` option to upgrade pip + setuptools to the latest on PyPI

Changed in version 3.4: Installs pip by default, added the `--without-pip` and `--copies` options

Changed in version 3.4: In earlier versions, if the target directory already existed, an error was raised, unless the `--clear` or `--upgrade` option was provided.

Note

While symlinks are supported on Windows, they are not recommended. Of particular note is that double-clicking `python.exe` in File Explorer will resolve the symlink eagerly and ignore the virtual environment.

Note

On Microsoft Windows, it may be required to enable the `Activate.ps1` script by setting the execution policy for the user. You can do this by issuing the following PowerShell command:

PS C:> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser

See About Execution Policies for more information.

The created `pyvenv.cfg` file also includes the `include-system-site-packages` key, set to `true` if `venv` is run with the `--system-site-packages` option, `false` otherwise.

Unless the `--without-pip` option is given, *ensurepip* will be invoked to bootstrap `pip` into the virtual environment.

Multiple paths can be given to `venv`, in which case an identical virtual environment will be created, according to the given options, at each provided path.

28.2.2 How venvs work

When a Python interpreter is running from a virtual environment, *sys.prefix* and *sys.exec_prefix* point to the directories of the virtual environment, whereas *sys.base_prefix* and *sys.base_exec_prefix* point to those of the base Python used to create the environment. It is sufficient to check `sys.prefix != sys.base_prefix` to determine if the current interpreter is running from a virtual environment.

A virtual environment may be "activated" using a script in its binary directory (`bin` on POSIX; `Scripts` on Windows). This will prepend that directory to your `PATH`, so that running **python** will invoke the environment's Python interpreter and you can run installed scripts without having to use their full path. The invocation of the activation script is platform-specific (*<venv>* must be replaced by the path to the directory containing the virtual environment):

Platform	Shell	Command to activate virtual environment
POSIX	bash/zsh	`$ source <venv>/bin/activate`
	fish	`$ source <venv>/bin/activate.fish`
	csh/tcsh	`$ source <venv>/bin/activate.csh`
	PowerShell	`$ <venv>/bin/Activate.ps1`
Windows	cmd.exe	`C:\> <venv>\Scripts\activate.bat`
	PowerShell	`PS C:\> <venv>\Scripts\Activate.ps1`

Added in version 3.4: **fish** and **csh** activation scripts.

Added in version 3.8: PowerShell activation scripts installed under POSIX for PowerShell Core support.

You don't specifically *need* to activate a virtual environment, as you can just specify the full path to that environment's Python interpreter when invoking Python. Furthermore, all scripts installed in the environment should be runnable without activating it.

In order to achieve this, scripts installed into virtual environments have a "shebang" line which points to the environment's Python interpreter, i.e. `#!/<path-to-venv>/bin/python`. This means that the script will run with that interpreter regardless of the value of `PATH`. On Windows, "shebang" line processing is supported if you have the launcher installed. Thus, double-clicking an installed script in a Windows Explorer window should run it with the correct interpreter without the environment needing to be activated or on the `PATH`.

When a virtual environment has been activated, the `VIRTUAL_ENV` environment variable is set to the path of the environment. Since explicitly activating a virtual environment is not required to use it, `VIRTUAL_ENV` cannot be relied upon to determine whether a virtual environment is being used.

Warning

Because scripts installed in environments should not expect the environment to be activated, their shebang lines contain the absolute paths to their environment's interpreters. Because of this, environments are inherently non-portable, in the general case. You should always have a simple means of recreating an environment (for example, if you have a requirements file `requirements.txt`, you can invoke `pip install -r requirements.txt` using the environment's `pip` to install all of the packages needed by the environment). If for any reason you need to move the environment to a new location, you should recreate it at the desired location and delete the one at the old location. If you move an environment because you moved a parent directory of it, you should recreate the environment in its new location. Otherwise, software installed into the environment may not work as expected.

You can deactivate a virtual environment by typing `deactivate` in your shell. The exact mechanism is platform-specific and is an internal implementation detail (typically, a script or shell function will be used).

28.2.3 API

The high-level method described above makes use of a simple API which provides mechanisms for third-party virtual environment creators to customize environment creation according to their needs, the *EnvBuilder* class.

class venv.**EnvBuilder**(*system_site_packages=False*, *clear=False*, *symlinks=False*, *upgrade=False*, *with_pip=False*, *prompt=None*, *upgrade_deps=False*, ***, *scm_ignore_files=frozenset()*)

The *EnvBuilder* class accepts the following keyword arguments on instantiation:

- `system_site_packages` – a Boolean value indicating that the system Python site-packages should be available to the environment (defaults to `False`).

- `clear` – a Boolean value which, if true, will delete the contents of any existing target directory, before creating the environment.
- `symlinks` – a Boolean value indicating whether to attempt to symlink the Python binary rather than copying.
- `upgrade` – a Boolean value which, if true, will upgrade an existing environment with the running Python - for use when that Python has been upgraded in-place (defaults to `False`).
- `with_pip` – a Boolean value which, if true, ensures pip is installed in the virtual environment. This uses *ensurepip* with the `--default-pip` option.
- `prompt` – a String to be used after virtual environment is activated (defaults to `None` which means directory name of the environment would be used). If the special string `"."` is provided, the basename of the current directory is used as the prompt.
- `upgrade_deps` – Update the base venv modules to the latest on PyPI
- `scm_ignore_files` – Create ignore files based for the specified source control managers (SCM) in the iterable. Support is defined by having a method named `create_{scm}_ignore_file`. The only value supported by default is `"git"` via *create_git_ignore_file()*.

Changed in version 3.4: Added the `with_pip` parameter

Changed in version 3.6: Added the `prompt` parameter

Changed in version 3.9: Added the `upgrade_deps` parameter

Changed in version 3.13: Added the `scm_ignore_files` parameter

Creators of third-party virtual environment tools will be free to use the provided *EnvBuilder* class as a base class.

The returned env-builder is an object which has a method, `create`:

create(*env_dir*)

Create a virtual environment by specifying the target directory (absolute or relative to the current directory) which is to contain the virtual environment. The `create` method will either create the environment in the specified directory, or raise an appropriate exception.

The `create` method of the *EnvBuilder* class illustrates the hooks available for subclass customization:

```
def create(self, env_dir):
    """
    Create a virtualized Python environment in a directory.
    env_dir is the target directory to create an environment in.
    """
    env_dir = os.path.abspath(env_dir)
    context = self.ensure_directories(env_dir)
    self.create_configuration(context)
    self.setup_python(context)
    self.setup_scripts(context)
    self.post_setup(context)
```

Each of the methods *ensure_directories()*, *create_configuration()*, *setup_python()*, *setup_scripts()* and *post_setup()* can be overridden.

ensure_directories(*env_dir*)

Creates the environment directory and all necessary subdirectories that don't already exist, and returns a context object. This context object is just a holder for attributes (such as paths) for use by the other methods. If the *EnvBuilder* is created with the arg `clear=True`, contents of the environment directory will be cleared and then all necessary subdirectories will be recreated.

The returned context object is a *types.SimpleNamespace* with the following attributes:

- `env_dir` - The location of the virtual environment. Used for `__VENV_DIR__` in activation scripts (see *`install_scripts()`*).
- `env_name` - The name of the virtual environment. Used for `__VENV_NAME__` in activation scripts (see *`install_scripts()`*).
- `prompt` - The prompt to be used by the activation scripts. Used for `__VENV_PROMPT__` in activation scripts (see *`install_scripts()`*).
- `executable` - The underlying Python executable used by the virtual environment. This takes into account the case where a virtual environment is created from another virtual environment.
- `inc_path` - The include path for the virtual environment.
- `lib_path` - The purelib path for the virtual environment.
- `bin_path` - The script path for the virtual environment.
- `bin_name` - The name of the script path relative to the virtual environment location. Used for `__VENV_BIN_NAME__` in activation scripts (see *`install_scripts()`*).
- `env_exe` - The name of the Python interpreter in the virtual environment. Used for `__VENV_PYTHON__` in activation scripts (see *`install_scripts()`*).
- `env_exec_cmd` - The name of the Python interpreter, taking into account filesystem redirections. This can be used to run Python in the virtual environment.

Changed in version 3.11: The *venv sysconfig installation scheme* is used to construct the paths of the created directories.

Changed in version 3.12: The attribute `lib_path` was added to the context, and the context object was documented.

`create_configuration`(*context*)

Creates the `pyvenv.cfg` configuration file in the environment.

`setup_python`(*context*)

Creates a copy or symlink to the Python executable in the environment. On POSIX systems, if a specific executable `python3.x` was used, symlinks to `python` and `python3` will be created pointing to that executable, unless files with those names already exist.

`setup_scripts`(*context*)

Installs activation scripts appropriate to the platform into the virtual environment.

`upgrade_dependencies`(*context*)

Upgrades the core venv dependency packages (currently `pip`) in the environment. This is done by shelling out to the `pip` executable in the environment.

Added in version 3.9.

Changed in version 3.12: `setuptools` is no longer a core venv dependency.

`post_setup`(*context*)

A placeholder method which can be overridden in third party implementations to pre-install packages in the virtual environment or perform other post-creation steps.

Changed in version 3.7.2: Windows now uses redirector scripts for `python[w].exe` instead of copying the actual binaries. In 3.7.2 only *`setup_python()`* does nothing unless running from a build in the source tree.

Changed in version 3.7.3: Windows copies the redirector scripts as part of *`setup_python()`* instead of *`setup_scripts()`*. This was not the case in 3.7.2. When using symlinks, the original executables will be linked.

In addition, *EnvBuilder* provides this utility method that can be called from *setup_scripts()* or *post_setup()* in subclasses to assist in installing custom scripts into the virtual environment.

install_scripts(*context*, *path*)

path is the path to a directory that should contain subdirectories "common", "posix", "nt", each containing scripts destined for the bin directory in the environment. The contents of "common" and the directory corresponding to *os.name* are copied after some text replacement of placeholders:

- __VENV_DIR__ is replaced with the absolute path of the environment directory.
- __VENV_NAME__ is replaced with the environment name (final path segment of environment directory).
- __VENV_PROMPT__ is replaced with the prompt (the environment name surrounded by parentheses and with a following space)
- __VENV_BIN_NAME__ is replaced with the name of the bin directory (either `bin` or `Scripts`).
- __VENV_PYTHON__ is replaced with the absolute path of the environment's executable.

The directories are allowed to exist (for when an existing environment is being upgraded).

create_git_ignore_file(*context*)

Creates a `.gitignore` file within the virtual environment that causes the entire directory to be ignored by the `git` source control manager.

Added in version 3.13.

There is also a module-level convenience function:

venv.**create**(*env_dir*, *system_site_packages=False*, *clear=False*, *symlinks=False*, *with_pip=False*, *prompt=None*, *upgrade_deps=False*, ***, *scm_ignore_files=frozenset()*)

Create an *EnvBuilder* with the given keyword arguments, and call its *create()* method with the *env_dir* argument.

Added in version 3.3.

Changed in version 3.4: Added the `with_pip` parameter

Changed in version 3.6: Added the `prompt` parameter

Changed in version 3.9: Added the `upgrade_deps` parameter

Changed in version 3.13: Added the `scm_ignore_files` parameter

28.2.4 An example of extending `EnvBuilder`

The following script shows how to extend *EnvBuilder* by implementing a subclass which installs setuptools and pip into a created virtual environment:

```
import os
import os.path
from subprocess import Popen, PIPE
import sys
from threading import Thread
from urllib.parse import urlparse
from urllib.request import urlretrieve
import venv

class ExtendedEnvBuilder(venv.EnvBuilder):
```

(continues on next page)

(continued from previous page)

```
    """
    This builder installs setuptools and pip so that you can pip or
    easy_install other packages into the created virtual environment.

    :param nodist: If true, setuptools and pip are not installed into the
                   created virtual environment.
    :param nopip: If true, pip is not installed into the created
                  virtual environment.
    :param progress: If setuptools or pip are installed, the progress of the
                     installation can be monitored by passing a progress
                     callable. If specified, it is called with two
                     arguments: a string indicating some progress, and a
                     context indicating where the string is coming from.
                     The context argument can have one of three values:
                     'main', indicating that it is called from virtualize()
                     itself, and 'stdout' and 'stderr', which are obtained
                     by reading lines from the output streams of a subprocess
                     which is used to install the app.

                     If a callable is not specified, default progress
                     information is output to sys.stderr.
    """

    def __init__(self, *args, **kwargs):
        self.nodist = kwargs.pop('nodist', False)
        self.nopip = kwargs.pop('nopip', False)
        self.progress = kwargs.pop('progress', None)
        self.verbose = kwargs.pop('verbose', False)
        super().__init__(*args, **kwargs)

    def post_setup(self, context):
        """
        Set up any packages which need to be pre-installed into the
        virtual environment being created.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        os.environ['VIRTUAL_ENV'] = context.env_dir
        if not self.nodist:
            self.install_setuptools(context)
        # Can't install pip without setuptools
        if not self.nopip and not self.nodist:
            self.install_pip(context)

    def reader(self, stream, context):
        """
        Read lines from a subprocess' output stream and either pass to a progress
        callable (if specified) or write progress information to sys.stderr.
        """
        progress = self.progress
        while True:
            s = stream.readline()
            if not s:
                break
            if progress is not None:
                progress(s, context)
```

(continues on next page)

(continued from previous page)

```
            else:
                if not self.verbose:
                    sys.stderr.write('.')
                else:
                    sys.stderr.write(s.decode('utf-8'))
                sys.stderr.flush()
        stream.close()

    def install_script(self, context, name, url):
        _, _, path, _, _, _ = urlparse(url)
        fn = os.path.split(path)[-1]
        binpath = context.bin_path
        distpath = os.path.join(binpath, fn)
        # Download script into the virtual environment's binaries folder
        urlretrieve(url, distpath)
        progress = self.progress
        if self.verbose:
            term = '\n'
        else:
            term = ''
        if progress is not None:
            progress('Installing %s ...%s' % (name, term), 'main')
        else:
            sys.stderr.write('Installing %s ...%s' % (name, term))
            sys.stderr.flush()
        # Install in the virtual environment
        args = [context.env_exe, fn]
        p = Popen(args, stdout=PIPE, stderr=PIPE, cwd=binpath)
        t1 = Thread(target=self.reader, args=(p.stdout, 'stdout'))
        t1.start()
        t2 = Thread(target=self.reader, args=(p.stderr, 'stderr'))
        t2.start()
        p.wait()
        t1.join()
        t2.join()
        if progress is not None:
            progress('done.', 'main')
        else:
            sys.stderr.write('done.\n')
        # Clean up - no longer needed
        os.unlink(distpath)

    def install_setuptools(self, context):
        """
        Install setuptools in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        url = "https://bootstrap.pypa.io/ez_setup.py"
        self.install_script(context, 'setuptools', url)
        # clear up the setuptools archive which gets downloaded
        pred = lambda o: o.startswith('setuptools-') and o.endswith('.tar.gz')
        files = filter(pred, os.listdir(context.bin_path))
        for f in files:
            f = os.path.join(context.bin_path, f)
            os.unlink(f)
```

(continues on next page)

(continued from previous page)

```
    def install_pip(self, context):
        """
        Install pip in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        url = 'https://bootstrap.pypa.io/get-pip.py'
        self.install_script(context, 'pip', url)


def main(args=None):
    import argparse

    parser = argparse.ArgumentParser(prog=__name__,
                                     description='Creates virtual Python '
                                                 'environments in one or '
                                                 'more target '
                                                 'directories.')
    parser.add_argument('dirs', metavar='ENV_DIR', nargs='+',
                        help='A directory in which to create the '
                             'virtual environment.')
    parser.add_argument('--no-setuptools', default=False,
                        action='store_true', dest='nodist',
                        help="Don't install setuptools or pip in the "
                             "virtual environment.")
    parser.add_argument('--no-pip', default=False,
                        action='store_true', dest='nopip',
                        help="Don't install pip in the virtual "
                             "environment.")
    parser.add_argument('--system-site-packages', default=False,
                        action='store_true', dest='system_site',
                        help='Give the virtual environment access to the '
                             'system site-packages dir.')
    if os.name == 'nt':
        use_symlinks = False
    else:
        use_symlinks = True
    parser.add_argument('--symlinks', default=use_symlinks,
                        action='store_true', dest='symlinks',
                        help='Try to use symlinks rather than copies, '
                             'when symlinks are not the default for '
                             'the platform.')
    parser.add_argument('--clear', default=False, action='store_true',
                        dest='clear', help='Delete the contents of the '
                                           'virtual environment '
                                           'directory if it already '
                                           'exists, before virtual '
                                           'environment creation.')
    parser.add_argument('--upgrade', default=False, action='store_true',
                        dest='upgrade', help='Upgrade the virtual '
                                             'environment directory to '
                                             'use this version of '
                                             'Python, assuming Python '
                                             'has been upgraded '
                                             'in-place.')
```

(continues on next page)

(continued from previous page)

```
    parser.add_argument('--verbose', default=False, action='store_true',
                        dest='verbose', help='Display the output '
                                             'from the scripts which '
                                             'install setuptools and pip.')
    options = parser.parse_args(args)
    if options.upgrade and options.clear:
        raise ValueError('you cannot supply --upgrade and --clear together.')
    builder = ExtendedEnvBuilder(system_site_packages=options.system_site,
                                 clear=options.clear,
                                 symlinks=options.symlinks,
                                 upgrade=options.upgrade,
                                 nodist=options.nodist,
                                 nopip=options.nopip,
                                 verbose=options.verbose)
    for d in options.dirs:
        builder.create(d)

if __name__ == '__main__':
    rc = 1
    try:
        main()
        rc = 0
    except Exception as e:
        print('Error: %s' % e, file=sys.stderr)
    sys.exit(rc)
```

This script is also available for download online.

28.3 `zipapp` — Manage executable Python zip archives

Added in version 3.5.

Source code: Lib/zipapp.py

This module provides tools to manage the creation of zip files containing Python code, which can be executed directly by the Python interpreter. The module provides both a *Command-Line Interface* and a *Python API*.

28.3.1 Basic Example

The following example shows how the *Command-Line Interface* can be used to create an executable archive from a directory containing Python code. When run, the archive will execute the `main` function from the module `myapp` in the archive.

```
$ python -m zipapp myapp -m "myapp:main"
$ python myapp.pyz
<output from myapp>
```

28.3.2 Command-Line Interface

When called as a program from the command line, the following form is used:

```
$ python -m zipapp source [options]
```

If *source* is a directory, this will create an archive from the contents of *source*. If *source* is a file, it should be an archive, and it will be copied to the target archive (or the contents of its shebang line will be displayed if the –info option is specified).

The following options are understood:

-o <output>, **--output**=<output>

Write the output to a file named *output*. If this option is not specified, the output filename will be the same as the input *source*, with the extension `.pyz` added. If an explicit filename is given, it is used as is (so a `.pyz` extension should be included if required).

An output filename must be specified if the *source* is an archive (and in that case, *output* must not be the same as *source*).

-p <interpreter>, **--python**=<interpreter>

Add a `#!` line to the archive specifying *interpreter* as the command to run. Also, on POSIX, make the archive executable. The default is to write no `#!` line, and not make the file executable.

-m <mainfn>, **--main**=<mainfn>

Write a `__main__.py` file to the archive that executes *mainfn*. The *mainfn* argument should have the form "pkg.mod:fn", where "pkg.mod" is a package/module in the archive, and "fn" is a callable in the given module. The `__main__.py` file will execute that callable.

--main cannot be specified when copying an archive.

-c, --compress

Compress files with the deflate method, reducing the size of the output file. By default, files are stored uncompressed in the archive.

--compress has no effect when copying an archive.

Added in version 3.7.

--info

Display the interpreter embedded in the archive, for diagnostic purposes. In this case, any other options are ignored and SOURCE must be an archive, not a directory.

-h, --help

Print a short usage message and exit.

28.3.3 Python API

The module defines two convenience functions:

zipapp.**create_archive**(*source*, *target=None*, *interpreter=None*, *main=None*, *filter=None*, *compressed=False*)

Create an application archive from *source*. The source can be any of the following:

- The name of a directory, or a *path-like object* referring to a directory, in which case a new application archive will be created from the content of that directory.
- The name of an existing application archive file, or a *path-like object* referring to such a file, in which case the file is copied to the target (modifying it to reflect the value given for the *interpreter* argument). The file name should include the `.pyz` extension, if required.

- A file object open for reading in bytes mode. The content of the file should be an application archive, and the file object is assumed to be positioned at the start of the archive.

The *target* argument determines where the resulting archive will be written:

- If it is the name of a file, or a *path-like object*, the archive will be written to that file.
- If it is an open file object, the archive will be written to that file object, which must be open for writing in bytes mode.
- If the target is omitted (or `None`), the source must be a directory and the target will be a file with the same name as the source, with a `.pyz` extension added.

The *interpreter* argument specifies the name of the Python interpreter with which the archive will be executed. It is written as a "shebang" line at the start of the archive. On POSIX, this will be interpreted by the OS, and on Windows it will be handled by the Python launcher. Omitting the *interpreter* results in no shebang line being written. If an interpreter is specified, and the target is a filename, the executable bit of the target file will be set.

The *main* argument specifies the name of a callable which will be used as the main program for the archive. It can only be specified if the source is a directory, and the source does not already contain a `__main__.py` file. The *main* argument should take the form "pkg.module:callable" and the archive will be run by importing "pkg.module" and executing the given callable with no arguments. It is an error to omit *main* if the source is a directory and does not contain a `__main__.py` file, as otherwise the resulting archive would not be executable.

The optional *filter* argument specifies a callback function that is passed a Path object representing the path to the file being added (relative to the source directory). It should return `True` if the file is to be added.

The optional *compressed* argument determines whether files are compressed. If set to `True`, files in the archive are compressed with the deflate method; otherwise, files are stored uncompressed. This argument has no effect when copying an existing archive.

If a file object is specified for *source* or *target*, it is the caller's responsibility to close it after calling create_archive.

When copying an existing archive, file objects supplied only need `read` and `readline`, or `write` methods. When creating an archive from a directory, if the target is a file object it will be passed to the `zipfile.ZipFile` class, and must supply the methods needed by that class.

Changed in version 3.7: Added the *filter* and *compressed* parameters.

`zipapp.`**`get_interpreter`**(*archive*)

Return the interpreter specified in the `#!` line at the start of the archive. If there is no `#!` line, return *None*. The *archive* argument can be a filename or a file-like object open for reading in bytes mode. It is assumed to be at the start of the archive.

28.3.4 Examples

Pack up a directory into an archive, and run it.

```
$ python -m zipapp myapp
$ python myapp.pyz
<output from myapp>
```

The same can be done using the *create_archive()* function:

```
>>> import zipapp
>>> zipapp.create_archive('myapp', 'myapp.pyz')
```

To make the application directly executable on POSIX, specify an interpreter to use.

```
$ python -m zipapp myapp -p "/usr/bin/env python"
$ ./myapp.pyz
<output from myapp>
```

To replace the shebang line on an existing archive, create a modified archive using the *create_archive()* function:

```
>>> import zipapp
>>> zipapp.create_archive('old_archive.pyz', 'new_archive.pyz', '/usr/bin/python3')
```

To update the file in place, do the replacement in memory using a *BytesIO* object, and then overwrite the source afterwards. Note that there is a risk when overwriting a file in place that an error will result in the loss of the original file. This code does not protect against such errors, but production code should do so. Also, this method will only work if the archive fits in memory:

```
>>> import zipapp
>>> import io
>>> temp = io.BytesIO()
>>> zipapp.create_archive('myapp.pyz', temp, '/usr/bin/python2')
>>> with open('myapp.pyz', 'wb') as f:
>>>     f.write(temp.getvalue())
```

28.3.5 Specifying the Interpreter

Note that if you specify an interpreter and then distribute your application archive, you need to ensure that the interpreter used is portable. The Python launcher for Windows supports most common forms of POSIX `#!` line, but there are other issues to consider:

- If you use "/usr/bin/env python" (or other forms of the "python" command, such as "/usr/bin/python"), you need to consider that your users may have either Python 2 or Python 3 as their default, and write your code to work under both versions.
- If you use an explicit version, for example "/usr/bin/env python3" your application will not work for users who do not have that version. (This may be what you want if you have not made your code Python 2 compatible).
- There is no way to say "python X.Y or later", so be careful of using an exact version like "/usr/bin/env python3.4" as you will need to change your shebang line for users of Python 3.5, for example.

Typically, you should use an "/usr/bin/env python2" or "/usr/bin/env python3", depending on whether your code is written for Python 2 or 3.

28.3.6 Creating Standalone Applications with zipapp

Using the *zipapp* module, it is possible to create self-contained Python programs, which can be distributed to end users who only need to have a suitable version of Python installed on their system. The key to doing this is to bundle all of the application's dependencies into the archive, along with the application code.

The steps to create a standalone archive are as follows:

1. Create your application in a directory as normal, so you have a `myapp` directory containing a `__main__.py` file, and any supporting application code.
2. Install all of your application's dependencies into the `myapp` directory, using pip:

   ```
   $ python -m pip install -r requirements.txt --target myapp
   ```

(this assumes you have your project requirements in a `requirements.txt` file - if not, you can just list the dependencies manually on the pip command line).

3. Package the application using:

```
$ python -m zipapp -p "interpreter" myapp
```

This will produce a standalone executable, which can be run on any machine with the appropriate interpreter available. See *Specifying the Interpreter* for details. It can be shipped to users as a single file.

On Unix, the `myapp.pyz` file is executable as it stands. You can rename the file to remove the `.pyz` extension if you prefer a "plain" command name. On Windows, the `myapp.pyz[w]` file is executable by virtue of the fact that the Python interpreter registers the `.pyz` and `.pyzw` file extensions when installed.

Caveats

If your application depends on a package that includes a C extension, that package cannot be run from a zip file (this is an OS limitation, as executable code must be present in the filesystem for the OS loader to load it). In this case, you can exclude that dependency from the zipfile, and either require your users to have it installed, or ship it alongside your zipfile and add code to your `__main__.py` to include the directory containing the unzipped module in `sys.path`. In this case, you will need to make sure to ship appropriate binaries for your target architecture(s) (and potentially pick the correct version to add to `sys.path` at runtime, based on the user's machine).

28.3.7 The Python Zip Application Archive Format

Python has been able to execute zip files which contain a `__main__.py` file since version 2.6. In order to be executed by Python, an application archive simply has to be a standard zip file containing a `__main__.py` file which will be run as the entry point for the application. As usual for any Python script, the parent of the script (in this case the zip file) will be placed on *`sys.path`* and thus further modules can be imported from the zip file.

The zip file format allows arbitrary data to be prepended to a zip file. The zip application format uses this ability to prepend a standard POSIX "shebang" line to the file (`#!/path/to/interpreter`).

Formally, the Python zip application format is therefore:

1. An optional shebang line, containing the characters `b'#!'` followed by an interpreter name, and then a newline (`b'\n'`) character. The interpreter name can be anything acceptable to the OS "shebang" processing, or the Python launcher on Windows. The interpreter should be encoded in UTF-8 on Windows, and in *`sys.getfilesystemencoding()`* on POSIX.
2. Standard zipfile data, as generated by the *`zipfile`* module. The zipfile content *must* include a file called `__main__.py` (which must be in the "root" of the zipfile - i.e., it cannot be in a subdirectory). The zipfile data can be compressed or uncompressed.

If an application archive has a shebang line, it may have the executable bit set on POSIX systems, to allow it to be executed directly.

There is no requirement that the tools in this module are used to create application archives - the module is a convenience, but archives in the above format created by any means are acceptable to Python.

CHAPTER

TWENTYNINE

PYTHON RUNTIME SERVICES

The modules described in this chapter provide a wide range of services related to the Python interpreter and its interaction with its environment. Here's an overview:

29.1 `sys` — System-specific parameters and functions

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available.

`sys.`**`abiflags`**

On POSIX systems where Python was built with the standard `configure` script, this contains the ABI flags as specified by **PEP 3149**.

Added in version 3.2.

Changed in version 3.8: Default flags became an empty string (`m` flag for pymalloc has been removed).

Availability: Unix.

`sys.`**`addaudithook`**`(`*`hook`*`)`

Append the callable *hook* to the list of active auditing hooks for the current (sub)interpreter.

When an auditing event is raised through the *`sys.audit()`* function, each hook will be called in the order it was added with the event name and the tuple of arguments. Native hooks added by `PySys_AddAuditHook()` are called first, followed by hooks added in the current (sub)interpreter. Hooks can then log the event, raise an exception to abort the operation, or terminate the process entirely.

Note that audit hooks are primarily for collecting information about internal or otherwise unobservable actions, whether by Python or libraries written in Python. They are not suitable for implementing a "sandbox". In particular, malicious code can trivially disable or bypass hooks added using this function. At a minimum, any security-sensitive hooks must be added using the C API `PySys_AddAuditHook()` before initialising the runtime, and any modules allowing arbitrary memory modification (such as *`ctypes`*) should be completely removed or closely monitored.

Calling *`sys.addaudithook()`* will itself raise an auditing event named `sys.addaudithook` with no arguments. If any existing hooks raise an exception derived from *`RuntimeError`*, the new hook will not be added and the exception suppressed. As a result, callers cannot assume that their hook has been added unless they control all existing hooks.

See the *audit events table* for all events raised by CPython, and **PEP 578** for the original design discussion.

Added in version 3.8.

Changed in version 3.8.1: Exceptions derived from *Exception* but not *RuntimeError* are no longer suppressed.

CPython implementation detail: When tracing is enabled (see *settrace()*), Python hooks are only traced if the callable has a `__cantrace__` member that is set to a true value. Otherwise, trace functions will skip the hook.

sys.**argv**

The list of command line arguments passed to a Python script. `argv[0]` is the script name (it is operating system dependent whether this is a full pathname or not). If the command was executed using the `-c` command line option to the interpreter, `argv[0]` is set to the string `'-c'`. If no script name was passed to the Python interpreter, `argv[0]` is the empty string.

To loop over the standard input, or the list of files given on the command line, see the *fileinput* module.

See also *sys.orig_argv*.

> **Note**
>
> On Unix, command line arguments are passed by bytes from OS. Python decodes them with filesystem encoding and "surrogateescape" error handler. When you need original bytes, you can get it by `[os.fsencode(arg) for arg in sys.argv]`.

sys.**audit**(*event*, **args*)

Raise an auditing event and trigger any active auditing hooks. *event* is a string identifying the event, and *args* may contain optional arguments with more information about the event. The number and types of arguments for a given event are considered a public and stable API and should not be modified between releases.

For example, one auditing event is named `os.chdir`. This event has one argument called *path* that will contain the requested new working directory.

sys.audit() will call the existing auditing hooks, passing the event name and arguments, and will re-raise the first exception from any hook. In general, if an exception is raised, it should not be handled and the process should be terminated as quickly as possible. This allows hook implementations to decide how to respond to particular events: they can merely log the event or abort the operation by raising an exception.

Hooks are added using the *sys.addaudithook()* or `PySys_AddAuditHook()` functions.

The native equivalent of this function is `PySys_Audit()`. Using the native function is preferred when possible.

See the *audit events table* for all events raised by CPython.

Added in version 3.8.

sys.**base_exec_prefix**

Set during Python startup, before `site.py` is run, to the same value as *exec_prefix*. If not running in a *virtual environment*, the values will stay the same; if `site.py` finds that a virtual environment is in use, the values of *prefix* and *exec_prefix* will be changed to point to the virtual environment, whereas *base_prefix* and *base_exec_prefix* will remain pointing to the base Python installation (the one which the virtual environment was created from).

Added in version 3.3.

sys.**base_prefix**

Set during Python startup, before `site.py` is run, to the same value as *prefix*. If not running in a *virtual environment*, the values will stay the same; if `site.py` finds that a virtual environment is in use, the values of *prefix* and *exec_prefix* will be changed to point to the virtual environment, whereas *base_prefix* and *base_exec_prefix* will remain pointing to the base Python installation (the one which the virtual environment was created from).

Added in version 3.3.

sys.**byteorder**

An indicator of the native byte order. This will have the value `'big'` on big-endian (most-significant byte first) platforms, and `'little'` on little-endian (least-significant byte first) platforms.

sys.**builtin_module_names**

A tuple of strings containing the names of all modules that are compiled into this Python interpreter. (This information is not available in any other way — `modules.keys()` only lists the imported modules.)

See also the *sys.stdlib_module_names* list.

sys.**call_tracing**(*func*, *args*)

Call `func(*args)`, while tracing is enabled. The tracing state is saved, and restored afterwards. This is intended to be called from a debugger from a checkpoint, to recursively debug or profile some other code.

Tracing is suspended while calling a tracing function set by *settrace()* or *setprofile()* to avoid infinite recursion. `call_tracing()` enables explicit recursion of the tracing function.

sys.**copyright**

A string containing the copyright pertaining to the Python interpreter.

sys.**_clear_type_cache**()

Clear the internal type cache. The type cache is used to speed up attribute and method lookups. Use the function *only* to drop unnecessary references during reference leak debugging.

This function should be used for internal and specialized purposes only.

Deprecated since version 3.13: Use the more general *_clear_internal_caches()* function instead.

sys.**_clear_internal_caches**()

Clear all internal performance-related caches. Use this function *only* to release unnecessary references and memory blocks when hunting for leaks.

Added in version 3.13.

sys.**_current_frames**()

Return a dictionary mapping each thread's identifier to the topmost stack frame currently active in that thread at the time the function is called. Note that functions in the *traceback* module can build the call stack given such a frame.

This is most useful for debugging deadlock: this function does not require the deadlocked threads' cooperation, and such threads' call stacks are frozen for as long as they remain deadlocked. The frame returned for a non-deadlocked thread may bear no relationship to that thread's current activity by the time calling code examines the frame.

This function should be used for internal and specialized purposes only.

Raises an *auditing event* `sys._current_frames` with no arguments.

sys.**_current_exceptions**()

Return a dictionary mapping each thread's identifier to the topmost exception currently active in that thread at the time the function is called. If a thread is not currently handling an exception, it is not included in the result dictionary.

This is most useful for statistical profiling.

This function should be used for internal and specialized purposes only.

Raises an *auditing event* `sys._current_exceptions` with no arguments.

Changed in version 3.12: Each value in the dictionary is now a single exception instance, rather than a 3-tuple as returned from `sys.exc_info()`.

sys.**breakpointhook**()

This hook function is called by built-in *breakpoint()*. By default, it drops you into the *pdb* debugger, but it can be set to any other function so that you can choose which debugger gets used.

The signature of this function is dependent on what it calls. For example, the default binding (e.g. `pdb.set_trace()`) expects no arguments, but you might bind it to a function that expects additional arguments (positional and/or keyword). The built-in `breakpoint()` function passes its `*args` and `**kws` straight through. Whatever `breakpointhooks()` returns is returned from `breakpoint()`.

The default implementation first consults the environment variable `PYTHONBREAKPOINT`. If that is set to `"0"` then this function returns immediately; i.e. it is a no-op. If the environment variable is not set, or is set to the empty string, `pdb.set_trace()` is called. Otherwise this variable should name a function to run, using Python's dotted-import nomenclature, e.g. `package.subpackage.module.function`. In this case, `package.subpackage.module` would be imported and the resulting module must have a callable named `function()`. This is run, passing in `*args` and `**kws`, and whatever `function()` returns, `sys.breakpointhook()` returns to the built-in *breakpoint()* function.

Note that if anything goes wrong while importing the callable named by `PYTHONBREAKPOINT`, a *RuntimeWarning* is reported and the breakpoint is ignored.

Also note that if `sys.breakpointhook()` is overridden programmatically, `PYTHONBREAKPOINT` is *not* consulted.

Added in version 3.7.

sys.**_debugmallocstats**()

Print low-level information to stderr about the state of CPython's memory allocator.

If Python is built in debug mode (`configure --with-pydebug option`), it also performs some expensive internal consistency checks.

Added in version 3.3.

CPython implementation detail: This function is specific to CPython. The exact output format is not defined here, and may change.

sys.**dllhandle**

Integer specifying the handle of the Python DLL.

Availability: Windows.

sys.**displayhook**(*value*)

If *value* is not `None`, this function prints `repr(value)` to `sys.stdout`, and saves *value* in `builtins._`. If `repr(value)` is not encodable to `sys.stdout.encoding` with `sys.stdout.errors` error handler (which is probably `'strict'`), encode it to `sys.stdout.encoding` with `'backslashreplace'` error handler.

`sys.displayhook` is called on the result of evaluating an *expression* entered in an interactive Python session. The display of these values can be customized by assigning another one-argument function to `sys.displayhook`.

Pseudo-code:

```
def displayhook(value):
    if value is None:
        return
    # Set '_' to None to avoid recursion
    builtins._ = None
    text = repr(value)
    try:
```

(continues on next page)

(continued from previous page)

```
        sys.stdout.write(text)
    except UnicodeEncodeError:
        bytes = text.encode(sys.stdout.encoding, 'backslashreplace')
        if hasattr(sys.stdout, 'buffer'):
            sys.stdout.buffer.write(bytes)
        else:
            text = bytes.decode(sys.stdout.encoding, 'strict')
            sys.stdout.write(text)
    sys.stdout.write("\n")
    builtins._ = value
```

Changed in version 3.2: Use `'backslashreplace'` error handler on *UnicodeEncodeError*.

sys.**dont_write_bytecode**

If this is true, Python won't try to write `.pyc` files on the import of source modules. This value is initially set to `True` or `False` depending on the `-B` command line option and the `PYTHONDONTWRITEBYTECODE` environment variable, but you can set it yourself to control bytecode file generation.

sys.**_emscripten_info**

A *named tuple* holding information about the environment on the *wasm32-emscripten* platform. The named tuple is provisional and may change in the future.

_emscripten_info.**emscripten_version**

Emscripten version as tuple of ints (major, minor, micro), e.g. `(3, 1, 8)`.

_emscripten_info.**runtime**

Runtime string, e.g. browser user agent, `'Node.js v14.18.2'`, or `'UNKNOWN'`.

_emscripten_info.**pthreads**

`True` if Python is compiled with Emscripten pthreads support.

_emscripten_info.**shared_memory**

`True` if Python is compiled with shared memory support.

Availability: Emscripten.

Added in version 3.11.

sys.**pycache_prefix**

If this is set (not `None`), Python will write bytecode-cache `.pyc` files to (and read them from) a parallel directory tree rooted at this directory, rather than from `__pycache__` directories in the source code tree. Any `__pycache__` directories in the source code tree will be ignored and new `.pyc` files written within the pycache prefix. Thus if you use *compileall* as a pre-build step, you must ensure you run it with the same pycache prefix (if any) that you will use at runtime.

A relative path is interpreted relative to the current working directory.

This value is initially set based on the value of the `-X pycache_prefix=PATH` command-line option or the `PYTHONPYCACHEPREFIX` environment variable (command-line takes precedence). If neither are set, it is `None`.

Added in version 3.8.

sys.**excepthook**(*type*, *value*, *traceback*)

This function prints out a given traceback and exception to `sys.stderr`.

When an exception other than *SystemExit* is raised and uncaught, the interpreter calls `sys.excepthook` with three arguments, the exception class, exception instance, and a traceback object. In an interactive session this happens just before control is returned to the prompt; in a Python program this happens just before the program

exits. The handling of such top-level exceptions can be customized by assigning another three-argument function to `sys.excepthook`.

Raise an auditing event `sys.excepthook` with arguments `hook`, `type`, `value`, `traceback` when an uncaught exception occurs. If no hook has been set, `hook` may be `None`. If any hook raises an exception derived from *RuntimeError* the call to the hook will be suppressed. Otherwise, the audit hook exception will be reported as unraisable and `sys.excepthook` will be called.

See also

The *sys.unraisablehook()* function handles unraisable exceptions and the *threading.excepthook()* function handles exception raised by *threading.Thread.run()*.

sys.**__breakpointhook__**

sys.**__displayhook__**

sys.**__excepthook__**

sys.**__unraisablehook__**

These objects contain the original values of `breakpointhook`, `displayhook`, `excepthook`, and `unraisablehook` at the start of the program. They are saved so that `breakpointhook`, `displayhook` and `excepthook`, `unraisablehook` can be restored in case they happen to get replaced with broken or alternative objects.

Added in version 3.7: __breakpointhook__

Added in version 3.8: __unraisablehook__

sys.**exception**()

This function, when called while an exception handler is executing (such as an `except` or `except*` clause), returns the exception instance that was caught by this handler. When exception handlers are nested within one another, only the exception handled by the innermost handler is accessible.

If no exception handler is executing, this function returns `None`.

Added in version 3.11.

sys.**exc_info**()

This function returns the old-style representation of the handled exception. If an exception `e` is currently handled (so *exception()* would return `e`), *exc_info()* returns the tuple `(type(e), e, e.__traceback__)`. That is, a tuple containing the type of the exception (a subclass of *BaseException*), the exception itself, and a traceback object which typically encapsulates the call stack at the point where the exception last occurred.

If no exception is being handled anywhere on the stack, this function return a tuple containing three `None` values.

Changed in version 3.11: The `type` and `traceback` fields are now derived from the `value` (the exception instance), so when an exception is modified while it is being handled, the changes are reflected in the results of subsequent calls to *exc_info()*.

sys.**exec_prefix**

A string giving the site-specific directory prefix where the platform-dependent Python files are installed; by default, this is also `'/usr/local'`. This can be set at build time with the `--exec-prefix` argument to the **configure** script. Specifically, all configuration files (e.g. the `pyconfig.h` header file) are installed in the directory *exec_prefix*`/lib/python`*X.Y*`/config`, and shared library modules are installed in *exec_prefix*`/lib/python`*X.Y*`/lib-dynload`, where *X.Y* is the version number of Python, for example `3.2`.

> **Note**
>
> If a *virtual environment* is in effect, this value will be changed in `site.py` to point to the virtual environment. The value for the Python installation will still be available, via *`base_exec_prefix`*.

`sys.`**`executable`**

A string giving the absolute path of the executable binary for the Python interpreter, on systems where this makes sense. If Python is unable to retrieve the real path to its executable, *`sys.executable`* will be an empty string or `None`.

`sys.`**`exit`**`(`[*arg*]`)`

Raise a *`SystemExit`* exception, signaling an intention to exit the interpreter.

The optional argument *arg* can be an integer giving the exit status (defaulting to zero), or another type of object. If it is an integer, zero is considered "successful termination" and any nonzero value is considered "abnormal termination" by shells and the like. Most systems require it to be in the range 0–127, and produce undefined results otherwise. Some systems have a convention for assigning specific meanings to specific exit codes, but these are generally underdeveloped; Unix programs generally use 2 for command line syntax errors and 1 for all other kind of errors. If another type of object is passed, `None` is equivalent to passing zero, and any other object is printed to *`stderr`* and results in an exit code of 1. In particular, `sys.exit("some error message")` is a quick way to exit a program when an error occurs.

Since *`exit()`* ultimately "only" raises an exception, it will only exit the process when called from the main thread, and the exception is not intercepted. Cleanup actions specified by finally clauses of `try` statements are honored, and it is possible to intercept the exit attempt at an outer level.

Changed in version 3.6: If an error occurs in the cleanup after the Python interpreter has caught *`SystemExit`* (such as an error flushing buffered data in the standard streams), the exit status is changed to 120.

`sys.`**`flags`**

The *named tuple* *flags* exposes the status of command line flags. The attributes are read only.

flags.**debug**	-d
flags.**inspect**	-i
flags.**interactive**	-i
flags.**isolated**	-I
flags.**optimize**	-O or -OO
flags.**dont_write_bytecode**	-B
flags.**no_user_site**	-s
flags.**no_site**	-S
flags.**ignore_environment**	-E
flags.**verbose**	-v
flags.**bytes_warning**	-b
flags.**quiet**	-q
flags.**hash_randomization**	-R
flags.**dev_mode**	-X dev (*Python Development Mode*)
flags.**utf8_mode**	-X utf8
flags.**safe_path**	-P
flags.**int_max_str_digits**	-X int_max_str_digits (*integer string conversion length limitation*)
flags.**warn_default_encoding**	-X warn_default_encoding

Changed in version 3.2: Added `quiet` attribute for the new `-q` flag.

Added in version 3.2.3: The `hash_randomization` attribute.

Changed in version 3.3: Removed obsolete `division_warning` attribute.

Changed in version 3.4: Added `isolated` attribute for `-I isolated` flag.

Changed in version 3.7: Added the `dev_mode` attribute for the new *Python Development Mode* and the `utf8_mode` attribute for the new `-X utf8` flag.

Changed in version 3.10: Added `warn_default_encoding` attribute for `-X warn_default_encoding` flag.

Changed in version 3.11: Added the `safe_path` attribute for `-P` option.

Changed in version 3.11: Added the `int_max_str_digits` attribute.

`sys.`**`float_info`**

A *named tuple* holding information about the float type. It contains low level information about the precision and internal representation. The values correspond to the various floating-point constants defined in the standard header file `float.h` for the 'C' programming language; see section 5.2.4.2.2 of the 1999 ISO/IEC C standard [C99], 'Characteristics of floating types', for details.

Table 1: Attributes of the `float_info` named tuple

attribute	float.h macro	explanation
`float_info.`**`epsilon`**	`DBL_EPSILON`	difference between 1.0 and the least value greater than 1.0 that is representable as a float. See also *`math.ulp()`*.
`float_info.`**`dig`**	`DBL_DIG`	The maximum number of decimal digits that can be faithfully represented in a float; see below.
`float_info.`**`mant_dig`**	`DBL_MANT_DIG`	Float precision: the number of base-`radix` digits in the significand of a float.
`float_info.`**`max`**	`DBL_MAX`	The maximum representable positive finite float.
`float_info.`**`max_exp`**	`DBL_MAX_EXP`	The maximum integer *e* such that `radix**(e-1)` is a representable finite float.
`float_info.`**`max_10_exp`**	`DBL_MAX_10_EXP`	The maximum integer *e* such that `10**e` is in the range of representable finite floats.
`float_info.`**`min`**	`DBL_MIN`	The minimum representable positive *normalized* float. Use *`math.ulp(0.0)`* to get the smallest positive *denormalized* representable float.
`float_info.`**`min_exp`**	`DBL_MIN_EXP`	The minimum integer *e* such that `radix**(e-1)` is a normalized float.
`float_info.`**`min_10_exp`**	`DBL_MIN_10_EXP`	The minimum integer *e* such that `10**e` is a normalized float.
`float_info.`**`radix`**	`FLT_RADIX`	The radix of exponent representation.
`float_info.`**`rounds`**	`FLT_ROUNDS`	An integer representing the rounding mode for floating-point arithmetic. This reflects the value of the system `FLT_ROUNDS` macro at interpreter startup time: • `-1`: indeterminable • `0`: toward zero • `1`: to nearest • `2`: toward positive infinity • `3`: toward negative infinity All other values for `FLT_ROUNDS` characterize implementation-defined rounding behavior.

The attribute *`sys.float_info.dig`* needs further explanation. If `s` is any string representing a decimal number with at most `sys.float_info.dig` significant digits, then converting `s` to a float and back again will recover a string representing the same decimal value:

```
>>> import sys
>>> sys.float_info.dig
15
>>> s = '3.14159265358979'    # decimal string with 15 significant digits
>>> format(float(s), '.15g')  # convert to float and back -> same value
'3.14159265358979'
```

But for strings with more than *sys.float_info.dig* significant digits, this isn't always true:

```
>>> s = '9876543211234567'    # 16 significant digits is too many!
>>> format(float(s), '.16g')  # conversion changes value
'9876543211234568'
```

sys.**float_repr_style**

A string indicating how the *repr()* function behaves for floats. If the string has value `'short'` then for a finite float `x`, `repr(x)` aims to produce a short string with the property that `float(repr(x)) == x`. This is the usual behaviour in Python 3.1 and later. Otherwise, `float_repr_style` has value `'legacy'` and `repr(x)` behaves in the same way as it did in versions of Python prior to 3.1.

Added in version 3.1.

sys.**getallocatedblocks**()

Return the number of memory blocks currently allocated by the interpreter, regardless of their size. This function is mainly useful for tracking and debugging memory leaks. Because of the interpreter's internal caches, the result can vary from call to call; you may have to call *_clear_internal_caches()* and *gc.collect()* to get more predictable results.

If a Python build or implementation cannot reasonably compute this information, *getallocatedblocks()* is allowed to return 0 instead.

Added in version 3.4.

sys.**getunicodeinternedsize**()

Return the number of unicode objects that have been interned.

Added in version 3.12.

sys.**getandroidapilevel**()

Return the build-time API level of Android as an integer. This represents the minimum version of Android this build of Python can run on. For runtime version information, see *platform.android_ver()*.

Availability: Android.

Added in version 3.7.

sys.**getdefaultencoding**()

Return the name of the current default string encoding used by the Unicode implementation.

sys.**getdlopenflags**()

Return the current value of the flags that are used for `dlopen()` calls. Symbolic names for the flag values can be found in the *os* module (`RTLD_`*xxx* constants, e.g. *os.RTLD_LAZY*).

Availability: Unix.

sys.**getfilesystemencoding**()

Get the *filesystem encoding*: the encoding used with the *filesystem error handler* to convert between Unicode filenames and bytes filenames. The filesystem error handler is returned from *getfilesystemencodeerrors()*.

For best compatibility, str should be used for filenames in all cases, although representing filenames as bytes is also supported. Functions accepting or returning filenames should support either str or bytes and internally convert to the system's preferred representation.

os.fsencode() and *os.fsdecode()* should be used to ensure that the correct encoding and errors mode are used.

The *filesystem encoding and error handler* are configured at Python startup by the PyConfig_Read() function: see filesystem_encoding and filesystem_errors members of PyConfig.

Changed in version 3.2: *getfilesystemencoding()* result cannot be None anymore.

Changed in version 3.6: Windows is no longer guaranteed to return 'mbcs'. See **PEP 529** and *_enablelegacywindowsfsencoding()* for more information.

Changed in version 3.7: Return 'utf-8' if the *Python UTF-8 Mode* is enabled.

sys.**getfilesystemencodeerrors**()

Get the *filesystem error handler*: the error handler used with the *filesystem encoding* to convert between Unicode filenames and bytes filenames. The filesystem encoding is returned from *getfilesystemencoding()*.

os.fsencode() and *os.fsdecode()* should be used to ensure that the correct encoding and errors mode are used.

The *filesystem encoding and error handler* are configured at Python startup by the PyConfig_Read() function: see filesystem_encoding and filesystem_errors members of PyConfig.

Added in version 3.6.

sys.**get_int_max_str_digits**()

Returns the current value for the *integer string conversion length limitation*. See also *set_int_max_str_digits()*.

Added in version 3.11.

sys.**getrefcount**(*object*)

Return the reference count of the *object*. The count returned is generally one higher than you might expect, because it includes the (temporary) reference as an argument to *getrefcount()*.

Note that the returned value may not actually reflect how many references to the object are actually held. For example, some objects are *immortal* and have a very high refcount that does not reflect the actual number of references. Consequently, do not rely on the returned value to be accurate, other than a value of 0 or 1.

Changed in version 3.12: Immortal objects have very large refcounts that do not match the actual number of references to the object.

sys.**getrecursionlimit**()

Return the current value of the recursion limit, the maximum depth of the Python interpreter stack. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python. It can be set by *setrecursionlimit()*.

sys.**getsizeof**(*object*[, *default*])

Return the size of an object in bytes. The object can be any type of object. All built-in objects will return correct results, but this does not have to hold true for third-party extensions as it is implementation specific.

Only the memory consumption directly attributed to the object is accounted for, not the memory consumption of objects it refers to.

If given, *default* will be returned if the object does not provide means to retrieve the size. Otherwise a *TypeError* will be raised.

getsizeof() calls the object's __sizeof__ method and adds an additional garbage collector overhead if the object is managed by the garbage collector.

See recursive sizeof recipe for an example of using *getsizeof()* recursively to find the size of containers and all their contents.

sys.**getswitchinterval**()

Return the interpreter's "thread switch interval"; see *setswitchinterval()*.

Added in version 3.2.

sys.**_getframe**([*depth*])

Return a frame object from the call stack. If optional integer *depth* is given, return the frame object that many calls below the top of the stack. If that is deeper than the call stack, *ValueError* is raised. The default for *depth* is zero, returning the frame at the top of the call stack.

Raises an *auditing event* sys._getframe with argument frame.

CPython implementation detail: This function should be used for internal and specialized purposes only. It is not guaranteed to exist in all implementations of Python.

sys.**_getframemodulename**([*depth*])

Return the name of a module from the call stack. If optional integer *depth* is given, return the module that many calls below the top of the stack. If that is deeper than the call stack, or if the module is unidentifiable, None is returned. The default for *depth* is zero, returning the module at the top of the call stack.

Raises an *auditing event* sys._getframemodulename with argument depth.

CPython implementation detail: This function should be used for internal and specialized purposes only. It is not guaranteed to exist in all implementations of Python.

sys.**getprofile**()

Get the profiler function as set by *setprofile()*.

sys.**gettrace**()

Get the trace function as set by *settrace()*.

CPython implementation detail: The *gettrace()* function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

sys.**getwindowsversion**()

Return a named tuple describing the Windows version currently running. The named elements are *major*, *minor*, *build*, *platform*, *service_pack*, *service_pack_minor*, *service_pack_major*, *suite_mask*, *product_type* and *platform_version*. *service_pack* contains a string, *platform_version* a 3-tuple and all other values are integers. The components can also be accessed by name, so sys.getwindowsversion()[0] is equivalent to sys.getwindowsversion().major. For compatibility with prior versions, only the first 5 elements are retrievable by indexing.

platform will be 2 (VER_PLATFORM_WIN32_NT).

product_type may be one of the following values:

Constant	Meaning
1 (VER_NT_WORKSTATION)	The system is a workstation.
2 (VER_NT_DOMAIN_CONTROLLER)	The system is a domain controller.
3 (VER_NT_SERVER)	The system is a server, but not a domain controller.

This function wraps the Win32 `GetVersionEx()` function; see the Microsoft documentation on `OSVERSIONINFOEX()` for more information about these fields.

platform_version returns the major version, minor version and build number of the current operating system, rather than the version that is being emulated for the process. It is intended for use in logging rather than for feature detection.

> **Note**
>
> *platform_version* derives the version from kernel32.dll which can be of a different version than the OS version. Please use `platform` module for achieving accurate OS version.

Availability: Windows.

Changed in version 3.2: Changed to a named tuple and added *service_pack_minor*, *service_pack_major*, *suite_mask*, and *product_type*.

Changed in version 3.6: Added *platform_version*

`sys.`**`get_asyncgen_hooks`**`()`

Returns an *asyncgen_hooks* object, which is similar to a `namedtuple` of the form `(firstiter, finalizer)`, where *firstiter* and *finalizer* are expected to be either `None` or functions which take an *asynchronous generator iterator* as an argument, and are used to schedule finalization of an asynchronous generator by an event loop.

Added in version 3.6: See **PEP 525** for more details.

> **Note**
>
> This function has been added on a provisional basis (see **PEP 411** for details.)

`sys.`**`get_coroutine_origin_tracking_depth`**`()`

Get the current coroutine origin tracking depth, as set by `set_coroutine_origin_tracking_depth()`.

Added in version 3.7.

> **Note**
>
> This function has been added on a provisional basis (see **PEP 411** for details.) Use it only for debugging purposes.

`sys.`**`hash_info`**

A *named tuple* giving parameters of the numeric hash implementation. For more details about hashing of numeric types, see *Hashing of numeric types*.

`hash_info.`**`width`**

The width in bits used for hash values

`hash_info.`**`modulus`**

The prime modulus P used for numeric hash scheme

`hash_info.`**`inf`**

The hash value returned for a positive infinity

hash_info.**nan**

(This attribute is no longer used)

hash_info.**imag**

The multiplier used for the imaginary part of a complex number

hash_info.**algorithm**

The name of the algorithm for hashing of str, bytes, and memoryview

hash_info.**hash_bits**

The internal output size of the hash algorithm

hash_info.**seed_bits**

The size of the seed key of the hash algorithm

Added in version 3.2.

Changed in version 3.4: Added *algorithm*, *hash_bits* and *seed_bits*

sys.**hexversion**

The version number encoded as a single integer. This is guaranteed to increase with each version, including proper support for non-production releases. For example, to test that the Python interpreter is at least version 1.5.2, use:

```
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

This is called `hexversion` since it only really looks meaningful when viewed as the result of passing it to the built-in *`hex()`* function. The *named tuple* *`sys.version_info`* may be used for a more human-friendly encoding of the same information.

More details of `hexversion` can be found at apiabiversion.

sys.**implementation**

An object containing information about the implementation of the currently running Python interpreter. The following attributes are required to exist in all Python implementations.

name is the implementation's identifier, e.g. `'cpython'`. The actual string is defined by the Python implementation, but it is guaranteed to be lower case.

version is a named tuple, in the same format as *`sys.version_info`*. It represents the version of the Python *implementation*. This has a distinct meaning from the specific version of the Python *language* to which the currently running interpreter conforms, which `sys.version_info` represents. For example, for PyPy 1.8 `sys.implementation.version` might be `sys.version_info(1, 8, 0, 'final', 0)`, whereas `sys.version_info` would be `sys.version_info(2, 7, 2, 'final', 0)`. For CPython they are the same value, since it is the reference implementation.

hexversion is the implementation version in hexadecimal format, like *`sys.hexversion`*.

cache_tag is the tag used by the import machinery in the filenames of cached modules. By convention, it would be a composite of the implementation's name and version, like `'cpython-33'`. However, a Python implementation may use some other value if appropriate. If `cache_tag` is set to `None`, it indicates that module caching should be disabled.

`sys.implementation` may contain additional attributes specific to the Python implementation. These non-standard attributes must start with an underscore, and are not described here. Regardless of its contents, *`sys.`*

implementation will not change during a run of the interpreter, nor between implementation versions. (It may change between Python language versions, however.) See **PEP 421** for more information.

Added in version 3.3.

> **Note**
>
> The addition of new required attributes must go through the normal PEP process. See **PEP 421** for more information.

sys.**int_info**

A *named tuple* that holds information about Python's internal representation of integers. The attributes are read only.

int_info.**bits_per_digit**

The number of bits held in each digit. Python integers are stored internally in base `2**int_info.bits_per_digit`.

int_info.**sizeof_digit**

The size in bytes of the C type used to represent a digit.

int_info.**default_max_str_digits**

The default value for *sys.get_int_max_str_digits()* when it is not otherwise explicitly configured.

int_info.**str_digits_check_threshold**

The minimum non-zero value for *sys.set_int_max_str_digits()*, PYTHONINTMAXSTRDIGITS, or `-X int_max_str_digits`.

Added in version 3.1.

Changed in version 3.11: Added *default_max_str_digits* and *str_digits_check_threshold*.

sys.**__interactivehook__**

When this attribute exists, its value is automatically called (with no arguments) when the interpreter is launched in interactive mode. This is done after the PYTHONSTARTUP file is read, so that you can set this hook there. The *site* module *sets this*.

Raises an *auditing event* `cpython.run_interactivehook` with the hook object as the argument when the hook is called on startup.

Added in version 3.4.

sys.**intern**(*string*)

Enter *string* in the table of "interned" strings and return the interned string – which is *string* itself or a copy. Interning strings is useful to gain a little performance on dictionary lookup – if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys.

Interned strings are not *immortal*; you must keep a reference to the return value of *intern()* around to benefit from it.

sys.**_is_gil_enabled**()

Return *True* if the *GIL* is enabled and *False* if it is disabled.

Added in version 3.13.

sys.**is_finalizing**()

Return *True* if the main Python interpreter is *shutting down*. Return *False* otherwise.

See also the *PythonFinalizationError* exception.

Added in version 3.5.

sys.**last_exc**

This variable is not always defined; it is set to the exception instance when an exception is not handled and the interpreter prints an error message and a stack traceback. Its intended use is to allow an interactive user to import a debugger module and engage in post-mortem debugging without having to re-execute the command that caused the error. (Typical use is `import pdb; pdb.pm()` to enter the post-mortem debugger; see *pdb* module for more information.)

Added in version 3.12.

sys.**_is_interned**(*string*)

Return *True* if the given string is "interned", *False* otherwise.

Added in version 3.13.

CPython implementation detail: It is not guaranteed to exist in all implementations of Python.

sys.**last_type**

sys.**last_value**

sys.**last_traceback**

These three variables are deprecated; use *sys.last_exc* instead. They hold the legacy representation of `sys.last_exc`, as returned from *exc_info()* above.

sys.**maxsize**

An integer giving the maximum value a variable of type `Py_ssize_t` can take. It's usually `2**31 - 1` on a 32-bit platform and `2**63 - 1` on a 64-bit platform.

sys.**maxunicode**

An integer giving the value of the largest Unicode code point, i.e. `1114111` (`0x10FFFF` in hexadecimal).

Changed in version 3.3: Before **PEP 393**, `sys.maxunicode` used to be either `0xFFFF` or `0x10FFFF`, depending on the configuration option that specified whether Unicode characters were stored as UCS-2 or UCS-4.

sys.**meta_path**

A list of *meta path finder* objects that have their *find_spec()* methods called to see if one of the objects can find the module to be imported. By default, it holds entries that implement Python's default import semantics. The *find_spec()* method is called with at least the absolute name of the module being imported. If the module to be imported is contained in a package, then the parent package's `__path__` attribute is passed in as a second argument. The method returns a *module spec*, or `None` if the module cannot be found.

See also

importlib.abc.MetaPathFinder
The abstract base class defining the interface of finder objects on *meta_path*.

importlib.machinery.ModuleSpec
The concrete class which *find_spec()* should return instances of.

Changed in version 3.4: *Module specs* were introduced in Python 3.4, by **PEP 451**.

Changed in version 3.12: Removed the fallback that looked for a `find_module()` method if a *meta_path* entry didn't have a *find_spec()* method.

sys.**modules**

This is a dictionary that maps module names to modules which have already been loaded. This can be manipulated to force reloading of modules and other tricks. However, replacing the dictionary will not necessarily work as expected and deleting essential items from the dictionary may cause Python to fail. If you want to iterate over this global dictionary always use `sys.modules.copy()` or `tuple(sys.modules)` to avoid exceptions as its size may change during iteration as a side effect of code or activity in other threads.

sys.**orig_argv**

The list of the original command line arguments passed to the Python executable.

The elements of *sys.orig_argv* are the arguments to the Python interpreter, while the elements of *sys.argv* are the arguments to the user's program. Arguments consumed by the interpreter itself will be present in *sys.orig_argv* and missing from *sys.argv*.

Added in version 3.10.

sys.**path**

A list of strings that specifies the search path for modules. Initialized from the environment variable `PYTHONPATH`, plus an installation-dependent default.

By default, as initialized upon program startup, a potentially unsafe path is prepended to *sys.path* (*before* the entries inserted as a result of `PYTHONPATH`):

- `python -m module` command line: prepend the current working directory.
- `python script.py` command line: prepend the script's directory. If it's a symbolic link, resolve symbolic links.
- `python -c code` and `python` (REPL) command lines: prepend an empty string, which means the current working directory.

To not prepend this potentially unsafe path, use the `-P` command line option or the `PYTHONSAFEPATH` environment variable.

A program is free to modify this list for its own purposes. Only strings should be added to *sys.path*; all other data types are ignored during import.

See also

- Module *site* This describes how to use .pth files to extend *sys.path*.

sys.**path_hooks**

A list of callables that take a path argument to try to create a *finder* for the path. If a finder can be created, it is to be returned by the callable, else raise *ImportError*.

Originally specified in **PEP 302**.

sys.**path_importer_cache**

A dictionary acting as a cache for *finder* objects. The keys are paths that have been passed to *sys.path_hooks* and the values are the finders that are found. If a path is a valid file system path but no finder is found on *sys.path_hooks* then `None` is stored.

Originally specified in **PEP 302**.

sys.**platform**

A string containing a platform identifier. Known values are:

System	`platform` value
AIX	`'aix'`
Android	`'android'`
Emscripten	`'emscripten'`
iOS	`'ios'`
Linux	`'linux'`
macOS	`'darwin'`
Windows	`'win32'`
Windows/Cygwin	`'cygwin'`
WASI	`'wasi'`

On Unix systems not listed in the table, the value is the lowercased OS name as returned by `uname -s`, with the first part of the version as returned by `uname -r` appended, e.g. `'sunos5'` or `'freebsd8'`, *at the time when Python was built*. Unless you want to test for a specific system version, it is therefore recommended to use the following idiom:

```
if sys.platform.startswith('freebsd'):
    # FreeBSD-specific code here...
```

Changed in version 3.3: On Linux, *sys.platform* doesn't contain the major version anymore. It is always `'linux'`, instead of `'linux2'` or `'linux3'`.

Changed in version 3.8: On AIX, *sys.platform* doesn't contain the major version anymore. It is always `'aix'`, instead of `'aix5'` or `'aix7'`.

Changed in version 3.13: On Android, *sys.platform* now returns `'android'` rather than `'linux'`.

See also

os.name has a coarser granularity. *os.uname()* gives system-dependent version information.

The *platform* module provides detailed checks for the system's identity.

`sys.`**`platlibdir`**

Name of the platform-specific library directory. It is used to build the path of standard library and the paths of installed extension modules.

It is equal to `"lib"` on most platforms. On Fedora and SuSE, it is equal to `"lib64"` on 64-bit platforms which gives the following `sys.path` paths (where `X.Y` is the Python `major.minor` version):

- `/usr/lib64/pythonX.Y/`: Standard library (like `os.py` of the *os* module)
- `/usr/lib64/pythonX.Y/lib-dynload/`: C extension modules of the standard library (like the *errno* module, the exact filename is platform specific)
- `/usr/lib/pythonX.Y/site-packages/` (always use `lib`, not *sys.platlibdir*): Third-party modules
- `/usr/lib64/pythonX.Y/site-packages/`: C extension modules of third-party packages

Added in version 3.9.

`sys.`**`prefix`**

A string giving the site-specific directory prefix where the platform independent Python files are installed; on Unix, the default is `/usr/local`. This can be set at build time with the `--prefix` argument to the **`configure`** script. See *Installation paths* for derived paths.

> **Note**
>
> If a *virtual environment* is in effect, this value will be changed in `site.py` to point to the virtual environment. The value for the Python installation will still be available, via `base_prefix`.

`sys.`**`ps1`**

`sys.`**`ps2`**

Strings specifying the primary and secondary prompt of the interpreter. These are only defined if the interpreter is in interactive mode. Their initial values in this case are `'>>> '` and `'... '`. If a non-string object is assigned to either variable, its `str()` is re-evaluated each time the interpreter prepares to read a new interactive command; this can be used to implement a dynamic prompt.

`sys.`**`setdlopenflags`**(*n*)

Set the flags used by the interpreter for `dlopen()` calls, such as when the interpreter loads extension modules. Among other things, this will enable a lazy resolving of symbols when importing a module, if called as `sys.setdlopenflags(0)`. To share symbols across extension modules, call as `sys.setdlopenflags(os.RTLD_GLOBAL)`. Symbolic names for the flag values can be found in the `os` module (`RTLD_xxx` constants, e.g. `os.RTLD_LAZY`).

Availability: Unix.

`sys.`**`set_int_max_str_digits`**(*maxdigits*)

Set the *integer string conversion length limitation* used by this interpreter. See also `get_int_max_str_digits()`.

Added in version 3.11.

`sys.`**`setprofile`**(*profilefunc*)

Set the system's profile function, which allows you to implement a Python source code profiler in Python. See chapter *The Python Profilers* for more information on the Python profiler. The system's profile function is called similarly to the system's trace function (see `settrace()`), but it is called with different events, for example it isn't called for each executed line of code (only on call and return, but the return event is reported even when an exception has been set). The function is thread-specific, but there is no way for the profiler to know about context switches between threads, so it does not make sense to use this in the presence of multiple threads. Also, its return value is not used, so it can simply return `None`. Error in the profile function will cause itself unset.

> **Note**
>
> The same tracing mechanism is used for `setprofile()` as `settrace()`. To trace calls with `setprofile()` inside a tracing function (e.g. in a debugger breakpoint), see `call_tracing()`.

Profile functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: `'call'`, `'return'`, `'c_call'`, `'c_return'`, or `'c_exception'`. *arg* depends on the event type.

The events have the following meaning:

`'call'`
A function is called (or some other code block entered). The profile function is called; *arg* is `None`.

`'return'`
A function (or other code block) is about to return. The profile function is called; *arg* is the value that will be returned, or `None` if the event is caused by an exception being raised.

'c_call'
A C function is about to be called. This may be an extension function or a built-in. *arg* is the C function object.

'c_return'
A C function has returned. *arg* is the C function object.

'c_exception'
A C function has raised an exception. *arg* is the C function object.

Raises an *auditing event* `sys.setprofile` with no arguments.

`sys.`**`setrecursionlimit`**(*limit*)

Set the maximum depth of the Python interpreter stack to *limit*. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.

The highest possible limit is platform-dependent. A user may need to set the limit higher when they have a program that requires deep recursion and a platform that supports a higher limit. This should be done with care, because a too-high limit can lead to a crash.

If the new limit is too low at the current recursion depth, a *RecursionError* exception is raised.

Changed in version 3.5.1: A *RecursionError* exception is now raised if the new limit is too low at the current recursion depth.

`sys.`**`setswitchinterval`**(*interval*)

Set the interpreter's thread switch interval (in seconds). This floating-point value determines the ideal duration of the "timeslices" allocated to concurrently running Python threads. Please note that the actual value can be higher, especially if long-running internal functions or methods are used. Also, which thread becomes scheduled at the end of the interval is the operating system's decision. The interpreter doesn't have its own scheduler.

Added in version 3.2.

`sys.`**`settrace`**(*tracefunc*)

Set the system's trace function, which allows you to implement a Python source code debugger in Python. The function is thread-specific; for a debugger to support multiple threads, it must register a trace function using *settrace()* for each thread being debugged or use *threading.settrace()*.

Trace functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: `'call'`, `'line'`, `'return'`, `'exception'` or `'opcode'`. *arg* depends on the event type.

The trace function is invoked (with *event* set to `'call'`) whenever a new local scope is entered; it should return a reference to a local trace function to be used for the new scope, or `None` if the scope shouldn't be traced.

The local trace function should return a reference to itself, or to another function which would then be used as the local trace function for the scope.

If there is any error occurred in the trace function, it will be unset, just like `settrace(None)` is called.

> **Note**
>
> Tracing is disabled while calling the trace function (e.g. a function set by `settrace()`). For recursive tracing see *call_tracing()*.

The events have the following meaning:

'call'
A function is called (or some other code block entered). The global trace function is called; *arg* is `None`; the return value specifies the local trace function.

'line'
The interpreter is about to execute a new line of code or re-execute the condition of a loop. The local trace function is called; *arg* is None; the return value specifies the new local trace function. See Objects/lnotab_notes.txt for a detailed explanation of how this works. Per-line events may be disabled for a frame by setting f_trace_lines to *False* on that frame.

'return'
A function (or other code block) is about to return. The local trace function is called; *arg* is the value that will be returned, or None if the event is caused by an exception being raised. The trace function's return value is ignored.

'exception'
An exception has occurred. The local trace function is called; *arg* is a tuple (exception, value, traceback); the return value specifies the new local trace function.

'opcode'
The interpreter is about to execute a new opcode (see *dis* for opcode details). The local trace function is called; *arg* is None; the return value specifies the new local trace function. Per-opcode events are not emitted by default: they must be explicitly requested by setting f_trace_opcodes to *True* on the frame.

Note that as an exception is propagated down the chain of callers, an 'exception' event is generated at each level.

For more fine-grained usage, it's possible to set a trace function by assigning frame.f_trace = tracefunc explicitly, rather than relying on it being set indirectly via the return value from an already installed trace function. This is also required for activating the trace function on the current frame, which *settrace()* doesn't do. Note that in order for this to work, a global tracing function must have been installed with *settrace()* in order to enable the runtime tracing machinery, but it doesn't need to be the same tracing function (e.g. it could be a low overhead tracing function that simply returns None to disable itself immediately on each frame).

For more information on code and frame objects, refer to types.

Raises an *auditing event* sys.settrace with no arguments.

CPython implementation detail: The *settrace()* function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

Changed in version 3.7: 'opcode' event type added; f_trace_lines and f_trace_opcodes attributes added to frames

sys.**set_asyncgen_hooks**(*[firstiter] [, finalizer]*)

Accepts two optional keyword arguments which are callables that accept an *asynchronous generator iterator* as an argument. The *firstiter* callable will be called when an asynchronous generator is iterated for the first time. The *finalizer* will be called when an asynchronous generator is about to be garbage collected.

Raises an *auditing event* sys.set_asyncgen_hooks_firstiter with no arguments.

Raises an *auditing event* sys.set_asyncgen_hooks_finalizer with no arguments.

Two auditing events are raised because the underlying API consists of two calls, each of which must raise its own event.

Added in version 3.6: See **PEP 525** for more details, and for a reference example of a *finalizer* method see the implementation of asyncio.Loop.shutdown_asyncgens in Lib/asyncio/base_events.py

Note

This function has been added on a provisional basis (see **PEP 411** for details.)

sys.**set_coroutine_origin_tracking_depth**(*depth*)

Allows enabling or disabling coroutine origin tracking. When enabled, the `cr_origin` attribute on coroutine objects will contain a tuple of (filename, line number, function name) tuples describing the traceback where the coroutine object was created, with the most recent call first. When disabled, `cr_origin` will be `None`.

To enable, pass a *depth* value greater than zero; this sets the number of frames whose information will be captured. To disable, pass set *depth* to zero.

This setting is thread-specific.

Added in version 3.7.

Note

This function has been added on a provisional basis (see **PEP 411** for details.) Use it only for debugging purposes.

sys.**activate_stack_trampoline**(*backend*, */*)

Activate the stack profiler trampoline *backend*. The only supported backend is `"perf"`.

Availability: Linux.

Added in version 3.12.

See also

- perf_profiling
- https://perf.wiki.kernel.org

sys.**deactivate_stack_trampoline**()

Deactivate the current stack profiler trampoline backend.

If no stack profiler is activated, this function has no effect.

Availability: Linux.

Added in version 3.12.

sys.**is_stack_trampoline_active**()

Return `True` if a stack profiler trampoline is active.

Availability: Linux.

Added in version 3.12.

sys.**_enablelegacywindowsfsencoding**()

Changes the *filesystem encoding and error handler* to 'mbcs' and 'replace' respectively, for consistency with versions of Python prior to 3.6.

This is equivalent to defining the `PYTHONLEGACYWINDOWSFSENCODING` environment variable before launching Python.

See also *`sys.getfilesystemencoding()`* and *`sys.getfilesystemencodeerrors()`*.

Availability: Windows.

> **Note**
>
> Changing the filesystem encoding after Python startup is risky because the old fsencoding or paths encoded by the old fsencoding may be cached somewhere. Use `PYTHONLEGACYWINDOWSFSENCODING` instead.

Added in version 3.6: See **PEP 529** for more details.

Deprecated since version 3.13, will be removed in version 3.16: Use `PYTHONLEGACYWINDOWSFSENCODING` instead.

`sys.`**`stdin`**

`sys.`**`stdout`**

`sys.`**`stderr`**

File objects used by the interpreter for standard input, output and errors:

- `stdin` is used for all interactive input (including calls to *`input()`*);
- `stdout` is used for the output of *`print()`* and *expression* statements and for the prompts of *`input()`*;
- The interpreter's own prompts and its error messages go to `stderr`.

These streams are regular *text files* like those returned by the *`open()`* function. Their parameters are chosen as follows:

- The encoding and error handling are is initialized from `PyConfig.stdio_encoding` and `PyConfig.stdio_errors`.

  On Windows, UTF-8 is used for the console device. Non-character devices such as disk files and pipes use the system locale encoding (i.e. the ANSI codepage). Non-console character devices such as NUL (i.e. where `isatty()` returns `True`) use the value of the console input and output codepages at startup, respectively for stdin and stdout/stderr. This defaults to the system *locale encoding* if the process is not initially attached to a console.

  The special behaviour of the console can be overridden by setting the environment variable PYTHONLEGACYWINDOWSSTDIO before starting Python. In that case, the console codepages are used as for any other character device.

  Under all platforms, you can override the character encoding by setting the `PYTHONIOENCODING` environment variable before starting Python or by using the new `-X utf8` command line option and `PYTHONUTF8` environment variable. However, for the Windows console, this only applies when `PYTHONLEGACYWINDOWSSTDIO` is also set.

- When interactive, the `stdout` stream is line-buffered. Otherwise, it is block-buffered like regular text files. The `stderr` stream is line-buffered in both cases. You can make both streams unbuffered by passing the `-u` command-line option or setting the `PYTHONUNBUFFERED` environment variable.

Changed in version 3.9: Non-interactive `stderr` is now line-buffered instead of fully buffered.

> **Note**
>
> To write or read binary data from/to the standard streams, use the underlying binary *`buffer`* object. For example, to write bytes to *`stdout`*, use `sys.stdout.buffer.write(b'abc')`.
>
> However, if you are writing a library (and do not control in which context its code will be executed), be aware that the standard streams may be replaced with file-like objects like *`io.StringIO`* which do not support the `buffer` attribute.

`sys.`**`__stdin__`**

sys.**__stdout__**

sys.**__stderr__**

These objects contain the original values of `stdin`, `stderr` and `stdout` at the start of the program. They are used during finalization, and could be useful to print to the actual standard stream no matter if the `sys.std*` object has been redirected.

It can also be used to restore the actual files to known working file objects in case they have been overwritten with a broken object. However, the preferred way to do this is to explicitly save the previous stream before replacing it, and restore the saved object.

> **Note**
>
> Under some conditions `stdin`, `stdout` and `stderr` as well as the original values `__stdin__`, `__stdout__` and `__stderr__` can be `None`. It is usually the case for Windows GUI apps that aren't connected to a console and Python apps started with **pythonw**.

sys.**stdlib_module_names**

A frozenset of strings containing the names of standard library modules.

It is the same on all platforms. Modules which are not available on some platforms and modules disabled at Python build are also listed. All module kinds are listed: pure Python, built-in, frozen and extension modules. Test modules are excluded.

For packages, only the main package is listed: sub-packages and sub-modules are not listed. For example, the `email` package is listed, but the `email.mime` sub-package and the `email.message` sub-module are not listed.

See also the *`sys.builtin_module_names`* list.

Added in version 3.10.

sys.**thread_info**

A *named tuple* holding information about the thread implementation.

thread_info.**name**

The name of the thread implementation:

- `"nt"`: Windows threads
- `"pthread"`: POSIX threads
- `"pthread-stubs"`: stub POSIX threads (on WebAssembly platforms without threading support)
- `"solaris"`: Solaris threads

thread_info.**lock**

The name of the lock implementation:

- `"semaphore"`: a lock uses a semaphore
- `"mutex+cond"`: a lock uses a mutex and a condition variable
- `None` if this information is unknown

thread_info.**version**

The name and version of the thread library. It is a string, or `None` if this information is unknown.

Added in version 3.3.

sys.**tracebacklimit**

When this variable is set to an integer value, it determines the maximum number of levels of traceback information printed when an unhandled exception occurs. The default is `1000`. When set to `0` or less, all traceback information is suppressed and only the exception type and value are printed.

sys.**unraisablehook**(*unraisable*, /)

Handle an unraisable exception.

Called when an exception has occurred but there is no way for Python to handle it. For example, when a destructor raises an exception or during garbage collection (*gc.collect()*).

The *unraisable* argument has the following attributes:

- `exc_type`: Exception type.
- `exc_value`: Exception value, can be `None`.
- `exc_traceback`: Exception traceback, can be `None`.
- `err_msg`: Error message, can be `None`.
- `object`: Object causing the exception, can be `None`.

The default hook formats `err_msg` and `object` as: `f'{err_msg}: {object!r}'`; use "Exception ignored in" error message if `err_msg` is `None`.

sys.unraisablehook() can be overridden to control how unraisable exceptions are handled.

See also

excepthook() which handles uncaught exceptions.

Warning

Storing `exc_value` using a custom hook can create a reference cycle. It should be cleared explicitly to break the reference cycle when the exception is no longer needed.

Storing `object` using a custom hook can resurrect it if it is set to an object which is being finalized. Avoid storing `object` after the custom hook completes to avoid resurrecting objects.

Raise an auditing event `sys.unraisablehook` with arguments *hook*, *unraisable* when an exception that cannot be handled occurs. The *unraisable* object is the same as what will be passed to the hook. If no hook has been set, *hook* may be `None`.

Added in version 3.8.

sys.**version**

A string containing the version number of the Python interpreter plus additional information on the build number and compiler used. This string is displayed when the interactive interpreter is started. Do not extract version information out of it, rather, use *version_info* and the functions provided by the *platform* module.

sys.**api_version**

The C API version for this interpreter. Programmers may find this useful when debugging version conflicts between Python and extension modules.

sys.**version_info**

A tuple containing the five components of the version number: *major*, *minor*, *micro*, *releaselevel*, and *serial*. All values except *releaselevel* are integers; the release level is `'alpha'`, `'beta'`, `'candidate'`, or `'final'`. The `version_info` value corresponding to the Python version 2.0 is `(2, 0, 0, 'final', 0)`. The components can also be accessed by name, so `sys.version_info[0]` is equivalent to `sys.version_info.major` and so on.

Changed in version 3.1: Added named component attributes.

sys.**warnoptions**

This is an implementation detail of the warnings framework; do not modify this value. Refer to the *warnings* module for more information on the warnings framework.

sys.**winver**

The version number used to form registry keys on Windows platforms. This is stored as string resource 1000 in the Python DLL. The value is normally the major and minor versions of the running Python interpreter. It is provided in the *sys* module for informational purposes; modifying this value has no effect on the registry keys used by Python.

Availability: Windows.

sys.**monitoring**

Namespace containing functions and constants for register callbacks and controlling monitoring events. See *sys.monitoring* for details.

sys.**_xoptions**

A dictionary of the various implementation-specific flags passed through the `-X` command-line option. Option names are either mapped to their values, if given explicitly, or to *True*. Example:

```
$ ./python -Xa=b -Xc
Python 3.2a3+ (py3k, Oct 16 2010, 20:14:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys._xoptions
{'a': 'b', 'c': True}
```

CPython implementation detail: This is a CPython-specific way of accessing options passed through `-X`. Other implementations may export them through other means, or not at all.

Added in version 3.2.

Citations

29.2 `sys.monitoring` — Execution event monitoring

Added in version 3.12.

Note

sys.monitoring is a namespace within the *sys* module, not an independent module, so there is no need to `import sys.monitoring`, simply `import sys` and then use `sys.monitoring`.

This namespace provides access to the functions and constants necessary to activate and control event monitoring.

As programs execute, events occur that might be of interest to tools that monitor execution. The *sys.monitoring* namespace provides means to receive callbacks when events of interest occur.

The monitoring API consists of three components:

- *Tool identifiers*
- *Events*
- *Callbacks*

29.2.1 Tool identifiers

A tool identifier is an integer and the associated name. Tool identifiers are used to discourage tools from interfering with each other and to allow multiple tools to operate at the same time. Currently tools are completely independent and cannot be used to monitor each other. This restriction may be lifted in the future.

Before registering or activating events, a tool should choose an identifier. Identifiers are integers in the range 0 to 5 inclusive.

Registering and using tools

sys.monitoring.**use_tool_id**(*tool_id:* int, *name:* str, /) → *None*

Must be called before *tool_id* can be used. *tool_id* must be in the range 0 to 5 inclusive. Raises a *ValueError* if *tool_id* is in use.

sys.monitoring.**free_tool_id**(*tool_id:* int, /) → *None*

Should be called once a tool no longer requires *tool_id*.

Note

free_tool_id() will not disable global or local events associated with *tool_id*, nor will it unregister any callback functions. This function is only intended to be used to notify the VM that the particular *tool_id* is no longer in use.

sys.monitoring.**get_tool**(*tool_id:* int, /) → *str* | *None*

Returns the name of the tool if *tool_id* is in use, otherwise it returns `None`. *tool_id* must be in the range 0 to 5 inclusive.

All IDs are treated the same by the VM with regard to events, but the following IDs are pre-defined to make co-operation of tools easier:

```
sys.monitoring.DEBUGGER_ID = 0
sys.monitoring.COVERAGE_ID = 1
sys.monitoring.PROFILER_ID = 2
sys.monitoring.OPTIMIZER_ID = 5
```

29.2.2 Events

The following events are supported:

`sys.monitoring.events.`**`BRANCH`**
A conditional branch is taken (or not).

`sys.monitoring.events.`**`CALL`**
A call in Python code (event occurs before the call).

`sys.monitoring.events.`**`C_RAISE`**
An exception raised from any callable, except for Python functions (event occurs after the exit).

`sys.monitoring.events.`**`C_RETURN`**
Return from any callable, except for Python functions (event occurs after the return).

`sys.monitoring.events.`**`EXCEPTION_HANDLED`**
An exception is handled.

`sys.monitoring.events.`**`INSTRUCTION`**
A VM instruction is about to be executed.

`sys.monitoring.events.`**`JUMP`**
An unconditional jump in the control flow graph is made.

`sys.monitoring.events.`**`LINE`**
An instruction is about to be executed that has a different line number from the preceding instruction.

`sys.monitoring.events.`**`PY_RESUME`**
Resumption of a Python function (for generator and coroutine functions), except for `throw()` calls.

`sys.monitoring.events.`**`PY_RETURN`**
Return from a Python function (occurs immediately before the return, the callee's frame will be on the stack).

`sys.monitoring.events.`**`PY_START`**
Start of a Python function (occurs immediately after the call, the callee's frame will be on the stack)

`sys.monitoring.events.`**`PY_THROW`**
A Python function is resumed by a `throw()` call.

`sys.monitoring.events.`**`PY_UNWIND`**
Exit from a Python function during exception unwinding.

`sys.monitoring.events.`**`PY_YIELD`**
Yield from a Python function (occurs immediately before the yield, the callee's frame will be on the stack).

`sys.monitoring.events.`**`RAISE`**
An exception is raised, except those that cause a *`STOP_ITERATION`* event.

`sys.monitoring.events.`**`RERAISE`**
An exception is re-raised, for example at the end of a `finally` block.

`sys.monitoring.events.`**`STOP_ITERATION`**
An artificial *`StopIteration`* is raised; see *the STOP_ITERATION event*.

More events may be added in the future.

These events are attributes of the `sys.monitoring.events` namespace. Each event is represented as a power-of-2 integer constant. To define a set of events, simply bitwise or the individual events together. For example, to specify both *`PY_RETURN`* and *`PY_START`* events, use the expression `PY_RETURN | PY_START`.

sys.monitoring.events.**NO_EVENTS**

An alias for 0 so users can do explicit comparisons like:

```
if get_events(DEBUGGER_ID) == NO_EVENTS:
    ...
```

Events are divided into three groups:

Local events

Local events are associated with normal execution of the program and happen at clearly defined locations. All local events can be disabled. The local events are:

- *PY_START*
- *PY_RESUME*
- *PY_RETURN*
- *PY_YIELD*
- *CALL*
- *LINE*
- *INSTRUCTION*
- *JUMP*
- *BRANCH*
- *STOP_ITERATION*

Ancillary events

Ancillary events can be monitored like other events, but are controlled by another event:

- *C_RAISE*
- *C_RETURN*

The *C_RETURN* and *C_RAISE* events are controlled by the *CALL* event. *C_RETURN* and *C_RAISE* events will only be seen if the corresponding *CALL* event is being monitored.

Other events

Other events are not necessarily tied to a specific location in the program and cannot be individually disabled.

The other events that can be monitored are:

- *PY_THROW*
- *PY_UNWIND*
- *RAISE*
- *EXCEPTION_HANDLED*

The STOP_ITERATION event

PEP 380 specifies that a *StopIteration* exception is raised when returning a value from a generator or coroutine. However, this is a very inefficient way to return a value, so some Python implementations, notably CPython 3.12+, do not raise an exception unless it would be visible to other code.

To allow tools to monitor for real exceptions without slowing down generators and coroutines, the *STOP_ITERATION* event is provided. *STOP_ITERATION* can be locally disabled, unlike *RAISE*.

29.2.3 Turning events on and off

In order to monitor an event, it must be turned on and a corresponding callback must be registered. Events can be turned on or off by setting the events either globally or for a particular code object.

Setting events globally

Events can be controlled globally by modifying the set of events being monitored.

`sys.monitoring.`**`get_events`**(*tool_id:* int, `/`) → *int*

Returns the `int` representing all the active events.

`sys.monitoring.`**`set_events`**(*tool_id:* int, *event_set:* int, `/`) → *None*

Activates all events which are set in *event_set*. Raises a *ValueError* if *tool_id* is not in use.

No events are active by default.

Per code object events

Events can also be controlled on a per code object basis. The functions defined below which accept a *types.CodeType* should be prepared to accept a look-alike object from functions which are not defined in Python (see monitoring).

`sys.monitoring.`**`get_local_events`**(*tool_id:* int, *code:* CodeType, `/`) → *int*

Returns all the local events for *code*

`sys.monitoring.`**`set_local_events`**(*tool_id:* int, *code:* CodeType, *event_set:* int, `/`) → *None*

Activates all the local events for *code* which are set in *event_set*. Raises a *ValueError* if *tool_id* is not in use.

Local events add to global events, but do not mask them. In other words, all global events will trigger for a code object, regardless of the local events.

Disabling events

`sys.monitoring.`**`DISABLE`**

A special value that can be returned from a callback function to disable events for the current code location.

Local events can be disabled for a specific code location by returning *sys.monitoring.DISABLE* from a callback function. This does not change which events are set, or any other code locations for the same event.

Disabling events for specific locations is very important for high performance monitoring. For example, a program can be run under a debugger with no overhead if the debugger disables all monitoring except for a few breakpoints.

`sys.monitoring.`**`restart_events`**() → *None*

Enable all the events that were disabled by *sys.monitoring.DISABLE* for all tools.

29.2.4 Registering callback functions

To register a callable for events call

sys.monitoring.**register_callback**(*tool_id:* int, *event:* int, *func:* Callable | None, /) → *Callable* | *None*

Registers the callable *func* for the *event* with the given *tool_id*

If another callback was registered for the given *tool_id* and *event*, it is unregistered and returned. Otherwise *register_callback()* returns None.

Functions can be unregistered by calling `sys.monitoring.register_callback(tool_id, event, None)`.

Callback functions can be registered and unregistered at any time.

Registering or unregistering a callback function will generate a *sys.audit()* event.

Callback function arguments

sys.monitoring.**MISSING**

A special value that is passed to a callback function to indicate that there are no arguments to the call.

When an active event occurs, the registered callback function is called. Different events will provide the callback function with different arguments, as follows:

- *PY_START* and *PY_RESUME*:

```
func(code: CodeType, instruction_offset: int) -> DISABLE | Any
```

- *PY_RETURN* and *PY_YIELD*:

```
func(code: CodeType, instruction_offset: int, retval: object) -> DISABLE | Any
```

- *CALL*, *C_RAISE* and *C_RETURN*:

```
func(code: CodeType, instruction_offset: int, callable: object, arg0: object |␣
↪MISSING) -> DISABLE | Any
```

If there are no arguments, *arg0* is set to *sys.monitoring.MISSING*.

- *RAISE*, *RERAISE*, *EXCEPTION_HANDLED*, *PY_UNWIND*, *PY_THROW* and *STOP_ITERATION*:

```
func(code: CodeType, instruction_offset: int, exception: BaseException) ->␣
↪DISABLE | Any
```

- *LINE*:

```
func(code: CodeType, line_number: int) -> DISABLE | Any
```

- *BRANCH* and *JUMP*:

```
func(code: CodeType, instruction_offset: int, destination_offset: int) -> DISABLE␣
↪| Any
```

Note that the *destination_offset* is where the code will next execute. For an untaken branch this will be the offset of the instruction following the branch.

- *INSTRUCTION*:

```
func(code: CodeType, instruction_offset: int) -> DISABLE | Any
```

29.3 sysconfig — Provide access to Python's configuration information

Added in version 3.2.

Source code: Lib/sysconfig

The *sysconfig* module provides access to Python's configuration information like the list of installation paths and the configuration variables relevant for the current platform.

29.3.1 Configuration variables

A Python distribution contains a `Makefile` and a `pyconfig.h` header file that are necessary to build both the Python binary itself and third-party C extensions compiled using `setuptools`.

sysconfig puts all variables found in these files in a dictionary that can be accessed using *get_config_vars()* or *get_config_var()*.

Notice that on Windows, it's a much smaller set.

sysconfig.**get_config_vars**(**args*)

With no arguments, return a dictionary of all configuration variables relevant for the current platform.

With arguments, return a list of values that result from looking up each argument in the configuration variable dictionary.

For each argument, if the value is not found, return `None`.

sysconfig.**get_config_var**(*name*)

Return the value of a single variable *name*. Equivalent to `get_config_vars().get(name)`.

If *name* is not found, return `None`.

Example of usage:

```
>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
'/usr/local/lib'
>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']
```

29.3.2 Installation paths

Python uses an installation scheme that differs depending on the platform and on the installation options. These schemes are stored in `sysconfig` under unique identifiers based on the value returned by `os.name`. The schemes are used by package installers to determine where to copy files to.

Python currently supports nine schemes:

- *posix_prefix*: scheme for POSIX platforms like Linux or macOS. This is the default scheme used when Python or a component is installed.
- *posix_home*: scheme for POSIX platforms, when the *home* option is used. This scheme defines paths located under a specific home prefix.
- *posix_user*: scheme for POSIX platforms, when the *user* option is used. This scheme defines paths located under the user's home directory (`site.USER_BASE`).
- *posix_venv*: scheme for `Python virtual environments` on POSIX platforms; by default it is the same as *posix_prefix*.
- *nt*: scheme for Windows. This is the default scheme used when Python or a component is installed.
- *nt_user*: scheme for Windows, when the *user* option is used.
- *nt_venv*: scheme for `Python virtual environments` on Windows; by default it is the same as *nt*.
- *venv*: a scheme with values from either *posix_venv* or *nt_venv* depending on the platform Python runs on.
- *osx_framework_user*: scheme for macOS, when the *user* option is used.

Each scheme is itself composed of a series of paths and each path has a unique identifier. Python currently uses eight paths:

- *stdlib*: directory containing the standard Python library files that are not platform-specific.
- *platstdlib*: directory containing the standard Python library files that are platform-specific.
- *platlib*: directory for site-specific, platform-specific files.
- *purelib*: directory for site-specific, non-platform-specific files ('pure' Python).
- *include*: directory for non-platform-specific header files for the Python C-API.
- *platinclude*: directory for platform-specific header files for the Python C-API.
- *scripts*: directory for script files.
- *data*: directory for data files.

29.3.3 User scheme

This scheme is designed to be the most convenient solution for users that don't have write permission to the global site-packages directory or don't want to install into it.

Files will be installed into subdirectories of `site.USER_BASE` (written as `userbase` hereafter). This scheme installs pure Python modules and extension modules in the same location (also known as `site.USER_SITE`).

posix_user

Path	Installation directory
stdlib	*userbase*/lib/python*X.Y*
platstdlib	*userbase*/lib/python*X.Y*
platlib	*userbase*/lib/python*X.Y*/site-packages
purelib	*userbase*/lib/python*X.Y*/site-packages
include	*userbase*/include/python*X.Y*
scripts	*userbase*/bin
data	*userbase*

nt_user

Path	Installation directory
stdlib	*userbase*\Python*XY*
platstdlib	*userbase*\Python*XY*
platlib	*userbase*\Python*XY*\site-packages
purelib	*userbase*\Python*XY*\site-packages
include	*userbase*\Python*XY*\Include
scripts	*userbase*\Python*XY*\Scripts
data	*userbase*

osx_framework_user

Path	Installation directory
stdlib	*userbase*/lib/python
platstdlib	*userbase*/lib/python
platlib	*userbase*/lib/python/site-packages
purelib	*userbase*/lib/python/site-packages
include	*userbase*/include/python*X.Y*
scripts	*userbase*/bin
data	*userbase*

29.3.4 Home scheme

The idea behind the "home scheme" is that you build and maintain a personal stash of Python modules. This scheme's name is derived from the idea of a "home" directory on Unix, since it's not unusual for a Unix user to make their home directory have a layout similar to `/usr/` or `/usr/local/`. This scheme can be used by anyone, regardless of the operating system they are installing for.

posix_home

Path	Installation directory
stdlib	*home*/lib/python
platstdlib	*home*/lib/python
platlib	*home*/lib/python
purelib	*home*/lib/python
include	*home*/include/python
platinclude	*home*/include/python
scripts	*home*/bin
data	*home*

29.3.5 Prefix scheme

The "prefix scheme" is useful when you wish to use one Python installation to perform the build/install (i.e., to run the setup script), but install modules into the third-party module directory of a different Python installation (or something that looks like a different Python installation). If this sounds a trifle unusual, it is—that's why the user and home schemes come before. However, there are at least two known cases where the prefix scheme will be useful.

First, consider that many Linux distributions put Python in /usr, rather than the more traditional /usr/local. This is entirely appropriate, since in those cases Python is part of "the system" rather than a local add-on. However, if you are installing Python modules from source, you probably want them to go in /usr/local/lib/python2.*X* rather than /usr/lib/python2.*X*.

Another possibility is a network filesystem where the name used to write to a remote directory is different from the name used to read it: for example, the Python interpreter accessed as /usr/local/bin/python might search for modules in /usr/local/lib/python2.*X*, but those modules would have to be installed to, say, /mnt/*@server*/export/lib/python2.*X*.

posix_prefix

Path	Installation directory
stdlib	*prefix*/lib/python*X.Y*
platstdlib	*prefix*/lib/python*X.Y*
platlib	*prefix*/lib/python*X.Y*/site-packages
purelib	*prefix*/lib/python*X.Y*/site-packages
include	*prefix*/include/python*X.Y*
platinclude	*prefix*/include/python*X.Y*
scripts	*prefix*/bin
data	*prefix*

nt

Path	Installation directory
stdlib	*prefix*\Lib
platstdlib	*prefix*\Lib
platlib	*prefix*\Lib\site-packages
purelib	*prefix*\Lib\site-packages
include	*prefix*\Include
platinclude	*prefix*\Include
scripts	*prefix*\Scripts
data	*prefix*

29.3.6 Installation path functions

sysconfig provides some functions to determine these installation paths.

sysconfig.**get_scheme_names**()

Return a tuple containing all schemes currently supported in *sysconfig*.

sysconfig.**get_default_scheme**()

Return the default scheme name for the current platform.

Added in version 3.10: This function was previously named `_get_default_scheme()` and considered an implementation detail.

Changed in version 3.11: When Python runs from a virtual environment, the *venv* scheme is returned.

sysconfig.**get_preferred_scheme**(*key*)

Return a preferred scheme name for an installation layout specified by *key*.

key must be either `"prefix"`, `"home"`, or `"user"`.

The return value is a scheme name listed in *get_scheme_names()*. It can be passed to *sysconfig* functions that take a *scheme* argument, such as *get_paths()*.

Added in version 3.10.

Changed in version 3.11: When Python runs from a virtual environment and `key="prefix"`, the *venv* scheme is returned.

sysconfig.**_get_preferred_schemes**()

Return a dict containing preferred scheme names on the current platform. Python implementers and redistributors may add their preferred schemes to the `_INSTALL_SCHEMES` module-level global value, and modify this function to return those scheme names, to e.g. provide different schemes for system and language package managers to use, so packages installed by either do not mix with those by the other.

End users should not use this function, but *get_default_scheme()* and *get_preferred_scheme()* instead.

Added in version 3.10.

sysconfig.**get_path_names**()

Return a tuple containing all path names currently supported in *sysconfig*.

sysconfig.**get_path**(*name*[, *scheme*[, *vars*[, *expand*]]])

Return an installation path corresponding to the path *name*, from the install scheme named *scheme*.

name has to be a value from the list returned by *get_path_names()*.

sysconfig stores installation paths corresponding to each path name, for each platform, with variables to be expanded. For instance the *stdlib* path for the *nt* scheme is: `{base}/Lib`.

get_path() will use the variables returned by *get_config_vars()* to expand the path. All variables have default values for each platform so one may call this function and get the default value.

If *scheme* is provided, it must be a value from the list returned by *get_scheme_names()*. Otherwise, the default scheme for the current platform is used.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary returned by *get_config_vars()*.

If *expand* is set to `False`, the path will not be expanded using the variables.

If *name* is not found, raise a *KeyError*.

sysconfig.**get_paths**([*scheme*[, *vars*[, *expand*]]])

Return a dictionary containing all installation paths corresponding to an installation scheme. See *get_path()* for more information.

If *scheme* is not provided, will use the default scheme for the current platform.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary used to expand the paths.

If *expand* is set to false, the paths will not be expanded.

If *scheme* is not an existing scheme, *get_paths()* will raise a *KeyError*.

29.3.7 Other functions

sysconfig.**get_python_version**()

Return the `MAJOR.MINOR` Python version number as a string. Similar to `'%d.%d' % sys.version_info[:2]`.

sysconfig.**get_platform**()

Return a string that identifies the current platform.

This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by *os.uname()*), although the exact information included depends on the OS; e.g., on Linux, the kernel version isn't particularly important.

Examples of returned values:

- linux-i586
- linux-alpha (?)
- solaris-2.6-sun4u

Windows will return one of:

- win-amd64 (64bit Windows on AMD64, aka x86_64, Intel64, and EM64T)
- win32 (all others - specifically, sys.platform is returned)

macOS can return:

- macosx-10.6-ppc

- macosx-10.4-ppc64
- macosx-10.3-i386
- macosx-10.4-fat

For other non-POSIX platforms, currently just returns *sys.platform*.

sysconfig.**is_python_build**()

Return `True` if the running Python interpreter was built from source and is being run from its built location, and not from a location resulting from e.g. running `make install` or installing via a binary installer.

sysconfig.**parse_config_h**(*fp*[, *vars*])

Parse a `config.h`-style file.

fp is a file-like object pointing to the `config.h`-like file.

A dictionary containing name/value pairs is returned. If an optional dictionary is passed in as the second argument, it is used instead of a new dictionary, and updated with the values read in the file.

sysconfig.**get_config_h_filename**()

Return the path of `pyconfig.h`.

sysconfig.**get_makefile_filename**()

Return the path of `Makefile`.

29.3.8 Using `sysconfig` as a script

You can use *sysconfig* as a script with Python's *-m* option:

```
$ python -m sysconfig
Platform: "macosx-10.4-i386"
Python version: "3.2"
Current installation scheme: "posix_prefix"

Paths:
        data = "/usr/local"
        include = "/Users/tarek/Dev/svn.python.org/py3k/Include"
        platinclude = "."
        platlib = "/usr/local/lib/python3.2/site-packages"
        platstdlib = "/usr/local/lib/python3.2"
        purelib = "/usr/local/lib/python3.2/site-packages"
        scripts = "/usr/local/bin"
        stdlib = "/usr/local/lib/python3.2"

Variables:
        AC_APPLE_UNIVERSAL_BUILD = "0"
        AIX_GENUINE_CPLUSPLUS = "0"
        AR = "ar"
        ARFLAGS = "rc"
        ...
```

This call will print in the standard output the information returned by *get_platform()*, *get_python_version()*, *get_path()* and *get_config_vars()*.

29.4 builtins — Built-in objects

This module provides direct access to all 'built-in' identifiers of Python; for example, builtins.open is the full name for the built-in function *open()*. See *Built-in Functions* and *Built-in Constants* for documentation.

This module is not normally accessed explicitly by most applications, but can be useful in modules that provide objects with the same name as a built-in value, but in which the built-in of that name is also needed. For example, in a module that wants to implement an *open()* function that wraps the built-in *open()*, this module can be used directly:

```
import builtins

def open(path):
    f = builtins.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''Wrapper around a file that converts output to uppercase.'''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

    # ...
```

As an implementation detail, most modules have the name __builtins__ made available as part of their globals. The value of __builtins__ is normally either this module or the value of this module's *__dict__* attribute. Since this is an implementation detail, it may not be used by alternate implementations of Python.

29.5 __main__ — Top-level code environment

In Python, the special name __main__ is used for two important constructs:

1. the name of the top-level environment of the program, which can be checked using the __name__ == '__main__' expression; and
2. the __main__.py file in Python packages.

Both of these mechanisms are related to Python modules; how users interact with them and how they interact with each other. They are explained in detail below. If you're new to Python modules, see the tutorial section tut-modules for an introduction.

29.5.1 __name__ == '__main__'

When a Python module or package is imported, __name__ is set to the module's name. Usually, this is the name of the Python file itself without the .py extension:

```
>>> import configparser
>>> configparser.__name__
'configparser'
```

If the file is part of a package, __name__ will also include the parent package's path:

```
>>> from concurrent.futures import process
>>> process.__name__
'concurrent.futures.process'
```

However, if the module is executed in the top-level code environment, its __name__ is set to the string '__main__'.

What is the "top-level code environment"?

__main__ is the name of the environment where top-level code is run. "Top-level code" is the first user-specified Python module that starts running. It's "top-level" because it imports all other modules that the program needs. Sometimes "top-level code" is called an *entry point* to the application.

The top-level code environment can be:

- the scope of an interactive prompt:

```
>>> __name__
'__main__'
```

- the Python module passed to the Python interpreter as a file argument:

```
$ python helloworld.py
Hello, world!
```

- the Python module or package passed to the Python interpreter with the -m argument:

```
$ python -m tarfile
usage: tarfile.py [-h] [-v] (...)
```

- Python code read by the Python interpreter from standard input:

```
$ echo "import this" | python
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
...
```

- Python code passed to the Python interpreter with the -c argument:

```
$ python -c "import this"
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
...
```

In each of these situations, the top-level module's `__name__` is set to `'__main__'`.

As a result, a module can discover whether or not it is running in the top-level environment by checking its own `__name__`, which allows a common idiom for conditionally executing code when the module is not initialized from an import statement:

```
if __name__ == '__main__':
    # Execute when the module is not initialized from an import statement.
    ...
```

See also

For a more detailed look at how `__name__` is set in all situations, see the tutorial section tut-modules.

Idiomatic Usage

Some modules contain code that is intended for script use only, like parsing command-line arguments or fetching data from standard input. If a module like this was imported from a different module, for example to unit test it, the script code would unintentionally execute as well.

This is where using the `if __name__ == '__main__'` code block comes in handy. Code within this block won't run unless the module is executed in the top-level environment.

Putting as few statements as possible in the block below `if __name__ == '__main__'` can improve code clarity and correctness. Most often, a function named `main` encapsulates the program's primary behavior:

```
# echo.py

import shlex
import sys

def echo(phrase: str) -> None:
   """A dummy wrapper around print."""
   # for demonstration purposes, you can imagine that there is some
   # valuable and reusable logic inside this function
   print(phrase)

def main() -> int:
    """Echo the input arguments to standard output"""
    phrase = shlex.join(sys.argv)
    echo(phrase)
    return 0

if __name__ == '__main__':
    sys.exit(main())  # next section explains the use of sys.exit
```

Note that if the module didn't encapsulate code inside the `main` function but instead put it directly within the `if __name__ == '__main__'` block, the `phrase` variable would be global to the entire module. This is error-prone as other functions within the module could be unintentionally using the global variable instead of a local name. A `main` function solves this problem.

Using a `main` function has the added benefit of the `echo` function itself being isolated and importable elsewhere. When `echo.py` is imported, the `echo` and `main` functions will be defined, but neither of them will be called, because `__name__ != '__main__'`.

Packaging Considerations

`main` functions are often used to create command-line tools by specifying them as entry points for console scripts. When this is done, pip inserts the function call into a template script, where the return value of `main` is passed into *`sys.exit()`*. For example:

```
sys.exit(main())
```

Since the call to `main` is wrapped in *`sys.exit()`*, the expectation is that your function will return some value acceptable as an input to *`sys.exit()`*; typically, an integer or `None` (which is implicitly returned if your function does not have a return statement).

By proactively following this convention ourselves, our module will have the same behavior when run directly (i.e. `python echo.py`) as it will have if we later package it as a console script entry-point in a pip-installable package.

In particular, be careful about returning strings from your `main` function. *`sys.exit()`* will interpret a string argument as a failure message, so your program will have an exit code of `1`, indicating failure, and the string will be written to *`sys.stderr`*. The `echo.py` example from earlier exemplifies using the `sys.exit(main())` convention.

See also

Python Packaging User Guide contains a collection of tutorials and references on how to distribute and install Python packages with modern tools.

29.5.2 `__main__.py` in Python Packages

If you are not familiar with Python packages, see section tut-packages of the tutorial. Most commonly, the `__main__.py` file is used to provide a command-line interface for a package. Consider the following hypothetical package, "bandclass":

```
bandclass
  ├── __init__.py
  ├── __main__.py
  └── student.py
```

`__main__.py` will be executed when the package itself is invoked directly from the command line using the `-m` flag. For example:

```
$ python -m bandclass
```

This command will cause `__main__.py` to run. How you utilize this mechanism will depend on the nature of the package you are writing, but in this hypothetical case, it might make sense to allow the teacher to search for students:

```
# bandclass/__main__.py

import sys
from .student import search_students

student_name = sys.argv[1] if len(sys.argv) >= 2 else ''
print(f'Found student: {search_students(student_name)}')
```

Note that `from .student import search_students` is an example of a relative import. This import style can be used when referencing modules within a package. For more details, see intra-package-references in the tut-modules section of the tutorial.

Idiomatic Usage

The content of `__main__.py` typically isn't fenced with an `if __name__ == '__main__'` block. Instead, those files are kept short and import functions to execute from other modules. Those other modules can then be easily unit-tested and are properly reusable.

If used, an `if __name__ == '__main__'` block will still work as expected for a `__main__.py` file within a package, because its `__name__` attribute will include the package's path if imported:

```
>>> import asyncio.__main__
>>> asyncio.__main__.__name__
'asyncio.__main__'
```

This won't work for `__main__.py` files in the root directory of a `.zip` file though. Hence, for consistency, a minimal `__main__.py` without a `__name__` check is preferred.

See also

See *venv* for an example of a package with a minimal `__main__.py` in the standard library. It doesn't contain a `if __name__ == '__main__'` block. You can invoke it with `python -m venv [directory]`.

See *runpy* for more details on the `-m` flag to the interpreter executable.

See *zipapp* for how to run applications packaged as *.zip* files. In this case Python looks for a `__main__.py` file in the root directory of the archive.

29.5.3 import __main__

Regardless of which module a Python program was started with, other modules running within that same program can import the top-level environment's scope (*namespace*) by importing the `__main__` module. This doesn't import a `__main__.py` file but rather whichever module that received the special name `'__main__'`.

Here is an example module that consumes the `__main__` namespace:

```
# namely.py

import __main__

def did_user_define_their_name():
    return 'my_name' in dir(__main__)

def print_user_name():
    if not did_user_define_their_name():
        raise ValueError('Define the variable `my_name`!')

    if '__file__' in dir(__main__):
        print(__main__.my_name, "found in file", __main__.__file__)
    else:
        print(__main__.my_name)
```

Example usage of this module could be as follows:

```
# start.py

import sys
```

(continues on next page)

(continued from previous page)

```
from namely import print_user_name

# my_name = "Dinsdale"

def main():
    try:
        print_user_name()
    except ValueError as ve:
        return str(ve)

if __name__ == "__main__":
    sys.exit(main())
```

Now, if we started our program, the result would look like this:

```
$ python start.py
Define the variable `my_name`!
```

The exit code of the program would be 1, indicating an error. Uncommenting the line with `my_name = "Dinsdale"` fixes the program and now it exits with status code 0, indicating success:

```
$ python start.py
Dinsdale found in file /path/to/start.py
```

Note that importing `__main__` doesn't cause any issues with unintentionally running top-level code meant for script use which is put in the `if __name__ == "__main__"` block of the `start` module. Why does this work?

Python inserts an empty `__main__` module in *sys.modules* at interpreter startup, and populates it by running top-level code. In our example this is the `start` module which runs line by line and imports `namely`. In turn, `namely` imports `__main__` (which is really `start`). That's an import cycle! Fortunately, since the partially populated `__main__` module is present in *sys.modules*, Python passes that to `namely`. See Special considerations for __main__ in the import system's reference for details on how this works.

The Python REPL is another example of a "top-level environment", so anything defined in the REPL becomes part of the `__main__` scope:

```
>>> import namely
>>> namely.did_user_define_their_name()
False
>>> namely.print_user_name()
Traceback (most recent call last):
...
ValueError: Define the variable `my_name`!
>>> my_name = 'Jabberwocky'
>>> namely.did_user_define_their_name()
True
>>> namely.print_user_name()
Jabberwocky
```

Note that in this case the `__main__` scope doesn't contain a `__file__` attribute as it's interactive.

The `__main__` scope is used in the implementation of *pdb* and *rlcompleter*.

29.6 `warnings` — Warning control

Source code: Lib/warnings.py

Warning messages are typically issued in situations where it is useful to alert the user of some condition in a program, where that condition (normally) doesn't warrant raising an exception and terminating the program. For example, one might want to issue a warning when a program uses an obsolete module.

Python programmers issue warnings by calling the *warn()* function defined in this module. (C programmers use `PyErr_WarnEx()`; see exceptionhandling for details).

Warning messages are normally written to *sys.stderr*, but their disposition can be changed flexibly, from ignoring all warnings to turning them into exceptions. The disposition of warnings can vary based on the *warning category*, the text of the warning message, and the source location where it is issued. Repetitions of a particular warning for the same source location are typically suppressed.

There are two stages in warning control: first, each time a warning is issued, a determination is made whether a message should be issued or not; next, if a message is to be issued, it is formatted and printed using a user-settable hook.

The determination whether to issue a warning message is controlled by the *warning filter*, which is a sequence of matching rules and actions. Rules can be added to the filter by calling *filterwarnings()* and reset to its default state by calling *resetwarnings()*.

The printing of warning messages is done by calling *showwarning()*, which may be overridden; the default implementation of this function formats the message by calling *formatwarning()*, which is also available for use by custom implementations.

See also

logging.captureWarnings() allows you to handle all warnings with the standard logging infrastructure.

29.6.1 Warning Categories

There are a number of built-in exceptions that represent warning categories. This categorization is useful to be able to filter out groups of warnings.

While these are technically *built-in exceptions*, they are documented here, because conceptually they belong to the warnings mechanism.

User code can define additional warning categories by subclassing one of the standard warning categories. A warning category must always be a subclass of the *Warning* class.

The following warnings category classes are currently defined:

Class	Description
Warning	This is the base class of all warning category classes. It is a subclass of *Exception*.
UserWarning	The default category for *warn()*.
DeprecationWarning	Base category for warnings about deprecated features when those warnings are intended for other Python developers (ignored by default, unless triggered by code in __main__).
SyntaxWarning	Base category for warnings about dubious syntactic features.
RuntimeWarning	Base category for warnings about dubious runtime features.
FutureWarning	Base category for warnings about deprecated features when those warnings are intended for end users of applications that are written in Python.
PendingDeprecationWarning	Base category for warnings about features that will be deprecated in the future (ignored by default).
ImportWarning	Base category for warnings triggered during the process of importing a module (ignored by default).
UnicodeWarning	Base category for warnings related to Unicode.
BytesWarning	Base category for warnings related to *bytes* and *bytearray*.
ResourceWarning	Base category for warnings related to resource usage (ignored by default).

Changed in version 3.7: Previously *DeprecationWarning* and *FutureWarning* were distinguished based on whether a feature was being removed entirely or changing its behaviour. They are now distinguished based on their intended audience and the way they're handled by the default warnings filters.

29.6.2 The Warnings Filter

The warnings filter controls whether warnings are ignored, displayed, or turned into errors (raising an exception).

Conceptually, the warnings filter maintains an ordered list of filter specifications; any specific warning is matched against each filter specification in the list in turn until a match is found; the filter determines the disposition of the match. Each entry is a tuple of the form (*action*, *message*, *category*, *module*, *lineno*), where:

- *action* is one of the following strings:

Value	Disposition
"default"	print the first occurrence of matching warnings for each location (module + line number) where the warning is issued
"error"	turn matching warnings into exceptions
"ignore"	never print matching warnings
"always"	always print matching warnings
"module"	print the first occurrence of matching warnings for each module where the warning is issued (regardless of line number)
"once"	print only the first occurrence of matching warnings, regardless of location

- *message* is a string containing a regular expression that the start of the warning message must match, case-insensitively. In -W and PYTHONWARNINGS, *message* is a literal string that the start of the warning message must contain (case-insensitively), ignoring any whitespace at the start or end of *message*.
- *category* is a class (a subclass of *Warning*) of which the warning category must be a subclass in order to match.

- *module* is a string containing a regular expression that the start of the fully qualified module name must match, case-sensitively. In `-W` and `PYTHONWARNINGS`, *module* is a literal string that the fully qualified module name must be equal to (case-sensitively), ignoring any whitespace at the start or end of *module*.
- *lineno* is an integer that the line number where the warning occurred must match, or `0` to match all line numbers.

Since the *Warning* class is derived from the built-in *Exception* class, to turn a warning into an error we simply raise `category(message)`.

If a warning is reported and doesn't match any registered filter then the "default" action is applied (hence its name).

Describing Warning Filters

The warnings filter is initialized by `-W` options passed to the Python interpreter command line and the `PYTHONWARNINGS` environment variable. The interpreter saves the arguments for all supplied entries without interpretation in *sys.warnoptions*; the *warnings* module parses these when it is first imported (invalid options are ignored, after printing a message to *sys.stderr*).

Individual warnings filters are specified as a sequence of fields separated by colons:

```
action:message:category:module:line
```

The meaning of each of these fields is as described in *The Warnings Filter*. When listing multiple filters on a single line (as for `PYTHONWARNINGS`), the individual filters are separated by commas and the filters listed later take precedence over those listed before them (as they're applied left-to-right, and the most recently applied filters take precedence over earlier ones).

Commonly used warning filters apply to either all warnings, warnings in a particular category, or warnings raised by particular modules or packages. Some examples:

```
default                       # Show all warnings (even those ignored by default)
ignore                        # Ignore all warnings
error                         # Convert all warnings to errors
error::ResourceWarning        # Treat ResourceWarning messages as errors
default::DeprecationWarning   # Show DeprecationWarning messages
ignore,default:::mymodule     # Only report warnings triggered by "mymodule"
error:::mymodule              # Convert warnings to errors in "mymodule"
```

Default Warning Filter

By default, Python installs several warning filters, which can be overridden by the `-W` command-line option, the `PYTHONWARNINGS` environment variable and calls to *filterwarnings()*.

In regular release builds, the default warning filter has the following entries (in order of precedence):

```
default::DeprecationWarning:__main__
ignore::DeprecationWarning
ignore::PendingDeprecationWarning
ignore::ImportWarning
ignore::ResourceWarning
```

In a debug build, the list of default warning filters is empty.

Changed in version 3.2: *DeprecationWarning* is now ignored by default in addition to *PendingDeprecationWarning*.

Changed in version 3.7: *DeprecationWarning* is once again shown by default when triggered directly by code in `__main__`.

Changed in version 3.7: *BytesWarning* no longer appears in the default filter list and is instead configured via *sys.warnoptions* when `-b` is specified twice.

Overriding the default filter

Developers of applications written in Python may wish to hide *all* Python level warnings from their users by default, and only display them when running tests or otherwise working on the application. The *sys.warnoptions* attribute used to pass filter configurations to the interpreter can be used as a marker to indicate whether or not warnings should be disabled:

```
import sys

if not sys.warnoptions:
    import warnings
    warnings.simplefilter("ignore")
```

Developers of test runners for Python code are advised to instead ensure that *all* warnings are displayed by default for the code under test, using code like:

```
import sys

if not sys.warnoptions:
    import os, warnings
    warnings.simplefilter("default") # Change the filter in this process
    os.environ["PYTHONWARNINGS"] = "default" # Also affect subprocesses
```

Finally, developers of interactive shells that run user code in a namespace other than `__main__` are advised to ensure that *DeprecationWarning* messages are made visible by default, using code like the following (where `user_ns` is the module used to execute code entered interactively):

```
import warnings
warnings.filterwarnings("default", category=DeprecationWarning,
                                   module=user_ns.get("__name__"))
```

29.6.3 Temporarily Suppressing Warnings

If you are using code that you know will raise a warning, such as a deprecated function, but do not want to see the warning (even when warnings have been explicitly configured via the command line), then it is possible to suppress the warning using the *catch_warnings* context manager:

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

While within the context manager all warnings will simply be ignored. This allows you to use known-deprecated code without having to see the warning while not suppressing the warning for other code that might not be aware of its use of deprecated code. Note: this can only be guaranteed in a single-threaded application. If two or more threads use the *catch_warnings* context manager at the same time, the behavior is undefined.

29.6.4 Testing Warnings

To test warnings raised by code, use the *catch_warnings* context manager. With it you can temporarily mutate the warnings filter to facilitate your testing. For instance, do the following to capture all raised warnings to check:

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # Cause all warnings to always be triggered.
    warnings.simplefilter("always")
    # Trigger a warning.
    fxn()
    # Verify some things
    assert len(w) == 1
    assert issubclass(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

One can also cause all warnings to be exceptions by using `error` instead of `always`. One thing to be aware of is that if a warning has already been raised because of a `once`/`default` rule, then no matter what filters are set the warning will not be seen again unless the warnings registry related to the warning has been cleared.

Once the context manager exits, the warnings filter is restored to its state when the context was entered. This prevents tests from changing the warnings filter in unexpected ways between tests and leading to indeterminate test results. The *showwarning()* function in the module is also restored to its original value. Note: this can only be guaranteed in a single-threaded application. If two or more threads use the *catch_warnings* context manager at the same time, the behavior is undefined.

When testing multiple operations that raise the same kind of warning, it is important to test them in a manner that confirms each operation is raising a new warning (e.g. set warnings to be raised as exceptions and check the operations raise exceptions, check that the length of the warning list continues to increase after each operation, or else delete the previous entries from the warnings list before each new operation).

29.6.5 Updating Code For New Versions of Dependencies

Warning categories that are primarily of interest to Python developers (rather than end users of applications written in Python) are ignored by default.

Notably, this "ignored by default" list includes *DeprecationWarning* (for every module except `__main__`), which means developers should make sure to test their code with typically ignored warnings made visible in order to receive timely notifications of future breaking API changes (whether in the standard library or third party packages).

In the ideal case, the code will have a suitable test suite, and the test runner will take care of implicitly enabling all warnings when running tests (the test runner provided by the *unittest* module does this).

In less ideal cases, applications can be checked for use of deprecated interfaces by passing `-Wd` to the Python interpreter (this is shorthand for `-W default`) or setting `PYTHONWARNINGS=default` in the environment. This enables default handling for all warnings, including those that are ignored by default. To change what action is taken for encountered warnings you can change what argument is passed to `-W` (e.g. `-W error`). See the `-W` flag for more details on what is possible.

29.6.6 Available Functions

warnings.**warn**(*message*, *category=None*, *stacklevel=1*, *source=None*, ***, *skip_file_prefixes=None*)

Issue a warning, or maybe ignore it or raise an exception. The *category* argument, if given, must be a *warning category class*; it defaults to `UserWarning`. Alternatively, *message* can be a `Warning` instance, in which case *category* will be ignored and `message.__class__` will be used. In this case, the message text will be `str(message)`. This function raises an exception if the particular warning issued is changed into an error by the *warnings filter*. The *stacklevel* argument can be used by wrapper functions written in Python, like this:

```
def deprecated_api(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

This makes the warning refer to `deprecated_api`'s caller, rather than to the source of `deprecated_api` itself (since the latter would defeat the purpose of the warning message).

The *skip_file_prefixes* keyword argument can be used to indicate which stack frames are ignored when counting stack levels. This can be useful when you want the warning to always appear at call sites outside of a package when a constant *stacklevel* does not fit all call paths or is otherwise challenging to maintain. If supplied, it must be a tuple of strings. When prefixes are supplied, stacklevel is implicitly overridden to be `max(2, stacklevel)`. To cause a warning to be attributed to the caller from outside of the current package you might write:

```
# example/lower.py
_warn_skips = (os.path.dirname(__file__),)

def one_way(r_luxury_yacht=None, t_wobbler_mangrove=None):
    if r_luxury_yacht:
        warnings.warn("Please migrate to t_wobbler_mangrove=.",
                      skip_file_prefixes=_warn_skips)

# example/higher.py
from . import lower

def another_way(**kw):
    lower.one_way(**kw)
```

This makes the warning refer to both the `example.lower.one_way()` and `package.higher.another_way()` call sites only from calling code living outside of `example` package.

source, if supplied, is the destroyed object which emitted a `ResourceWarning`.

Changed in version 3.6: Added *source* parameter.

Changed in version 3.12: Added *skip_file_prefixes*.

warnings.**warn_explicit**(*message*, *category*, *filename*, *lineno*, *module=None*, *registry=None*, *module_globals=None*, *source=None*)

This is a low-level interface to the functionality of `warn()`, passing in explicitly the message, category, filename and line number, and optionally the module name and the registry (which should be the `__warningregistry__` dictionary of the module). The module name defaults to the filename with `.py` stripped; if no registry is passed, the warning is never suppressed. *message* must be a string and *category* a subclass of `Warning` or *message* may be a `Warning` instance, in which case *category* will be ignored.

module_globals, if supplied, should be the global namespace in use by the code for which the warning is issued. (This argument is used to support displaying source for modules found in zipfiles or other non-filesystem import sources).

source, if supplied, is the destroyed object which emitted a `ResourceWarning`.

Changed in version 3.6: Add the *source* parameter.

warnings.**showwarning**(*message*, *category*, *filename*, *lineno*, *file=None*, *line=None*)

Write a warning to a file. The default implementation calls `formatwarning(message, category, filename, lineno, line)` and writes the resulting string to *file*, which defaults to *sys.stderr*. You may replace this function with any callable by assigning to `warnings.showwarning`. *line* is a line of source code to be included in the warning message; if *line* is not supplied, *showwarning()* will try to read the line specified by *filename* and *lineno*.

warnings.**formatwarning**(*message*, *category*, *filename*, *lineno*, *line=None*)

Format a warning the standard way. This returns a string which may contain embedded newlines and ends in a newline. *line* is a line of source code to be included in the warning message; if *line* is not supplied, *formatwarning()* will try to read the line specified by *filename* and *lineno*.

warnings.**filterwarnings**(*action*, *message=''*, *category=Warning*, *module=''*, *lineno=0*, *append=False*)

Insert an entry into the list of *warnings filter specifications*. The entry is inserted at the front by default; if *append* is true, it is inserted at the end. This checks the types of the arguments, compiles the *message* and *module* regular expressions, and inserts them as a tuple in the list of warnings filters. Entries closer to the front of the list override entries later in the list, if both match a particular warning. Omitted arguments default to a value that matches everything.

warnings.**simplefilter**(*action*, *category=Warning*, *lineno=0*, *append=False*)

Insert a simple entry into the list of *warnings filter specifications*. The meaning of the function parameters is as for *filterwarnings()*, but regular expressions are not needed as the filter inserted always matches any message in any module as long as the category and line number match.

warnings.**resetwarnings**()

Reset the warnings filter. This discards the effect of all previous calls to *filterwarnings()*, including that of the `-W` command line options and calls to *simplefilter()*.

@warnings.**deprecated**(*msg*, ***, *category=DeprecationWarning*, *stacklevel=1*)

Decorator to indicate that a class, function or overload is deprecated.

When this decorator is applied to an object, deprecation warnings may be emitted at runtime when the object is used. *static type checkers* will also generate a diagnostic on usage of the deprecated object.

Usage:

```
from warnings import deprecated
from typing import overload

@deprecated("Use B instead")
class A:
    pass

@deprecated("Use g instead")
def f():
    pass

@overload
@deprecated("int support is deprecated")
def g(x: int) -> int: ...
@overload
def g(x: str) -> int: ...
```

The warning specified by *category* will be emitted at runtime on use of deprecated objects. For functions, that happens on calls; for classes, on instantiation and on creation of subclasses. If the *category* is `None`, no warning is emitted at runtime. The *stacklevel* determines where the warning is emitted. If it is `1` (the default), the warning is emitted at the direct caller of the deprecated object; if it is higher, it is emitted further up the stack. Static type checker behavior is not affected by the *category* and *stacklevel* arguments.

The deprecation message passed to the decorator is saved in the `__deprecated__` attribute on the decorated object. If applied to an overload, the decorator must be after the `@overload` decorator for the attribute to exist on the overload as returned by `typing.get_overloads()`.

Added in version 3.13: See **PEP 702**.

29.6.7 Available Context Managers

class warnings.**catch_warnings**(*, *record=False*, *module=None*, *action=None*, *category=Warning*, *lineno=0*, *append=False*)

A context manager that copies and, upon exit, restores the warnings filter and the `showwarning()` function. If the *record* argument is `False` (the default) the context manager returns `None` on entry. If *record* is `True`, a list is returned that is progressively populated with objects as seen by a custom `showwarning()` function (which also suppresses output to `sys.stdout`). Each object in the list has attributes with the same names as the arguments to `showwarning()`.

The *module* argument takes a module that will be used instead of the module returned when you import `warnings` whose filter will be protected. This argument exists primarily for testing the `warnings` module itself.

If the *action* argument is not `None`, the remaining arguments are passed to `simplefilter()` as if it were called immediately on entering the context.

> **Note**
>
> The `catch_warnings` manager works by replacing and then later restoring the module's `showwarning()` function and internal list of filter specifications. This means the context manager is modifying global state and therefore is not thread-safe.

Changed in version 3.11: Added the *action*, *category*, *lineno*, and *append* parameters.

29.7 `dataclasses` — Data Classes

Source code: Lib/dataclasses.py

This module provides a decorator and functions for automatically adding generated *special methods* such as `__init__()` and `__repr__()` to user-defined classes. It was originally described in **PEP 557**.

The member variables to use in these generated methods are defined using **PEP 526** type annotations. For example, this code:

```
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

will add, among other things, a `__init__()` that looks like:

```
def __init__(self, name: str, unit_price: float, quantity_on_hand: int = 0):
    self.name = name
    self.unit_price = unit_price
    self.quantity_on_hand = quantity_on_hand
```

Note that this method is automatically added to the class: it is not directly specified in the `InventoryItem` definition shown above.

Added in version 3.7.

29.7.1 Module contents

`@dataclasses.dataclass`(*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*, *match_args=True*, *kw_only=False*, *slots=False*, *weakref_slot=False*)

This function is a *decorator* that is used to add generated *special methods* to classes, as described below.

The `@dataclass` decorator examines the class to find `field`s. A `field` is defined as a class variable that has a *type annotation*. With two exceptions described below, nothing in `@dataclass` examines the type specified in the variable annotation.

The order of the fields in all of the generated methods is the order in which they appear in the class definition.

The `@dataclass` decorator will add various "dunder" methods to the class, described below. If any of the added methods already exist in the class, the behavior depends on the parameter, as documented below. The decorator returns the same class that it is called on; no new class is created.

If `@dataclass` is used just as a simple decorator with no parameters, it acts as if it has the default values documented in this signature. That is, these three uses of `@dataclass` are equivalent:

```
@dataclass
class C:
    ...

@dataclass()
class C:
    ...

@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False,␣
↪frozen=False,
           match_args=True, kw_only=False, slots=False, weakref_slot=False)
class C:
    ...
```

The parameters to `@dataclass` are:

- *init*: If true (the default), a `__init__()` method will be generated.

  If the class already defines `__init__()`, this parameter is ignored.

- *repr*: If true (the default), a `__repr__()` method will be generated. The generated repr string will have the class name and the name and repr of each field, in the order they are defined in the class. Fields that are marked as being excluded from the repr are not included. For example: `InventoryItem(name='widget', unit_price=3.0, quantity_on_hand=10)`.

  If the class already defines `__repr__()`, this parameter is ignored.

- *eq*: If true (the default), an `__eq__()` method will be generated. This method compares the class as if it were a tuple of its fields, in order. Both instances in the comparison must be of the identical type.

  If the class already defines `__eq__()`, this parameter is ignored.

- *order*: If true (the default is `False`), `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` methods will be generated. These compare the class as if it were a tuple of its fields, in order. Both instances in the comparison must be of the identical type. If *order* is true and *eq* is false, a *ValueError* is raised.

  If the class already defines any of `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()`, then *TypeError* is raised.

- *unsafe_hash*: If `False` (the default), a `__hash__()` method is generated according to how *eq* and *frozen* are set.

  `__hash__()` is used by built-in *hash()*, and when objects are added to hashed collections such as dictionaries and sets. Having a `__hash__()` implies that instances of the class are immutable. Mutability is a complicated property that depends on the programmer's intent, the existence and behavior of `__eq__()`, and the values of the *eq* and *frozen* flags in the `@dataclass` decorator.

  By default, `@dataclass` will not implicitly add a `__hash__()` method unless it is safe to do so. Neither will it add or change an existing explicitly defined `__hash__()` method. Setting the class attribute `__hash__ = None` has a specific meaning to Python, as described in the `__hash__()` documentation.

  If `__hash__()` is not explicitly defined, or if it is set to `None`, then `@dataclass` *may* add an implicit `__hash__()` method. Although not recommended, you can force `@dataclass` to create a `__hash__()` method with `unsafe_hash=True`. This might be the case if your class is logically immutable but can still be mutated. This is a specialized use case and should be considered carefully.

  Here are the rules governing implicit creation of a `__hash__()` method. Note that you cannot both have an explicit `__hash__()` method in your dataclass and set `unsafe_hash=True`; this will result in a *TypeError*.

  If *eq* and *frozen* are both true, by default `@dataclass` will generate a `__hash__()` method for you. If *eq* is true and *frozen* is false, `__hash__()` will be set to `None`, marking it unhashable (which it is, since it is mutable). If *eq* is false, `__hash__()` will be left untouched meaning the `__hash__()` method of the superclass will be used (if the superclass is *object*, this means it will fall back to id-based hashing).

- *frozen*: If true (the default is `False`), assigning to fields will generate an exception. This emulates read-only frozen instances. If `__setattr__()` or `__delattr__()` is defined in the class, then *TypeError* is raised. See the discussion below.

- *match_args*: If true (the default is `True`), the `__match_args__` tuple will be created from the list of parameters to the generated `__init__()` method (even if `__init__()` is not generated, see above). If false, or if `__match_args__` is already defined in the class, then `__match_args__` will not be generated.

  Added in version 3.10.

- *kw_only*: If true (the default value is `False`), then all fields will be marked as keyword-only. If a field is marked as keyword-only, then the only effect is that the `__init__()` parameter generated from a keyword-only field must be specified with a keyword when `__init__()` is called. There is no effect on any other aspect of dataclasses. See the *parameter* glossary entry for details. Also see the *KW_ONLY* section.

  Added in version 3.10.

- *slots*: If true (the default is `False`), `__slots__` attribute will be generated and new class will be returned instead of the original one. If `__slots__` is already defined in the class, then *TypeError* is raised. Calling no-arg *super()* in dataclasses using `slots=True` will result in the following exception being

raised: `TypeError: super(type, obj): obj must be an instance or subtype of type`. The two-arg *super()* is a valid workaround. See gh-90562 for full details.

Added in version 3.10.

Changed in version 3.11: If a field name is already included in the `__slots__` of a base class, it will not be included in the generated `__slots__` to prevent overriding them. Therefore, do not use `__slots__` to retrieve the field names of a dataclass. Use *fields()* instead. To be able to determine inherited slots, base class `__slots__` may be any iterable, but *not* an iterator.

- *weakref_slot*: If true (the default is `False`), add a slot named "__weakref__", which is required to make an instance weakref-able. It is an error to specify `weakref_slot=True` without also specifying `slots=True`.

  Added in version 3.11.

`fields` may optionally specify a default value, using normal Python syntax:

```
@dataclass
class C:
    a: int       # 'a' has no default value
    b: int = 0   # assign a default value for 'b'
```

In this example, both `a` and `b` will be included in the added `__init__()` method, which will be defined as:

```
def __init__(self, a: int, b: int = 0):
```

TypeError will be raised if a field without a default value follows a field with a default value. This is true whether this occurs in a single class, or as a result of class inheritance.

`dataclasses.`**`field`**`(*, default=MISSING, default_factory=MISSING, init=True, repr=True, hash=None, compare=True, metadata=None, kw_only=MISSING)`

For common and simple use cases, no other functionality is required. There are, however, some dataclass features that require additional per-field information. To satisfy this need for additional information, you can replace the default field value with a call to the provided `field()` function. For example:

```
@dataclass
class C:
    mylist: list[int] = field(default_factory=list)

c = C()
c.mylist += [1, 2, 3]
```

As shown above, the *MISSING* value is a sentinel object used to detect if some parameters are provided by the user. This sentinel is used because `None` is a valid value for some parameters with a distinct meaning. No code should directly use the *MISSING* value.

The parameters to `field()` are:

- *default*: If provided, this will be the default value for this field. This is needed because the `field()` call itself replaces the normal position of the default value.
- *default_factory*: If provided, it must be a zero-argument callable that will be called when a default value is needed for this field. Among other purposes, this can be used to specify fields with mutable default values, as discussed below. It is an error to specify both *default* and *default_factory*.
- *init*: If true (the default), this field is included as a parameter to the generated `__init__()` method.
- *repr*: If true (the default), this field is included in the string returned by the generated `__repr__()` method.

- *hash*: This can be a bool or `None`. If true, this field is included in the generated `__hash__()` method. If `None` (the default), use the value of *compare*: this would normally be the expected behavior. A field should be considered in the hash if it's used for comparisons. Setting this value to anything other than `None` is discouraged.

  One possible reason to set `hash=False` but `compare=True` would be if a field is expensive to compute a hash value for, that field is needed for equality testing, and there are other fields that contribute to the type's hash value. Even if a field is excluded from the hash, it will still be used for comparisons.

- *compare*: If true (the default), this field is included in the generated equality and comparison methods (`__eq__()`, `__gt__()`, et al.).

- *metadata*: This can be a mapping or `None`. `None` is treated as an empty dict. This value is wrapped in *`MappingProxyType()`* to make it read-only, and exposed on the *`Field`* object. It is not used at all by Data Classes, and is provided as a third-party extension mechanism. Multiple third-parties can each have their own key, to use as a namespace in the metadata.

- *kw_only*: If true, this field will be marked as keyword-only. This is used when the generated `__init__()` method's parameters are computed.

  Added in version 3.10.

If the default value of a field is specified by a call to `field()`, then the class attribute for this field will be replaced by the specified *default* value. If *default* is not provided, then the class attribute will be deleted. The intent is that after the *`@dataclass`* decorator runs, the class attributes will all contain the default values for the fields, just as if the default value itself were specified. For example, after:

```
@dataclass
class C:
    x: int
    y: int = field(repr=False)
    z: int = field(repr=False, default=10)
    t: int = 20
```

The class attribute `C.z` will be `10`, the class attribute `C.t` will be `20`, and the class attributes `C.x` and `C.y` will not be set.

class dataclasses.**Field**

`Field` objects describe each defined field. These objects are created internally, and are returned by the *`fields()`* module-level method (see below). Users should never instantiate a `Field` object directly. Its documented attributes are:

- `name`: The name of the field.
- `type`: The type of the field.
- `default`, `default_factory`, `init`, `repr`, `hash`, `compare`, `metadata`, and `kw_only` have the identical meaning and values as they do in the *`field()`* function.

Other attributes may exist, but they are private and must not be inspected or relied on.

dataclasses.**fields**(*class_or_instance*)

Returns a tuple of *`Field`* objects that define the fields for this dataclass. Accepts either a dataclass, or an instance of a dataclass. Raises *`TypeError`* if not passed a dataclass or instance of one. Does not return pseudo-fields which are `ClassVar` or `InitVar`.

dataclasses.**asdict**(*obj*, *, *dict_factory=dict*)

Converts the dataclass *obj* to a dict (by using the factory function *dict_factory*). Each dataclass is converted to a dict of its fields, as `name: value` pairs. dataclasses, dicts, lists, and tuples are recursed into. Other objects are copied with *`copy.deepcopy()`*.

Example of using `asdict()` on nested dataclasses:

```
@dataclass
class Point:
     x: int
     y: int

@dataclass
class C:
     mylist: list[Point]

p = Point(10, 20)
assert asdict(p) == {'x': 10, 'y': 20}

c = C([Point(0, 0), Point(10, 4)])
assert asdict(c) == {'mylist': [{'x': 0, 'y': 0}, {'x': 10, 'y': 4}]}
```

To create a shallow copy, the following workaround may be used:

```
{field.name: getattr(obj, field.name) for field in fields(obj)}
```

`asdict()` raises *TypeError* if *obj* is not a dataclass instance.

`dataclasses.`**`astuple`**(*obj*, ***, *tuple_factory=tuple*)

Converts the dataclass *obj* to a tuple (by using the factory function *tuple_factory*). Each dataclass is converted to a tuple of its field values. dataclasses, dicts, lists, and tuples are recursed into. Other objects are copied with *copy.deepcopy()*.

Continuing from the previous example:

```
assert astuple(p) == (10, 20)
assert astuple(c) == ([(0, 0), (10, 4)],)
```

To create a shallow copy, the following workaround may be used:

```
tuple(getattr(obj, field.name) for field in dataclasses.fields(obj))
```

`astuple()` raises *TypeError* if *obj* is not a dataclass instance.

`dataclasses.`**`make_dataclass`**(*cls_name*, *fields*, ***, *bases=()*, *namespace=None*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*, *match_args=True*, *kw_only=False*, *slots=False*, *weakref_slot=False*, *module=None*)

Creates a new dataclass with name *cls_name*, fields as defined in *fields*, base classes as given in *bases*, and initialized with a namespace as given in *namespace*. *fields* is an iterable whose elements are each either `name`, `(name, type)`, or `(name, type, Field)`. If just `name` is supplied, *typing.Any* is used for `type`. The values of *init*, *repr*, *eq*, *order*, *unsafe_hash*, *frozen*, *match_args*, *kw_only*, *slots*, and *weakref_slot* have the same meaning as they do in *@dataclass*.

If *module* is defined, the `__module__` attribute of the dataclass is set to that value. By default, it is set to the module name of the caller.

This function is not strictly required, because any Python mechanism for creating a new class with `__annotations__` can then apply the *@dataclass* function to convert that class to a dataclass. This function is provided as a convenience. For example:

```
C = make_dataclass('C',
                   [('x', int),
                     'y',
```

(continues on next page)

(continued from previous page)

```
                    ('z', int, field(default=5))],
                   namespace={'add_one': lambda self: self.x + 1})
```

Is equivalent to:

```
@dataclass
class C:
    x: int
    y: 'typing.Any'
    z: int = 5

    def add_one(self):
        return self.x + 1
```

dataclasses.**replace**(*obj*, */*, ***changes*)

Creates a new object of the same type as *obj*, replacing fields with values from *changes*. If *obj* is not a Data Class, raises *TypeError*. If keys in *changes* are not field names of the given dataclass, raises *TypeError*.

The newly returned object is created by calling the `__init__()` method of the dataclass. This ensures that *__post_init__()*, if present, is also called.

Init-only variables without default values, if any exist, must be specified on the call to `replace()` so that they can be passed to `__init__()` and *__post_init__()*.

It is an error for *changes* to contain any fields that are defined as having `init=False`. A *ValueError* will be raised in this case.

Be forewarned about how `init=False` fields work during a call to `replace()`. They are not copied from the source object, but rather are initialized in *__post_init__()*, if they're initialized at all. It is expected that `init=False` fields will be rarely and judiciously used. If they are used, it might be wise to have alternate class constructors, or perhaps a custom `replace()` (or similarly named) method which handles instance copying.

Dataclass instances are also supported by generic function *copy.replace()*.

dataclasses.**is_dataclass**(*obj*)

Return `True` if its parameter is a dataclass (including subclasses of a dataclass) or an instance of one, otherwise return `False`.

If you need to know if a class is an instance of a dataclass (and not a dataclass itself), then add a further check for `not isinstance(obj, type)`:

```
def is_dataclass_instance(obj):
    return is_dataclass(obj) and not isinstance(obj, type)
```

dataclasses.**MISSING**

A sentinel value signifying a missing default or default_factory.

dataclasses.**KW_ONLY**

A sentinel value used as a type annotation. Any fields after a pseudo-field with the type of `KW_ONLY` are marked as keyword-only fields. Note that a pseudo-field of type `KW_ONLY` is otherwise completely ignored. This includes the name of such a field. By convention, a name of `_` is used for a `KW_ONLY` field. Keyword-only fields signify `__init__()` parameters that must be specified as keywords when the class is instantiated.

In this example, the fields `y` and `z` will be marked as keyword-only fields:

```
@dataclass
class Point:
```

(continues on next page)

(continued from previous page)

```
    x: float
    _: KW_ONLY
    y: float
    z: float

p = Point(0, y=1.5, z=2.0)
```

In a single dataclass, it is an error to specify more than one field whose type is `KW_ONLY`.

Added in version 3.10.

exception dataclasses.**FrozenInstanceError**

Raised when an implicitly defined `__setattr__()` or `__delattr__()` is called on a dataclass which was defined with `frozen=True`. It is a subclass of *AttributeError*.

29.7.2 Post-init processing

dataclasses.**__post_init__**()

When defined on the class, it will be called by the generated `__init__()`, normally as `self.__post_init__()`. However, if any `InitVar` fields are defined, they will also be passed to `__post_init__()` in the order they were defined in the class. If no `__init__()` method is generated, then `__post_init__()` will not automatically be called.

Among other uses, this allows for initializing field values that depend on one or more other fields. For example:

```
@dataclass
class C:
    a: float
    b: float
    c: float = field(init=False)

    def __post_init__(self):
        self.c = self.a + self.b
```

The `__init__()` method generated by *@dataclass* does not call base class `__init__()` methods. If the base class has an `__init__()` method that has to be called, it is common to call this method in a *__post_init__()* method:

```
class Rectangle:
    def __init__(self, height, width):
        self.height = height
        self.width = width

@dataclass
class Square(Rectangle):
    side: float

    def __post_init__(self):
        super().__init__(self.side, self.side)
```

Note, however, that in general the dataclass-generated `__init__()` methods don't need to be called, since the derived dataclass will take care of initializing all fields of any base class that is a dataclass itself.

See the section below on init-only variables for ways to pass parameters to `__post_init__()`. Also see the warning about how *replace()* handles `init=False` fields.

29.7.3 Class variables

One of the few places where *@dataclass* actually inspects the type of a field is to determine if a field is a class variable as defined in **PEP 526**. It does this by checking if the type of the field is *typing.ClassVar*. If a field is a `ClassVar`, it is excluded from consideration as a field and is ignored by the dataclass mechanisms. Such `ClassVar` pseudo-fields are not returned by the module-level *fields()* function.

29.7.4 Init-only variables

Another place where *@dataclass* inspects a type annotation is to determine if a field is an init-only variable. It does this by seeing if the type of a field is of type `dataclasses.InitVar`. If a field is an `InitVar`, it is considered a pseudo-field called an init-only field. As it is not a true field, it is not returned by the module-level *fields()* function. Init-only fields are added as parameters to the generated `__init__()` method, and are passed to the optional *__post_init__()* method. They are not otherwise used by dataclasses.

For example, suppose a field will be initialized from a database, if a value is not provided when creating the class:

```
@dataclass
class C:
    i: int
    j: int | None = None
    database: InitVar[DatabaseType | None] = None

    def __post_init__(self, database):
        if self.j is None and database is not None:
            self.j = database.lookup('j')

c = C(10, database=my_database)
```

In this case, *fields()* will return *Field* objects for `i` and `j`, but not for `database`.

29.7.5 Frozen instances

It is not possible to create truly immutable Python objects. However, by passing `frozen=True` to the *@dataclass* decorator you can emulate immutability. In that case, dataclasses will add `__setattr__()` and `__delattr__()` methods to the class. These methods will raise a *FrozenInstanceError* when invoked.

There is a tiny performance penalty when using `frozen=True`: `__init__()` cannot use simple assignment to initialize fields, and must use `object.__setattr__()`.

29.7.6 Inheritance

When the dataclass is being created by the *@dataclass* decorator, it looks through all of the class's base classes in reverse MRO (that is, starting at *object*) and, for each dataclass that it finds, adds the fields from that base class to an ordered mapping of fields. After all of the base class fields are added, it adds its own fields to the ordered mapping. All of the generated methods will use this combined, calculated ordered mapping of fields. Because the fields are in insertion order, derived classes override base classes. An example:

```
@dataclass
class Base:
    x: Any = 15.0
    y: int = 0
```

(continues on next page)

(continued from previous page)

```
@dataclass
class C(Base):
    z: int = 10
    x: int = 15
```

The final list of fields is, in order, `x`, `y`, `z`. The final type of `x` is *int*, as specified in class `C`.

The generated `__init__()` method for `C` will look like:

```
def __init__(self, x: int = 15, y: int = 0, z: int = 10):
```

29.7.7 Re-ordering of keyword-only parameters in `__init__()`

After the parameters needed for `__init__()` are computed, any keyword-only parameters are moved to come after all regular (non-keyword-only) parameters. This is a requirement of how keyword-only parameters are implemented in Python: they must come after non-keyword-only parameters.

In this example, `Base.y`, `Base.w`, and `D.t` are keyword-only fields, and `Base.x` and `D.z` are regular fields:

```
@dataclass
class Base:
    x: Any = 15.0
    _: KW_ONLY
    y: int = 0
    w: int = 1

@dataclass
class D(Base):
    z: int = 10
    t: int = field(kw_only=True, default=0)
```

The generated `__init__()` method for `D` will look like:

```
def __init__(self, x: Any = 15.0, z: int = 10, *, y: int = 0, w: int = 1, t: int = 0):
```

Note that the parameters have been re-ordered from how they appear in the list of fields: parameters derived from regular fields are followed by parameters derived from keyword-only fields.

The relative ordering of keyword-only parameters is maintained in the re-ordered `__init__()` parameter list.

29.7.8 Default factory functions

If a *field()* specifies a *default_factory*, it is called with zero arguments when a default value for the field is needed. For example, to create a new instance of a list, use:

```
mylist: list = field(default_factory=list)
```

If a field is excluded from `__init__()` (using `init=False`) and the field also specifies *default_factory*, then the default factory function will always be called from the generated `__init__()` function. This happens because there is no other way to give the field an initial value.

29.7.9 Mutable default values

Python stores default member variable values in class attributes. Consider this example, not using dataclasses:

```
class C:
    x = []
    def add(self, element):
        self.x.append(element)

o1 = C()
o2 = C()
o1.add(1)
o2.add(2)
assert o1.x == [1, 2]
assert o1.x is o2.x
```

Note that the two instances of class `C` share the same class variable `x`, as expected.

Using dataclasses, *if* this code was valid:

```
@dataclass
class D:
    x: list = []      # This code raises ValueError
    def add(self, element):
        self.x.append(element)
```

it would generate code similar to:

```
class D:
    x = []
    def __init__(self, x=x):
        self.x = x
    def add(self, element):
        self.x.append(element)

assert D().x is D().x
```

This has the same issue as the original example using class `C`. That is, two instances of class `D` that do not specify a value for `x` when creating a class instance will share the same copy of `x`. Because dataclasses just use normal Python class creation they also share this behavior. There is no general way for Data Classes to detect this condition. Instead, the *`@dataclass`* decorator will raise a *`ValueError`* if it detects an unhashable default parameter. The assumption is that if a value is unhashable, it is mutable. This is a partial solution, but it does protect against many common errors.

Using default factory functions is a way to create new instances of mutable types as default values for fields:

```
@dataclass
class D:
    x: list = field(default_factory=list)

assert D().x is not D().x
```

Changed in version 3.11: Instead of looking for and disallowing objects of type *`list`*, *`dict`*, or *`set`*, unhashable objects are now not allowed as default values. Unhashability is used to approximate mutability.

29.7.10 Descriptor-typed fields

Fields that are assigned descriptor objects as their default value have the following special behaviors:

- The value for the field passed to the dataclass's `__init__()` method is passed to the descriptor's `__set__()` method rather than overwriting the descriptor object.
- Similarly, when getting or setting the field, the descriptor's `__get__()` or `__set__()` method is called rather than returning or overwriting the descriptor object.
- To determine whether a field contains a default value, *@dataclass* will call the descriptor's `__get__()` method using its class access form: `descriptor.__get__(obj=None, type=cls)`. If the descriptor returns a value in this case, it will be used as the field's default. On the other hand, if the descriptor raises *AttributeError* in this situation, no default value will be provided for the field.

```
class IntConversionDescriptor:
    def __init__(self, *, default):
        self._default = default

    def __set_name__(self, owner, name):
        self._name = "_" + name

    def __get__(self, obj, type):
        if obj is None:
            return self._default

        return getattr(obj, self._name, self._default)

    def __set__(self, obj, value):
        setattr(obj, self._name, int(value))

@dataclass
class InventoryItem:
    quantity_on_hand: IntConversionDescriptor = IntConversionDescriptor(default=100)

i = InventoryItem()
print(i.quantity_on_hand)    # 100
i.quantity_on_hand = 2.5     # calls __set__ with 2.5
print(i.quantity_on_hand)    # 2
```

Note that if a field is annotated with a descriptor type, but is not assigned a descriptor object as its default value, the field will act like a normal field.

29.8 `contextlib` — Utilities for `with`-statement contexts

Source code: Lib/contextlib.py

This module provides utilities for common tasks involving the `with` statement. For more information see also *Context Manager Types* and context-managers.

29.8.1 Utilities

Functions and classes provided:

class contextlib.**AbstractContextManager**

An *abstract base class* for classes that implement `object.__enter__()` and `object.__exit__()`. A default implementation for `object.__enter__()` is provided which returns `self` while `object.__exit__()` is an abstract method which by default returns `None`. See also the definition of *Context Manager Types*.

Added in version 3.6.

class contextlib.**AbstractAsyncContextManager**

An *abstract base class* for classes that implement `object.__aenter__()` and `object.__aexit__()`. A default implementation for `object.__aenter__()` is provided which returns `self` while `object.__aexit__()` is an abstract method which by default returns `None`. See also the definition of async-context-managers.

Added in version 3.7.

@contextlib.**contextmanager**

This function is a *decorator* that can be used to define a factory function for `with` statement context managers, without needing to create a class or separate `__enter__()` and `__exit__()` methods.

While many objects natively support use in with statements, sometimes a resource needs to be managed that isn't a context manager in its own right, and doesn't implement a `close()` method for use with `contextlib.closing`

An abstract example would be the following to ensure correct resource management:

```
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwds):
    # Code to acquire resource, e.g.:
    resource = acquire_resource(*args, **kwds)
    try:
        yield resource
    finally:
        # Code to release resource, e.g.:
        release_resource(resource)
```

The function can then be used like this:

```
>>> with managed_resource(timeout=3600) as resource:
...     # Resource is released at the end of this block,
...     # even if code in the block raises an exception
```

The function being decorated must return a *generator*-iterator when called. This iterator must yield exactly one value, which will be bound to the targets in the `with` statement's `as` clause, if any.

At the point where the generator yields, the block nested in the `with` statement is executed. The generator is then resumed after the block is exited. If an unhandled exception occurs in the block, it is reraised inside the generator at the point where the yield occurred. Thus, you can use a `try...except...finally` statement to trap the error (if any), or ensure that some cleanup takes place. If an exception is trapped merely in order to log it or to perform some action (rather than to suppress it entirely), the generator must reraise that exception. Otherwise the generator context manager will indicate to the `with` statement that the exception has been handled, and execution will resume with the statement immediately following the `with` statement.

contextmanager() uses *ContextDecorator* so the context managers it creates can be used as decorators as well as in `with` statements. When used as a decorator, a new generator instance is implicitly created on each function call (this allows the otherwise "one-shot" context managers created by *contextmanager()* to meet the requirement that context managers support multiple invocations in order to be used as decorators).

Changed in version 3.2: Use of *ContextDecorator*.

`@contextlib.`**`asynccontextmanager`**

Similar to *contextmanager()*, but creates an asynchronous context manager.

This function is a *decorator* that can be used to define a factory function for `async with` statement asynchronous context managers, without needing to create a class or separate `__aenter__()` and `__aexit__()` methods. It must be applied to an *asynchronous generator* function.

A simple example:

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def get_connection():
    conn = await acquire_db_connection()
    try:
        yield conn
    finally:
        await release_db_connection(conn)

async def get_all_users():
    async with get_connection() as conn:
        return conn.query('SELECT ...')
```

Added in version 3.7.

Context managers defined with *asynccontextmanager()* can be used either as decorators or with `async with` statements:

```
import time
from contextlib import asynccontextmanager

@asynccontextmanager
async def timeit():
    now = time.monotonic()
    try:
        yield
    finally:
        print(f'it took {time.monotonic() - now}s to run')

@timeit()
async def main():
    # ... async code ...
```

When used as a decorator, a new generator instance is implicitly created on each function call. This allows the otherwise "one-shot" context managers created by *asynccontextmanager()* to meet the requirement that context managers support multiple invocations in order to be used as decorators.

Changed in version 3.10: Async context managers created with *asynccontextmanager()* can be used as decorators.

`contextlib.`**`closing`**(*thing*)

Return a context manager that closes *thing* upon completion of the block. This is basically equivalent to:

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

And lets you write code like this:

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('https://www.python.org')) as page:
    for line in page:
        print(line)
```

without needing to explicitly close `page`. Even if an error occurs, `page.close()` will be called when the `with` block is exited.

Note

Most types managing resources support the *context manager* protocol, which closes *thing* on leaving the `with` statement. As such, `closing()` is most useful for third party types that don't support context managers. This example is purely for illustration purposes, as *`urlopen()`* would normally be used in a context manager.

`contextlib.`**`aclosing`**(*thing*)

Return an async context manager that calls the `aclose()` method of *thing* upon completion of the block. This is basically equivalent to:

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def aclosing(thing):
    try:
        yield thing
    finally:
        await thing.aclose()
```

Significantly, `aclosing()` supports deterministic cleanup of async generators when they happen to exit early by `break` or an exception. For example:

```
from contextlib import aclosing

async with aclosing(my_generator()) as values:
    async for value in values:
        if value == 42:
            break
```

This pattern ensures that the generator's async exit code is executed in the same context as its iterations (so that exceptions and context variables work as expected, and the exit code isn't run after the lifetime of some task it depends on).

Added in version 3.10.

contextlib.**nullcontext**(*enter_result=None*)

Return a context manager that returns *enter_result* from `__enter__`, but otherwise does nothing. It is intended to be used as a stand-in for an optional context manager, for example:

```
def myfunction(arg, ignore_exceptions=False):
    if ignore_exceptions:
        # Use suppress to ignore all exceptions.
        cm = contextlib.suppress(Exception)
    else:
        # Do not ignore any exceptions, cm has no effect.
        cm = contextlib.nullcontext()
    with cm:
        # Do something
```

An example using *enter_result*:

```
def process_file(file_or_path):
    if isinstance(file_or_path, str):
        # If string, open file
        cm = open(file_or_path)
    else:
        # Caller is responsible for closing file
        cm = nullcontext(file_or_path)

    with cm as file:
        # Perform processing on the file
```

It can also be used as a stand-in for asynchronous context managers:

```
async def send_http(session=None):
    if not session:
        # If no http session, create it with aiohttp
        cm = aiohttp.ClientSession()
    else:
        # Caller is responsible for closing the session
        cm = nullcontext(session)

    async with cm as session:
        # Send http requests with session
```

Added in version 3.7.

Changed in version 3.10: *asynchronous context manager* support was added.

contextlib.**suppress**(**exceptions*)

Return a context manager that suppresses any of the specified exceptions if they occur in the body of a `with` statement and then resumes execution with the first statement following the end of the `with` statement.

As with any other mechanism that completely suppresses exceptions, this context manager should be used only to cover very specific errors where silently continuing with program execution is known to be the right thing to do.

For example:

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove('somefile.tmp')
```

(continues on next page)

(continued from previous page)

```
with suppress(FileNotFoundError):
    os.remove('someotherfile.tmp')
```

This code is equivalent to:

```
try:
    os.remove('somefile.tmp')
except FileNotFoundError:
    pass

try:
    os.remove('someotherfile.tmp')
except FileNotFoundError:
    pass
```

This context manager is *reentrant*.

If the code within the `with` block raises a *BaseExceptionGroup*, suppressed exceptions are removed from the group. Any exceptions of the group which are not suppressed are re-raised in a new group which is created using the original group's *derive()* method.

Added in version 3.4.

Changed in version 3.12: `suppress` now supports suppressing exceptions raised as part of a *BaseExceptionGroup*.

`contextlib.`**`redirect_stdout`**(*new_target*)

Context manager for temporarily redirecting *sys.stdout* to another file or file-like object.

This tool adds flexibility to existing functions or classes whose output is hardwired to stdout.

For example, the output of *help()* normally is sent to *sys.stdout*. You can capture that output in a string by redirecting the output to an *io.StringIO* object. The replacement stream is returned from the `__enter__` method and so is available as the target of the `with` statement:

```
with redirect_stdout(io.StringIO()) as f:
    help(pow)
s = f.getvalue()
```

To send the output of *help()* to a file on disk, redirect the output to a regular file:

```
with open('help.txt', 'w') as f:
    with redirect_stdout(f):
        help(pow)
```

To send the output of *help()* to *sys.stderr*:

```
with redirect_stdout(sys.stderr):
    help(pow)
```

Note that the global side effect on *sys.stdout* means that this context manager is not suitable for use in library code and most threaded applications. It also has no effect on the output of subprocesses. However, it is still a useful approach for many utility scripts.

This context manager is *reentrant*.

Added in version 3.4.

contextlib.**redirect_stderr**(*new_target*)

Similar to *redirect_stdout()* but redirecting *sys.stderr* to another file or file-like object.

This context manager is *reentrant*.

Added in version 3.5.

contextlib.**chdir**(*path*)

Non parallel-safe context manager to change the current working directory. As this changes a global state, the working directory, it is not suitable for use in most threaded or async contexts. It is also not suitable for most non-linear code execution, like generators, where the program execution is temporarily relinquished – unless explicitly desired, you should not yield when this context manager is active.

This is a simple wrapper around *chdir()*, it changes the current working directory upon entering and restores the old one on exit.

This context manager is *reentrant*.

Added in version 3.11.

class contextlib.**ContextDecorator**

A base class that enables a context manager to also be used as a decorator.

Context managers inheriting from `ContextDecorator` have to implement `__enter__` and `__exit__` as normal. `__exit__` retains its optional exception handling even when used as a decorator.

`ContextDecorator` is used by *contextmanager()*, so you get this functionality automatically.

Example of `ContextDecorator`:

```
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False
```

The class can then be used like this:

```
>>> @mycontext()
... def function():
...     print('The bit in the middle')
...
>>> function()
Starting
The bit in the middle
Finishing

>>> with mycontext():
...     print('The bit in the middle')
...
Starting
The bit in the middle
Finishing
```

This change is just syntactic sugar for any construct of the following form:

```
def f():
    with cm():
        # Do stuff
```

`ContextDecorator` lets you instead write:

```
@cm()
def f():
    # Do stuff
```

It makes it clear that the `cm` applies to the whole function, rather than just a piece of it (and saving an indentation level is nice, too).

Existing context managers that already have a base class can be extended by using `ContextDecorator` as a mixin class:

```
from contextlib import ContextDecorator

class mycontext(ContextBaseClass, ContextDecorator):
    def __enter__(self):
        return self

    def __exit__(self, *exc):
        return False
```

Note

As the decorated function must be able to be called multiple times, the underlying context manager must support use in multiple `with` statements. If this is not the case, then the original construct with the explicit `with` statement inside the function should be used.

Added in version 3.2.

class contextlib.**AsyncContextDecorator**

Similar to *ContextDecorator* but only for asynchronous functions.

Example of `AsyncContextDecorator`:

```
from asyncio import run
from contextlib import AsyncContextDecorator

class mycontext(AsyncContextDecorator):
    async def __aenter__(self):
        print('Starting')
        return self

    async def __aexit__(self, *exc):
        print('Finishing')
        return False
```

The class can then be used like this:

```
>>> @mycontext()
... async def function():
...     print('The bit in the middle')
...
```

(continues on next page)

(continued from previous page)

```
>>> run(function())
Starting
The bit in the middle
Finishing

>>> async def function():
...     async with mycontext():
...         print('The bit in the middle')
...
>>> run(function())
Starting
The bit in the middle
Finishing
```

Added in version 3.10.

class contextlib.**ExitStack**

A context manager that is designed to make it easy to programmatically combine other context managers and cleanup functions, especially those that are optional or otherwise driven by input data.

For example, a set of files may easily be handled in a single with statement as follows:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # All opened files will automatically be closed at the end of
    # the with statement, even if attempts to open files later
    # in the list raise an exception
```

The `__enter__()` method returns the *ExitStack* instance, and performs no additional operations.

Each instance maintains a stack of registered callbacks that are called in reverse order when the instance is closed (either explicitly or implicitly at the end of a `with` statement). Note that callbacks are *not* invoked implicitly when the context stack instance is garbage collected.

This stack model is used so that context managers that acquire their resources in their `__init__` method (such as file objects) can be handled correctly.

Since registered callbacks are invoked in the reverse order of registration, this ends up behaving as if multiple nested `with` statements had been used with the registered set of callbacks. This even extends to exception handling - if an inner callback suppresses or replaces an exception, then outer callbacks will be passed arguments based on that updated state.

This is a relatively low level API that takes care of the details of correctly unwinding the stack of exit callbacks. It provides a suitable foundation for higher level context managers that manipulate the exit stack in application specific ways.

Added in version 3.3.

enter_context(*cm*)

Enters a new context manager and adds its `__exit__()` method to the callback stack. The return value is the result of the context manager's own `__enter__()` method.

These context managers may suppress exceptions just as they normally would if used directly as part of a `with` statement.

Changed in version 3.11: Raises *TypeError* instead of *AttributeError* if *cm* is not a context manager.

push(*exit*)

Adds a context manager's `__exit__()` method to the callback stack.

As `__enter__` is *not* invoked, this method can be used to cover part of an `__enter__()` implementation with a context manager's own `__exit__()` method.

If passed an object that is not a context manager, this method assumes it is a callback with the same signature as a context manager's `__exit__()` method and adds it directly to the callback stack.

By returning true values, these callbacks can suppress exceptions the same way context manager `__exit__()` methods can.

The passed in object is returned from the function, allowing this method to be used as a function decorator.

callback(*callback*, */*, **args*, ***kwds*)

Accepts an arbitrary callback function and arguments and adds it to the callback stack.

Unlike the other methods, callbacks added this way cannot suppress exceptions (as they are never passed the exception details).

The passed in callback is returned from the function, allowing this method to be used as a function decorator.

pop_all()

Transfers the callback stack to a fresh *ExitStack* instance and returns it. No callbacks are invoked by this operation - instead, they will now be invoked when the new stack is closed (either explicitly or implicitly at the end of a `with` statement).

For example, a group of files can be opened as an "all or nothing" operation as follows:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # Hold onto the close method, but don't call it yet.
    close_files = stack.pop_all().close
    # If opening any file fails, all previously opened files will be
    # closed automatically. If all files are opened successfully,
    # they will remain open even after the with statement ends.
    # close_files() can then be invoked explicitly to close them all.
```

close()

Immediately unwinds the callback stack, invoking callbacks in the reverse order of registration. For any context managers and exit callbacks registered, the arguments passed in will indicate that no exception occurred.

class contextlib.**AsyncExitStack**

An asynchronous context manager, similar to *ExitStack*, that supports combining both synchronous and asynchronous context managers, as well as having coroutines for cleanup logic.

The *close()* method is not implemented; *aclose()* must be used instead.

coroutine enter_async_context(*cm*)

Similar to *ExitStack.enter_context()* but expects an asynchronous context manager.

Changed in version 3.11: Raises *TypeError* instead of *AttributeError* if *cm* is not an asynchronous context manager.

push_async_exit(*exit*)

Similar to *ExitStack.push()* but expects either an asynchronous context manager or a coroutine function.

push_async_callback(*callback*, */*, **args*, ***kwds*)

Similar to *ExitStack.callback()* but expects a coroutine function.

coroutine aclose()

Similar to *ExitStack.close()* but properly handles awaitables.

Continuing the example for *asynccontextmanager()*:

```
async with AsyncExitStack() as stack:
    connections = [await stack.enter_async_context(get_connection())
        for i in range(5)]
    # All opened connections will automatically be released at the end of
    # the async with statement, even if attempts to open a connection
    # later in the list raise an exception.
```

Added in version 3.7.

29.8.2 Examples and Recipes

This section describes some examples and recipes for making effective use of the tools provided by *contextlib*.

Supporting a variable number of context managers

The primary use case for *ExitStack* is the one given in the class documentation: supporting a variable number of context managers and other cleanup operations in a single `with` statement. The variability may come from the number of context managers needed being driven by user input (such as opening a user specified collection of files), or from some of the context managers being optional:

```
with ExitStack() as stack:
    for resource in resources:
        stack.enter_context(resource)
    if need_special_resource():
        special = acquire_special_resource()
        stack.callback(release_special_resource, special)
    # Perform operations that use the acquired resources
```

As shown, *ExitStack* also makes it quite easy to use `with` statements to manage arbitrary resources that don't natively support the context management protocol.

Catching exceptions from `__enter__` methods

It is occasionally desirable to catch exceptions from an `__enter__` method implementation, *without* inadvertently catching exceptions from the `with` statement body or the context manager's `__exit__` method. By using *ExitStack* the steps in the context management protocol can be separated slightly in order to allow this:

```
stack = ExitStack()
try:
    x = stack.enter_context(cm)
except Exception:
    # handle __enter__ exception
else:
    with stack:
        # Handle normal case
```

Actually needing to do this is likely to indicate that the underlying API should be providing a direct resource management interface for use with `try`/`except`/`finally` statements, but not all APIs are well designed in that regard. When a context manager is the only resource management API provided, then *ExitStack* can make it easier to handle various situations that can't be handled directly in a `with` statement.

Cleaning up in an `__enter__` implementation

As noted in the documentation of *`ExitStack.push()`*, this method can be useful in cleaning up an already allocated resource if later steps in the `__enter__()` implementation fail.

Here's an example of doing this for a context manager that accepts resource acquisition and release functions, along with an optional validation function, and maps them to the context management protocol:

```
from contextlib import contextmanager, AbstractContextManager, ExitStack

class ResourceManager(AbstractContextManager):

    def __init__(self, acquire_resource, release_resource, check_resource_ok=None):
        self.acquire_resource = acquire_resource
        self.release_resource = release_resource
        if check_resource_ok is None:
            def check_resource_ok(resource):
                return True
        self.check_resource_ok = check_resource_ok

    @contextmanager
    def _cleanup_on_error(self):
        with ExitStack() as stack:
            stack.push(self)
            yield
            # The validation check passed and didn't raise an exception
            # Accordingly, we want to keep the resource, and pass it
            # back to our caller
            stack.pop_all()

    def __enter__(self):
        resource = self.acquire_resource()
        with self._cleanup_on_error():
            if not self.check_resource_ok(resource):
                msg = "Failed validation for {!r}"
                raise RuntimeError(msg.format(resource))
        return resource

    def __exit__(self, *exc_details):
        # We don't need to duplicate any of our resource release logic
        self.release_resource()
```

Replacing any use of `try-finally` and flag variables

A pattern you will sometimes see is a `try-finally` statement with a flag variable to indicate whether or not the body of the `finally` clause should be executed. In its simplest form (that can't already be handled just by using an `except` clause instead), it looks something like this:

```
cleanup_needed = True
try:
    result = perform_operation()
    if result:
        cleanup_needed = False
finally:
    if cleanup_needed:
        cleanup_resources()
```

As with any `try` statement based code, this can cause problems for development and review, because the setup code and the cleanup code can end up being separated by arbitrarily long sections of code.

ExitStack makes it possible to instead register a callback for execution at the end of a `with` statement, and then later decide to skip executing that callback:

```
from contextlib import ExitStack

with ExitStack() as stack:
    stack.callback(cleanup_resources)
    result = perform_operation()
    if result:
        stack.pop_all()
```

This allows the intended cleanup behaviour to be made explicit up front, rather than requiring a separate flag variable.

If a particular application uses this pattern a lot, it can be simplified even further by means of a small helper class:

```
from contextlib import ExitStack

class Callback(ExitStack):
    def __init__(self, callback, /, *args, **kwds):
        super().__init__()
        self.callback(callback, *args, **kwds)

    def cancel(self):
        self.pop_all()

with Callback(cleanup_resources) as cb:
    result = perform_operation()
    if result:
        cb.cancel()
```

If the resource cleanup isn't already neatly bundled into a standalone function, then it is still possible to use the decorator form of *ExitStack.callback()* to declare the resource cleanup in advance:

```
from contextlib import ExitStack

with ExitStack() as stack:
    @stack.callback
    def cleanup_resources():
        ...
    result = perform_operation()
    if result:
        stack.pop_all()
```

Due to the way the decorator protocol works, a callback function declared this way cannot take any parameters. Instead, any resources to be released must be accessed as closure variables.

Using a context manager as a function decorator

ContextDecorator makes it possible to use a context manager in both an ordinary `with` statement and also as a function decorator.

For example, it is sometimes useful to wrap functions or groups of statements with a logger that can track the time of entry and time of exit. Rather than writing both a function decorator and a context manager for the task, inheriting from *ContextDecorator* provides both capabilities in a single definition:

```
from contextlib import ContextDecorator
import logging

logging.basicConfig(level=logging.INFO)

class track_entry_and_exit(ContextDecorator):
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        logging.info('Entering: %s', self.name)

    def __exit__(self, exc_type, exc, exc_tb):
        logging.info('Exiting: %s', self.name)
```

Instances of this class can be used as both a context manager:

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

And also as a function decorator:

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

Note that there is one additional limitation when using context managers as function decorators: there's no way to access the return value of `__enter__()`. If that value is needed, then it is still necessary to use an explicit `with` statement.

See also

PEP 343 - The "with" statement
The specification, background, and examples for the Python `with` statement.

29.8.3 Single use, reusable and reentrant context managers

Most context managers are written in a way that means they can only be used effectively in a `with` statement once. These single use context managers must be created afresh each time they're used - attempting to use them a second time will trigger an exception or otherwise not work correctly.

This common limitation means that it is generally advisable to create context managers directly in the header of the `with` statement where they are used (as shown in all of the usage examples above).

Files are an example of effectively single use context managers, since the first `with` statement will close the file, preventing any further IO operations using that file object.

Context managers created using *contextmanager()* are also single use context managers, and will complain about the underlying generator failing to yield if an attempt is made to use them a second time:

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def singleuse():
...     print("Before")
...     yield
...     print("After")
...
>>> cm = singleuse()
>>> with cm:
...     pass
...
Before
After
>>> with cm:
...     pass
...
Traceback (most recent call last):
    ...
RuntimeError: generator didn't yield
```

Reentrant context managers

More sophisticated context managers may be "reentrant". These context managers can not only be used in multiple `with` statements, but may also be used *inside* a `with` statement that is already using the same context manager.

threading.RLock is an example of a reentrant context manager, as are *suppress()*, *redirect_stdout()*, and *chdir()*. Here's a very simple example of reentrant use:

```
>>> from contextlib import redirect_stdout
>>> from io import StringIO
>>> stream = StringIO()
>>> write_to_stream = redirect_stdout(stream)
>>> with write_to_stream:
...     print("This is written to the stream rather than stdout")
...     with write_to_stream:
...         print("This is also written to the stream")
...
>>> print("This is written directly to stdout")
This is written directly to stdout
>>> print(stream.getvalue())
This is written to the stream rather than stdout
This is also written to the stream
```

Real world examples of reentrancy are more likely to involve multiple functions calling each other and hence be far more complicated than this example.

Note also that being reentrant is *not* the same thing as being thread safe. *redirect_stdout()*, for example, is definitely not thread safe, as it makes a global modification to the system state by binding *sys.stdout* to a different stream.

Reusable context managers

Distinct from both single use and reentrant context managers are "reusable" context managers (or, to be completely explicit, "reusable, but not reentrant" context managers, since reentrant context managers are also reusable). These context managers support being used multiple times, but will fail (or otherwise not work correctly) if the specific context manager instance has already been used in a containing with statement.

threading.Lock is an example of a reusable, but not reentrant, context manager (for a reentrant lock, it is necessary to use *threading.RLock* instead).

Another example of a reusable, but not reentrant, context manager is *ExitStack*, as it invokes *all* currently registered callbacks when leaving any with statement, regardless of where those callbacks were added:

```
>>> from contextlib import ExitStack
>>> stack = ExitStack()
>>> with stack:
...     stack.callback(print, "Callback: from first context")
...     print("Leaving first context")
...
Leaving first context
Callback: from first context
>>> with stack:
...     stack.callback(print, "Callback: from second context")
...     print("Leaving second context")
...
Leaving second context
Callback: from second context
>>> with stack:
...     stack.callback(print, "Callback: from outer context")
...     with stack:
...         stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Callback: from outer context
Leaving outer context
```

As the output from the example shows, reusing a single stack object across multiple with statements works correctly, but attempting to nest them will cause the stack to be cleared at the end of the innermost with statement, which is unlikely to be desirable behaviour.

Using separate *ExitStack* instances instead of reusing a single instance avoids that problem:

```
>>> from contextlib import ExitStack
>>> with ExitStack() as outer_stack:
...     outer_stack.callback(print, "Callback: from outer context")
...     with ExitStack() as inner_stack:
...         inner_stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Leaving outer context
Callback: from outer context
```

29.9 abc — Abstract Base Classes

Source code: Lib/abc.py

This module provides the infrastructure for defining *abstract base classes* (ABCs) in Python, as outlined in **PEP 3119**; see the PEP for why this was added to Python. (See also **PEP 3141** and the `numbers` module regarding a type hierarchy for numbers based on ABCs.)

The `collections` module has some concrete classes that derive from ABCs; these can, of course, be further derived. In addition, the `collections.abc` submodule has some ABCs that can be used to test whether a class or instance provides a particular interface, for example, if it is *hashable* or if it is a *mapping*.

This module provides the metaclass `ABCMeta` for defining ABCs and a helper class `ABC` to alternatively define ABCs through inheritance:

class abc.**ABC**

A helper class that has `ABCMeta` as its metaclass. With this class, an abstract base class can be created by simply deriving from `ABC` avoiding sometimes confusing metaclass usage, for example:

```
from abc import ABC

class MyABC(ABC):
    pass
```

Note that the type of `ABC` is still `ABCMeta`, therefore inheriting from `ABC` requires the usual precautions regarding metaclass usage, as multiple inheritance may lead to metaclass conflicts. One may also define an abstract base class by passing the metaclass keyword and using `ABCMeta` directly, for example:

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass
```

Added in version 3.4.

class abc.**ABCMeta**

Metaclass for defining Abstract Base Classes (ABCs).

Use this metaclass to create an ABC. An ABC can be subclassed directly, and then acts as a mix-in class. You can also register unrelated concrete classes (even built-in classes) and unrelated ABCs as "virtual subclasses" – these and their descendants will be considered subclasses of the registering ABC by the built-in `issubclass()` function, but the registering ABC won't show up in their MRO (Method Resolution Order) nor will method implementations defined by the registering ABC be callable (not even via `super()`).[1]

Classes created with a metaclass of `ABCMeta` have the following method:

register(*subclass*)

Register *subclass* as a "virtual subclass" of this ABC. For example:

```
from abc import ABC

class MyABC(ABC):
    pass
```

(continues on next page)

[1] C++ programmers should note that Python's virtual base class concept is not the same as C++'s.

(continued from previous page)

```
MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

Changed in version 3.3: Returns the registered subclass, to allow usage as a class decorator.

Changed in version 3.4: To detect calls to `register()`, you can use the *get_cache_token()* function.

You can also override this method in an abstract base class:

__subclasshook__(*subclass*)

(Must be defined as a class method.)

Check whether *subclass* is considered a subclass of this ABC. This means that you can customize the behavior of *issubclass()* further without the need to call *register()* on every class you want to consider a subclass of the ABC. (This class method is called from the `__subclasscheck__()` method of the ABC.)

This method should return `True`, `False` or *NotImplemented*. If it returns `True`, the *subclass* is considered a subclass of this ABC. If it returns `False`, the *subclass* is not considered a subclass of this ABC, even if it would normally be one. If it returns `NotImplemented`, the subclass check is continued with the usual mechanism.

For a demonstration of these concepts, look at this example ABC definition:

```
class Foo:
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)

class MyIterable(ABC):

    @abstractmethod
    def __iter__(self):
        while False:
            yield None

    def get_iterator(self):
        return self.__iter__()

    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented

MyIterable.register(Foo)
```

The ABC `MyIterable` defines the standard iterable method, *__iter__()*, as an abstract method. The implementation given here can still be called from subclasses. The `get_iterator()` method is also part of the `MyIterable` abstract base class, but it does not have to be overridden in non-abstract derived classes.

The *__subclasshook__()* class method defined here says that any class that has an *__iter__()* method in its *__dict__* (or in that of one of its base classes, accessed via the *__mro__* list) is considered a `MyIterable` too.

Finally, the last line makes `Foo` a virtual subclass of `MyIterable`, even though it does not define an *`__iter__()`* method (it uses the old-style iterable protocol, defined in terms of `__len__()` and `__getitem__()`). Note that this will not make `get_iterator` available as a method of `Foo`, so it is provided separately.

The `abc` module also provides the following decorator:

`@abc.`**`abstractmethod`**

A decorator indicating abstract methods.

Using this decorator requires that the class's metaclass is *`ABCMeta`* or is derived from it. A class that has a metaclass derived from `ABCMeta` cannot be instantiated unless all of its abstract methods and properties are overridden. The abstract methods can be called using any of the normal 'super' call mechanisms. `abstractmethod()` may be used to declare abstract methods for properties and descriptors.

Dynamically adding abstract methods to a class, or attempting to modify the abstraction status of a method or class once it is created, are only supported using the *`update_abstractmethods()`* function. The `abstractmethod()` only affects subclasses derived using regular inheritance; "virtual subclasses" registered with the ABC's *`register()`* method are not affected.

When `abstractmethod()` is applied in combination with other method descriptors, it should be applied as the innermost decorator, as shown in the following usage examples:

```
class C(ABC):
    @abstractmethod
    def my_abstract_method(self, arg1):
        ...
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg2):
        ...
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg3):
        ...

    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
    @my_abstract_property.setter
    @abstractmethod
    def my_abstract_property(self, val):
        ...

    @abstractmethod
    def _get_x(self):
        ...
    @abstractmethod
    def _set_x(self, val):
        ...
    x = property(_get_x, _set_x)
```

In order to correctly interoperate with the abstract base class machinery, the descriptor must identify itself as abstract using `__isabstractmethod__`. In general, this attribute should be `True` if any of the methods used to compose the descriptor are abstract. For example, Python's built-in *`property`* does the equivalent of:

```
class Descriptor:
    ...
    @property
    def __isabstractmethod__(self):
        return any(getattr(f, '__isabstractmethod__', False) for
                   f in (self._fget, self._fset, self._fdel))
```

Note

Unlike Java abstract methods, these abstract methods may have an implementation. This implementation can be called via the *super()* mechanism from the class that overrides it. This could be useful as an end-point for a super-call in a framework that uses cooperative multiple-inheritance.

The abc module also supports the following legacy decorators:

@abc.**abstractclassmethod**

Added in version 3.2.

Deprecated since version 3.3: It is now possible to use *classmethod* with *abstractmethod()*, making this decorator redundant.

A subclass of the built-in *classmethod()*, indicating an abstract classmethod. Otherwise it is similar to *abstractmethod()*.

This special case is deprecated, as the *classmethod()* decorator is now correctly identified as abstract when applied to an abstract method:

```
class C(ABC):
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg):
        ...
```

@abc.**abstractstaticmethod**

Added in version 3.2.

Deprecated since version 3.3: It is now possible to use *staticmethod* with *abstractmethod()*, making this decorator redundant.

A subclass of the built-in *staticmethod()*, indicating an abstract staticmethod. Otherwise it is similar to *abstractmethod()*.

This special case is deprecated, as the *staticmethod()* decorator is now correctly identified as abstract when applied to an abstract method:

```
class C(ABC):
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg):
        ...
```

@abc.**abstractproperty**

Deprecated since version 3.3: It is now possible to use *property*, *property.getter()*, *property.setter()* and *property.deleter()* with *abstractmethod()*, making this decorator redundant.

A subclass of the built-in *property()*, indicating an abstract property.

This special case is deprecated, as the *property()* decorator is now correctly identified as abstract when applied to an abstract method:

```
class C(ABC):
    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
```

The above example defines a read-only property; you can also define a read-write abstract property by appropriately marking one or more of the underlying methods as abstract:

```
class C(ABC):
    @property
    def x(self):
        ...

    @x.setter
    @abstractmethod
    def x(self, val):
        ...
```

If only some components are abstract, only those components need to be updated to create a concrete property in a subclass:

```
class D(C):
    @C.x.setter
    def x(self, val):
        ...
```

The `abc` module also provides the following functions:

`abc.`**`get_cache_token`**`()`

Returns the current abstract base class cache token.

The token is an opaque object (that supports equality testing) identifying the current version of the abstract base class cache for virtual subclasses. The token changes with every call to *ABCMeta.register()* on any ABC.

Added in version 3.4.

`abc.`**`update_abstractmethods`**`(`*cls*`)`

A function to recalculate an abstract class's abstraction status. This function should be called if a class's abstract methods have been implemented or changed after it was created. Usually, this function should be called from within a class decorator.

Returns *cls*, to allow usage as a class decorator.

If *cls* is not an instance of *ABCMeta*, does nothing.

Note

This function assumes that *cls*'s superclasses are already updated. It does not update any subclasses.

Added in version 3.10.

29.10 atexit — Exit handlers

The *atexit* module defines functions to register and unregister cleanup functions. Functions thus registered are automatically executed upon normal interpreter termination. *atexit* runs these functions in the *reverse* order in which they were registered; if you register A, B, and C, at interpreter termination time they will be run in the order C, B, A.

Note: The functions registered via this module are not called when the program is killed by a signal not handled by Python, when a Python fatal internal error is detected, or when *os._exit()* is called.

Note: The effect of registering or unregistering functions from within a cleanup function is undefined.

Changed in version 3.7: When used with C-API subinterpreters, registered functions are local to the interpreter they were registered in.

atexit.**register**(*func*, **args*, ***kwargs*)

Register *func* as a function to be executed at termination. Any optional arguments that are to be passed to *func* must be passed as arguments to *register()*. It is possible to register the same function and arguments more than once.

At normal program termination (for instance, if *sys.exit()* is called or the main module's execution completes), all functions registered are called in last in, first out order. The assumption is that lower level modules will normally be imported before higher level modules and thus must be cleaned up later.

If an exception is raised during execution of the exit handlers, a traceback is printed (unless *SystemExit* is raised) and the exception information is saved. After all exit handlers have had a chance to run, the last exception to be raised is re-raised.

This function returns *func*, which makes it possible to use it as a decorator.

> **Warning**
>
> Starting new threads or calling *os.fork()* from a registered function can lead to race condition between the main Python runtime thread freeing thread states while internal *threading* routines or the new process try to use that state. This can lead to crashes rather than clean shutdown.

Changed in version 3.12: Attempts to start a new thread or *os.fork()* a new process in a registered function now leads to *RuntimeError*.

atexit.**unregister**(*func*)

Remove *func* from the list of functions to be run at interpreter shutdown. *unregister()* silently does nothing if *func* was not previously registered. If *func* has been registered more than once, every occurrence of that function in the *atexit* call stack will be removed. Equality comparisons (==) are used internally during unregistration, so function references do not need to have matching identities.

> **See also**
>
> **Module *readline***
> Useful example of *atexit* to read and write *readline* history files.

29.10.1 `atexit` Example

The following simple example demonstrates how a module can initialize a counter from a file when it is imported and save the counter's updated value automatically when the program terminates without relying on the application making an explicit call into this module at termination.

```
try:
    with open('counterfile') as infile:
        _count = int(infile.read())
except FileNotFoundError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    with open('counterfile', 'w') as outfile:
        outfile.write('%d' % _count)

import atexit

atexit.register(savecounter)
```

Positional and keyword arguments may also be passed to *register()* to be passed along to the registered function when it is called:

```
def goodbye(name, adjective):
    print('Goodbye %s, it was %s to meet you.' % (name, adjective))

import atexit

atexit.register(goodbye, 'Donny', 'nice')
# or:
atexit.register(goodbye, adjective='nice', name='Donny')
```

Usage as a *decorator*:

```
import atexit

@atexit.register
def goodbye():
    print('You are now leaving the Python sector.')
```

This only works with functions that can be called without arguments.

29.11 `traceback` — Print or retrieve a stack traceback

Source code: Lib/traceback.py

This module provides a standard interface to extract, format and print stack traces of Python programs. It exactly mimics the behavior of the Python interpreter when it prints a stack trace. This is useful when you want to print stack traces under program control, such as in a "wrapper" around the interpreter.

The module uses traceback objects — these are objects of type *types.TracebackType*, which are assigned to the *__traceback__* field of *BaseException* instances.

See also

Module *faulthandler*
Used to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal.

Module *pdb*
Interactive source code debugger for Python programs.

The module defines the following functions:

traceback.**print_tb**(*tb*, *limit=None*, *file=None*)

Print up to *limit* stack trace entries from traceback object *tb* (starting from the caller's frame) if *limit* is positive. Otherwise, print the last `abs(limit)` entries. If *limit* is omitted or `None`, all entries are printed. If *file* is omitted or `None`, the output goes to *sys.stderr*; otherwise it should be an open *file* or *file-like object* to receive the output.

Changed in version 3.5: Added negative *limit* support.

traceback.**print_exception**(*exc*, /, [*value*, *tb*,]*limit=None*, *file=None*, *chain=True*)

Print exception information and stack trace entries from traceback object *tb* to *file*. This differs from *print_tb()* in the following ways:

- if *tb* is not `None`, it prints a header `Traceback (most recent call last):`
- it prints the exception type and *value* after the stack trace
- if *type(value)* is *SyntaxError* and *value* has the appropriate format, it prints the line where the syntax error occurred with a caret indicating the approximate position of the error.

Since Python 3.10, instead of passing *value* and *tb*, an exception object can be passed as the first argument. If *value* and *tb* are provided, the first argument is ignored in order to provide backwards compatibility.

The optional *limit* argument has the same meaning as for *print_tb()*. If *chain* is true (the default), then chained exceptions (the *__cause__* or *__context__* attributes of the exception) will be printed as well, like the interpreter itself does when printing an unhandled exception.

Changed in version 3.5: The *etype* argument is ignored and inferred from the type of *value*.

Changed in version 3.10: The *etype* parameter has been renamed to *exc* and is now positional-only.

traceback.**print_exc**(*limit=None*, *file=None*, *chain=True*)

This is a shorthand for `print_exception(sys.exception(), limit, file, chain)`.

traceback.**print_last**(*limit=None*, *file=None*, *chain=True*)

This is a shorthand for `print_exception(sys.last_exc, limit, file, chain)`. In general it will work only after an exception has reached an interactive prompt (see *sys.last_exc*).

traceback.**print_stack**(*f=None*, *limit=None*, *file=None*)

Print up to *limit* stack trace entries (starting from the invocation point) if *limit* is positive. Otherwise, print the last `abs(limit)` entries. If *limit* is omitted or `None`, all entries are printed. The optional *f* argument can be used to specify an alternate stack frame to start. The optional *file* argument has the same meaning as for *print_tb()*.

Changed in version 3.5: Added negative *limit* support.

traceback.**extract_tb**(*tb*, *limit=None*)

Return a *StackSummary* object representing a list of "pre-processed" stack trace entries extracted from the traceback object *tb*. It is useful for alternate formatting of stack traces. The optional *limit* argument has the same meaning as for *print_tb()*. A "pre-processed" stack trace entry is a *FrameSummary* object containing attributes *filename*, *lineno*, *name*, and *line* representing the information that is usually printed for a stack trace.

traceback.**extract_stack**(*f=None*, *limit=None*)

Extract the raw traceback from the current stack frame. The return value has the same format as for *extract_tb()*. The optional *f* and *limit* arguments have the same meaning as for *print_stack()*.

traceback.**format_list**(*extracted_list*)

Given a list of tuples or *FrameSummary* objects as returned by *extract_tb()* or *extract_stack()*, return a list of strings ready for printing. Each string in the resulting list corresponds to the item with the same index in the argument list. Each string ends in a newline; the strings may contain internal newlines as well, for those items whose source text line is not `None`.

traceback.**format_exception_only**(*exc*, */*, [*value*,]*, *show_group=False*)

Format the exception part of a traceback using an exception value such as given by *sys.last_value*. The return value is a list of strings, each ending in a newline. The list contains the exception's message, which is normally a single string; however, for *SyntaxError* exceptions, it contains several lines that (when printed) display detailed information about where the syntax error occurred. Following the message, the list contains the exception's *notes*.

Since Python 3.10, instead of passing *value*, an exception object can be passed as the first argument. If *value* is provided, the first argument is ignored in order to provide backwards compatibility.

When *show_group* is `True`, and the exception is an instance of *BaseExceptionGroup*, the nested exceptions are included as well, recursively, with indentation relative to their nesting depth.

Changed in version 3.10: The *etype* parameter has been renamed to *exc* and is now positional-only.

Changed in version 3.11: The returned list now includes any *notes* attached to the exception.

Changed in version 3.13: *show_group* parameter was added.

traceback.**format_exception**(*exc*, */*, [*value*, *tb*,]*limit=None*, *chain=True*)

Format a stack trace and the exception information. The arguments have the same meaning as the corresponding arguments to *print_exception()*. The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does *print_exception()*.

Changed in version 3.5: The *etype* argument is ignored and inferred from the type of *value*.

Changed in version 3.10: This function's behavior and signature were modified to match *print_exception()*.

traceback.**format_exc**(*limit=None*, *chain=True*)

This is like `print_exc(limit)` but returns a string instead of printing to a file.

traceback.**format_tb**(*tb*, *limit=None*)

A shorthand for `format_list(extract_tb(tb, limit))`.

traceback.**format_stack**(*f=None*, *limit=None*)

A shorthand for `format_list(extract_stack(f, limit))`.

traceback.**clear_frames**(*tb*)

Clears the local variables of all the stack frames in a traceback *tb* by calling the `clear()` method of each frame object.

Added in version 3.4.

traceback.**walk_stack**(*f*)

Walk a stack following `f.f_back` from the given frame, yielding the frame and line number for each frame. If *f* is `None`, the current stack is used. This helper is used with *StackSummary.extract()*.

Added in version 3.5.

traceback.**walk_tb**(*tb*)

Walk a traceback following `tb_next` yielding the frame and line number for each frame. This helper is used with *StackSummary.extract()*.

Added in version 3.5.

The module also defines the following classes:

29.11.1 `TracebackException` Objects

Added in version 3.5.

`TracebackException` objects are created from actual exceptions to capture data for later printing in a lightweight fashion.

class traceback.**TracebackException**(*exc_type*, *exc_value*, *exc_traceback*, ***, *limit=None*, *lookup_lines=True*, *capture_locals=False*, *compact=False*, *max_group_width=15*, *max_group_depth=10*)

Capture an exception for later rendering. *limit*, *lookup_lines* and *capture_locals* are as for the *StackSummary* class.

If *compact* is true, only data that is required by `TracebackException`'s *format()* method is saved in the class attributes. In particular, the *__context__* field is calculated only if *__cause__* is `None` and *__suppress_context__* is false.

Note that when locals are captured, they are also shown in the traceback.

max_group_width and *max_group_depth* control the formatting of exception groups (see *BaseExceptionGroup*). The depth refers to the nesting level of the group, and the width refers to the size of a single exception group's exceptions array. The formatted output is truncated when either limit is exceeded.

Changed in version 3.10: Added the *compact* parameter.

Changed in version 3.11: Added the *max_group_width* and *max_group_depth* parameters.

__cause__

A `TracebackException` of the original *__cause__*.

__context__

A `TracebackException` of the original *__context__*.

exceptions

If `self` represents an *ExceptionGroup*, this field holds a list of `TracebackException` instances representing the nested exceptions. Otherwise it is `None`.

Added in version 3.11.

__suppress_context__

The *__suppress_context__* value from the original exception.

__notes__

The *__notes__* value from the original exception, or `None` if the exception does not have any notes. If it is not `None` is it formatted in the traceback after the exception string.

Added in version 3.11.

stack

A *StackSummary* representing the traceback.

exc_type

The class of the original traceback.

Deprecated since version 3.13.

exc_type_str

String display of the class of the original exception.

Added in version 3.13.

filename

For syntax errors - the file name where the error occurred.

lineno

For syntax errors - the line number where the error occurred.

end_lineno

For syntax errors - the end line number where the error occurred. Can be `None` if not present.

Added in version 3.10.

text

For syntax errors - the text where the error occurred.

offset

For syntax errors - the offset into the text where the error occurred.

end_offset

For syntax errors - the end offset into the text where the error occurred. Can be `None` if not present.

Added in version 3.10.

msg

For syntax errors - the compiler error message.

classmethod from_exception(*exc*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

Capture an exception for later rendering. *limit*, *lookup_lines* and *capture_locals* are as for the *StackSummary* class.

Note that when locals are captured, they are also shown in the traceback.

print(*, *file=None*, *chain=True*)

Print to *file* (default `sys.stderr`) the exception information returned by *format()*.

Added in version 3.11.

format(*, *chain=True*)

Format the exception.

If *chain* is not `True`, *__cause__* and *__context__* will not be formatted.

The return value is a generator of strings, each ending in a newline and some containing internal newlines. *print_exception()* is a wrapper around this method which just prints the lines to a file.

format_exception_only(*, *show_group=False*)

Format the exception part of the traceback.

The return value is a generator of strings, each ending in a newline.

When *show_group* is `False`, the generator emits the exception's message followed by its notes (if it has any). The exception message is normally a single string; however, for *SyntaxError* exceptions, it consists of several lines that (when printed) display detailed information about where the syntax error occurred.

When *show_group* is `True`, and the exception is an instance of *BaseExceptionGroup*, the nested exceptions are included as well, recursively, with indentation relative to their nesting depth.

Changed in version 3.11: The exception's *notes* are now included in the output.

Changed in version 3.13: Added the *show_group* parameter.

29.11.2 StackSummary Objects

Added in version 3.5.

`StackSummary` objects represent a call stack ready for formatting.

class traceback.**StackSummary**

classmethod extract(*frame_gen*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

Construct a `StackSummary` object from a frame generator (such as is returned by *walk_stack()* or *walk_tb()*).

If *limit* is supplied, only this many frames are taken from *frame_gen*. If *lookup_lines* is `False`, the returned *FrameSummary* objects will not have read their lines in yet, making the cost of creating the `StackSummary` cheaper (which may be valuable if it may not actually get formatted). If *capture_locals* is `True` the local variables in each `FrameSummary` are captured as object representations.

Changed in version 3.12: Exceptions raised from *repr()* on a local variable (when *capture_locals* is `True`) are no longer propagated to the caller.

classmethod from_list(*a_list*)

Construct a `StackSummary` object from a supplied list of *FrameSummary* objects or old-style list of tuples. Each tuple should be a 4-tuple with *filename*, *lineno*, *name*, *line* as the elements.

format()

Returns a list of strings ready for printing. Each string in the resulting list corresponds to a single frame from the stack. Each string ends in a newline; the strings may contain internal newlines as well, for those items with source text lines.

For long sequences of the same frame and line, the first few repetitions are shown, followed by a summary line stating the exact number of further repetitions.

Changed in version 3.6: Long sequences of repeated frames are now abbreviated.

format_frame_summary(*frame_summary*)

Returns a string for printing one of the frames involved in the stack. This method is called for each *FrameSummary* object to be printed by *StackSummary.format()*. If it returns `None`, the frame is omitted from the output.

Added in version 3.11.

29.11.3 `FrameSummary` Objects

Added in version 3.5.

A `FrameSummary` object represents a single frame in a traceback.

class `traceback.`**FrameSummary**(*filename*, *lineno*, *name*, *lookup_line=True*, *locals=None*, *line=None*)

Represents a single frame in the traceback or stack that is being formatted or printed. It may optionally have a stringified version of the frame's locals included in it. If *lookup_line* is `False`, the source code is not looked up until the `FrameSummary` has the *line* attribute accessed (which also happens when casting it to a *tuple*). *line* may be directly provided, and will prevent line lookups happening at all. *locals* is an optional local variable mapping, and if supplied the variable representations are stored in the summary for later display.

`FrameSummary` instances have the following attributes:

filename

The filename of the source code for this frame. Equivalent to accessing `f.f_code.co_filename` on a frame object *f*.

lineno

The line number of the source code for this frame.

name

Equivalent to accessing `f.f_code.co_name` on a frame object *f*.

line

A string representing the source code for this frame, with leading and trailing whitespace stripped. If the source is not available, it is `None`.

29.11.4 Traceback Examples

This simple example implements a basic read-eval-print loop, similar to (but less useful than) the standard Python interactive interpreter loop. For a more complete implementation of the interpreter loop, refer to the *code* module.

```
import sys, traceback

def run_user_code(envdir):
    source = input(">>> ")
    try:
        exec(source, envdir)
    except Exception:
        print("Exception in user code:")
        print("-"*60)
        traceback.print_exc(file=sys.stdout)
        print("-"*60)

envdir = {}
while True:
    run_user_code(envdir)
```

The following example demonstrates the different ways to print and format the exception and traceback:

```
import sys, traceback

def lumberjack():
    bright_side_of_life()
```

(continues on next page)

(continued from previous page)

```
def bright_side_of_life():
    return tuple()[0]

try:
    lumberjack()
except IndexError:
    exc = sys.exception()
    print("*** print_tb:")
    traceback.print_tb(exc.__traceback__, limit=1, file=sys.stdout)
    print("*** print_exception:")
    traceback.print_exception(exc, limit=2, file=sys.stdout)
    print("*** print_exc:")
    traceback.print_exc(limit=2, file=sys.stdout)
    print("*** format_exc, first and last line:")
    formatted_lines = traceback.format_exc().splitlines()
    print(formatted_lines[0])
    print(formatted_lines[-1])
    print("*** format_exception:")
    print(repr(traceback.format_exception(exc)))
    print("*** extract_tb:")
    print(repr(traceback.extract_tb(exc.__traceback__)))
    print("*** format_tb:")
    print(repr(traceback.format_tb(exc.__traceback__)))
    print("*** tb_lineno:", exc.__traceback__.tb_lineno)
```

The output for the example would look similar to this:

```
*** print_tb:
  File "<doctest...>", line 10, in <module>
    lumberjack()
    ~~~~~~~~~~^^
*** print_exception:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
    ~~~~~~~~~~^^
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~^^
IndexError: tuple index out of range
*** print_exc:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
    ~~~~~~~~~~^^
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~^^
IndexError: tuple index out of range
*** format_exc, first and last line:
Traceback (most recent call last):
IndexError: tuple index out of range
*** format_exception:
['Traceback (most recent call last):\n',
 '  File "<doctest default[0]>", line 10, in <module>\n    lumberjack()\n    ~~~~~~~~~
↪~^^\n',
 '  File "<doctest default[0]>", line 4, in lumberjack\n    bright_side_of_life()\n  ␣
```

(continues on next page)

(continued from previous page)

```
↪ ~~~~~~~~~~~~~~~~~~^^\n',
 '  File "<doctest default[0]>", line 7, in bright_side_of_life\n    return␣
↪tuple()[0]\n           ~~~~~~~^^^\n',
 'IndexError: tuple index out of range\n']
*** extract_tb:
[<FrameSummary file <doctest...>, line 10 in <module>>,
 <FrameSummary file <doctest...>, line 4 in lumberjack>,
 <FrameSummary file <doctest...>, line 7 in bright_side_of_life>]
*** format_tb:
['  File "<doctest default[0]>", line 10, in <module>\n    lumberjack()\n    ~~~~~~~~~
↪~^^\n',
 '  File "<doctest default[0]>", line 4, in lumberjack\n    bright_side_of_life()\n  ␣
↪ ~~~~~~~~~~~~~~~~~~^^\n',
 '  File "<doctest default[0]>", line 7, in bright_side_of_life\n    return␣
↪tuple()[0]\n           ~~~~~~~^^^\n']
*** tb_lineno: 10
```

The following example shows the different ways to print and format the stack:

```
>>> import traceback
>>> def another_function():
...     lumberstack()
...
>>> def lumberstack():
...     traceback.print_stack()
...     print(repr(traceback.extract_stack()))
...     print(repr(traceback.format_stack()))
...
>>> another_function()
  File "<doctest>", line 10, in <module>
    another_function()
  File "<doctest>", line 3, in another_function
    lumberstack()
  File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('<doctest>', 10, '<module>', 'another_function()'),
 ('<doctest>', 3, 'another_function', 'lumberstack()'),
 ('<doctest>', 7, 'lumberstack', 'print(repr(traceback.extract_stack()))')]
['  File "<doctest>", line 10, in <module>\n    another_function()\n',
 '  File "<doctest>", line 3, in another_function\n    lumberstack()\n',
 '  File "<doctest>", line 8, in lumberstack\n    print(repr(traceback.format_
↪stack()))\n']
```

This last example demonstrates the final few formatting functions:

```
>>> import traceback
>>> traceback.format_list([('spam.py', 3, '<module>', 'spam.eggs()'),
...                        ('eggs.py', 42, 'eggs', 'return "bacon"')])
['  File "spam.py", line 3, in <module>\n    spam.eggs()\n',
 '  File "eggs.py", line 42, in eggs\n    return "bacon"\n']
>>> an_error = IndexError('tuple index out of range')
>>> traceback.format_exception_only(type(an_error), an_error)
['IndexError: tuple index out of range\n']
```

29.12 `__future__` — Future statement definitions

Source code: Lib/__future__.py

Imports of the form `from __future__ import feature` are called future statements. These are special-cased by the Python compiler to allow the use of new Python features in modules containing the future statement before the release in which the feature becomes standard.

While these future statements are given additional special meaning by the Python compiler, they are still executed like any other import statement and the *__future__* exists and is handled by the import system the same way any other Python module would be. This design serves three purposes:

- To avoid confusing existing tools that analyze import statements and expect to find the modules they're importing.
- To document when incompatible changes were introduced, and when they will be — or were — made mandatory. This is a form of executable documentation, and can be inspected programmatically via importing *__future__* and examining its contents.
- To ensure that future statements run under releases prior to Python 2.1 at least yield runtime exceptions (the import of *__future__* will fail, because there was no module of that name prior to 2.1).

29.12.1 Module Contents

No feature description will ever be deleted from *__future__*. Since its introduction in Python 2.1 the following features have found their way into the language using this mechanism:

feature	optional in	mandatory in	effect
nested_scopes	2.1.0b1	2.2	**PEP 227**: *Statically Nested Scopes*
generators	2.2.0a1	2.3	**PEP 255**: *Simple Generators*
division	2.2.0a2	3.0	**PEP 238**: *Changing the Division Operator*
absolute_import	2.5.0a1	3.0	**PEP 328**: *Imports: Multi-Line and Absolute/Relative*
with_statement	2.5.0a1	2.6	**PEP 343**: *The "with" Statement*
print_function	2.6.0a2	3.0	**PEP 3105**: *Make print a function*
unicode_literals	2.6.0a2	3.0	**PEP 3112**: *Bytes literals in Python 3000*
generator_stop	3.5.0b1	3.7	**PEP 479**: *StopIteration handling inside generators*
annotations	3.7.0b1	TBD[1]	**PEP 563**: *Postponed evaluation of annotations*

class `__future__._Feature`

Each statement in `__future__.py` is of the form:

```
FeatureName = _Feature(OptionalRelease, MandatoryRelease,
                       CompilerFlag)
```

where, normally, *OptionalRelease* is less than *MandatoryRelease*, and both are 5-tuples of the same form as *sys.version_info*:

[1] `from __future__ import annotations` was previously scheduled to become mandatory in Python 3.10, but the Python Steering Council twice decided to delay the change (announcement for Python 3.10; announcement for Python 3.11). No final decision has been made yet. See also **PEP 563** and **PEP 649**.

```
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
 PY_MINOR_VERSION, # the 1; an int
 PY_MICRO_VERSION, # the 0; an int
 PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; string
 PY_RELEASE_SERIAL # the 3; an int
)
```

_Feature.**getOptionalRelease**()

OptionalRelease records the first release in which the feature was accepted.

_Feature.**getMandatoryRelease**()

In the case of a *MandatoryRelease* that has not yet occurred, *MandatoryRelease* predicts the release in which the feature will become part of the language.

Else *MandatoryRelease* records when the feature became part of the language; in releases at or after that, modules no longer need a future statement to use the feature in question, but may continue to use such imports.

MandatoryRelease may also be `None`, meaning that a planned feature got dropped or that it is not yet decided.

_Feature.**compiler_flag**

CompilerFlag is the (bitfield) flag that should be passed in the fourth argument to the built-in function *compile()* to enable the feature in dynamically compiled code. This flag is stored in the *_Feature.compiler_flag* attribute on *_Feature* instances.

See also

future
How the compiler treats future imports.

PEP 236 - Back to the __future__
The original proposal for the __future__ mechanism.

29.13 gc — Garbage Collector interface

This module provides an interface to the optional garbage collector. It provides the ability to disable the collector, tune the collection frequency, and set debugging options. It also provides access to unreachable objects that the collector found but cannot free. Since the collector supplements the reference counting already used in Python, you can disable the collector if you are sure your program does not create reference cycles. Automatic collection can be disabled by calling `gc.disable()`. To debug a leaking program call `gc.set_debug(gc.DEBUG_LEAK)`. Notice that this includes `gc.DEBUG_SAVEALL`, causing garbage-collected objects to be saved in gc.garbage for inspection.

The *gc* module provides the following functions:

gc.**enable**()

Enable automatic garbage collection.

gc.**disable**()

Disable automatic garbage collection.

gc.**isenabled**()

Return `True` if automatic collection is enabled.

gc.**collect**(*generation=2*)

With no arguments, run a full collection. The optional argument *generation* may be an integer specifying which generation to collect (from 0 to 2). A *ValueError* is raised if the generation number is invalid. The sum of collected objects and uncollectable objects is returned.

The free lists maintained for a number of built-in types are cleared whenever a full collection or collection of the highest generation (2) is run. Not all items in some free lists may be freed due to the particular implementation, in particular *float*.

The effect of calling `gc.collect()` while the interpreter is already performing a collection is undefined.

gc.**set_debug**(*flags*)

Set the garbage collection debugging flags. Debugging information will be written to `sys.stderr`. See below for a list of debugging flags which can be combined using bit operations to control debugging.

gc.**get_debug**()

Return the debugging flags currently set.

gc.**get_objects**(*generation=None*)

Returns a list of all objects tracked by the collector, excluding the list returned. If *generation* is not `None`, return only the objects tracked by the collector that are in that generation.

Changed in version 3.8: New *generation* parameter.

Raises an *auditing event* `gc.get_objects` with argument `generation`.

gc.**get_stats**()

Return a list of three per-generation dictionaries containing collection statistics since interpreter start. The number of keys may change in the future, but currently each dictionary will contain the following items:

- `collections` is the number of times this generation was collected;
- `collected` is the total number of objects collected inside this generation;
- `uncollectable` is the total number of objects which were found to be uncollectable (and were therefore moved to the *garbage* list) inside this generation.

Added in version 3.4.

gc.**set_threshold**(*threshold0*[, *threshold1*[, *threshold2*]])

Set the garbage collection thresholds (the collection frequency). Setting *threshold0* to zero disables collection.

The GC classifies objects into three generations depending on how many collection sweeps they have survived. New objects are placed in the youngest generation (generation `0`). If an object survives a collection it is moved into the next older generation. Since generation `2` is the oldest generation, objects in that generation remain there after a collection. In order to decide when to run, the collector keeps track of the number object allocations and deallocations since the last collection. When the number of allocations minus the number of deallocations exceeds *threshold0*, collection starts. Initially only generation `0` is examined. If generation `0` has been examined more than *threshold1* times since generation `1` has been examined, then generation `1` is examined as well. With the third generation, things are a bit more complicated, see Collecting the oldest generation for more information.

gc.**get_count**()

Return the current collection counts as a tuple of `(count0, count1, count2)`.

gc.**get_threshold**()

Return the current collection thresholds as a tuple of `(threshold0, threshold1, threshold2)`.

gc.**get_referrers**(**objs*)

Return the list of objects that directly refer to any of objs. This function will only locate those containers which

support garbage collection; extension types which do refer to other objects but do not support garbage collection will not be found.

Note that objects which have already been dereferenced, but which live in cycles and have not yet been collected by the garbage collector can be listed among the resulting referrers. To get only currently live objects, call *collect()* before calling *get_referrers()*.

Warning

Care must be taken when using objects returned by *get_referrers()* because some of them could still be under construction and hence in a temporarily invalid state. Avoid using *get_referrers()* for any purpose other than debugging.

Raises an *auditing event* `gc.get_referrers` with argument `objs`.

`gc.`**`get_referents`**`(`*objs*`)`

Return a list of objects directly referred to by any of the arguments. The referents returned are those objects visited by the arguments' C-level `tp_traverse` methods (if any), and may not be all objects actually directly reachable. `tp_traverse` methods are supported only by objects that support garbage collection, and are only required to visit objects that may be involved in a cycle. So, for example, if an integer is directly reachable from an argument, that integer object may or may not appear in the result list.

Raises an *auditing event* `gc.get_referents` with argument `objs`.

`gc.`**`is_tracked`**`(`*obj*`)`

Returns `True` if the object is currently tracked by the garbage collector, `False` otherwise. As a general rule, instances of atomic types aren't tracked and instances of non-atomic types (containers, user-defined objects…) are. However, some type-specific optimizations can be present in order to suppress the garbage collector footprint of simple instances (e.g. dicts containing only atomic keys and values):

```
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
False
>>> gc.is_tracked({"a": []})
True
```

Added in version 3.1.

`gc.`**`is_finalized`**`(`*obj*`)`

Returns `True` if the given object has been finalized by the garbage collector, `False` otherwise.

```
>>> x = None
>>> class Lazarus:
...     def __del__(self):
...         global x
...         x = self
...
>>> lazarus = Lazarus()
>>> gc.is_finalized(lazarus)
```

(continues on next page)

(continued from previous page)

```
False
>>> del lazarus
>>> gc.is_finalized(x)
True
```

Added in version 3.9.

gc.**freeze**()

Freeze all the objects tracked by the garbage collector; move them to a permanent generation and ignore them in all the future collections.

If a process will `fork()` without `exec()`, avoiding unnecessary copy-on-write in child processes will maximize memory sharing and reduce overall memory usage. This requires both avoiding creation of freed "holes" in memory pages in the parent process and ensuring that GC collections in child processes won't touch the `gc_refs` counter of long-lived objects originating in the parent process. To accomplish both, call `gc.disable()` early in the parent process, `gc.freeze()` right before `fork()`, and `gc.enable()` early in child processes.

Added in version 3.7.

gc.**unfreeze**()

Unfreeze the objects in the permanent generation, put them back into the oldest generation.

Added in version 3.7.

gc.**get_freeze_count**()

Return the number of objects in the permanent generation.

Added in version 3.7.

The following variables are provided for read-only access (you can mutate the values but should not rebind them):

gc.**garbage**

A list of objects which the collector found to be unreachable but could not be freed (uncollectable objects). Starting with Python 3.4, this list should be empty most of the time, except when using instances of C extension types with a non-`NULL` `tp_del` slot.

If *DEBUG_SAVEALL* is set, then all unreachable objects will be added to this list rather than freed.

Changed in version 3.2: If this list is non-empty at *interpreter shutdown*, a *ResourceWarning* is emitted, which is silent by default. If *DEBUG_UNCOLLECTABLE* is set, in addition all uncollectable objects are printed.

Changed in version 3.4: Following **PEP 442**, objects with a `__del__()` method don't end up in *gc.garbage* anymore.

gc.**callbacks**

A list of callbacks that will be invoked by the garbage collector before and after collection. The callbacks will be called with two arguments, *phase* and *info*.

phase can be one of two values:

"start": The garbage collection is about to start.

"stop": The garbage collection has finished.

info is a dict providing more information for the callback. The following keys are currently defined:

"generation": The oldest generation being collected.

"collected": When *phase* is "stop", the number of objects successfully collected.

"uncollectable": When *phase* is "stop", the number of objects that could not be collected and were put in *garbage*.

Applications can add their own callbacks to this list. The primary use cases are:

Gathering statistics about garbage collection, such as how often various generations are collected, and how long the collection takes.

Allowing applications to identify and clear their own uncollectable types when they appear in *garbage*.

Added in version 3.3.

The following constants are provided for use with *set_debug()*:

gc.**DEBUG_STATS**

Print statistics during collection. This information can be useful when tuning the collection frequency.

gc.**DEBUG_COLLECTABLE**

Print information on collectable objects found.

gc.**DEBUG_UNCOLLECTABLE**

Print information of uncollectable objects found (objects which are not reachable but cannot be freed by the collector). These objects will be added to the `garbage` list.

Changed in version 3.2: Also print the contents of the *garbage* list at *interpreter shutdown*, if it isn't empty.

gc.**DEBUG_SAVEALL**

When set, all unreachable objects found will be appended to *garbage* rather than being freed. This can be useful for debugging a leaking program.

gc.**DEBUG_LEAK**

The debugging flags necessary for the collector to print information about a leaking program (equal to `DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_SAVEALL`).

29.14 `inspect` — Inspect live objects

Source code: Lib/inspect.py

The *inspect* module provides several useful functions to help get information about live objects such as modules, classes, methods, functions, tracebacks, frame objects, and code objects. For example, it can help you examine the contents of a class, retrieve the source code of a method, extract and format the argument list for a function, or get all the information you need to display a detailed traceback.

There are four main kinds of services provided by this module: type checking, getting source code, inspecting classes and functions, and examining the interpreter stack.

29.14.1 Types and members

The *getmembers()* function retrieves the members of an object such as a class or module. The functions whose names begin with "is" are mainly provided as convenient choices for the second argument to *getmembers()*. They also help you determine when you can expect to find the following special attributes (see import-mod-attrs for module attributes):

Type	Attribute	Description
class	__doc__	documentation string
	__name__	name with which this class was defined

continues on

Table 2 – continued from previous page

Type	Attribute	Description
	__qualname__	qualified name
	__module__	name of module in which this class was defined
	__type_params__	A tuple containing the type parameters of a generic class
method	__doc__	documentation string
	__name__	name with which this method was defined
	__qualname__	qualified name
	__func__	function object containing implementation of method
	__self__	instance to which this method is bound, or `None`
	__module__	name of module in which this method was defined
function	__doc__	documentation string
	__name__	name with which this function was defined
	__qualname__	qualified name
	__code__	code object containing compiled function *bytecode*
	__defaults__	tuple of any default values for positional or keyword parameters
	__kwdefaults__	mapping of any default values for keyword-only parameters
	__globals__	global namespace in which this function was defined
	__builtins__	builtins namespace
	__annotations__	mapping of parameters names to annotations; `"return"` key is reserved for return annotation
	__type_params__	A tuple containing the type parameters of a generic function
	__module__	name of module in which this function was defined
traceback	tb_frame	frame object at this level
	tb_lasti	index of last attempted instruction in bytecode
	tb_lineno	current line number in Python source code
	tb_next	next inner traceback object (called by this level)
frame	f_back	next outer frame object (this frame's caller)
	f_builtins	builtins namespace seen by this frame
	f_code	code object being executed in this frame
	f_globals	global namespace seen by this frame
	f_lasti	index of last attempted instruction in bytecode
	f_lineno	current line number in Python source code
	f_locals	local namespace seen by this frame
	f_trace	tracing function for this frame, or `None`
code	co_argcount	number of arguments (not including keyword only arguments, * or ** args)
	co_code	string of raw compiled bytecode
	co_cellvars	tuple of names of cell variables (referenced by containing scopes)
	co_consts	tuple of constants used in the bytecode
	co_filename	name of file in which this code object was created
	co_firstlineno	number of first line in Python source code
	co_flags	bitmap of `CO_*` flags, read more *here*
	co_lnotab	encoded mapping of line numbers to bytecode indices
	co_freevars	tuple of names of free variables (referenced via a function's closure)
	co_posonlyargcount	number of positional only arguments
	co_kwonlyargcount	number of keyword only arguments (not including ** arg)
	co_name	name with which this code object was defined
	co_qualname	fully qualified name with which this code object was defined
	co_names	tuple of names other than arguments and function locals
	co_nlocals	number of local variables
	co_stacksize	virtual machine stack space required
	co_varnames	tuple of names of arguments and local variables
generator	__name__	name

continues on

Table 2 – continued from previous page

Type	Attribute	Description
	__qualname__	qualified name
	gi_frame	frame
	gi_running	is the generator running?
	gi_code	code
	gi_yieldfrom	object being iterated by `yield from`, or `None`
async generator	__name__	name
	__qualname__	qualified name
	ag_await	object being awaited on, or `None`
	ag_frame	frame
	ag_running	is the generator running?
	ag_code	code
coroutine	__name__	name
	__qualname__	qualified name
	cr_await	object being awaited on, or `None`
	cr_frame	frame
	cr_running	is the coroutine running?
	cr_code	code
	cr_origin	where coroutine was created, or `None`. See *sys.set_coroutine_origin_tracking*
builtin	__doc__	documentation string
	__name__	original name of this function or method
	__qualname__	qualified name
	__self__	instance to which a method is bound, or `None`

Changed in version 3.5: Add `__qualname__` and `gi_yieldfrom` attributes to generators.

The `__name__` attribute of generators is now set from the function name, instead of the code name, and it can now be modified.

Changed in version 3.7: Add `cr_origin` attribute to coroutines.

Changed in version 3.10: Add `__builtins__` attribute to functions.

`inspect.`**`getmembers`**(*object*[, *predicate*])

Return all the members of an object in a list of `(name, value)` pairs sorted by name. If the optional *predicate* argument—which will be called with the `value` object of each member—is supplied, only members for which the predicate returns a true value are included.

> **Note**
>
> *getmembers()* will only return class attributes defined in the metaclass when the argument is a class and those attributes have been listed in the metaclass' custom `__dir__()`.

`inspect.`**`getmembers_static`**(*object*[, *predicate*])

Return all the members of an object in a list of `(name, value)` pairs sorted by name without triggering dynamic lookup via the descriptor protocol, __getattr__ or __getattribute__. Optionally, only return members that satisfy a given predicate.

> **Note**
>
> *getmembers_static()* may not be able to retrieve all members that getmembers can fetch (like dynamically created attributes) and may find members that getmembers can't (like descriptors that raise AttributeEr-

ror). It can also return descriptor objects instead of instance members in some cases.

Added in version 3.11.

inspect.**getmodulename**(*path*)

Return the name of the module named by the file *path*, without including the names of enclosing packages. The file extension is checked against all of the entries in *importlib.machinery.all_suffixes()*. If it matches, the final path component is returned with the extension removed. Otherwise, `None` is returned.

Note that this function *only* returns a meaningful name for actual Python modules - paths that potentially refer to Python packages will still return `None`.

Changed in version 3.3: The function is based directly on *importlib*.

inspect.**ismodule**(*object*)

Return `True` if the object is a module.

inspect.**isclass**(*object*)

Return `True` if the object is a class, whether built-in or created in Python code.

inspect.**ismethod**(*object*)

Return `True` if the object is a bound method written in Python.

inspect.**isfunction**(*object*)

Return `True` if the object is a Python function, which includes functions created by a *lambda* expression.

inspect.**isgeneratorfunction**(*object*)

Return `True` if the object is a Python generator function.

Changed in version 3.8: Functions wrapped in *functools.partial()* now return `True` if the wrapped function is a Python generator function.

Changed in version 3.13: Functions wrapped in *functools.partialmethod()* now return `True` if the wrapped function is a Python generator function.

inspect.**isgenerator**(*object*)

Return `True` if the object is a generator.

inspect.**iscoroutinefunction**(*object*)

Return `True` if the object is a *coroutine function* (a function defined with an `async def` syntax), a *functools.partial()* wrapping a *coroutine function*, or a sync function marked with *markcoroutinefunction()*.

Added in version 3.5.

Changed in version 3.8: Functions wrapped in *functools.partial()* now return `True` if the wrapped function is a *coroutine function*.

Changed in version 3.12: Sync functions marked with *markcoroutinefunction()* now return `True`.

Changed in version 3.13: Functions wrapped in *functools.partialmethod()* now return `True` if the wrapped function is a *coroutine function*.

inspect.**markcoroutinefunction**(*func*)

Decorator to mark a callable as a *coroutine function* if it would not otherwise be detected by *iscoroutinefunction()*.

This may be of use for sync functions that return a *coroutine*, if the function is passed to an API that requires *iscoroutinefunction()*.

When possible, using an `async def` function is preferred. Also acceptable is calling the function and testing the return with *`iscoroutine()`*.

Added in version 3.12.

`inspect.`**`iscoroutine`**(*object*)

Return `True` if the object is a *coroutine* created by an `async def` function.

Added in version 3.5.

`inspect.`**`isawaitable`**(*object*)

Return `True` if the object can be used in `await` expression.

Can also be used to distinguish generator-based coroutines from regular generators:

```
import types

def gen():
    yield
@types.coroutine
def gen_coro():
    yield

assert not isawaitable(gen())
assert isawaitable(gen_coro())
```

Added in version 3.5.

`inspect.`**`isasyncgenfunction`**(*object*)

Return `True` if the object is an *asynchronous generator* function, for example:

```
>>> async def agen():
...     yield 1
...
>>> inspect.isasyncgenfunction(agen)
True
```

Added in version 3.6.

Changed in version 3.8: Functions wrapped in *`functools.partial()`* now return `True` if the wrapped function is an *asynchronous generator* function.

Changed in version 3.13: Functions wrapped in *`functools.partialmethod()`* now return `True` if the wrapped function is a *coroutine function*.

`inspect.`**`isasyncgen`**(*object*)

Return `True` if the object is an *asynchronous generator iterator* created by an *asynchronous generator* function.

Added in version 3.6.

`inspect.`**`istraceback`**(*object*)

Return `True` if the object is a traceback.

`inspect.`**`isframe`**(*object*)

Return `True` if the object is a frame.

`inspect.`**`iscode`**(*object*)

Return `True` if the object is a code.

inspect.**isbuiltin**(*object*)

Return True if the object is a built-in function or a bound built-in method.

inspect.**ismethodwrapper**(*object*)

Return True if the type of object is a *MethodWrapperType*.

These are instances of *MethodWrapperType*, such as __str__(), __eq__() and __repr__().

Added in version 3.11.

inspect.**isroutine**(*object*)

Return True if the object is a user-defined or built-in function or method.

inspect.**isabstract**(*object*)

Return True if the object is an abstract base class.

inspect.**ismethoddescriptor**(*object*)

Return True if the object is a method descriptor, but not if *ismethod()*, *isclass()*, *isfunction()* or *isbuiltin()* are true.

This, for example, is true of int.__add__. An object passing this test has a __get__() method, but not a __set__() method or a __delete__() method. Beyond that, the set of attributes varies. A *__name__* attribute is usually sensible, and __doc__ often is.

Methods implemented via descriptors that also pass one of the other tests return False from the *ismethoddescriptor()* test, simply because the other tests promise more – you can, e.g., count on having the __func__ attribute (etc) when an object passes *ismethod()*.

Changed in version 3.13: This function no longer incorrectly reports objects with __get__() and __delete__(), but not __set__(), as being method descriptors (such objects are data descriptors, not method descriptors).

inspect.**isdatadescriptor**(*object*)

Return True if the object is a data descriptor.

Data descriptors have a __set__ or a __delete__ method. Examples are properties (defined in Python), getsets, and members. The latter two are defined in C and there are more specific tests available for those types, which is robust across Python implementations. Typically, data descriptors will also have *__name__* and __doc__ attributes (properties, getsets, and members have both of these attributes), but this is not guaranteed.

inspect.**isgetsetdescriptor**(*object*)

Return True if the object is a getset descriptor.

CPython implementation detail: getsets are attributes defined in extension modules via PyGetSetDef structures. For Python implementations without such types, this method will always return False.

inspect.**ismemberdescriptor**(*object*)

Return True if the object is a member descriptor.

CPython implementation detail: Member descriptors are attributes defined in extension modules via PyMemberDef structures. For Python implementations without such types, this method will always return False.

29.14.2 Retrieving source code

inspect.**getdoc**(*object*)

Get the documentation string for an object, cleaned up with *cleandoc()*. If the documentation string for an object is not provided and the object is a class, a method, a property or a descriptor, retrieve the documentation string from the inheritance hierarchy. Return None if the documentation string is invalid or missing.

Changed in version 3.5: Documentation strings are now inherited if not overridden.

inspect.**getcomments**(*object*)

Return in a single string any lines of comments immediately preceding the object's source code (for a class, function, or method), or at the top of the Python source file (if the object is a module). If the object's source code is unavailable, return None. This could happen if the object has been defined in C or the interactive shell.

inspect.**getfile**(*object*)

Return the name of the (text or binary) file in which an object was defined. This will fail with a *TypeError* if the object is a built-in module, class, or function.

inspect.**getmodule**(*object*)

Try to guess which module an object was defined in. Return None if the module cannot be determined.

inspect.**getsourcefile**(*object*)

Return the name of the Python source file in which an object was defined or None if no way can be identified to get the source. This will fail with a *TypeError* if the object is a built-in module, class, or function.

inspect.**getsourcelines**(*object*)

Return a list of source lines and starting line number for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of the lines corresponding to the object and the line number indicates where in the original source file the first line of code was found. An *OSError* is raised if the source code cannot be retrieved. A *TypeError* is raised if the object is a built-in module, class, or function.

Changed in version 3.3: *OSError* is raised instead of *IOError*, now an alias of the former.

inspect.**getsource**(*object*)

Return the text of the source code for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An *OSError* is raised if the source code cannot be retrieved. A *TypeError* is raised if the object is a built-in module, class, or function.

Changed in version 3.3: *OSError* is raised instead of *IOError*, now an alias of the former.

inspect.**cleandoc**(*doc*)

Clean up indentation from docstrings that are indented to line up with blocks of code.

All leading whitespace is removed from the first line. Any leading whitespace that can be uniformly removed from the second line onwards is removed. Empty lines at the beginning and end are subsequently removed. Also, all tabs are expanded to spaces.

29.14.3 Introspecting callables with the Signature object

Added in version 3.3.

The *Signature* object represents the call signature of a callable object and its return annotation. To retrieve a Signature object, use the signature() function.

inspect.**signature**(*callable*, *, *follow_wrapped=True*, *globals=None*, *locals=None*, *eval_str=False*)

Return a *Signature* object for the given *callable*:

```
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...     pass

>>> sig = signature(foo)

>>> str(sig)
'(a, *, b: int, **kwargs)'

>>> str(sig.parameters['b'])
'b: int'

>>> sig.parameters['b'].annotation
<class 'int'>
```

Accepts a wide range of Python callables, from plain functions and classes to *functools.partial()* objects.

For objects defined in modules using stringized annotations (from __future__ import annotations), *signature()* will attempt to automatically un-stringize the annotations using *get_annotations()*. The *globals*, *locals*, and *eval_str* parameters are passed into *get_annotations()* when resolving the annotations; see the documentation for *get_annotations()* for instructions on how to use these parameters.

Raises *ValueError* if no signature can be provided, and *TypeError* if that type of object is not supported. Also, if the annotations are stringized, and *eval_str* is not false, the eval() call(s) to un-stringize the annotations in *get_annotations()* could potentially raise any kind of exception.

A slash(/) in the signature of a function denotes that the parameters prior to it are positional-only. For more info, see the FAQ entry on positional-only parameters.

Changed in version 3.5: The *follow_wrapped* parameter was added. Pass False to get a signature of *callable* specifically (callable.__wrapped__ will not be used to unwrap decorated callables.)

Changed in version 3.10: The *globals*, *locals*, and *eval_str* parameters were added.

> **Note**
>
> Some callables may not be introspectable in certain implementations of Python. For example, in CPython, some built-in functions defined in C provide no metadata about their arguments.

CPython implementation detail: If the passed object has a __signature__ attribute, we may use it to create the signature. The exact semantics are an implementation detail and are subject to unannounced changes. Consult the source code for current semantics.

class inspect.**Signature**(*parameters=None*, *, *return_annotation=Signature.empty*)

A Signature object represents the call signature of a function and its return annotation. For each parameter accepted by the function it stores a *Parameter* object in its *parameters* collection.

The optional *parameters* argument is a sequence of *Parameter* objects, which is validated to check that there are no parameters with duplicate names, and that the parameters are in the right order, i.e. positional-only first, then positional-or-keyword, and that parameters with defaults follow parameters without defaults.

The optional *return_annotation* argument can be an arbitrary Python object. It represents the "return" annotation of the callable.

`Signature` objects are *immutable*. Use *Signature.replace()* or *copy.replace()* to make a modified copy.

Changed in version 3.5: `Signature` objects are now picklable and *hashable*.

empty

A special class-level marker to specify absence of a return annotation.

parameters

An ordered mapping of parameters' names to the corresponding *Parameter* objects. Parameters appear in strict definition order, including keyword-only parameters.

Changed in version 3.7: Python only explicitly guaranteed that it preserved the declaration order of keyword-only parameters as of version 3.7, although in practice this order had always been preserved in Python 3.

return_annotation

The "return" annotation for the callable. If the callable has no "return" annotation, this attribute is set to *Signature.empty*.

bind(**args*, ***kwargs*)

Create a mapping from positional and keyword arguments to parameters. Returns *BoundArguments* if `*args` and `**kwargs` match the signature, or raises a *TypeError*.

bind_partial(**args*, ***kwargs*)

Works the same way as *Signature.bind()*, but allows the omission of some required arguments (mimics *functools.partial()* behavior.) Returns *BoundArguments*, or raises a *TypeError* if the passed arguments do not match the signature.

replace(**[, parameters][, return_annotation]*)

Create a new *Signature* instance based on the instance *replace()* was invoked on. It is possible to pass different *parameters* and/or *return_annotation* to override the corresponding properties of the base signature. To remove `return_annotation` from the copied `Signature`, pass in *Signature.empty*.

```
>>> def test(a, b):
...     pass
...
>>> sig = signature(test)
>>> new_sig = sig.replace(return_annotation="new return anno")
>>> str(new_sig)
"(a, b) -> 'new return anno'"
```

Signature objects are also supported by the generic function *copy.replace()*.

format(**, max_width=None*)

Create a string representation of the *Signature* object.

If *max_width* is passed, the method will attempt to fit the signature into lines of at most *max_width* characters. If the signature is longer than *max_width*, all parameters will be on separate lines.

Added in version 3.13.

classmethod from_callable(*obj*, *, *follow_wrapped=True*, *globals=None*, *locals=None*, *eval_str=False*)

Return a *Signature* (or its subclass) object for a given callable *obj*.

This method simplifies subclassing of *Signature*:

```
class MySignature(Signature):
    pass
sig = MySignature.from_callable(sum)
assert isinstance(sig, MySignature)
```

Its behavior is otherwise identical to that of *signature()*.

Added in version 3.5.

Changed in version 3.10: The *globals*, *locals*, and *eval_str* parameters were added.

class inspect.**Parameter**(*name*, *kind*, *, *default=Parameter.empty*, *annotation=Parameter.empty*)

`Parameter` objects are *immutable*. Instead of modifying a `Parameter` object, you can use *Parameter.replace()* or *copy.replace()* to create a modified copy.

Changed in version 3.5: Parameter objects are now picklable and *hashable*.

empty

A special class-level marker to specify absence of default values and annotations.

name

The name of the parameter as a string. The name must be a valid Python identifier.

CPython implementation detail: CPython generates implicit parameter names of the form `.0` on the code objects used to implement comprehensions and generator expressions.

Changed in version 3.6: These parameter names are now exposed by this module as names like `implicit0`.

default

The default value for the parameter. If the parameter has no default value, this attribute is set to *Parameter.empty*.

annotation

The annotation for the parameter. If the parameter has no annotation, this attribute is set to *Parameter.empty*.

kind

Describes how argument values are bound to the parameter. The possible values are accessible via *Parameter* (like `Parameter.KEYWORD_ONLY`), and support comparison and ordering, in the following order:

Name	Meaning
POSITIONAL_ONLY	Value must be supplied as a positional argument. Positional only parameters are those which appear before a `/` entry (if present) in a Python function definition.
POSITIONAL_OR_KEYWORD	Value may be supplied as either a keyword or positional argument (this is the standard binding behaviour for functions implemented in Python.)
VAR_POSITIONAL	A tuple of positional arguments that aren't bound to any other parameter. This corresponds to a `*args` parameter in a Python function definition.
KEYWORD_ONLY	Value must be supplied as a keyword argument. Keyword only parameters are those which appear after a `*` or `*args` entry in a Python function definition.
VAR_KEYWORD	A dict of keyword arguments that aren't bound to any other parameter. This corresponds to a `**kwargs` parameter in a Python function definition.

Example: print all keyword-only arguments without default values:

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     if (param.kind == param.KEYWORD_ONLY and
...                       param.default is param.empty):
...         print('Parameter:', param)
Parameter: c
```

`kind.`**`description`**

Describes an enum value of *`Parameter.kind`*.

Added in version 3.8.

Example: print all descriptions of arguments:

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     print(param.kind.description)
positional or keyword
positional or keyword
keyword-only
keyword-only
```

`replace`(**[, name][, kind][, default][, annotation]*)

Create a new *`Parameter`* instance based on the instance replaced was invoked on. To override a `Parameter` attribute, pass the corresponding argument. To remove a default value or/and an annotation from a `Parameter`, pass *`Parameter.empty`*.

```
>>> from inspect import Parameter
>>> param = Parameter('foo', Parameter.KEYWORD_ONLY, default=42)
```

(continues on next page)

(continued from previous page)

```
>>> str(param)
'foo=42'

>>> str(param.replace()) # Will create a shallow copy of 'param'
'foo=42'

>>> str(param.replace(default=Parameter.empty, annotation='spam'))
"foo: 'spam'"
```

Parameter objects are also supported by the generic function *copy.replace()*.

Changed in version 3.4: In Python 3.3 *Parameter* objects were allowed to have `name` set to `None` if their `kind` was set to `POSITIONAL_ONLY`. This is no longer permitted.

class inspect.**BoundArguments**

Result of a *Signature.bind()* or *Signature.bind_partial()* call. Holds the mapping of arguments to the function's parameters.

arguments

A mutable mapping of parameters' names to arguments' values. Contains only explicitly bound arguments. Changes in *arguments* will reflect in *args* and *kwargs*.

Should be used in conjunction with *Signature.parameters* for any argument processing purposes.

> **Note**
>
> Arguments for which *Signature.bind()* or *Signature.bind_partial()* relied on a default value are skipped. However, if needed, use *BoundArguments.apply_defaults()* to add them.

Changed in version 3.9: *arguments* is now of type *dict*. Formerly, it was of type *collections.OrderedDict*.

args

A tuple of positional arguments values. Dynamically computed from the *arguments* attribute.

kwargs

A dict of keyword arguments values. Dynamically computed from the *arguments* attribute.

signature

A reference to the parent *Signature* object.

apply_defaults()

Set default values for missing arguments.

For variable-positional arguments (`*args`) the default is an empty tuple.

For variable-keyword arguments (`**kwargs`) the default is an empty dict.

```
>>> def foo(a, b='ham', *args): pass
>>> ba = inspect.signature(foo).bind('spam')
>>> ba.apply_defaults()
>>> ba.arguments
{'a': 'spam', 'b': 'ham', 'args': ()}
```

Added in version 3.5.

The *args* and *kwargs* properties can be used to invoke functions:

```
def test(a, *, b):
    ...

sig = signature(test)
ba = sig.bind(10, b=20)
test(*ba.args, **ba.kwargs)
```

See also

PEP 362 - Function Signature Object.
The detailed specification, implementation details and examples.

29.14.4 Classes and functions

`inspect.`**`getclasstree`**(*classes*, *unique=False*)

Arrange the given list of classes into a hierarchy of nested lists. Where a nested list appears, it contains classes derived from the class whose entry immediately precedes the list. Each entry is a 2-tuple containing a class and a tuple of its base classes. If the *unique* argument is true, exactly one entry appears in the returned structure for each class in the given list. Otherwise, classes using multiple inheritance and their descendants will appear multiple times.

`inspect.`**`getfullargspec`**(*func*)

Get the names and default values of a Python function's parameters. A *named tuple* is returned:

`FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults, annotations)`

args is a list of the positional parameter names. *varargs* is the name of the `*` parameter or `None` if arbitrary positional arguments are not accepted. *varkw* is the name of the `**` parameter or `None` if arbitrary keyword arguments are not accepted. *defaults* is an *n*-tuple of default argument values corresponding to the last *n* positional parameters, or `None` if there are no such defaults defined. *kwonlyargs* is a list of keyword-only parameter names in declaration order. *kwonlydefaults* is a dictionary mapping parameter names from *kwonlyargs* to the default values used if no argument is supplied. *annotations* is a dictionary mapping parameter names to annotations. The special key `"return"` is used to report the function return value annotation (if any).

Note that *`signature()`* and *Signature Object* provide the recommended API for callable introspection, and support additional behaviours (like positional-only arguments) that are sometimes encountered in extension module APIs. This function is retained primarily for use in code that needs to maintain compatibility with the Python 2 `inspect` module API.

Changed in version 3.4: This function is now based on *`signature()`*, but still ignores `__wrapped__` attributes and includes the already bound first parameter in the signature output for bound methods.

Changed in version 3.6: This method was previously documented as deprecated in favour of *`signature()`* in Python 3.5, but that decision has been reversed in order to restore a clearly supported standard interface for single-source Python 2/3 code migrating away from the legacy `getargspec()` API.

Changed in version 3.7: Python only explicitly guaranteed that it preserved the declaration order of keyword-only parameters as of version 3.7, although in practice this order had always been preserved in Python 3.

`inspect.`**`getargvalues`**(*frame*)

Get information about arguments passed into a particular frame. A *named tuple* `ArgInfo(args, varargs, keywords, locals)` is returned. *args* is a list of the argument names. *varargs* and *keywords* are the names of the `*` and `**` arguments or `None`. *locals* is the locals dictionary of the given frame.

> **Note**
>
> This function was inadvertently marked as deprecated in Python 3.5.

inspect.**formatargvalues**(*args*[, *varargs*, *varkw*, *locals*, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*])

Format a pretty argument spec from the four values returned by *getargvalues()*. The format* arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

> **Note**
>
> This function was inadvertently marked as deprecated in Python 3.5.

inspect.**getmro**(*cls*)

Return a tuple of class cls's base classes, including cls, in method resolution order. No class appears more than once in this tuple. Note that the method resolution order depends on cls's type. Unless a very peculiar user-defined metatype is in use, cls will be the first element of the tuple.

inspect.**getcallargs**(*func*, */*, **args*, ***kwds*)

Bind the *args* and *kwds* to the argument names of the Python function or method *func*, as if it was called with them. For bound methods, bind also the first argument (typically named `self`) to the associated instance. A dict is returned, mapping the argument names (including the names of the `*` and `**` arguments, if any) to their values from *args* and *kwds*. In case of invoking *func* incorrectly, i.e. whenever `func(*args, **kwds)` would raise an exception because of incompatible signature, an exception of the same type and the same or similar message is raised. For example:

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
...
>>> getcallargs(f, 1, 2, 3) == {'a': 1, 'named': {}, 'b': 2, 'pos': (3,)}
True
>>> getcallargs(f, a=2, x=4) == {'a': 2, 'named': {'x': 4}, 'b': 1, 'pos': ()}
True
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() missing 1 required positional argument: 'a'
```

Added in version 3.2.

Deprecated since version 3.5: Use *Signature.bind()* and *Signature.bind_partial()* instead.

inspect.**getclosurevars**(*func*)

Get the mapping of external name references in a Python function or method *func* to their current values. A *named tuple* `ClosureVars(nonlocals, globals, builtins, unbound)` is returned. *nonlocals* maps referenced names to lexical closure variables, *globals* to the function's module globals and *builtins* to the builtins visible from the function body. *unbound* is the set of names referenced in the function that could not be resolved at all given the current module globals and builtins.

TypeError is raised if *func* is not a Python function or method.

Added in version 3.3.

inspect.**unwrap**(*func*, *, *stop=None*)

Get the object wrapped by *func*. It follows the chain of __wrapped__ attributes returning the last object in the chain.

stop is an optional callback accepting an object in the wrapper chain as its sole argument that allows the unwrapping to be terminated early if the callback returns a true value. If the callback never returns a true value, the last object in the chain is returned as usual. For example, *signature()* uses this to stop unwrapping if any object in the chain has a __signature__ attribute defined.

ValueError is raised if a cycle is encountered.

Added in version 3.4.

inspect.**get_annotations**(*obj*, *, *globals=None*, *locals=None*, *eval_str=False*)

Compute the annotations dict for an object.

obj may be a callable, class, or module. Passing in an object of any other type raises *TypeError*.

Returns a dict. get_annotations() returns a new dict every time it's called; calling it twice on the same object will return two different but equivalent dicts.

This function handles several details for you:

- If eval_str is true, values of type str will be un-stringized using *eval()*. This is intended for use with stringized annotations (from __future__ import annotations).
- If obj doesn't have an annotations dict, returns an empty dict. (Functions and methods always have an annotations dict; classes, modules, and other types of callables may not.)
- Ignores inherited annotations on classes. If a class doesn't have its own annotations dict, returns an empty dict.
- All accesses to object members and dict values are done using getattr() and dict.get() for safety.
- Always, always, always returns a freshly created dict.

eval_str controls whether or not values of type str are replaced with the result of calling *eval()* on those values:

- If eval_str is true, *eval()* is called on values of type str. (Note that get_annotations doesn't catch exceptions; if *eval()* raises an exception, it will unwind the stack past the get_annotations call.)
- If eval_str is false (the default), values of type str are unchanged.

globals and locals are passed in to *eval()*; see the documentation for *eval()* for more information. If globals or locals is None, this function may replace that value with a context-specific default, contingent on type(obj):

- If obj is a module, globals defaults to obj.__dict__.
- If obj is a class, globals defaults to sys.modules[obj.__module__].__dict__ and locals defaults to the obj class namespace.
- If obj is a callable, globals defaults to obj.__globals__, although if obj is a wrapped function (using *functools.update_wrapper()*) it is first unwrapped.

Calling get_annotations is best practice for accessing the annotations dict of any object. See annotations-howto for more information on annotations best practices.

Added in version 3.10.

29.14.5 The interpreter stack

Some of the following functions return *FrameInfo* objects. For backwards compatibility these objects allow tuple-like operations on all attributes except `positions`. This behavior is considered deprecated and may be removed in the future.

class inspect.**FrameInfo**

frame

The frame object that the record corresponds to.

filename

The file name associated with the code being executed by the frame this record corresponds to.

lineno

The line number of the current line associated with the code being executed by the frame this record corresponds to.

function

The function name that is being executed by the frame this record corresponds to.

code_context

A list of lines of context from the source code that's being executed by the frame this record corresponds to.

index

The index of the current line being executed in the *code_context* list.

positions

A *dis.Positions* object containing the start line number, end line number, start column offset, and end column offset associated with the instruction being executed by the frame this record corresponds to.

Changed in version 3.5: Return a *named tuple* instead of a *tuple*.

Changed in version 3.11: `FrameInfo` is now a class instance (that is backwards compatible with the previous *named tuple*).

class inspect.**Traceback**

filename

The file name associated with the code being executed by the frame this traceback corresponds to.

lineno

The line number of the current line associated with the code being executed by the frame this traceback corresponds to.

function

The function name that is being executed by the frame this traceback corresponds to.

code_context

A list of lines of context from the source code that's being executed by the frame this traceback corresponds to.

index

The index of the current line being executed in the *code_context* list.

positions

A *dis.Positions* object containing the start line number, end line number, start column offset, and end column offset associated with the instruction being executed by the frame this traceback corresponds to.

Changed in version 3.11: `Traceback` is now a class instance (that is backwards compatible with the previous *named tuple*).

Note

Keeping references to frame objects, as found in the first element of the frame records these functions return, can cause your program to create reference cycles. Once a reference cycle has been created, the lifespan of all objects which can be accessed from the objects which form the cycle can become much longer even if Python's optional cycle detector is enabled. If such cycles must be created, it is important to ensure they are explicitly broken to avoid the delayed destruction of objects and increased memory consumption which occurs.

Though the cycle detector will catch these, destruction of the frames (and local variables) can be made deterministic by removing the cycle in a `finally` clause. This is also important if the cycle detector was disabled when Python was compiled or using *`gc.disable()`*. For example:

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

If you want to keep the frame around (for example to print a traceback later), you can also break reference cycles by using the `frame.clear()` method.

The optional *context* argument supported by most of these functions specifies the number of lines of context to return, which are centered around the current line.

`inspect.`**`getframeinfo`**(*frame*, *context=1*)

Get information about a frame or traceback object. A *`Traceback`* object is returned.

Changed in version 3.11: A *`Traceback`* object is returned instead of a named tuple.

`inspect.`**`getouterframes`**(*frame*, *context=1*)

Get a list of *`FrameInfo`* objects for a frame and all outer frames. These frames represent the calls that lead to the creation of *frame*. The first entry in the returned list represents *frame*; the last entry represents the outermost call on *frame*'s stack.

Changed in version 3.5: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

Changed in version 3.11: A list of *`FrameInfo`* objects is returned.

`inspect.`**`getinnerframes`**(*traceback*, *context=1*)

Get a list of *`FrameInfo`* objects for a traceback's frame and all inner frames. These frames represent calls made as a consequence of *frame*. The first entry in the list represents *traceback*; the last entry represents where the exception was raised.

Changed in version 3.5: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

Changed in version 3.11: A list of *`FrameInfo`* objects is returned.

`inspect.`**`currentframe`**()

Return the frame object for the caller's stack frame.

CPython implementation detail: This function relies on Python stack frame support in the interpreter, which isn't guaranteed to exist in all implementations of Python. If running in an implementation without Python stack frame support this function returns `None`.

inspect.**stack**(*context=1*)

Return a list of *FrameInfo* objects for the caller's stack. The first entry in the returned list represents the caller; the last entry represents the outermost call on the stack.

Changed in version 3.5: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

Changed in version 3.11: A list of *FrameInfo* objects is returned.

inspect.**trace**(*context=1*)

Return a list of *FrameInfo* objects for the stack between the current frame and the frame in which an exception currently being handled was raised in. The first entry in the list represents the caller; the last entry represents where the exception was raised.

Changed in version 3.5: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

Changed in version 3.11: A list of *FrameInfo* objects is returned.

29.14.6 Fetching attributes statically

Both *getattr()* and *hasattr()* can trigger code execution when fetching or checking for the existence of attributes. Descriptors, like properties, will be invoked and `__getattr__()` and `__getattribute__()` may be called.

For cases where you want passive introspection, like documentation tools, this can be inconvenient. *getattr_static()* has the same signature as *getattr()* but avoids executing code when it fetches attributes.

inspect.**getattr_static**(*obj*, *attr*, *default=None*)

Retrieve attributes without triggering dynamic lookup via the descriptor protocol, `__getattr__()` or `__getattribute__()`.

Note: this function may not be able to retrieve all attributes that getattr can fetch (like dynamically created attributes) and may find attributes that getattr can't (like descriptors that raise AttributeError). It can also return descriptors objects instead of instance members.

If the instance *__dict__* is shadowed by another member (for example a property) then this function will be unable to find instance members.

Added in version 3.2.

getattr_static() does not resolve descriptors, for example slot descriptors or getset descriptors on objects implemented in C. The descriptor object is returned instead of the underlying attribute.

You can handle these with code like the following. Note that for arbitrary getset descriptors invoking these may trigger code execution:

```
# example code for resolving the builtin descriptor types
class _foo:
    __slots__ = ['foo']

slot_descriptor = type(_foo.foo)
getset_descriptor = type(type(open(__file__)).name)
wrapper_descriptor = type(str.__dict__['__add__'])
descriptor_types = (slot_descriptor, getset_descriptor, wrapper_descriptor)

result = getattr_static(some_object, 'foo')
if type(result) in descriptor_types:
    try:
        result = result.__get__()
```

(continues on next page)

(continued from previous page)

```
    except AttributeError:
        # descriptors can raise AttributeError to
        # indicate there is no underlying value
        # in which case the descriptor itself will
        # have to do
        pass
```

29.14.7 Current State of Generators, Coroutines, and Asynchronous Generators

When implementing coroutine schedulers and for other advanced uses of generators, it is useful to determine whether a generator is currently executing, is waiting to start or resume or execution, or has already terminated. *getgeneratorstate()* allows the current state of a generator to be determined easily.

inspect.**getgeneratorstate**(*generator*)

Get current state of a generator-iterator.

Possible states are:

- GEN_CREATED: Waiting to start execution.
- GEN_RUNNING: Currently being executed by the interpreter.
- GEN_SUSPENDED: Currently suspended at a yield expression.
- GEN_CLOSED: Execution has completed.

Added in version 3.2.

inspect.**getcoroutinestate**(*coroutine*)

Get current state of a coroutine object. The function is intended to be used with coroutine objects created by `async def` functions, but will accept any coroutine-like object that has `cr_running` and `cr_frame` attributes.

Possible states are:

- CORO_CREATED: Waiting to start execution.
- CORO_RUNNING: Currently being executed by the interpreter.
- CORO_SUSPENDED: Currently suspended at an await expression.
- CORO_CLOSED: Execution has completed.

Added in version 3.5.

inspect.**getasyncgenstate**(*agen*)

Get current state of an asynchronous generator object. The function is intended to be used with asynchronous iterator objects created by `async def` functions which use the `yield` statement, but will accept any asynchronous generator-like object that has `ag_running` and `ag_frame` attributes.

Possible states are:

- AGEN_CREATED: Waiting to start execution.
- AGEN_RUNNING: Currently being executed by the interpreter.
- AGEN_SUSPENDED: Currently suspended at a yield expression.
- AGEN_CLOSED: Execution has completed.

Added in version 3.12.

The current internal state of the generator can also be queried. This is mostly useful for testing purposes, to ensure that internal state is being updated as expected:

inspect.**getgeneratorlocals**(*generator*)

Get the mapping of live local variables in *generator* to their current values. A dictionary is returned that maps from variable names to values. This is the equivalent of calling *locals()* in the body of the generator, and all the same caveats apply.

If *generator* is a *generator* with no currently associated frame, then an empty dictionary is returned. *TypeError* is raised if *generator* is not a Python generator object.

CPython implementation detail: This function relies on the generator exposing a Python stack frame for introspection, which isn't guaranteed to be the case in all implementations of Python. In such cases, this function will always return an empty dictionary.

Added in version 3.3.

inspect.**getcoroutinelocals**(*coroutine*)

This function is analogous to *getgeneratorlocals()*, but works for coroutine objects created by `async def` functions.

Added in version 3.5.

inspect.**getasyncgenlocals**(*agen*)

This function is analogous to *getgeneratorlocals()*, but works for asynchronous generator objects created by `async def` functions which use the `yield` statement.

Added in version 3.12.

29.14.8 Code Objects Bit Flags

Python code objects have a `co_flags` attribute, which is a bitmap of the following flags:

inspect.**CO_OPTIMIZED**

The code object is optimized, using fast locals.

inspect.**CO_NEWLOCALS**

If set, a new dict will be created for the frame's `f_locals` when the code object is executed.

inspect.**CO_VARARGS**

The code object has a variable positional parameter (`*args`-like).

inspect.**CO_VARKEYWORDS**

The code object has a variable keyword parameter (`**kwargs`-like).

inspect.**CO_NESTED**

The flag is set when the code object is a nested function.

inspect.**CO_GENERATOR**

The flag is set when the code object is a generator function, i.e. a generator object is returned when the code object is executed.

inspect.**CO_COROUTINE**

The flag is set when the code object is a coroutine function. When the code object is executed it returns a coroutine object. See **PEP 492** for more details.

Added in version 3.5.

inspect.**CO_ITERABLE_COROUTINE**

The flag is used to transform generators into generator-based coroutines. Generator objects with this flag can be used in `await` expression, and can `yield from` coroutine objects. See **PEP 492** for more details.

Added in version 3.5.

inspect.**CO_ASYNC_GENERATOR**

The flag is set when the code object is an asynchronous generator function. When the code object is executed it returns an asynchronous generator object. See **PEP 525** for more details.

Added in version 3.6.

Note

The flags are specific to CPython, and may not be defined in other Python implementations. Furthermore, the flags are an implementation detail, and can be removed or deprecated in future Python releases. It's recommended to use public APIs from the *inspect* module for any introspection needs.

29.14.9 Buffer flags

class inspect.**BufferFlags**

This is an *enum.IntFlag* that represents the flags that can be passed to the `__buffer__()` method of objects implementing the buffer protocol.

The meaning of the flags is explained at buffer-request-types.

SIMPLE

WRITABLE

FORMAT

ND

STRIDES

C_CONTIGUOUS

F_CONTIGUOUS

ANY_CONTIGUOUS

INDIRECT

CONTIG

CONTIG_RO

STRIDED

STRIDED_RO

RECORDS

RECORDS_RO

FULL

FULL_RO

READ

WRITE

Added in version 3.12.

29.14.10 Command Line Interface

The *inspect* module also provides a basic introspection capability from the command line.

By default, accepts the name of a module and prints the source of that module. A class or function within the module can be printed instead by appended a colon and the qualified name of the target object.

--details

Print information about the specified object rather than the source code

29.15 site — Site-specific configuration hook

Source code: Lib/site.py

This module is automatically imported during initialization. The automatic import can be suppressed using the interpreter's -S option.

Importing this module will append site-specific paths to the module search path and add a few builtins, unless -S was used. In that case, this module can be safely imported with no automatic modifications to the module search path or additions to the builtins. To explicitly trigger the usual site-specific additions, call the *main()* function.

Changed in version 3.3: Importing the module used to trigger paths manipulation even when using -S.

It starts by constructing up to four directories from a head and a tail part. For the head part, it uses sys.prefix and sys.exec_prefix; empty heads are skipped. For the tail part, it uses the empty string and then lib/site-packages (on Windows) or lib/python*X.Y*/site-packages (on Unix and macOS). For each of the distinct head-tail combinations, it sees if it refers to an existing directory, and if so, adds it to sys.path and also inspects the newly added path for configuration files.

Changed in version 3.5: Support for the "site-python" directory has been removed.

If a file named "pyvenv.cfg" exists one directory above sys.executable, sys.prefix and sys.exec_prefix are set to that directory and it is also checked for site-packages (sys.base_prefix and sys.base_exec_prefix will always be the "real" prefixes of the Python installation). If "pyvenv.cfg" (a bootstrap configuration file) contains the key "include-system-site-packages" set to anything other than "true" (case-insensitive), the system-level prefixes will not be searched for site-packages; otherwise they will.

A path configuration file is a file whose name has the form *name*.pth and exists in one of the four directories mentioned above; its contents are additional items (one per line) to be added to sys.path. Non-existing items are never added to sys.path, and no check is made that the item refers to a directory rather than a file. No item is added to sys.path more than once. Blank lines and lines beginning with # are skipped. Lines starting with import (followed by space or tab) are executed.

Note

An executable line in a `.pth` file is run at every Python startup, regardless of whether a particular module is actually going to be used. Its impact should thus be kept to a minimum. The primary intended purpose of executable lines is to make the corresponding module(s) importable (load 3rd-party import hooks, adjust `PATH` etc). Any other initialization is supposed to be done upon a module's actual import, if and when it happens. Limiting a code chunk to a single line is a deliberate measure to discourage putting anything more complex here.

Changed in version 3.13: The `.pth` files are now decoded by UTF-8 at first and then by the *locale encoding* if it fails.

For example, suppose `sys.prefix` and `sys.exec_prefix` are set to `/usr/local`. The Python X.Y library is then installed in `/usr/local/lib/pythonX.Y`. Suppose this has a subdirectory `/usr/local/lib/pythonX.Y/site-packages` with three subsubdirectories, `foo`, `bar` and `spam`, and two path configuration files, `foo.pth` and `bar.pth`. Assume `foo.pth` contains the following:

```
# foo package configuration

foo
bar
bletch
```

and `bar.pth` contains:

```
# bar package configuration

bar
```

Then the following version-specific directories are added to `sys.path`, in this order:

```
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

Note that `bletch` is omitted because it doesn't exist; the `bar` directory precedes the `foo` directory because `bar.pth` comes alphabetically before `foo.pth`; and `spam` is omitted because it is not mentioned in either path configuration file.

29.15.1 `sitecustomize`

After these path manipulations, an attempt is made to import a module named *`sitecustomize`*, which can perform arbitrary site-specific customizations. It is typically created by a system administrator in the site-packages directory. If this import fails with an *`ImportError`* or its subclass exception, and the exception's *`name`* attribute equals to `'sitecustomize'`, it is silently ignored. If Python is started without output streams available, as with `pythonw.exe` on Windows (which is used by default to start IDLE), attempted output from *`sitecustomize`* is ignored. Any other exception causes a silent and perhaps mysterious failure of the process.

29.15.2 usercustomize

After this, an attempt is made to import a module named *usercustomize*, which can perform arbitrary user-specific customizations, if *ENABLE_USER_SITE* is true. This file is intended to be created in the user site-packages directory (see below), which is part of `sys.path` unless disabled by `-s`. If this import fails with an *ImportError* or its subclass exception, and the exception's *name* attribute equals to `'usercustomize'`, it is silently ignored.

Note that for some non-Unix systems, `sys.prefix` and `sys.exec_prefix` are empty, and the path manipulations are skipped; however the import of *sitecustomize* and *usercustomize* is still attempted.

29.15.3 Readline configuration

On systems that support *readline*, this module will also import and configure the *rlcompleter* module, if Python is started in interactive mode and without the `-S` option. The default behavior is enable tab-completion and to use `~/.python_history` as the history save file. To disable it, delete (or override) the *sys.__interactivehook__* attribute in your *sitecustomize* or *usercustomize* module or your `PYTHONSTARTUP` file.

Changed in version 3.4: Activation of rlcompleter and history was made automatic.

29.15.4 Module contents

`site.`**`PREFIXES`**

A list of prefixes for site-packages directories.

`site.`**`ENABLE_USER_SITE`**

Flag showing the status of the user site-packages directory. `True` means that it is enabled and was added to `sys.path`. `False` means that it was disabled by user request (with `-s` or `PYTHONNOUSERSITE`). `None` means it was disabled for security reasons (mismatch between user or group id and effective id) or by an administrator.

`site.`**`USER_SITE`**

Path to the user site-packages for the running Python. Can be `None` if *getusersitepackages()* hasn't been called yet. Default value is `~/.local/lib/pythonX.Y/site-packages` for UNIX and non-framework macOS builds, `~/Library/Python/X.Y/lib/python/site-packages` for macOS framework builds, and `%APPDATA%\Python\PythonXY\site-packages` on Windows. This directory is a site directory, which means that `.pth` files in it will be processed.

`site.`**`USER_BASE`**

Path to the base directory for the user site-packages. Can be `None` if *getuserbase()* hasn't been called yet. Default value is `~/.local` for UNIX and macOS non-framework builds, `~/Library/Python/X.Y` for macOS framework builds, and `%APPDATA%\Python` for Windows. This value is used to compute the installation directories for scripts, data files, Python modules, etc. for the *user installation scheme*. See also `PYTHONUSERBASE`.

`site.`**`main`**`()`

Adds all the standard site-specific directories to the module search path. This function is called automatically when this module is imported, unless the Python interpreter was started with the `-S` flag.

Changed in version 3.3: This function used to be called unconditionally.

`site.`**`addsitedir`**`(`*sitedir*, *known_paths=None*`)`

Add a directory to sys.path and process its `.pth` files. Typically used in *sitecustomize* or *usercustomize* (see above).

site.**getsitepackages**()

Return a list containing all global site-packages directories.

Added in version 3.2.

site.**getuserbase**()

Return the path of the user base directory, *USER_BASE*. If it is not initialized yet, this function will also set it, respecting PYTHONUSERBASE.

Added in version 3.2.

site.**getusersitepackages**()

Return the path of the user-specific site-packages directory, *USER_SITE*. If it is not initialized yet, this function will also set it, respecting *USER_BASE*. To determine if the user-specific site-packages was added to sys.path *ENABLE_USER_SITE* should be used.

Added in version 3.2.

29.15.5 Command Line Interface

The *site* module also provides a way to get the user directories from the command line:

```
$ python -m site --user-site
/home/user/.local/lib/python3.11/site-packages
```

If it is called without arguments, it will print the contents of *sys.path* on the standard output, followed by the value of *USER_BASE* and whether the directory exists, then the same thing for *USER_SITE*, and finally the value of *ENABLE_USER_SITE*.

--user-base

Print the path to the user base directory.

--user-site

Print the path to the user site-packages directory.

If both options are given, user base and user site will be printed (always in this order), separated by *os.pathsep*.

If any option is given, the script will exit with one of these values: 0 if the user site-packages directory is enabled, 1 if it was disabled by the user, 2 if it is disabled for security reasons or by an administrator, and a value greater than 2 if there is an error.

See also

- **PEP 370** – Per user site-packages directory
- *The initialization of the sys.path module search path* – The initialization of *sys.path*.

CHAPTER

THIRTY

CUSTOM PYTHON INTERPRETERS

The modules described in this chapter allow writing interfaces similar to Python's interactive interpreter. If you want a Python interpreter that supports some special feature in addition to the Python language, you should look at the *code* module. (The *codeop* module is lower-level, used to support compiling a possibly incomplete chunk of Python code.)

The full list of modules described in this chapter is:

30.1 code — Interpreter base classes

Source code: Lib/code.py

The `code` module provides facilities to implement read-eval-print loops in Python. Two classes and convenience functions are included which can be used to build applications which provide an interactive interpreter prompt.

class `code.`**InteractiveInterpreter**(*locals=None*)

This class deals with parsing and interpreter state (the user's namespace); it does not deal with input buffering or prompting or input file naming (the filename is always passed in explicitly). The optional *locals* argument specifies a mapping to use as the namespace in which code will be executed; it defaults to a newly created dictionary with key `'__name__'` set to `'__console__'` and key `'__doc__'` set to `None`.

class `code.`**InteractiveConsole**(*locals=None*, *filename='<console>'*, *local_exit=False*)

Closely emulate the behavior of the interactive Python interpreter. This class builds on *InteractiveInterpreter* and adds prompting using the familiar `sys.ps1` and `sys.ps2`, and input buffering. If *local_exit* is true, `exit()` and `quit()` in the console will not raise *SystemExit*, but instead return to the calling code.

Changed in version 3.13: Added *local_exit* parameter.

`code.`**interact**(*banner=None*, *readfunc=None*, *local=None*, *exitmsg=None*, *local_exit=False*)

Convenience function to run a read-eval-print loop. This creates a new instance of *InteractiveConsole* and sets *readfunc* to be used as the *InteractiveConsole.raw_input()* method, if provided. If *local* is provided, it is passed to the *InteractiveConsole* constructor for use as the default namespace for the interpreter loop. If *local_exit* is provided, it is passed to the *InteractiveConsole* constructor. The *interact()* method of the instance is then run with *banner* and *exitmsg* passed as the banner and exit message to use, if provided. The console object is discarded after use.

Changed in version 3.6: Added *exitmsg* parameter.

Changed in version 3.13: Added *local_exit* parameter.

code.**compile_command**(*source*, *filename='<input>'*, *symbol='single'*)

This function is useful for programs that want to emulate Python's interpreter main loop (a.k.a. the read-eval-print loop). The tricky part is to determine when the user has entered an incomplete command that can be completed by entering more text (as opposed to a complete command or a syntax error). This function *almost* always makes the same decision as the real interpreter main loop.

source is the source string; *filename* is the optional filename from which source was read, defaulting to `'<input>'`; and *symbol* is the optional grammar start symbol, which should be `'single'` (the default), `'eval'` or `'exec'`.

Returns a code object (the same as `compile(source, filename, symbol)`) if the command is complete and valid; `None` if the command is incomplete; raises *SyntaxError* if the command is complete and contains a syntax error, or raises *OverflowError* or *ValueError* if the command contains an invalid literal.

30.1.1 Interactive Interpreter Objects

InteractiveInterpreter.**runsource**(*source*, *filename='<input>'*, *symbol='single'*)

Compile and run some source in the interpreter. Arguments are the same as for *compile_command()*; the default for *filename* is `'<input>'`, and for *symbol* is `'single'`. One of several things can happen:

- The input is incorrect; *compile_command()* raised an exception (*SyntaxError* or *OverflowError*). A syntax traceback will be printed by calling the *showsyntaxerror()* method. *runsource()* returns `False`.
- The input is incomplete, and more input is required; *compile_command()* returned `None`. *runsource()* returns `True`.
- The input is complete; *compile_command()* returned a code object. The code is executed by calling the *runcode()* (which also handles run-time exceptions, except for *SystemExit*). *runsource()* returns `False`.

The return value can be used to decide whether to use `sys.ps1` or `sys.ps2` to prompt the next line.

InteractiveInterpreter.**runcode**(*code*)

Execute a code object. When an exception occurs, *showtraceback()* is called to display a traceback. All exceptions are caught except *SystemExit*, which is allowed to propagate.

A note about *KeyboardInterrupt*: this exception may occur elsewhere in this code, and may not always be caught. The caller should be prepared to deal with it.

InteractiveInterpreter.**showsyntaxerror**(*filename=None*)

Display the syntax error that just occurred. This does not display a stack trace because there isn't one for syntax errors. If *filename* is given, it is stuffed into the exception instead of the default filename provided by Python's parser, because it always uses `'<string>'` when reading from a string. The output is written by the *write()* method.

InteractiveInterpreter.**showtraceback**()

Display the exception that just occurred. We remove the first stack item because it is within the interpreter object implementation. The output is written by the *write()* method.

Changed in version 3.5: The full chained traceback is displayed instead of just the primary traceback.

InteractiveInterpreter.**write**(*data*)

Write a string to the standard error stream (`sys.stderr`). Derived classes should override this to provide the appropriate output handling as needed.

30.1.2 Interactive Console Objects

The *InteractiveConsole* class is a subclass of *InteractiveInterpreter*, and so offers all the methods of the interpreter objects as well as the following additions.

InteractiveConsole.**interact**(*banner=None*, *exitmsg=None*)

Closely emulate the interactive Python console. The optional *banner* argument specify the banner to print before the first interaction; by default it prints a banner similar to the one printed by the standard Python interpreter, followed by the class name of the console object in parentheses (so as not to confuse this with the real interpreter – since it's so close!).

The optional *exitmsg* argument specifies an exit message printed when exiting. Pass the empty string to suppress the exit message. If *exitmsg* is not given or `None`, a default message is printed.

Changed in version 3.4: To suppress printing any banner, pass an empty string.

Changed in version 3.6: Print an exit message when exiting.

InteractiveConsole.**push**(*line*)

Push a line of source text to the interpreter. The line should not have a trailing newline; it may have internal newlines. The line is appended to a buffer and the interpreter's *runsource()* method is called with the concatenated contents of the buffer as source. If this indicates that the command was executed or invalid, the buffer is reset; otherwise, the command is incomplete, and the buffer is left as it was after the line was appended. The return value is `True` if more input is required, `False` if the line was dealt with in some way (this is the same as `runsource()`).

InteractiveConsole.**resetbuffer**()

Remove any unhandled source text from the input buffer.

InteractiveConsole.**raw_input**(*prompt=''*)

Write a prompt and read a line. The returned line does not include the trailing newline. When the user enters the EOF key sequence, *EOFError* is raised. The base implementation reads from `sys.stdin`; a subclass may replace this with a different implementation.

30.2 codeop — Compile Python code

Source code: Lib/codeop.py

The *codeop* module provides utilities upon which the Python read-eval-print loop can be emulated, as is done in the *code* module. As a result, you probably don't want to use the module directly; if you want to include such a loop in your program you probably want to use the *code* module instead.

There are two parts to this job:

1. Being able to tell if a line of input completes a Python statement: in short, telling whether to print '>>>' or '...' next.
2. Remembering which future statements the user has entered, so subsequent input can be compiled with these in effect.

The *codeop* module provides a way of doing each of these things, and a way of doing them both.

To do just the former:

codeop.**compile_command**(*source*, *filename='<input>'*, *symbol='single'*)

Tries to compile *source*, which should be a string of Python code and return a code object if *source* is valid Python code. In that case, the filename attribute of the code object will be *filename*, which defaults to `'<input>'`. Returns `None` if *source* is *not* valid Python code, but is a prefix of valid Python code.

If there is a problem with *source*, an exception will be raised. *SyntaxError* is raised if there is invalid Python syntax, and *OverflowError* or *ValueError* if there is an invalid literal.

The *symbol* argument determines whether *source* is compiled as a statement (`'single'`, the default), as a sequence of *statement* (`'exec'`) or as an *expression* (`'eval'`). Any other value will cause *ValueError* to be raised.

> **Note**
>
> It is possible (but not likely) that the parser stops parsing with a successful outcome before reaching the end of the source; in this case, trailing symbols may be ignored instead of causing an error. For example, a backslash followed by two newlines may be followed by arbitrary garbage. This will be fixed once the API for the parser is better.

class codeop.**Compile**

Instances of this class have `__call__()` methods identical in signature to the built-in function *compile()*, but with the difference that if the instance compiles program text containing a *__future__* statement, the instance 'remembers' and compiles all subsequent program texts with the statement in force.

class codeop.**CommandCompiler**

Instances of this class have `__call__()` methods identical in signature to *compile_command()*; the difference is that if the instance compiles program text containing a *__future__* statement, the instance 'remembers' and compiles all subsequent program texts with the statement in force.

CHAPTER

THIRTYONE

IMPORTING MODULES

The modules described in this chapter provide new ways to import other Python modules and hooks for customizing the import process.

The full list of modules described in this chapter is:

31.1 `zipimport` — Import modules from Zip archives

Source code: Lib/zipimport.py

This module adds the ability to import Python modules (`*.py`, `*.pyc`) and packages from ZIP-format archives. It is usually not needed to use the *`zipimport`* module explicitly; it is automatically used by the built-in `import` mechanism for *`sys.path`* items that are paths to ZIP archives.

Typically, *`sys.path`* is a list of directory names as strings. This module also allows an item of *`sys.path`* to be a string naming a ZIP file archive. The ZIP archive can contain a subdirectory structure to support package imports, and a path within the archive can be specified to only import from a subdirectory. For example, the path `example.zip/lib/` would only import from the `lib/` subdirectory within the archive.

Any files may be present in the ZIP archive, but importers are only invoked for `.py` and `.pyc` files. ZIP import of dynamic modules (`.pyd`, `.so`) is disallowed. Note that if an archive only contains `.py` files, Python will not attempt to modify the archive by adding the corresponding `.pyc` file, meaning that if a ZIP archive doesn't contain `.pyc` files, importing may be rather slow.

Changed in version 3.13: ZIP64 is supported

Changed in version 3.8: Previously, ZIP archives with an archive comment were not supported.

See also

PKZIP Application Note
Documentation on the ZIP file format by Phil Katz, the creator of the format and algorithms used.

PEP 273 - Import Modules from Zip Archives
Written by James C. Ahlstrom, who also provided an implementation. Python 2.3 follows the specification in PEP 273, but uses an implementation written by Just van Rossum that uses the import hooks described in PEP 302.

***`importlib`* - The implementation of the import machinery**
Package providing the relevant protocols for all importers to implement.

This module defines an exception:

exception zipimport.**ZipImportError**

Exception raised by zipimporter objects. It's a subclass of *ImportError*, so it can be caught as *ImportError*, too.

31.1.1 zipimporter Objects

zipimporter is the class for importing ZIP files.

class zipimport.**zipimporter**(*archivepath*)

Create a new zipimporter instance. *archivepath* must be a path to a ZIP file, or to a specific path within a ZIP file. For example, an *archivepath* of `foo/bar.zip/lib` will look for modules in the `lib` directory inside the ZIP file `foo/bar.zip` (provided that it exists).

ZipImportError is raised if *archivepath* doesn't point to a valid ZIP archive.

Changed in version 3.12: Methods `find_loader()` and `find_module()`, deprecated in 3.10 are now removed. Use *find_spec()* instead.

create_module(*spec*)

Implementation of *importlib.abc.Loader.create_module()* that returns *None* to explicitly request the default semantics.

Added in version 3.10.

exec_module(*module*)

Implementation of *importlib.abc.Loader.exec_module()*.

Added in version 3.10.

find_spec(*fullname*, *target=None*)

An implementation of *importlib.abc.PathEntryFinder.find_spec()*.

Added in version 3.10.

get_code(*fullname*)

Return the code object for the specified module. Raise *ZipImportError* if the module couldn't be imported.

get_data(*pathname*)

Return the data associated with *pathname*. Raise *OSError* if the file wasn't found.

Changed in version 3.3: *IOError* used to be raised, it is now an alias of *OSError*.

get_filename(*fullname*)

Return the value `__file__` would be set to if the specified module was imported. Raise *ZipImportError* if the module couldn't be imported.

Added in version 3.1.

get_source(*fullname*)

Return the source code for the specified module. Raise *ZipImportError* if the module couldn't be found, return *None* if the archive does contain the module, but has no source for it.

is_package(*fullname*)

Return `True` if the module specified by *fullname* is a package. Raise *ZipImportError* if the module couldn't be found.

load_module(*fullname*)

Load the module specified by *fullname*. *fullname* must be the fully qualified (dotted) module name. Returns the imported module on success, raises *ZipImportError* on failure.

Deprecated since version 3.10: Use *exec_module()* instead.

invalidate_caches()

Clear out the internal cache of information about files found within the ZIP archive.

Added in version 3.10.

archive

The file name of the importer's associated ZIP file, without a possible subpath.

prefix

The subpath within the ZIP file where modules are searched. This is the empty string for zipimporter objects which point to the root of the ZIP file.

The *archive* and *prefix* attributes, when combined with a slash, equal the original *archivepath* argument given to the *zipimporter* constructor.

31.1.2 Examples

Here is an example that imports a module from a ZIP archive - note that the *zipimport* module is not explicitly used.

```
$ unzip -l example.zip
Archive:  example.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
     8467  11-26-02 22:30   jwzthreading.py
 --------                   -------
     8467                   1 file
$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, 'example.zip')  # Add .zip file to front of path
>>> import jwzthreading
>>> jwzthreading.__file__
'example.zip/jwzthreading.py'
```

31.2 pkgutil — Package extension utility

Source code: Lib/pkgutil.py

This module provides utilities for the import system, in particular package support.

class pkgutil.**ModuleInfo**(*module_finder*, *name*, *ispkg*)

A namedtuple that holds a brief summary of a module's info.

Added in version 3.6.

pkgutil.**extend_path**(*path*, *name*)

Extend the search path for the modules which comprise a package. Intended use is to place the following code in a package's __init__.py:

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

For each directory on *sys.path* that has a subdirectory that matches the package name, add the subdirectory to the package's __path__. This is useful if one wants to distribute different parts of a single logical package as multiple directories.

It also looks for *.pkg files beginning where * matches the *name* argument. This feature is similar to *.pth files (see the *site* module for more information), except that it doesn't special-case lines starting with import. A *.pkg file is trusted at face value: apart from skipping blank lines and ignoring comments, all entries found in a *.pkg file are added to the path, regardless of whether they exist on the filesystem (this is a feature).

If the input path is not a list (as is the case for frozen packages) it is returned unchanged. The input path is not modified; an extended copy is returned. Items are only appended to the copy at the end.

It is assumed that *sys.path* is a sequence. Items of *sys.path* that are not strings referring to existing directories are ignored. Unicode items on *sys.path* that cause errors when used as filenames may cause this function to raise an exception (in line with *os.path.isdir()* behavior).

pkgutil.**find_loader**(*fullname*)

Retrieve a module *loader* for the given *fullname*.

This is a backwards compatibility wrapper around *importlib.util.find_spec()* that converts most failures to *ImportError* and only returns the loader rather than the full *importlib.machinery.ModuleSpec*.

Changed in version 3.3: Updated to be based directly on *importlib* rather than relying on the package internal **PEP 302** import emulation.

Changed in version 3.4: Updated to be based on **PEP 451**

Deprecated since version 3.12, will be removed in version 3.14: Use *importlib.util.find_spec()* instead.

pkgutil.**get_importer**(*path_item*)

Retrieve a *finder* for the given *path_item*.

The returned finder is cached in *sys.path_importer_cache* if it was newly created by a path hook.

The cache (or part of it) can be cleared manually if a rescan of *sys.path_hooks* is necessary.

Changed in version 3.3: Updated to be based directly on *importlib* rather than relying on the package internal **PEP 302** import emulation.

pkgutil.**get_loader**(*module_or_name*)

Get a *loader* object for *module_or_name*.

If the module or package is accessible via the normal import mechanism, a wrapper around the relevant part of that machinery is returned. Returns None if the module cannot be found or imported. If the named module is not already imported, its containing package (if any) is imported, in order to establish the package __path__.

Changed in version 3.3: Updated to be based directly on *importlib* rather than relying on the package internal **PEP 302** import emulation.

Changed in version 3.4: Updated to be based on **PEP 451**

Deprecated since version 3.12, will be removed in version 3.14: Use *importlib.util.find_spec()* instead.

pkgutil.**iter_importers**(*fullname=''*)

Yield *finder* objects for the given module name.

If fullname contains a `'.'`, the finders will be for the package containing fullname, otherwise they will be all registered top level finders (i.e. those on both *sys.meta_path* and *sys.path_hooks*).

If the named module is in a package, that package is imported as a side effect of invoking this function.

If no module name is specified, all top level finders are produced.

Changed in version 3.3: Updated to be based directly on *importlib* rather than relying on the package internal **PEP 302** import emulation.

pkgutil.**iter_modules**(*path=None*, *prefix=''*)

Yields *ModuleInfo* for all submodules on *path*, or, if *path* is `None`, all top-level modules on *sys.path*.

path should be either `None` or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

> **Note**
>
> Only works for a *finder* which defines an `iter_modules()` method. This interface is non-standard, so the module also provides implementations for *importlib.machinery.FileFinder* and *zipimport.zipimporter*.

Changed in version 3.3: Updated to be based directly on *importlib* rather than relying on the package internal **PEP 302** import emulation.

pkgutil.**walk_packages**(*path=None*, *prefix=''*, *onerror=None*)

Yields *ModuleInfo* for all modules recursively on *path*, or, if *path* is `None`, all accessible modules.

path should be either `None` or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

Note that this function must import all *packages* (*not* all modules!) on the given *path*, in order to access the `__path__` attribute to find submodules.

onerror is a function which gets called with one argument (the name of the package which was being imported) if any exception occurs while trying to import a package. If no *onerror* function is supplied, *ImportError*s are caught and ignored, while all other exceptions are propagated, terminating the search.

Examples:

```
# list all modules python can access
walk_packages()

# list all submodules of ctypes
walk_packages(ctypes.__path__, ctypes.__name__ + '.')
```

> **Note**
>
> Only works for a *finder* which defines an `iter_modules()` method. This interface is non-standard, so the module also provides implementations for *importlib.machinery.FileFinder* and *zipimport.zipimporter*.

Changed in version 3.3: Updated to be based directly on *importlib* rather than relying on the package internal **PEP 302** import emulation.

pkgutil.**get_data**(*package*, *resource*)

Get a resource from a package.

This is a wrapper for the *loader* *get_data* API. The *package* argument should be the name of a package, in standard module format (`foo.bar`). The *resource* argument should be in the form of a relative filename, using `/` as the path separator. The parent directory name `..` is not allowed, and nor is a rooted name (starting with a `/`).

The function returns a binary string that is the contents of the specified resource.

For packages located in the filesystem, which have already been imported, this is the rough equivalent of:

```
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()
```

If the package cannot be located or loaded, or it uses a *loader* which does not support *get_data*, then `None` is returned. In particular, the *loader* for *namespace packages* does not support *get_data*.

pkgutil.**resolve_name**(*name*)

Resolve a name to an object.

This functionality is used in numerous places in the standard library (see bpo-12915) - and equivalent functionality is also in widely used third-party packages such as setuptools, Django and Pyramid.

It is expected that *name* will be a string in one of the following formats, where W is shorthand for a valid Python identifier and dot stands for a literal period in these pseudo-regexes:

- `W(.W)*`
- `W(.W)*:(W(.W)*)?`

The first form is intended for backward compatibility only. It assumes that some part of the dotted name is a package, and the rest is an object somewhere within that package, possibly nested inside other objects. Because the place where the package stops and the object hierarchy starts can't be inferred by inspection, repeated attempts to import must be done with this form.

In the second form, the caller makes the division point clear through the provision of a single colon: the dotted name to the left of the colon is a package to be imported, and the dotted name to the right is the object hierarchy within that package. Only one import is needed in this form. If it ends with the colon, then a module object is returned.

The function will return an object (which might be a module), or raise one of the following exceptions:

ValueError – if *name* isn't in a recognised format.

ImportError – if an import failed when it shouldn't have.

AttributeError – If a failure occurred when traversing the object hierarchy within the imported package to get to the desired object.

Added in version 3.9.

31.3 modulefinder — Find modules used by a script

Source code: Lib/modulefinder.py

This module provides a *ModuleFinder* class that can be used to determine the set of modules imported by a script. `modulefinder.py` can also be run as a script, giving the filename of a Python script as its argument, after which a report of the imported modules will be printed.

`modulefinder.`**AddPackagePath**(*pkg_name*, *path*)

Record that the package named *pkg_name* can be found in the specified *path*.

`modulefinder.`**ReplacePackage**(*oldname*, *newname*)

Allows specifying that the module named *oldname* is in fact the package named *newname*.

class `modulefinder.`**ModuleFinder**(*path=None*, *debug=0*, *excludes=[]*, *replace_paths=[]*)

This class provides *run_script()* and *report()* methods to determine the set of modules imported by a script. *path* can be a list of directories to search for modules; if not specified, `sys.path` is used. *debug* sets the debugging level; higher values make the class print debugging messages about what it's doing. *excludes* is a list of module names to exclude from the analysis. *replace_paths* is a list of `(oldpath, newpath)` tuples that will be replaced in module paths.

report()

Print a report to standard output that lists the modules imported by the script and their paths, as well as modules that are missing or seem to be missing.

run_script(*pathname*)

Analyze the contents of the *pathname* file, which must contain Python code.

modules

A dictionary mapping module names to modules. See *Example usage of ModuleFinder*.

31.3.1 Example usage of ModuleFinder

The script that is going to get analyzed later on (bacon.py):

```
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

The script that will output the report of bacon.py:

```
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')
```

(continues on next page)

(continued from previous page)

```
print('Loaded modules:')
for name, mod in finder.modules.items():
    print('%s: ' % name, end='')
    print(','.join(list(mod.globalnames.keys())[:3]))

print('-'*50)
print('Modules not imported:')
print('\n'.join(finder.badmodules.keys()))
```

Sample output (may vary depending on the architecture):

```
Loaded modules:
_types:
copyreg:  _inverted_registry,_slotnames,__all__
re._compiler:  isstring,_sre,_optimize_unicode
_sre:
re._constants:  REPEAT_ONE,makedict,AT_END_LINE
sys:
re:  __module__,finditer,_expand
itertools:
__main__:  re,itertools,baconhameggs
re._parser:  _PATTERNENDERS,SRE_FLAG_UNICODE
array:
types:  __module__,IntType,TypeType
----------------------------------------------------
Modules not imported:
guido.python.ham
baconhameggs
```

31.4 `runpy` — Locating and executing Python modules

Source code: Lib/runpy.py

The *runpy* module is used to locate and run Python modules without importing them first. Its main use is to implement the `-m` command line switch that allows scripts to be located using the Python module namespace rather than the filesystem.

Note that this is *not* a sandbox module - all code is executed in the current process, and any side effects (such as cached imports of other modules) will remain in place after the functions have returned.

Furthermore, any functions and classes defined by the executed code are not guaranteed to work correctly after a *runpy* function has returned. If that limitation is not acceptable for a given use case, *importlib* is likely to be a more suitable choice than this module.

The *runpy* module provides two functions:

`runpy.`**`run_module`**(*mod_name*, *init_globals=None*, *run_name=None*, *alter_sys=False*)

Execute the code of the specified module and return the resulting module's globals dictionary. The module's code is first located using the standard import mechanism (refer to **PEP 302** for details) and then executed in a fresh module namespace.

The *mod_name* argument should be an absolute module name. If the module name refers to a package rather than a normal module, then that package is imported and the *__main__* submodule within that package is then executed and the resulting module globals dictionary returned.

The optional dictionary argument *init_globals* may be used to pre-populate the module's globals dictionary before the code is executed. *init_globals* will not be modified. If any of the special global variables below are defined in *init_globals*, those definitions are overridden by *run_module()*.

The special global variables `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed. (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail.)

`__name__` is set to *run_name* if this optional argument is not *None*, to `mod_name + '.__main__'` if the named module is a package and to the *mod_name* argument otherwise.

`__spec__` will be set appropriately for the *actually* imported module (that is, `__spec__.name` will always be *mod_name* or `mod_name + '.__main__'`, never *run_name*).

`__file__`, `__cached__`, `__loader__` and `__package__` are set as normal based on the module spec.

If the argument *alter_sys* is supplied and evaluates to *True*, then `sys.argv[0]` is updated with the value of `__file__` and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. Both `sys.argv[0]` and `sys.modules[__name__]` are restored to their original values before the function returns.

Note that this manipulation of *sys* is not thread-safe. Other threads may see the partially initialised module, as well as the altered list of arguments. It is recommended that the `sys` module be left alone when invoking this function from threaded code.

See also

The `-m` option offering equivalent functionality from the command line.

Changed in version 3.1: Added ability to execute packages by looking for a *__main__* submodule.

Changed in version 3.2: Added `__cached__` global variable (see **PEP 3147**).

Changed in version 3.4: Updated to take advantage of the module spec feature added by **PEP 451**. This allows `__cached__` to be set correctly for modules run this way, as well as ensuring the real module name is always accessible as `__spec__.name`.

Changed in version 3.12: The setting of `__cached__`, `__loader__`, and `__package__` are deprecated. See *ModuleSpec* for alternatives.

`runpy.`**`run_path`**(*path_name*, *init_globals=None*, *run_name=None*)

Execute the code at the named filesystem location and return the resulting module's globals dictionary. As with a script name supplied to the CPython command line, *file_path* may refer to a Python source file, a compiled bytecode file or a valid *sys.path* entry containing a *__main__* module (e.g. a zipfile containing a top-level `__main__.py` file).

For a simple script, the specified code is simply executed in a fresh module namespace. For a valid *sys.path* entry (typically a zipfile or directory), the entry is first added to the beginning of `sys.path`. The function then looks for and executes a *__main__* module using the updated path. Note that there is no special protection against invoking an existing `__main__` entry located elsewhere on `sys.path` if there is no such module at the specified location.

The optional dictionary argument *init_globals* may be used to pre-populate the module's globals dictionary before the code is executed. *init_globals* will not be modified. If any of the special global variables below are defined in *init_globals*, those definitions are overridden by *run_path()*.

The special global variables `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed. (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail.)

__name__ is set to *run_name* if this optional argument is not *None* and to '<run_path>' otherwise.

If *file_path* directly references a script file (whether as source or as precompiled byte code), then __file__ will be set to *file_path*, and __spec__, __cached__, __loader__ and __package__ will all be set to *None*.

If *file_path* is a reference to a valid *sys.path* entry, then __spec__ will be set appropriately for the imported *__main__* module (that is, __spec__.name will always be __main__). __file__, __cached__, __loader__ and __package__ will be set as normal based on the module spec.

A number of alterations are also made to the *sys* module. Firstly, *sys.path* may be altered as described above. sys.argv[0] is updated with the value of *file_path* and sys.modules[__name__] is updated with a temporary module object for the module being executed. All modifications to items in *sys* are reverted before the function returns.

Note that, unlike *run_module()*, the alterations made to *sys* are not optional in this function as these adjustments are essential to allowing the execution of *sys.path* entries. As the thread-safety limitations still apply, use of this function in threaded code should be either serialised with the import lock or delegated to a separate process.

See also

using-on-interface-options for equivalent functionality on the command line (python path/to/script).

Added in version 3.2.

Changed in version 3.4: Updated to take advantage of the module spec feature added by **PEP 451**. This allows __cached__ to be set correctly in the case where __main__ is imported from a valid *sys.path* entry rather than being executed directly.

Changed in version 3.12: The setting of __cached__, __loader__, and __package__ are deprecated.

See also

PEP 338 – Executing modules as scripts
PEP written and implemented by Nick Coghlan.

PEP 366 – Main module explicit relative imports
PEP written and implemented by Nick Coghlan.

PEP 451 – A ModuleSpec Type for the Import System
PEP written and implemented by Eric Snow

using-on-general - CPython command line details

The *importlib.import_module()* function

31.5 importlib — The implementation of import

Added in version 3.1.

Source code: Lib/importlib/__init__.py

31.5.1 Introduction

The purpose of the *importlib* package is three-fold.

One is to provide the implementation of the `import` statement (and thus, by extension, the *__import__()* function) in Python source code. This provides an implementation of `import` which is portable to any Python interpreter. This also provides an implementation which is easier to comprehend than one implemented in a programming language other than Python.

Two, the components to implement `import` are exposed in this package, making it easier for users to create their own custom objects (known generically as an *importer*) to participate in the import process.

Three, the package contains modules exposing additional functionality for managing aspects of Python packages:

- *importlib.metadata* presents access to metadata from third-party distributions.
- *importlib.resources* provides routines for accessing non-code "resources" from Python packages.

See also

import
The language reference for the `import` statement.

Packages specification
Original specification of packages. Some semantics have changed since the writing of this document (e.g. redirecting based on `None` in *sys.modules*).

The *__import__()* function
The `import` statement is syntactic sugar for this function.

The initialization of the sys.path module search path
The initialization of *sys.path*.

PEP 235
Import on Case-Insensitive Platforms

PEP 263
Defining Python Source Code Encodings

PEP 302
New Import Hooks

PEP 328
Imports: Multi-Line and Absolute/Relative

PEP 366
Main module explicit relative imports

PEP 420
Implicit namespace packages

PEP 451
A ModuleSpec Type for the Import System

PEP 488
Elimination of PYO files

PEP 489
Multi-phase extension module initialization

PEP 552
Deterministic pycs

PEP 3120
Using UTF-8 as the Default Source Encoding

PEP 3147
PYC Repository Directories

31.5.2 Functions

`importlib.__import__`(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)

An implementation of the built-in `__import__()` function.

> **Note**
>
> Programmatic importing of modules should use `import_module()` instead of this function.

`importlib.import_module`(*name*, *package=None*)

Import a module. The *name* argument specifies what module to import in absolute or relative terms (e.g. either `pkg.mod` or `..mod`). If the name is specified in relative terms, then the *package* argument must be set to the name of the package which is to act as the anchor for resolving the package name (e.g. `import_module('..mod', 'pkg.subpkg')` will import `pkg.mod`).

The `import_module()` function acts as a simplifying wrapper around `importlib.__import__()`. This means all semantics of the function are derived from `importlib.__import__()`. The most important difference between these two functions is that `import_module()` returns the specified package or module (e.g. `pkg.mod`), while `__import__()` returns the top-level package or module (e.g. `pkg`).

If you are dynamically importing a module that was created since the interpreter began execution (e.g., created a Python source file), you may need to call `invalidate_caches()` in order for the new module to be noticed by the import system.

Changed in version 3.3: Parent packages are automatically imported.

`importlib.invalidate_caches`()

Invalidate the internal caches of finders stored at `sys.meta_path`. If a finder implements `invalidate_caches()` then it will be called to perform the invalidation. This function should be called if any modules are created/installed while your program is running to guarantee all finders will notice the new module's existence.

Added in version 3.3.

Changed in version 3.10: Namespace packages created/installed in a different `sys.path` location after the same namespace was already imported are noticed.

`importlib.reload`(*module*)

Reload a previously imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (which can be different if re-importing causes a different object to be placed in `sys.modules`).

When `reload()` is executed:

- Python module's code is recompiled and the module-level code re-executed, defining a new set of objects which are bound to names in the module's dictionary by reusing the *loader* which originally loaded the module. The `init` function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.

- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module's advantage if it maintains a global table or cache of objects — with a `try` statement it can test for the table's presence and skip its initialization if desired:

```
try:
    cache
except NameError:
    cache = {}
```

It is generally not very useful to reload built-in or dynamically loaded modules. Reloading *sys*, *__main__*, *builtins* and other key modules is not recommended. In many cases extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from` … `import` …, calling *reload()* for the other module does not redefine the objects imported from it — one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (*module.name*) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

Added in version 3.4.

Changed in version 3.7: *ModuleNotFoundError* is raised when the module being reloaded lacks a *ModuleSpec*.

31.5.3 `importlib.abc` – Abstract base classes related to import

Source code: Lib/importlib/abc.py

The *importlib.abc* module contains all of the core abstract base classes used by `import`. Some subclasses of the core abstract base classes are also provided to help in implementing the core ABCs.

ABC hierarchy:

```
object
 +-- MetaPathFinder
 +-- PathEntryFinder
 +-- Loader
      +-- ResourceLoader --------+
      +-- InspectLoader          |
           +-- ExecutionLoader --+
                                 +-- FileLoader
                                 +-- SourceLoader
```

class importlib.abc.**MetaPathFinder**

An abstract base class representing a *meta path finder*.

Added in version 3.3.

Changed in version 3.10: No longer a subclass of `Finder`.

find_spec(*fullname*, *path*, *target=None*)

An abstract method for finding a *spec* for the specified module. If this is a top-level import, *path* will be `None`. Otherwise, this is a search for a subpackage or module and *path* will be the value of `__path__` from the parent package. If a spec cannot be found, `None` is returned. When passed in, `target` is a module object that the finder may use to make a more educated guess about what spec to return. *importlib.util.spec_from_loader()* may be useful for implementing concrete `MetaPathFinders`.

Added in version 3.4.

invalidate_caches()

An optional method which, when called, should invalidate any internal cache used by the finder. Used by *importlib.invalidate_caches()* when invalidating the caches of all finders on *sys.meta_path*.

Changed in version 3.4: Returns `None` when called instead of *NotImplemented*.

class importlib.abc.**PathEntryFinder**

An abstract base class representing a *path entry finder*. Though it bears some similarities to *MetaPathFinder*, `PathEntryFinder` is meant for use only within the path-based import subsystem provided by *importlib.machinery.PathFinder*.

Added in version 3.3.

Changed in version 3.10: No longer a subclass of `Finder`.

find_spec(*fullname*, *target=None*)

An abstract method for finding a *spec* for the specified module. The finder will search for the module only within the *path entry* to which it is assigned. If a spec cannot be found, `None` is returned. When passed in, `target` is a module object that the finder may use to make a more educated guess about what spec to return. *importlib.util.spec_from_loader()* may be useful for implementing concrete `PathEntryFinders`.

Added in version 3.4.

invalidate_caches()

An optional method which, when called, should invalidate any internal cache used by the finder. Used by *importlib.machinery.PathFinder.invalidate_caches()* when invalidating the caches of all cached finders.

class importlib.abc.**Loader**

An abstract base class for a *loader*. See **PEP 302** for the exact definition for a loader.

Loaders that wish to support resource reading should implement a `get_resource_reader()` method as specified by *importlib.resources.abc.ResourceReader*.

Changed in version 3.7: Introduced the optional `get_resource_reader()` method.

create_module(*spec*)

A method that returns the module object to use when importing a module. This method may return `None`, indicating that default module creation semantics should take place.

Added in version 3.4.

Changed in version 3.6: This method is no longer optional when *exec_module()* is defined.

exec_module(*module*)

An abstract method that executes the module in its own namespace when a module is imported or reloaded. The module should already be initialized when *exec_module()* is called. When this method exists, *create_module()* must be defined.

Added in version 3.4.

Changed in version 3.6: *create_module()* must also be defined.

load_module(*fullname*)

A legacy method for loading a module. If the module cannot be loaded, *ImportError* is raised, otherwise the loaded module is returned.

If the requested module already exists in *sys.modules*, that module should be used and reloaded. Otherwise the loader should create a new module and insert it into *sys.modules* before any loading begins, to prevent recursion from the import. If the loader inserted a module and the load fails, it must be removed by the loader from *sys.modules*; modules already in *sys.modules* before the loader began execution should be left alone.

The loader should set several attributes on the module (note that some of these attributes can change when a module is reloaded):

- **__name__**
  The module's fully qualified name. It is `'__main__'` for an executed module.
- **__file__**
  The location the *loader* used to load the module. For example, for modules loaded from a .py file this is the filename. It is not set on all modules (e.g. built-in modules).
- **__cached__**
  The filename of a compiled version of the module's code. It is not set on all modules (e.g. built-in modules).
- **__path__**
  The list of locations where the package's submodules will be found. Most of the time this is a single directory. The import system passes this attribute to `__import__()` and to finders in the same way as *sys.path* but just for the package. It is not set on non-package modules so it can be used as an indicator that the module is a package.
- **__package__**
  The fully qualified name of the package the module is in (or the empty string for a top-level module). If the module is a package then this is the same as `__name__`.
- **__loader__**
  The *loader* used to load the module.

When *exec_module()* is available then backwards-compatible functionality is provided.

Changed in version 3.4: Raise *ImportError* when called instead of *NotImplementedError*. Functionality provided when *exec_module()* is available.

Deprecated since version 3.4: The recommended API for loading a module is *exec_module()* (and *create_module()*). Loaders should implement it instead of *load_module()*. The import machinery takes care of all the other responsibilities of *load_module()* when *exec_module()* is implemented.

class importlib.abc.**ResourceLoader**

An abstract base class for a *loader* which implements the optional **PEP 302** protocol for loading arbitrary resources from the storage back-end.

Deprecated since version 3.7: This ABC is deprecated in favour of supporting resource loading through *importlib.resources.abc.ResourceReader*.

abstractmethod get_data(*path*)

An abstract method to return the bytes for the data located at *path*. Loaders that have a file-like storage back-end that allows storing arbitrary data can implement this abstract method to give direct access to the data stored. *OSError* is to be raised if the *path* cannot be found. The *path* is expected to be constructed using a module's `__file__` attribute or an item from a package's `__path__`.

Changed in version 3.4: Raises *OSError* instead of *NotImplementedError*.

class importlib.abc.**InspectLoader**

An abstract base class for a *loader* which implements the optional **PEP 302** protocol for loaders that inspect modules.

get_code(*fullname*)

Return the code object for a module, or `None` if the module does not have a code object (as would be the case, for example, for a built-in module). Raise an *ImportError* if loader cannot find the requested module.

> **Note**
>
> While the method has a default implementation, it is suggested that it be overridden if possible for performance.

Changed in version 3.4: No longer abstract and a concrete implementation is provided.

abstractmethod get_source(*fullname*)

An abstract method to return the source of a module. It is returned as a text string using *universal newlines*, translating all recognized line separators into `'\n'` characters. Returns `None` if no source is available (e.g. a built-in module). Raises *ImportError* if the loader cannot find the module specified.

Changed in version 3.4: Raises *ImportError* instead of *NotImplementedError*.

is_package(*fullname*)

An optional method to return a true value if the module is a package, a false value otherwise. *ImportError* is raised if the *loader* cannot find the module.

Changed in version 3.4: Raises *ImportError* instead of *NotImplementedError*.

static source_to_code(*data*, *path='<string>'*)

Create a code object from Python source.

The *data* argument can be whatever the *compile()* function supports (i.e. string or bytes). The *path* argument should be the "path" to where the source code originated from, which can be an abstract concept (e.g. location in a zip file).

With the subsequent code object one can execute it in a module by running `exec(code, module.__dict__)`.

Added in version 3.4.

Changed in version 3.5: Made the method static.

exec_module(*module*)

Implementation of *Loader.exec_module()*.

Added in version 3.4.

load_module(*fullname*)

Implementation of *Loader.load_module()*.

Deprecated since version 3.4: use *exec_module()* instead.

class importlib.abc.**ExecutionLoader**

An abstract base class which inherits from *InspectLoader* that, when implemented, helps a module to be executed as a script. The ABC represents an optional **PEP 302** protocol.

abstractmethod get_filename(*fullname*)

An abstract method that is to return the value of `__file__` for the specified module. If no path is available, *ImportError* is raised.

If source code is available, then the method should return the path to the source file, regardless of whether a bytecode was used to load the module.

Changed in version 3.4: Raises *ImportError* instead of *NotImplementedError*.

class importlib.abc.**FileLoader**(*fullname*, *path*)

An abstract base class which inherits from *ResourceLoader* and *ExecutionLoader*, providing concrete implementations of *ResourceLoader.get_data()* and *ExecutionLoader.get_filename()*.

The *fullname* argument is a fully resolved name of the module the loader is to handle. The *path* argument is the path to the file for the module.

Added in version 3.3.

name

The name of the module the loader can handle.

path

Path to the file of the module.

load_module(*fullname*)

Calls super's `load_module()`.

Deprecated since version 3.4: Use *Loader.exec_module()* instead.

abstractmethod get_filename(*fullname*)

Returns *path*.

abstractmethod get_data(*path*)

Reads *path* as a binary file and returns the bytes from it.

class importlib.abc.**SourceLoader**

An abstract base class for implementing source (and optionally bytecode) file loading. The class inherits from both *ResourceLoader* and *ExecutionLoader*, requiring the implementation of:

- *ResourceLoader.get_data()*
- ***ExecutionLoader.get_filename()***
  Should only return the path to the source file; sourceless loading is not supported.

The abstract methods defined by this class are to add optional bytecode file support. Not implementing these optional methods (or causing them to raise *NotImplementedError*) causes the loader to only work with source code. Implementing the methods allows the loader to work with source *and* bytecode files; it does not allow for *sourceless* loading where only bytecode is provided. Bytecode files are an optimization to speed up loading by removing the parsing step of Python's compiler, and so no bytecode-specific API is exposed.

path_stats(*path*)

Optional abstract method which returns a *dict* containing metadata about the specified path. Supported dictionary keys are:

- `'mtime'` (mandatory): an integer or floating-point number representing the modification time of the source code;
- `'size'` (optional): the size in bytes of the source code.

Any other keys in the dictionary are ignored, to allow for future extensions. If the path cannot be handled, *OSError* is raised.

Added in version 3.3.

Changed in version 3.4: Raise *OSError* instead of *NotImplementedError*.

path_mtime(*path*)

Optional abstract method which returns the modification time for the specified path.

Deprecated since version 3.3: This method is deprecated in favour of *path_stats()*. You don't have to implement it, but it is still available for compatibility purposes. Raise *OSError* if the path cannot be handled.

Changed in version 3.4: Raise *OSError* instead of *NotImplementedError*.

set_data(*path*, *data*)

Optional abstract method which writes the specified bytes to a file path. Any intermediate directories which do not exist are to be created automatically.

When writing to the path fails because the path is read-only (*errno.EACCES*/*PermissionError*), do not propagate the exception.

Changed in version 3.4: No longer raises *NotImplementedError* when called.

get_code(*fullname*)

Concrete implementation of *InspectLoader.get_code()*.

exec_module(*module*)

Concrete implementation of *Loader.exec_module()*.

Added in version 3.4.

load_module(*fullname*)

Concrete implementation of *Loader.load_module()*.

Deprecated since version 3.4: Use *exec_module()* instead.

get_source(*fullname*)

Concrete implementation of *InspectLoader.get_source()*.

is_package(*fullname*)

Concrete implementation of *InspectLoader.is_package()*. A module is determined to be a package if its file path (as provided by *ExecutionLoader.get_filename()*) is a file named `__init__` when the file extension is removed **and** the module name itself does not end in `__init__`.

class importlib.abc.**ResourceReader**

Superseded by TraversableResources

An *abstract base class* to provide the ability to read *resources*.

From the perspective of this ABC, a *resource* is a binary artifact that is shipped within a package. Typically this is something like a data file that lives next to the `__init__.py` file of the package. The purpose of this class is to help abstract out the accessing of such data files so that it does not matter if the package and its data file(s) are stored e.g. in a zip file versus on the file system.

For any of methods of this class, a *resource* argument is expected to be a *path-like object* which represents conceptually just a file name. This means that no subdirectory paths should be included in the *resource* argument. This is because the location of the package the reader is for, acts as the "directory". Hence the metaphor for directories and file names is packages and resources, respectively. This is also why instances of this class are expected to directly correlate to a specific package (instead of potentially representing multiple packages or a module).

Loaders that wish to support resource reading are expected to provide a method called `get_resource_reader(fullname)` which returns an object implementing this ABC's interface. If

the module specified by fullname is not a package, this method should return *None*. An object compatible with this ABC should only be returned when the specified module is a package.

Added in version 3.7.

Deprecated since version 3.12, will be removed in version 3.14: Use *importlib.resources.abc.TraversableResources* instead.

abstractmethod open_resource(*resource*)

Returns an opened, *file-like object* for binary reading of the *resource*.

If the resource cannot be found, *FileNotFoundError* is raised.

abstractmethod resource_path(*resource*)

Returns the file system path to the *resource*.

If the resource does not concretely exist on the file system, raise *FileNotFoundError*.

abstractmethod is_resource(*name*)

Returns `True` if the named *name* is considered a resource. *FileNotFoundError* is raised if *name* does not exist.

abstractmethod contents()

Returns an *iterable* of strings over the contents of the package. Do note that it is not required that all names returned by the iterator be actual resources, e.g. it is acceptable to return names for which *is_resource()* would be false.

Allowing non-resource names to be returned is to allow for situations where how a package and its resources are stored are known a priori and the non-resource names would be useful. For instance, returning subdirectory names is allowed so that when it is known that the package and resources are stored on the file system then those subdirectory names can be used directly.

The abstract method returns an iterable of no items.

class importlib.abc.**Traversable**

An object with a subset of *pathlib.Path* methods suitable for traversing directories and opening files.

For a representation of the object on the file-system, use *importlib.resources.as_file()*.

Added in version 3.9.

Deprecated since version 3.12, will be removed in version 3.14: Use *importlib.resources.abc.Traversable* instead.

name

Abstract. The base name of this object without any parent references.

abstractmethod iterdir()

Yield `Traversable` objects in `self`.

abstractmethod is_dir()

Return `True` if `self` is a directory.

abstractmethod is_file()

Return `True` if `self` is a file.

abstractmethod joinpath(*child*)

Return Traversable child in `self`.

abstractmethod __truediv__(*child*)

Return `Traversable` child in `self`.

abstractmethod open(*mode='r'*, **args*, ***kwargs*)

mode may be 'r' or 'rb' to open as text or binary. Return a handle suitable for reading (same as *pathlib.Path.open*).

When opening as text, accepts encoding parameters such as those accepted by *io.TextIOWrapper*.

read_bytes()

Read contents of `self` as bytes.

read_text(*encoding=None*)

Read contents of `self` as text.

class importlib.abc.**TraversableResources**

An abstract base class for resource readers capable of serving the *importlib.resources.files()* interface. Subclasses *importlib.resources.abc.ResourceReader* and provides concrete implementations of the *importlib.resources.abc.ResourceReader*'s abstract methods. Therefore, any loader supplying *importlib.abc.TraversableResources* also supplies ResourceReader.

Loaders that wish to support resource reading are expected to implement this interface.

Added in version 3.9.

Deprecated since version 3.12, will be removed in version 3.14: Use *importlib.resources.abc.TraversableResources* instead.

abstractmethod files()

Returns a *importlib.resources.abc.Traversable* object for the loaded package.

31.5.4 importlib.machinery – Importers and path hooks

Source code: Lib/importlib/machinery.py

This module contains the various objects that help `import` find and load modules.

importlib.machinery.**SOURCE_SUFFIXES**

A list of strings representing the recognized file suffixes for source modules.

Added in version 3.3.

importlib.machinery.**DEBUG_BYTECODE_SUFFIXES**

A list of strings representing the file suffixes for non-optimized bytecode modules.

Added in version 3.3.

Deprecated since version 3.5: Use *BYTECODE_SUFFIXES* instead.

importlib.machinery.**OPTIMIZED_BYTECODE_SUFFIXES**

A list of strings representing the file suffixes for optimized bytecode modules.

Added in version 3.3.

Deprecated since version 3.5: Use *BYTECODE_SUFFIXES* instead.

importlib.machinery.**BYTECODE_SUFFIXES**

A list of strings representing the recognized file suffixes for bytecode modules (including the leading dot).

Added in version 3.3.

Changed in version 3.5: The value is no longer dependent on `__debug__`.

importlib.machinery.**EXTENSION_SUFFIXES**

A list of strings representing the recognized file suffixes for extension modules.

Added in version 3.3.

importlib.machinery.**all_suffixes**()

Returns a combined list of strings representing all file suffixes for modules recognized by the standard import machinery. This is a helper for code which simply needs to know if a filesystem path potentially refers to a module without needing any details on the kind of module (for example, *inspect.getmodulename()*).

Added in version 3.3.

class importlib.machinery.**BuiltinImporter**

An *importer* for built-in modules. All known built-in modules are listed in *sys.builtin_module_names*. This class implements the *importlib.abc.MetaPathFinder* and *importlib.abc.InspectLoader* ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

Changed in version 3.5: As part of **PEP 489**, the builtin importer now implements `Loader.create_module()` and `Loader.exec_module()`

class importlib.machinery.**FrozenImporter**

An *importer* for frozen modules. This class implements the *importlib.abc.MetaPathFinder* and *importlib.abc.InspectLoader* ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

Changed in version 3.4: Gained `create_module()` and `exec_module()` methods.

class importlib.machinery.**WindowsRegistryFinder**

Finder for modules declared in the Windows registry. This class implements the *importlib.abc.MetaPathFinder* ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

Added in version 3.3.

Deprecated since version 3.6: Use *site* configuration instead. Future versions of Python may not enable this finder by default.

class importlib.machinery.**PathFinder**

A *Finder* for *sys.path* and package `__path__` attributes. This class implements the *importlib.abc.MetaPathFinder* ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

classmethod find_spec(*fullname*, *path=None*, *target=None*)

Class method that attempts to find a *spec* for the module specified by *fullname* on *sys.path* or, if defined, on *path*. For each path entry that is searched, *sys.path_importer_cache* is checked. If a non-false object is found then it is used as the *path entry finder* to look for the module being searched for. If no entry is found in *sys.path_importer_cache*, then *sys.path_hooks* is searched for a finder for the path entry and, if found, is stored in *sys.path_importer_cache* along with being queried about the module. If no finder is ever found then `None` is both stored in the cache and returned.

Added in version 3.4.

Changed in version 3.5: If the current working directory – represented by an empty string – is no longer valid then `None` is returned but no value is cached in *sys.path_importer_cache*.

classmethod invalidate_caches()

Calls *importlib.abc.PathEntryFinder.invalidate_caches()* on all finders stored in *sys.path_importer_cache* that define the method. Otherwise entries in *sys.path_importer_cache* set to `None` are deleted.

Changed in version 3.7: Entries of `None` in *sys.path_importer_cache* are deleted.

Changed in version 3.4: Calls objects in *sys.path_hooks* with the current working directory for `''` (i.e. the empty string).

class importlib.machinery.**FileFinder**(*path*, **loader_details*)

A concrete implementation of *importlib.abc.PathEntryFinder* which caches results from the file system.

The *path* argument is the directory for which the finder is in charge of searching.

The *loader_details* argument is a variable number of 2-item tuples each containing a loader and a sequence of file suffixes the loader recognizes. The loaders are expected to be callables which accept two arguments of the module's name and the path to the file found.

The finder will cache the directory contents as necessary, making stat calls for each module search to verify the cache is not outdated. Because cache staleness relies upon the granularity of the operating system's state information of the file system, there is a potential race condition of searching for a module, creating a new file, and then searching for the module the new file represents. If the operations happen fast enough to fit within the granularity of stat calls, then the module search will fail. To prevent this from happening, when you create a module dynamically, make sure to call *importlib.invalidate_caches()*.

Added in version 3.3.

path

The path the finder will search in.

find_spec(*fullname*, *target=None*)

Attempt to find the spec to handle *fullname* within *path*.

Added in version 3.4.

invalidate_caches()

Clear out the internal cache.

classmethod path_hook(**loader_details*)

A class method which returns a closure for use on *sys.path_hooks*. An instance of *FileFinder* is returned by the closure using the path argument given to the closure directly and *loader_details* indirectly.

If the argument to the closure is not an existing directory, *ImportError* is raised.

class importlib.machinery.**SourceFileLoader**(*fullname*, *path*)

A concrete implementation of *importlib.abc.SourceLoader* by subclassing *importlib.abc.FileLoader* and providing some concrete implementations of other methods.

Added in version 3.3.

name

The name of the module that this loader will handle.

path

The path to the source file.

is_package(*fullname*)

Return `True` if *path* appears to be for a package.

path_stats(*path*)

Concrete implementation of *importlib.abc.SourceLoader.path_stats()*.

set_data(*path*, *data*)

Concrete implementation of *importlib.abc.SourceLoader.set_data()*.

load_module(*name=None*)

Concrete implementation of *importlib.abc.Loader.load_module()* where specifying the name of the module to load is optional.

Deprecated since version 3.6: Use *importlib.abc.Loader.exec_module()* instead.

class importlib.machinery.**SourcelessFileLoader**(*fullname*, *path*)

A concrete implementation of *importlib.abc.FileLoader* which can import bytecode files (i.e. no source code files exist).

Please note that direct use of bytecode files (and thus not source code files) inhibits your modules from being usable by all Python implementations or new versions of Python which change the bytecode format.

Added in version 3.3.

name

The name of the module the loader will handle.

path

The path to the bytecode file.

is_package(*fullname*)

Determines if the module is a package based on *path*.

get_code(*fullname*)

Returns the code object for *name* created from *path*.

get_source(*fullname*)

Returns `None` as bytecode files have no source when this loader is used.

load_module(*name=None*)

Concrete implementation of *importlib.abc.Loader.load_module()* where specifying the name of the module to load is optional.

Deprecated since version 3.6: Use *importlib.abc.Loader.exec_module()* instead.

class importlib.machinery.**ExtensionFileLoader**(*fullname*, *path*)

A concrete implementation of *importlib.abc.ExecutionLoader* for extension modules.

The *fullname* argument specifies the name of the module the loader is to support. The *path* argument is the path to the extension module's file.

Note that, by default, importing an extension module will fail in subinterpreters if it doesn't implement multi-phase init (see **PEP 489**), even if it would otherwise import successfully.

Added in version 3.3.

Changed in version 3.12: Multi-phase init is now required for use in subinterpreters.

name

Name of the module the loader supports.

path

Path to the extension module.

create_module(*spec*)

Creates the module object from the given specification in accordance with **PEP 489**.

Added in version 3.5.

exec_module(*module*)

Initializes the given module object in accordance with **PEP 489**.

Added in version 3.5.

is_package(*fullname*)

Returns `True` if the file path points to a package's `__init__` module based on *EXTENSION_SUFFIXES*.

get_code(*fullname*)

Returns `None` as extension modules lack a code object.

get_source(*fullname*)

Returns `None` as extension modules do not have source code.

get_filename(*fullname*)

Returns *path*.

Added in version 3.4.

class importlib.machinery.**NamespaceLoader**(*name*, *path*, *path_finder*)

A concrete implementation of *importlib.abc.InspectLoader* for namespace packages. This is an alias for a private class and is only made public for introspecting the `__loader__` attribute on namespace packages:

```
>>> from importlib.machinery import NamespaceLoader
>>> import my_namespace
>>> isinstance(my_namespace.__loader__, NamespaceLoader)
True
>>> import importlib.abc
>>> isinstance(my_namespace.__loader__, importlib.abc.Loader)
True
```

Added in version 3.11.

class importlib.machinery.**ModuleSpec**(*name*, *loader*, ***, *origin=None*, *loader_state=None*, *is_package=None*)

A specification for a module's import-system-related state. This is typically exposed as the module's `__spec__` attribute. In the descriptions below, the names in parentheses give the corresponding attribute available directly on the module object, e.g. `module.__spec__.origin == module.__file__`. Note, however, that while the *values* are usually equivalent, they can differ since there is no synchronization between the two objects. For example, it is possible to update the module's `__file__` at runtime and this will not be automatically reflected in the module's `__spec__.origin`, and vice versa.

Added in version 3.4.

name

(`__name__`)

The module's fully qualified name. The *finder* should always set this attribute to a non-empty string.

loader

(`__loader__`)

The *loader* used to load the module. The *finder* should always set this attribute.

origin

(__file__)

The location the *loader* should use to load the module. For example, for modules loaded from a .py file this is the filename. The *finder* should always set this attribute to a meaningful value for the *loader* to use. In the uncommon case that there is not one (like for namespace packages), it should be set to None.

submodule_search_locations

(__path__)

The list of locations where the package's submodules will be found. Most of the time this is a single directory. The *finder* should set this attribute to a list, even an empty one, to indicate to the import system that the module is a package. It should be set to None for non-package modules. It is set automatically later to a special object for namespace packages.

loader_state

The *finder* may set this attribute to an object containing additional, module-specific data to use when loading the module. Otherwise it should be set to None.

cached

(__cached__)

The filename of a compiled version of the module's code. The *finder* should always set this attribute but it may be None for modules that do not need compiled code stored.

parent

(__package__)

(Read-only) The fully qualified name of the package the module is in (or the empty string for a top-level module). If the module is a package then this is the same as *name*.

has_location

True if the spec's *origin* refers to a loadable location,
False otherwise. This value impacts how *origin* is interpreted and how the module's __file__ is populated.

class importlib.machinery.**AppleFrameworkLoader**(*name*, *path*)

A specialization of *importlib.machinery.ExtensionFileLoader* that is able to load extension modules in Framework format.

For compatibility with the iOS App Store, *all* binary modules in an iOS app must be dynamic libraries, contained in a framework with appropriate metadata, stored in the Frameworks folder of the packaged app. There can be only a single binary per framework, and there can be no executable binary material outside the Frameworks folder.

To accomodate this requirement, when running on iOS, extension module binaries are *not* packaged as .so files on sys.path, but as individual standalone frameworks. To discover those frameworks, this loader is be registered against the .fwork file extension, with a .fwork file acting as a placeholder in the original location of the binary on sys.path. The .fwork file contains the path of the actual binary in the Frameworks folder, relative to the app bundle. To allow for resolving a framework-packaged binary back to the original location, the framework is expected to contain a .origin file that contains the location of the .fwork file, relative to the app bundle.

For example, consider the case of an import from foo.bar import _whiz, where _whiz is implemented with the binary module sources/foo/bar/_whiz.abi3.so, with sources being the location registered on sys.path, relative to the application bundle. This module *must* be distributed as Frameworks/foo.bar._whiz.framework/foo.bar._whiz (creating the framework name from the full import path

of the module), with an `Info.plist` file in the `.framework` directory identifying the binary as a framework. The `foo.bar._whiz` module would be represented in the original location with a `sources/foo/bar/_whiz.abi3.fwork` marker file, containing the path `Frameworks/foo.bar._whiz/foo.bar._whiz`. The framework would also contain `Frameworks/foo.bar._whiz.framework/foo.bar._whiz.origin`, containing the path to the `.fwork` file.

When a module is loaded with this loader, the `__file__` for the module will report as the location of the `.fwork` file. This allows code to use the `__file__` of a module as an anchor for file system traveral. However, the spec origin will reference the location of the *actual* binary in the `.framework` folder.

The Xcode project building the app is responsible for converting any `.so` files from wherever they exist in the `PYTHONPATH` into frameworks in the `Frameworks` folder (including stripping extensions from the module file, the addition of framework metadata, and signing the resulting framework), and creating the `.fwork` and `.origin` files. This will usually be done with a build step in the Xcode project; see the iOS documentation for details on how to construct this build step.

Added in version 3.13.

Availability: iOS.

name

Name of the module the loader supports.

path

Path to the `.fwork` file for the extension module.

31.5.5 `importlib.util` – Utility code for importers

Source code: Lib/importlib/util.py

This module contains the various objects that help in the construction of an *importer*.

`importlib.util.`**`MAGIC_NUMBER`**

The bytes which represent the bytecode version number. If you need help with loading/writing bytecode then consider *`importlib.abc.SourceLoader`*.

Added in version 3.4.

`importlib.util.`**`cache_from_source`**(*path*, *debug_override=None*, ***, *optimization=None*)

Return the **PEP 3147**/**PEP 488** path to the byte-compiled file associated with the source *path*. For example, if *path* is `/foo/bar/baz.py` the return value would be `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. The `cpython-32` string comes from the current magic tag (see `get_tag()`; if `sys.implementation.cache_tag` is not defined then *`NotImplementedError`* will be raised).

The *optimization* parameter is used to specify the optimization level of the bytecode file. An empty string represents no optimization, so `/foo/bar/baz.py` with an *optimization* of `''` will result in a bytecode path of `/foo/bar/__pycache__/baz.cpython-32.pyc`. `None` causes the interpreter's optimization level to be used. Any other value's string representation is used, so `/foo/bar/baz.py` with an *optimization* of `2` will lead to the bytecode path of `/foo/bar/__pycache__/baz.cpython-32.opt-2.pyc`. The string representation of *optimization* can only be alphanumeric, else *`ValueError`* is raised.

The *debug_override* parameter is deprecated and can be used to override the system's value for `__debug__`. A `True` value is the equivalent of setting *optimization* to the empty string. A `False` value is the same as setting *optimization* to `1`. If both *debug_override* an *optimization* are not `None` then *`TypeError`* is raised.

Added in version 3.4.

Changed in version 3.5: The *optimization* parameter was added and the *debug_override* parameter was deprecated.

Changed in version 3.6: Accepts a *path-like object*.

importlib.util.**source_from_cache**(*path*)

Given the *path* to a **PEP 3147** file name, return the associated source code file path. For example, if *path* is `/foo/bar/__pycache__/baz.cpython-32.pyc` the returned path would be `/foo/bar/baz.py`. *path* need not exist, however if it does not conform to **PEP 3147** or **PEP 488** format, a *ValueError* is raised. If `sys.implementation.cache_tag` is not defined, *NotImplementedError* is raised.

Added in version 3.4.

Changed in version 3.6: Accepts a *path-like object*.

importlib.util.**decode_source**(*source_bytes*)

Decode the given bytes representing source code and return it as a string with universal newlines (as required by *importlib.abc.InspectLoader.get_source()*).

Added in version 3.4.

importlib.util.**resolve_name**(*name*, *package*)

Resolve a relative module name to an absolute one.

If **name** has no leading dots, then **name** is simply returned. This allows for usage such as `importlib.util.resolve_name('sys', __spec__.parent)` without doing a check to see if the **package** argument is needed.

ImportError is raised if **name** is a relative module name but **package** is a false value (e.g. `None` or the empty string). *ImportError* is also raised if a relative name would escape its containing package (e.g. requesting `..bacon` from within the `spam` package).

Added in version 3.3.

Changed in version 3.9: To improve consistency with import statements, raise *ImportError* instead of *ValueError* for invalid relative import attempts.

importlib.util.**find_spec**(*name*, *package=None*)

Find the *spec* for a module, optionally relative to the specified **package** name. If the module is in *sys.modules*, then `sys.modules[name].__spec__` is returned (unless the spec would be `None` or is not set, in which case *ValueError* is raised). Otherwise a search using *sys.meta_path* is done. `None` is returned if no spec is found.

If **name** is for a submodule (contains a dot), the parent module is automatically imported.

name and **package** work the same as for `import_module()`.

Added in version 3.4.

Changed in version 3.7: Raises *ModuleNotFoundError* instead of *AttributeError* if **package** is in fact not a package (i.e. lacks a `__path__` attribute).

importlib.util.**module_from_spec**(*spec*)

Create a new module based on **spec** and *spec.loader.create_module*.

If *spec.loader.create_module* does not return `None`, then any pre-existing attributes will not be reset. Also, no *AttributeError* will be raised if triggered while accessing **spec** or setting an attribute on the module.

This function is preferred over using *types.ModuleType* to create a new module as **spec** is used to set as many import-controlled attributes on the module as possible.

Added in version 3.5.

importlib.util.**spec_from_loader**(*name*, *loader*, *, *origin=None*, *is_package=None*)

A factory function for creating a *ModuleSpec* instance based on a loader. The parameters have the same meaning as they do for ModuleSpec. The function uses available *loader* APIs, such as `InspectLoader.is_package()`, to fill in any missing information on the spec.

Added in version 3.4.

importlib.util.**spec_from_file_location**(*name*, *location*, *, *loader=None*, *submodule_search_locations=None*)

A factory function for creating a *ModuleSpec* instance based on the path to a file. Missing information will be filled in on the spec by making use of loader APIs and by the implication that the module will be file-based.

Added in version 3.4.

Changed in version 3.6: Accepts a *path-like object*.

importlib.util.**source_hash**(*source_bytes*)

Return the hash of *source_bytes* as bytes. A hash-based `.pyc` file embeds the *source_hash()* of the corresponding source file's contents in its header.

Added in version 3.7.

importlib.util.**_incompatible_extension_module_restrictions**(*, *disable_check*)

A context manager that can temporarily skip the compatibility check for extension modules. By default the check is enabled and will fail when a single-phase init module is imported in a subinterpreter. It will also fail for a multi-phase init module that doesn't explicitly support a per-interpreter GIL, when imported in an interpreter with its own GIL.

Note that this function is meant to accommodate an unusual case; one which is likely to eventually go away. There's is a pretty good chance this is not what you were looking for.

You can get the same effect as this function by implementing the basic interface of multi-phase init (**PEP 489**) and lying about support for multiple interpreters (or per-interpreter GIL).

Warning

Using this function to disable the check can lead to unexpected behavior and even crashes. It should only be used during extension module development.

Added in version 3.12.

class importlib.util.**LazyLoader**(*loader*)

A class which postpones the execution of the loader of a module until the module has an attribute accessed.

This class **only** works with loaders that define *exec_module()* as control over what module type is used for the module is required. For those same reasons, the loader's *create_module()* method must return `None` or a type for which its `__class__` attribute can be mutated along with not using *slots*. Finally, modules which substitute the object placed into *sys.modules* will not work as there is no way to properly replace the module references throughout the interpreter safely; *ValueError* is raised if such a substitution is detected.

Note

For projects where startup time is critical, this class allows for potentially minimizing the cost of loading a module if it is never used. For projects where startup time is not essential then use of this class is **heavily** discouraged due to error messages created during loading being postponed and thus occurring out of context.

Added in version 3.5.

Changed in version 3.6: Began calling *create_module()*, removing the compatibility warning for *importlib.machinery.BuiltinImporter* and *importlib.machinery.ExtensionFileLoader*.

classmethod factory(*loader*)

A class method which returns a callable that creates a lazy loader. This is meant to be used in situations where the loader is passed by class instead of by instance.

```
suffixes = importlib.machinery.SOURCE_SUFFIXES
loader = importlib.machinery.SourceFileLoader
lazy_loader = importlib.util.LazyLoader.factory(loader)
finder = importlib.machinery.FileFinder(path, (lazy_loader, suffixes))
```

31.5.6 Examples

Importing programmatically

To programmatically import a module, use *importlib.import_module()*.

```
import importlib

itertools = importlib.import_module('itertools')
```

Checking if a module can be imported

If you need to find out if a module can be imported without actually doing the import, then you should use *importlib.util.find_spec()*.

Note that if `name` is a submodule (contains a dot), *importlib.util.find_spec()* will import the parent module.

```
import importlib.util
import sys

# For illustrative purposes.
name = 'itertools'

if name in sys.modules:
    print(f"{name!r} already in sys.modules")
elif (spec := importlib.util.find_spec(name)) is not None:
    # If you chose to perform the actual import ...
    module = importlib.util.module_from_spec(spec)
    sys.modules[name] = module
    spec.loader.exec_module(module)
    print(f"{name!r} has been imported")
else:
    print(f"can't find the {name!r} module")
```

Importing a source file directly

To import a Python source file directly, use the following recipe:

```
import importlib.util
import sys

# For illustrative purposes.
import tokenize
file_path = tokenize.__file__
module_name = tokenize.__name__

spec = importlib.util.spec_from_file_location(module_name, file_path)
module = importlib.util.module_from_spec(spec)
sys.modules[module_name] = module
spec.loader.exec_module(module)
```

Implementing lazy imports

The example below shows how to implement lazy imports:

```
>>> import importlib.util
>>> import sys
>>> def lazy_import(name):
...     spec = importlib.util.find_spec(name)
...     loader = importlib.util.LazyLoader(spec.loader)
...     spec.loader = loader
...     module = importlib.util.module_from_spec(spec)
...     sys.modules[name] = module
...     loader.exec_module(module)
...     return module
...
>>> lazy_typing = lazy_import("typing")
>>> #lazy_typing is a real module object,
>>> #but it is not loaded in memory yet.
>>> lazy_typing.TYPE_CHECKING
False
```

Setting up an importer

For deep customizations of import, you typically want to implement an *importer*. This means managing both the *finder* and *loader* side of things. For finders there are two flavours to choose from depending on your needs: a *meta path finder* or a *path entry finder*. The former is what you would put on `sys.meta_path` while the latter is what you create using a *path entry hook* on `sys.path_hooks` which works with `sys.path` entries to potentially create a finder. This example will show you how to register your own importers so that import will use them (for creating an importer for yourself, read the documentation for the appropriate classes defined within this package):

```
import importlib.machinery
import sys

# For illustrative purposes only.
SpamMetaPathFinder = importlib.machinery.PathFinder
SpamPathEntryFinder = importlib.machinery.FileFinder
loader_details = (importlib.machinery.SourceFileLoader,
```

(continues on next page)

(continued from previous page)

```
                  importlib.machinery.SOURCE_SUFFIXES)

# Setting up a meta path finder.
# Make sure to put the finder in the proper location in the list in terms of
# priority.
sys.meta_path.append(SpamMetaPathFinder)

# Setting up a path entry finder.
# Make sure to put the path hook in the proper location in the list in terms
# of priority.
sys.path_hooks.append(SpamPathEntryFinder.path_hook(loader_details))
```

Approximating `importlib.import_module()`

Import itself is implemented in Python code, making it possible to expose most of the import machinery through importlib. The following helps illustrate the various APIs that importlib exposes by providing an approximate implementation of *importlib.import_module()*:

```
import importlib.util
import sys

def import_module(name, package=None):
    """An approximate implementation of import."""
    absolute_name = importlib.util.resolve_name(name, package)
    try:
        return sys.modules[absolute_name]
    except KeyError:
        pass

    path = None
    if '.' in absolute_name:
        parent_name, _, child_name = absolute_name.rpartition('.')
        parent_module = import_module(parent_name)
        path = parent_module.__spec__.submodule_search_locations
    for finder in sys.meta_path:
        spec = finder.find_spec(absolute_name, path)
        if spec is not None:
            break
    else:
        msg = f'No module named {absolute_name!r}'
        raise ModuleNotFoundError(msg, name=absolute_name)
    module = importlib.util.module_from_spec(spec)
    sys.modules[absolute_name] = module
    spec.loader.exec_module(module)
    if path is not None:
        setattr(parent_module, child_name, module)
    return module
```

31.6 importlib.resources – Package resource reading, opening and access

Source code: Lib/importlib/resources/__init__.py

Added in version 3.7.

This module leverages Python's import system to provide access to *resources* within *packages*.

"Resources" are file-like resources associated with a module or package in Python. The resources may be contained directly in a package, within a subdirectory contained in that package, or adjacent to modules outside a package. Resources may be text or binary. As a result, Python module sources (.py) of a package and compilation artifacts (pycache) are technically de-facto resources of that package. In practice, however, resources are primarily those non-Python artifacts exposed specifically by the package author.

Resources can be opened or read in either binary or text mode.

Resources are roughly akin to files inside directories, though it's important to keep in mind that this is just a metaphor. Resources and packages **do not** have to exist as physical files and directories on the file system: for example, a package and its resources can be imported from a zip file using *zipimport*.

Note

This module provides functionality similar to pkg_resources Basic Resource Access without the performance overhead of that package. This makes reading resources included in packages easier, with more stable and consistent semantics.

The standalone backport of this module provides more information on using importlib.resources and migrating from pkg_resources to importlib.resources.

Loaders that wish to support resource reading should implement a `get_resource_reader(fullname)` method as specified by *importlib.resources.abc.ResourceReader*.

class importlib.resources.**Anchor**

Represents an anchor for resources, either a *module object* or a module name as a string. Defined as `Union[str, ModuleType]`.

importlib.resources.**files**(*anchor:* Anchor | None = *None*)

Returns a *Traversable* object representing the resource container (think directory) and its resources (think files). A Traversable may contain other containers (think subdirectories).

anchor is an optional *Anchor*. If the anchor is a package, resources are resolved from that package. If a module, resources are resolved adjacent to that module (in the same package or the package root). If the anchor is omitted, the caller's module is used.

Added in version 3.9.

Changed in version 3.12: *package* parameter was renamed to *anchor*. *anchor* can now be a non-package module and if omitted will default to the caller's module. *package* is still accepted for compatibility but will raise a *DeprecationWarning*. Consider passing the anchor positionally or using `importlib_resources >= 5.10` for a compatible interface on older Pythons.

importlib.resources.**as_file**(*traversable*)

Given a *Traversable* object representing a file or directory, typically from *importlib.resources.files()*, return a context manager for use in a `with` statement. The context manager provides a *pathlib.Path* object.

Exiting the context manager cleans up any temporary file or directory created when the resource was extracted from e.g. a zip file.

Use `as_file` when the Traversable methods (`read_text`, etc) are insufficient and an actual file or directory on the file system is required.

Added in version 3.9.

Changed in version 3.12: Added support for *traversable* representing a directory.

31.6.1 Functional API

A set of simplified, backwards-compatible helpers is available. These allow common operations in a single function call.

For all the following functions:

- *anchor* is an *Anchor*, as in *files()*. Unlike in `files`, it may not be omitted.
- *path_names* are components of a resource's path name, relative to the anchor. For example, to get the text of resource named `info.txt`, use:

```
importlib.resources.read_text(my_module, "info.txt")
```

Like *Traversable.joinpath*, The individual components should use forward slashes (`/`) as path separators. For example, the following are equivalent:

```
importlib.resources.read_binary(my_module, "pics/painting.png")
importlib.resources.read_binary(my_module, "pics", "painting.png")
```

For backward compatibility reasons, functions that read text require an explicit *encoding* argument if multiple *path_names* are given. For example, to get the text of `info/chapter1.txt`, use:

```
importlib.resources.read_text(my_module, "info", "chapter1.txt",
                              encoding='utf-8')
```

`importlib.resources.`**`open_binary`**(*anchor*, **path_names*)

Open the named resource for binary reading.

See *the introduction* for details on *anchor* and *path_names*.

This function returns a *BinaryIO* object, that is, a binary stream open for reading.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).open('rb')
```

Changed in version 3.13: Multiple *path_names* are accepted.

`importlib.resources.`**`open_text`**(*anchor*, **path_names*, *encoding='utf-8'*, *errors='strict'*)

Open the named resource for text reading. By default, the contents are read as strict UTF-8.

See *the introduction* for details on *anchor* and *path_names*. *encoding* and *errors* have the same meaning as in built-in *open()*.

For backward compatibility reasons, the *encoding* argument must be given explicitly if there are multiple *path_names*. This limitation is scheduled to be removed in Python 3.15.

This function returns a *TextIO* object, that is, a text stream open for reading.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).open('r', encoding=encoding)
```

Changed in version 3.13: Multiple *path_names* are accepted. *encoding* and *errors* must be given as keyword arguments.

importlib.resources.**read_binary**(*anchor*, **path_names*)

Read and return the contents of the named resource as *bytes*.

See *the introduction* for details on *anchor* and *path_names*.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).read_bytes()
```

Changed in version 3.13: Multiple *path_names* are accepted.

importlib.resources.**read_text**(*anchor*, **path_names*, *encoding='utf-8'*, *errors='strict'*)

Read and return the contents of the named resource as *str*. By default, the contents are read as strict UTF-8.

See *the introduction* for details on *anchor* and *path_names*. *encoding* and *errors* have the same meaning as in built-in *open()*.

For backward compatibility reasons, the *encoding* argument must be given explicitly if there are multiple *path_names*. This limitation is scheduled to be removed in Python 3.15.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).read_text(encoding=encoding)
```

Changed in version 3.13: Multiple *path_names* are accepted. *encoding* and *errors* must be given as keyword arguments.

importlib.resources.**path**(*anchor*, **path_names*)

Provides the path to the *resource* as an actual file system path. This function returns a context manager for use in a `with` statement. The context manager provides a *pathlib.Path* object.

Exiting the context manager cleans up any temporary files created, e.g. when the resource needs to be extracted from a zip file.

For example, the *stat()* method requires an actual file system path; it can be used like this:

```
with importlib.resources.path(anchor, "resource.txt") as fspath:
    result = fspath.stat()
```

See *the introduction* for details on *anchor* and *path_names*.

This function is roughly equivalent to:

```
as_file(files(anchor).joinpath(*path_names))
```

Changed in version 3.13: Multiple *path_names* are accepted. *encoding* and *errors* must be given as keyword arguments.

importlib.resources.**is_resource**(*anchor*, **path_names*)

Return `True` if the named resource exists, otherwise `False`. This function does not consider directories to be resources.

See *the introduction* for details on *anchor* and *path_names*.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).is_file()
```

Changed in version 3.13: Multiple *path_names* are accepted.

importlib.resources.**contents**(*anchor*, **path_names*)

Return an iterable over the named items within the package or path. The iterable returns names of resources (e.g. files) and non-resources (e.g. directories) as *str*. The iterable does not recurse into subdirectories.

See *the introduction* for details on *anchor* and *path_names*.

This function is roughly equivalent to:

```
for resource in files(anchor).joinpath(*path_names).iterdir():
    yield resource.name
```

Deprecated since version 3.11: Prefer `iterdir()` as above, which offers more control over the results and richer functionality.

31.7 importlib.resources.abc – Abstract base classes for resources

Source code: Lib/importlib/resources/abc.py

Added in version 3.11.

class importlib.resources.abc.**ResourceReader**

Superseded by TraversableResources

An *abstract base class* to provide the ability to read *resources*.

From the perspective of this ABC, a *resource* is a binary artifact that is shipped within a package. Typically this is something like a data file that lives next to the `__init__.py` file of the package. The purpose of this class is to help abstract out the accessing of such data files so that it does not matter if the package and its data file(s) are stored e.g. in a zip file versus on the file system.

For any of methods of this class, a *resource* argument is expected to be a *path-like object* which represents conceptually just a file name. This means that no subdirectory paths should be included in the *resource* argument. This is because the location of the package the reader is for, acts as the "directory". Hence the metaphor for directories and file names is packages and resources, respectively. This is also why instances of this class are expected to directly correlate to a specific package (instead of potentially representing multiple packages or a module).

Loaders that wish to support resource reading are expected to provide a method called `get_resource_reader(fullname)` which returns an object implementing this ABC's interface. If the module specified by fullname is not a package, this method should return *None*. An object compatible with this ABC should only be returned when the specified module is a package.

Deprecated since version 3.12, will be removed in version 3.14: Use *importlib.resources.abc.TraversableResources* instead.

abstractmethod **open_resource**(*resource*)

Returns an opened, *file-like object* for binary reading of the *resource*.

If the resource cannot be found, *FileNotFoundError* is raised.

abstractmethod resource_path(*resource*)

Returns the file system path to the *resource*.

If the resource does not concretely exist on the file system, raise *FileNotFoundError*.

abstractmethod is_resource(*name*)

Returns `True` if the named *name* is considered a resource. *FileNotFoundError* is raised if *name* does not exist.

abstractmethod contents()

Returns an *iterable* of strings over the contents of the package. Do note that it is not required that all names returned by the iterator be actual resources, e.g. it is acceptable to return names for which *is_resource()* would be false.

Allowing non-resource names to be returned is to allow for situations where how a package and its resources are stored are known a priori and the non-resource names would be useful. For instance, returning subdirectory names is allowed so that when it is known that the package and resources are stored on the file system then those subdirectory names can be used directly.

The abstract method returns an iterable of no items.

class importlib.resources.abc.**Traversable**

An object with a subset of *pathlib.Path* methods suitable for traversing directories and opening files.

For a representation of the object on the file-system, use *importlib.resources.as_file()*.

name

Abstract. The base name of this object without any parent references.

abstractmethod iterdir()

Yield Traversable objects in self.

abstractmethod is_dir()

Return `True` if self is a directory.

abstractmethod is_file()

Return `True` if self is a file.

abstractmethod joinpath(**pathsegments*)

Traverse directories according to *pathsegments* and return the result as `Traversable`.

Each *pathsegments* argument may contain multiple names separated by forward slashes (`/`, `posixpath.sep`). For example, the following are equivalent:

```
files.joinpath('subdir', 'subsuddir', 'file.txt')
files.joinpath('subdir/subsuddir/file.txt')
```

Note that some `Traversable` implementations might not be updated to the latest version of the protocol. For compatibility with such implementations, provide a single argument without path separators to each call to `joinpath`. For example:

```
files.joinpath('subdir').joinpath('subsubdir').joinpath('file.txt')
```

Changed in version 3.11: `joinpath` accepts multiple *pathsegments*, and these segments may contain forward slashes as path separators. Previously, only a single *child* argument was accepted.

abstractmethod __truediv__(*child*)

Return Traversable child in self. Equivalent to `joinpath(child)`.

abstractmethod open(*mode='r'*, **args*, ***kwargs*)

mode may be 'r' or 'rb' to open as text or binary. Return a handle suitable for reading (same as *pathlib.Path.open*).

When opening as text, accepts encoding parameters such as those accepted by *io.TextIOWrapper*.

read_bytes()

Read contents of self as bytes.

read_text(*encoding=None*)

Read contents of self as text.

class importlib.resources.abc.**TraversableResources**

An abstract base class for resource readers capable of serving the *importlib.resources.files()* interface. Subclasses *ResourceReader* and provides concrete implementations of the ResourceReader's abstract methods. Therefore, any loader supplying TraversableResources also supplies ResourceReader.

Loaders that wish to support resource reading are expected to implement this interface.

abstractmethod files()

Returns a *importlib.resources.abc.Traversable* object for the loaded package.

31.8 importlib.metadata – Accessing package metadata

Added in version 3.8.

Changed in version 3.10: importlib.metadata is no longer provisional.

Source code: Lib/importlib/metadata/__init__.py

importlib.metadata is a library that provides access to the metadata of an installed Distribution Package, such as its entry points or its top-level names (Import Packages, modules, if any). Built in part on Python's import system, this library intends to replace similar functionality in the entry point API and metadata API of pkg_resources. Along with *importlib.resources*, this package can eliminate the need to use the older and less efficient pkg_resources package.

importlib.metadata operates on third-party *distribution packages* installed into Python's site-packages directory via tools such as pip. Specifically, it works with distributions with discoverable dist-info or egg-info directories, and metadata defined by the Core metadata specifications.

Important

These are *not* necessarily equivalent to or correspond 1:1 with the top-level *import package* names that can be imported inside Python code. One *distribution package* can contain multiple *import packages* (and single modules), and one top-level *import package* may map to multiple *distribution packages* if it is a namespace package. You can use *packages_distributions()* to get a mapping between them.

By default, distribution metadata can live on the file system or in zip archives on *sys.path*. Through an extension mechanism, the metadata can live almost anywhere.

See also

https://importlib-metadata.readthedocs.io/
The documentation for importlib_metadata, which supplies a backport of importlib.metadata.

This includes an API reference for this module's classes and functions, as well as a migration guide for existing users of `pkg_resources`.

31.8.1 Overview

Let's say you wanted to get the version string for a Distribution Package you've installed using `pip`. We start by creating a virtual environment and installing something into it:

```
$ python -m venv example
$ source example/bin/activate
(example) $ python -m pip install wheel
```

You can get the version string for `wheel` by running the following:

```
(example) $ python
>>> from importlib.metadata import version
>>> version('wheel')
'0.32.3'
```

You can also get a collection of entry points selectable by properties of the EntryPoint (typically 'group' or 'name'), such as `console_scripts`, `distutils.commands` and others. Each group contains a collection of *EntryPoint* objects.

You can get the *metadata for a distribution*:

```
>>> list(metadata('wheel'))
['Metadata-Version', 'Name', 'Version', 'Summary', 'Home-page', 'Author', 'Author-
↪email', 'Maintainer', 'Maintainer-email', 'License', 'Project-URL', 'Project-URL',
↪'Project-URL', 'Keywords', 'Platform', 'Classifier', 'Classifier', 'Classifier',
↪'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier',
↪'Classifier', 'Classifier', 'Requires-Python', 'Provides-Extra',
↪'Requires-Dist', 'Requires-Dist']
```

You can also get a *distribution's version number*, list its *constituent files*, and get a list of the distribution's *Distribution requirements*.

31.8.2 Functional API

This package provides the following functionality via its public API.

Entry points

The `entry_points()` function returns a collection of entry points. Entry points are represented by `EntryPoint` instances; each `EntryPoint` has a `.name`, `.group`, and `.value` attributes and a `.load()` method to resolve the value. There are also `.module`, `.attr`, and `.extras` attributes for getting the components of the `.value` attribute.

Query all entry points:

```
>>> eps = entry_points()
```

The `entry_points()` function returns an `EntryPoints` object, a collection of all `EntryPoint` objects with `names` and `groups` attributes for convenience:

```
>>> sorted(eps.groups)
['console_scripts', 'distutils.commands', 'distutils.setup_keywords', 'egg_info.
↪writers', 'setuptools.installation']
```

`EntryPoints` has a `select` method to select entry points matching specific properties. Select entry points in the `console_scripts` group:

```
>>> scripts = eps.select(group='console_scripts')
```

Equivalently, since `entry_points` passes keyword arguments through to select:

```
>>> scripts = entry_points(group='console_scripts')
```

Pick out a specific script named "wheel" (found in the wheel project):

```
>>> 'wheel' in scripts.names
True
>>> wheel = scripts['wheel']
```

Equivalently, query for that entry point during selection:

```
>>> (wheel,) = entry_points(group='console_scripts', name='wheel')
>>> (wheel,) = entry_points().select(group='console_scripts', name='wheel')
```

Inspect the resolved entry point:

```
>>> wheel
EntryPoint(name='wheel', value='wheel.cli:main', group='console_scripts')
>>> wheel.module
'wheel.cli'
>>> wheel.attr
'main'
>>> wheel.extras
[]
>>> main = wheel.load()
>>> main
<function main at 0x103528488>
```

The `group` and `name` are arbitrary values defined by the package author and usually a client will wish to resolve all entry points for a particular group. Read the setuptools docs for more information on entry points, their definition, and usage.

Changed in version 3.12: The "selectable" entry points were introduced in `importlib_metadata` 3.6 and Python 3.10. Prior to those changes, `entry_points` accepted no parameters and always returned a dictionary of entry points, keyed by group. With `importlib_metadata` 5.0 and Python 3.12, `entry_points` always returns an `EntryPoints` object. See backports.entry_points_selectable for compatibility options.

Changed in version 3.13: `EntryPoint` objects no longer present a tuple-like interface (`__getitem__()`).

Distribution metadata

Every Distribution Package includes some metadata, which you can extract using the `metadata()` function:

```
>>> wheel_metadata = metadata('wheel')
```

The keys of the returned data structure, a `PackageMetadata`, name the metadata keywords, and the values are returned unparsed from the distribution metadata:

```
>>> wheel_metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
```

`PackageMetadata` also presents a `json` attribute that returns all the metadata in a JSON-compatible form per **PEP 566**:

```
>>> wheel_metadata.json['requires_python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
```

Note

The actual type of the object returned by `metadata()` is an implementation detail and should be accessed only through the interface described by the PackageMetadata protocol.

Changed in version 3.10: The `Description` is now included in the metadata when presented through the payload. Line continuation characters have been removed.

The `json` attribute was added.

Distribution versions

The `version()` function is the quickest way to get a Distribution Package's version number, as a string:

```
>>> version('wheel')
'0.32.3'
```

Distribution files

You can also get the full set of files contained within a distribution. The `files()` function takes a Distribution Package name and returns all of the files installed by this distribution. Each file object returned is a `PackagePath`, a *`pathlib.PurePath`* derived object with additional `dist`, `size`, and `hash` properties as indicated by the metadata. For example:

```
>>> util = [p for p in files('wheel') if 'util.py' in str(p)][0]
>>> util
PackagePath('wheel/util.py')
>>> util.size
859
>>> util.dist
<importlib.metadata._hooks.PathDistribution object at 0x101e0cef0>
>>> util.hash
<FileHash mode: sha256 value: bYkw5oMccfazVCoYQwKkkemoVyMAFoR34mmKBx8R1NI>
```

Once you have the file, you can also read its contents:

```
>>> print(util.read_text())
import base64
import sys
...
def as_bytes(s):
    if isinstance(s, text_type):
        return s.encode('utf-8')
    return s
```

You can also use the `locate` method to get a the absolute path to the file:

```
>>> util.locate()
PosixPath('/home/gustav/example/lib/site-packages/wheel/util.py')
```

In the case where the metadata file listing files (RECORD or SOURCES.txt) is missing, `files()` will return `None`. The caller may wish to wrap calls to `files()` in always_iterable or otherwise guard against this condition if the target distribution is not known to have the metadata present.

Distribution requirements

To get the full set of requirements for a Distribution Package, use the `requires()` function:

```
>>> requires('wheel')
["pytest (>=3.0.0) ; extra == 'test'", "pytest-cov ; extra == 'test'"]
```

Mapping import to distribution packages

A convenience method to resolve the Distribution Package name (or names, in the case of a namespace package) that provide each importable top-level Python module or Import Package:

```
>>> packages_distributions()
{'importlib_metadata': ['importlib-metadata'], 'yaml': ['PyYAML'], 'jaraco': ['jaraco.
↪classes', 'jaraco.functools'], ...}
```

Some editable installs, do not supply top-level names, and thus this function is not reliable with such installs.

Added in version 3.10.

31.8.3 Distributions

While the above API is the most common and convenient usage, you can get all of that information from the `Distribution` class. A `Distribution` is an abstract object that represents the metadata for a Python Distribution Package. You can get the `Distribution` instance:

```
>>> from importlib.metadata import distribution
>>> dist = distribution('wheel')
```

Thus, an alternative way to get the version number is through the `Distribution` instance:

```
>>> dist.version
'0.32.3'
```

There are all kinds of additional metadata available on the `Distribution` instance:

```
>>> dist.metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
>>> dist.metadata['License']
'MIT'
```

For editable packages, an `origin` property may present **PEP 610** metadata:

```
>>> dist.origin.url
'file:///path/to/wheel-0.32.3.editable-py3-none-any.whl'
```

The full set of available metadata is not described here. See the Core metadata specifications for additional details.

Added in version 3.13: The `.origin` property was added.

31.8.4 Distribution Discovery

By default, this package provides built-in support for discovery of metadata for file system and zip file Distribution Packages. This metadata finder search defaults to `sys.path`, but varies slightly in how it interprets those values from how other import machinery does. In particular:

- `importlib.metadata` does not honor *bytes* objects on `sys.path`.
- `importlib.metadata` will incidentally honor *pathlib.Path* objects on `sys.path` even though such values will be ignored for imports.

31.8.5 Extending the search algorithm

Because Distribution Package metadata is not available through *sys.path* searches, or package loaders directly, the metadata for a distribution is found through import system finders. To find a distribution package's metadata, `importlib.metadata` queries the list of *meta path finders* on *sys.meta_path*.

By default `importlib.metadata` installs a finder for distribution packages found on the file system. This finder doesn't actually find any *distributions*, but it can find their metadata.

The abstract class *importlib.abc.MetaPathFinder* defines the interface expected of finders by Python's import system. `importlib.metadata` extends this protocol by looking for an optional `find_distributions` callable on the finders from *sys.meta_path* and presents this extended interface as the `DistributionFinder` abstract base class, which defines this abstract method:

```
@abc.abstractmethod
def find_distributions(context=DistributionFinder.Context()):
    """Return an iterable of all Distribution instances capable of
    loading the metadata for packages for the indicated ``context``.
    """
```

The `DistributionFinder.Context` object provides `.path` and `.name` properties indicating the path to search and name to match and may supply other relevant context.

What this means in practice is that to support finding distribution package metadata in locations other than the file system, subclass `Distribution` and implement the abstract methods. Then from a custom finder, return instances of this derived `Distribution` in the `find_distributions()` method.

Example

Consider for example a custom finder that loads Python modules from a database:

```
class DatabaseImporter(importlib.abc.MetaPathFinder):
    def __init__(self, db):
        self.db = db

    def find_spec(self, fullname, target=None) -> ModuleSpec:
        return self.db.spec_from_name(fullname)

sys.meta_path.append(DatabaseImporter(connect_db(...)))
```

That importer now presumably provides importable modules from a database, but it provides no metadata or entry points. For this custom importer to provide metadata, it would also need to implement `DistributionFinder`:

```
from importlib.metadata import DistributionFinder

class DatabaseImporter(DistributionFinder):
    ...

    def find_distributions(self, context=DistributionFinder.Context()):
        query = dict(name=context.name) if context.name else {}
        for dist_record in self.db.query_distributions(query):
            yield DatabaseDistribution(dist_record)
```

In this way, `query_distributions` would return records for each distribution served by the database matching the query. For example, if `requests-1.0` is in the database, `find_distributions` would yield a `DatabaseDistribution` for `Context(name='requests')` or `Context(name=None)`.

For the sake of simplicity, this example ignores `context.path`. The `path` attribute defaults to `sys.path` and is the set of import paths to be considered in the search. A `DatabaseImporter` could potentially function without any concern for a search path. Assuming the importer does no partitioning, the "path" would be irrelevant. In order to illustrate the purpose of `path`, the example would need to illustrate a more complex `DatabaseImporter` whose behavior varied depending on `sys.path`/`PYTHONPATH`. In that case, the `find_distributions` should honor the `context.path` and only yield `Distributions` pertinent to that path.

`DatabaseDistribution`, then, would look something like:

```
class DatabaseDistribution(importlib.metadata.Distributon):
    def __init__(self, record):
        self.record = record

    def read_text(self, filename):
        """
        Read a file like "METADATA" for the current distribution.
        """
        if filename == "METADATA":
            return f"""Name: {self.record.name}
Version: {self.record.version}
"""
        if filename == "entry_points.txt":
            return "\n".join(
              f"""[{ep.group}]\n{ep.name}={ep.value}"""
              for ep in self.record.entry_points)

    def locate_file(self, path):
        raise RuntimeError("This distribution has no file system")
```

This basic implementation should provide metadata and entry points for packages served by the `DatabaseImporter`, assuming that the `record` supplies suitable `.name`, `.version`, and `.entry_points` attributes.

The `DatabaseDistribution` may also provide other metadata files, like `RECORD` (required for `Distribution.files`) or override the implementation of `Distribution.files`. See the source for more inspiration.

31.9 The initialization of the `sys.path` module search path

A module search path is initialized when Python starts. This module search path may be accessed at *`sys.path`*.

The first entry in the module search path is the directory that contains the input script, if there is one. Otherwise, the first entry is the current directory, which is the case when executing the interactive shell, a `-c` command, or `-m` module.

The `PYTHONPATH` environment variable is often used to add directories to the search path. If this environment variable is found then the contents are added to the module search path.

Note

`PYTHONPATH` will affect all installed Python versions/environments. Be wary of setting this in your shell profile or global environment variables. The *`site`* module offers more nuanced techniques as mentioned below.

The next items added are the directories containing standard Python modules as well as any *extension module*s that these modules depend on. Extension modules are `.pyd` files on Windows and `.so` files on other platforms. The directory with the platform-independent Python modules is called `prefix`. The directory with the extension modules is called `exec_prefix`.

The `PYTHONHOME` environment variable may be used to set the `prefix` and `exec_prefix` locations. Otherwise these directories are found by using the Python executable as a starting point and then looking for various 'landmark' files and directories. Note that any symbolic links are followed so the real Python executable location is used as the search starting point. The Python executable location is called `home`.

Once `home` is determined, the `prefix` directory is found by first looking for `python`*`majorversionminorversion`*`.zip` (`python311.zip`). On Windows the zip archive is searched for in `home` and on Unix the archive is expected to be in `lib`. Note that the expected zip archive location is added to the module search path even if the archive does not exist. If no archive was found, Python on Windows will continue the search for `prefix` by looking for `Lib\os.py`. Python on Unix will look for `lib/python`*`majorversion`*`.`*`minorversion`*`/os.py` (`lib/python3.11/os.py`). On Windows `prefix` and `exec_prefix` are the same, however on other platforms `lib/python`*`majorversion`*`.`*`minorversion`*`/lib-dynload` (`lib/python3.11/lib-dynload`) is searched for and used as an anchor for `exec_prefix`. On some platforms `lib` may be `lib64` or another value, see *`sys.platlibdir`* and `PYTHONPLATLIBDIR`.

Once found, `prefix` and `exec_prefix` are available at *`sys.prefix`* and *`sys.exec_prefix`* respectively.

Finally, the *`site`* module is processed and `site-packages` directories are added to the module search path. A common way to customize the search path is to create *`sitecustomize`* or *`usercustomize`* modules as described in the *`site`* module documentation.

Note

Certain command line options may further affect path calculations. See `-E`, `-I`, `-s` and `-S` for further details.

31.9.1 Virtual environments

If Python is run in a virtual environment (as described at tut-venv) then `prefix` and `exec_prefix` are specific to the virtual environment.

If a `pyvenv.cfg` file is found alongside the main executable, or in the directory one level above the executable, the following variations apply:

- If `home` is an absolute path and `PYTHONHOME` is not set, this path is used instead of the path to the main executable when deducing `prefix` and `exec_prefix`.

31.9.2 _pth files

To completely override *sys.path* create a `._pth` file with the same name as the shared library or executable (`python._pth` or `python311._pth`). The shared library path is always known on Windows, however it may not be available on other platforms. In the `._pth` file specify one line for each path to add to *sys.path*. The file based on the shared library name overrides the one based on the executable, which allows paths to be restricted for any program loading the runtime if desired.

When the file exists, all registry and environment variables are ignored, isolated mode is enabled, and *site* is not imported unless one line in the file specifies `import site`. Blank paths and lines starting with `#` are ignored. Each path may be absolute or relative to the location of the file. Import statements other than to `site` are not permitted, and arbitrary code cannot be specified.

Note that `.pth` files (without leading underscore) will be processed normally by the *site* module when `import site` has been specified.

31.9.3 Embedded Python

If Python is embedded within another application `Py_InitializeFromConfig()` and the `PyConfig` structure can be used to initialize Python. The path specific details are described at init-path-config.

See also

- windows_finding_modules for detailed Windows notes.
- using-on-unix for Unix details.

CHAPTER

THIRTYTWO

PYTHON LANGUAGE SERVICES

Python provides a number of modules to assist in working with the Python language. These modules support tokenizing, parsing, syntax analysis, bytecode disassembly, and various other facilities.

These modules include:

32.1 `ast` — Abstract Syntax Trees

Source code: Lib/ast.py

The *ast* module helps Python applications to process trees of the Python abstract syntax grammar. The abstract syntax itself might change with each Python release; this module helps to find out programmatically what the current grammar looks like.

An abstract syntax tree can be generated by passing *ast.PyCF_ONLY_AST* as a flag to the *compile()* built-in function, or using the *parse()* helper provided in this module. The result will be a tree of objects whose classes all inherit from *ast.AST*. An abstract syntax tree can be compiled into a Python code object using the built-in *compile()* function.

32.1.1 Abstract Grammar

The abstract grammar is currently defined as follows:

```
-- ASDL's 4 builtin types are:
-- identifier, int, string, constant

module Python
{
    mod = Module(stmt* body, type_ignore* type_ignores)
        | Interactive(stmt* body)
        | Expression(expr body)
        | FunctionType(expr* argtypes, expr returns)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns,
                       string? type_comment, type_param* type_params)
         | AsyncFunctionDef(identifier name, arguments args,
                            stmt* body, expr* decorator_list, expr? returns,
                            string? type_comment, type_param* type_params)
```

(continues on next page)

(continued from previous page)

```
          | ClassDef(identifier name,
             expr* bases,
             keyword* keywords,
             stmt* body,
             expr* decorator_list,
             type_param* type_params)
          | Return(expr? value)

          | Delete(expr* targets)
          | Assign(expr* targets, expr value, string? type_comment)
          | TypeAlias(expr name, type_param* type_params, expr value)
          | AugAssign(expr target, operator op, expr value)
          -- 'simple' indicates that we annotate simple name without parens
          | AnnAssign(expr target, expr annotation, expr? value, int simple)

          -- use 'orelse' because else is a keyword in target languages
          | For(expr target, expr iter, stmt* body, stmt* orelse, string? type_
↪comment)
          | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse, string? type_
↪comment)
          | While(expr test, stmt* body, stmt* orelse)
          | If(expr test, stmt* body, stmt* orelse)
          | With(withitem* items, stmt* body, string? type_comment)
          | AsyncWith(withitem* items, stmt* body, string? type_comment)

          | Match(expr subject, match_case* cases)

          | Raise(expr? exc, expr? cause)
          | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
          | TryStar(stmt* body, excepthandler* handlers, stmt* orelse, stmt*␣
↪finalbody)
          | Assert(expr test, expr? msg)

          | Import(alias* names)
          | ImportFrom(identifier? module, alias* names, int? level)

          | Global(identifier* names)
          | Nonlocal(identifier* names)
          | Expr(expr value)
          | Pass | Break | Continue

          -- col_offset is the byte offset in the utf8 string the parser uses
          attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

          -- BoolOp() can use left & right?
    expr = BoolOp(boolop op, expr* values)
         | NamedExpr(expr target, expr value)
         | BinOp(expr left, operator op, expr right)
         | UnaryOp(unaryop op, expr operand)
         | Lambda(arguments args, expr body)
         | IfExp(expr test, expr body, expr orelse)
         | Dict(expr* keys, expr* values)
         | Set(expr* elts)
         | ListComp(expr elt, comprehension* generators)
         | SetComp(expr elt, comprehension* generators)
         | DictComp(expr key, expr value, comprehension* generators)
```

(continues on next page)

(continued from previous page)

```
         | GeneratorExp(expr elt, comprehension* generators)
         -- the grammar constrains where yield expressions can occur
         | Await(expr value)
         | Yield(expr? value)
         | YieldFrom(expr value)
         -- need sequences for compare to distinguish between
         -- x < 4 < 3 and (x < 4) < 3
         | Compare(expr left, cmpop* ops, expr* comparators)
         | Call(expr func, expr* args, keyword* keywords)
         | FormattedValue(expr value, int conversion, expr? format_spec)
         | JoinedStr(expr* values)
         | Constant(constant value, string? kind)

         -- the following expression can appear in assignment context
         | Attribute(expr value, identifier attr, expr_context ctx)
         | Subscript(expr value, expr slice, expr_context ctx)
         | Starred(expr value, expr_context ctx)
         | Name(identifier id, expr_context ctx)
         | List(expr* elts, expr_context ctx)
         | Tuple(expr* elts, expr_context ctx)

         -- can appear only in Subscript
         | Slice(expr? lower, expr? upper, expr? step)

          -- col_offset is the byte offset in the utf8 string the parser uses
          attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

    expr_context = Load | Store | Del

    boolop = And | Or

    operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
                 | RShift | BitOr | BitXor | BitAnd | FloorDiv

    unaryop = Invert | Not | UAdd | USub

    cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

    comprehension = (expr target, expr iter, expr* ifs, int is_async)

    excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
                    attributes (int lineno, int col_offset, int? end_lineno, int? end_
↪col_offset)

    arguments = (arg* posonlyargs, arg* args, arg? vararg, arg* kwonlyargs,
                 expr* kw_defaults, arg? kwarg, expr* defaults)

    arg = (identifier arg, expr? annotation, string? type_comment)
           attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

    -- keyword arguments supplied to call (NULL identifier for **kwargs)
    keyword = (identifier? arg, expr value)
               attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)
```

(continues on next page)

(continued from previous page)

```
    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)
             attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

    withitem = (expr context_expr, expr? optional_vars)

    match_case = (pattern pattern, expr? guard, stmt* body)

    pattern = MatchValue(expr value)
            | MatchSingleton(constant value)
            | MatchSequence(pattern* patterns)
            | MatchMapping(expr* keys, pattern* patterns, identifier? rest)
            | MatchClass(expr cls, pattern* patterns, identifier* kwd_attrs, pattern*␣
↪kwd_patterns)

            | MatchStar(identifier? name)
            -- The optional "rest" MatchMapping parameter handles capturing extra␣
↪mapping keys

            | MatchAs(pattern? pattern, identifier? name)
            | MatchOr(pattern* patterns)

             attributes (int lineno, int col_offset, int end_lineno, int end_col_
↪offset)

    type_ignore = TypeIgnore(int lineno, string tag)

    type_param = TypeVar(identifier name, expr? bound, expr? default_value)
               | ParamSpec(identifier name, expr? default_value)
               | TypeVarTuple(identifier name, expr? default_value)
              attributes (int lineno, int col_offset, int end_lineno, int end_col_
↪offset)
}
```

32.1.2 Node classes

class ast.**AST**

This is the base of all AST node classes. The actual node classes are derived from the `Parser/Python.asdl` file, which is reproduced *above*. They are defined in the `_ast` C module and re-exported in *ast*.

There is one class defined for each left-hand side symbol in the abstract grammar (for example, `ast.stmt` or `ast.expr`). In addition, there is one class defined for each constructor on the right-hand side; these classes inherit from the classes for the left-hand side trees. For example, *ast.BinOp* inherits from `ast.expr`. For production rules with alternatives (aka "sums"), the left-hand side class is abstract: only instances of specific constructor nodes are ever created.

_fields

Each concrete class has an attribute `_fields` which gives the names of all child nodes.

Each instance of a concrete class has one attribute for each child node, of the type as defined in the grammar. For example, *ast.BinOp* instances have an attribute `left` of type `ast.expr`.

If these attributes are marked as optional in the grammar (using a question mark), the value might be `None`. If the attributes can have zero-or-more values (marked with an asterisk), the values are represented as Python lists. All possible attributes must be present and have valid values when compiling an AST with *compile()*.

_field_types

The `_field_types` attribute on each concrete class is a dictionary mapping field names (as also listed in *_fields*) to their types.

```
>>> ast.TypeVar._field_types
{'name': <class 'str'>, 'bound': ast.expr | None, 'default_value': ast.expr |␣
↪None}
```

Added in version 3.13.

lineno

col_offset

end_lineno

end_col_offset

Instances of `ast.expr` and `ast.stmt` subclasses have *lineno*, *col_offset*, *end_lineno*, and *end_col_offset* attributes. The *lineno* and *end_lineno* are the first and last line numbers of source text span (1-indexed so the first line is line 1) and the *col_offset* and *end_col_offset* are the corresponding UTF-8 byte offsets of the first and last tokens that generated the node. The UTF-8 offset is recorded because the parser uses UTF-8 internally.

Note that the end positions are not required by the compiler and are therefore optional. The end offset is *after* the last symbol, for example one can get the source segment of a one-line expression node using `source_line[node.col_offset : node.end_col_offset]`.

The constructor of a class `ast.T` parses its arguments as follows:

- If there are positional arguments, there must be as many as there are items in `T._fields`; they will be assigned as attributes of these names.
- If there are keyword arguments, they will set the attributes of the same names to the given values.

For example, to create and populate an *ast.UnaryOp* node, you could use

```
node = ast.UnaryOp(ast.USub(), ast.Constant(5, lineno=0, col_offset=0),
                   lineno=0, col_offset=0)
```

If a field that is optional in the grammar is omitted from the constructor, it defaults to `None`. If a list field is omitted, it defaults to the empty list. If a field of type `ast.expr_context` is omitted, it defaults to *Load()*. If any other field is omitted, a *DeprecationWarning* is raised and the AST node will not have this field. In Python 3.15, this condition will raise an error.

Changed in version 3.8: Class *ast.Constant* is now used for all constants.

Changed in version 3.9: Simple indices are represented by their value, extended slices are represented as tuples.

Deprecated since version 3.8: Old classes `ast.Num`, `ast.Str`, `ast.Bytes`, `ast.NameConstant` and `ast.Ellipsis` are still available, but they will be removed in future Python releases. In the meantime, instantiating them will return an instance of a different class.

Deprecated since version 3.9: Old classes `ast.Index` and `ast.ExtSlice` are still available, but they will be removed in future Python releases. In the meantime, instantiating them will return an instance of a different class.

Deprecated since version 3.13, will be removed in version 3.15: Previous versions of Python allowed the creation of AST nodes that were missing required fields. Similarly, AST node constructors allowed arbitrary keyword arguments that were set as attributes of the AST node, even if they did not match any of the fields of the AST node. This behavior is deprecated and will be removed in Python 3.15.

Note

The descriptions of the specific node classes displayed here were initially adapted from the fantastic Green Tree Snakes project and all its contributors.

Root nodes

class ast.**Module**(*body*, *type_ignores*)

A Python module, as with file input. Node type generated by `ast.parse()` in the default `"exec"` *mode*.

body is a `list` of the module's *Statements*.

type_ignores is a `list` of the module's type ignore comments; see `ast.parse()` for more details.

```
>>> print(ast.dump(ast.parse('x = 1'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=Constant(value=1))])
```

class ast.**Expression**(*body*)

A single Python expression input. Node type generated by `ast.parse()` when *mode* is `"eval"`.

body is a single node, one of the *expression types*.

```
>>> print(ast.dump(ast.parse('123', mode='eval'), indent=4))
Expression(
    body=Constant(value=123))
```

class ast.**Interactive**(*body*)

A single interactive input, like in tut-interac. Node type generated by `ast.parse()` when *mode* is `"single"`.

body is a `list` of *statement nodes*.

```
>>> print(ast.dump(ast.parse('x = 1; y = 2', mode='single'), indent=4))
Interactive(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=Constant(value=1)),
        Assign(
            targets=[
                Name(id='y', ctx=Store())],
            value=Constant(value=2))])
```

class ast.**FunctionType**(*argtypes*, *returns*)

A representation of an old-style type comments for functions, as Python versions prior to 3.5 didn't support **PEP 484** annotations. Node type generated by `ast.parse()` when *mode* is `"func_type"`.

Such type comments would look like this:

```
def sum_two_number(a, b):
    # type: (int, int) -> int
    return a + b
```

argtypes is a `list` of *expression nodes*.

returns is a single *expression node*.

```
>>> print(ast.dump(ast.parse('(int, str) -> List[int]', mode='func_type'),␣
↪indent=4))
FunctionType(
    argtypes=[
        Name(id='int', ctx=Load()),
        Name(id='str', ctx=Load())],
    returns=Subscript(
        value=Name(id='List', ctx=Load()),
        slice=Name(id='int', ctx=Load()),
        ctx=Load()))
```

Added in version 3.8.

Literals

class `ast.`**Constant**(*value*)

A constant value. The `value` attribute of the `Constant` literal contains the Python object it represents. The values represented can be simple types such as a number, string or `None`, but also immutable container types (tuples and frozensets) if all of their elements are constant.

```
>>> print(ast.dump(ast.parse('123', mode='eval'), indent=4))
Expression(
    body=Constant(value=123))
```

class `ast.`**FormattedValue**(*value*, *conversion*, *format_spec*)

Node representing a single formatting field in an f-string. If the string contains a single formatting field and nothing else the node can be isolated otherwise it appears in *JoinedStr*.

- `value` is any expression node (such as a literal, a variable, or a function call).
- `conversion` is an integer:
  - -1: no formatting
  - 115: `!s` string formatting
  - 114: `!r` repr formatting
  - 97: `!a` ascii formatting
- `format_spec` is a *JoinedStr* node representing the formatting of the value, or `None` if no format was specified. Both `conversion` and `format_spec` can be set at the same time.

class `ast.`**JoinedStr**(*values*)

An f-string, comprising a series of *FormattedValue* and *Constant* nodes.

```
>>> print(ast.dump(ast.parse('f"sin({a}) is {sin(a):.3}"', mode='eval'),␣
↪indent=4))
Expression(
    body=JoinedStr(
```

(continues on next page)

(continued from previous page)

```
        values=[
            Constant(value='sin('),
            FormattedValue(
                value=Name(id='a', ctx=Load()),
                conversion=-1),
            Constant(value=') is '),
            FormattedValue(
                value=Call(
                    func=Name(id='sin', ctx=Load()),
                    args=[
                        Name(id='a', ctx=Load())]),
                conversion=-1,
                format_spec=JoinedStr(
                    values=[
                        Constant(value='.3')]))]))
```

class ast.**List**(*elts*, *ctx*)

class ast.**Tuple**(*elts*, *ctx*)

A list or tuple. `elts` holds a list of nodes representing the elements. `ctx` is *Store* if the container is an assignment target (i.e. `(x,y)=something`), and *Load* otherwise.

```
>>> print(ast.dump(ast.parse('[1, 2, 3]', mode='eval'), indent=4))
Expression(
    body=List(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)],
        ctx=Load()))
>>> print(ast.dump(ast.parse('(1, 2, 3)', mode='eval'), indent=4))
Expression(
    body=Tuple(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)],
        ctx=Load()))
```

class ast.**Set**(*elts*)

A set. `elts` holds a list of nodes representing the set's elements.

```
>>> print(ast.dump(ast.parse('{1, 2, 3}', mode='eval'), indent=4))
Expression(
    body=Set(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)]))
```

class ast.**Dict**(*keys*, *values*)

A dictionary. `keys` and `values` hold lists of nodes representing the keys and the values respectively, in matching order (what would be returned when calling `dictionary.keys()` and `dictionary.values()`).

When doing dictionary unpacking using dictionary literals the expression to be expanded goes in the `values` list, with a `None` at the corresponding position in `keys`.

```
>>> print(ast.dump(ast.parse('{"a":1, **d}', mode='eval'), indent=4))
Expression(
    body=Dict(
        keys=[
            Constant(value='a'),
            None],
        values=[
            Constant(value=1),
            Name(id='d', ctx=Load())]))
```

Variables

class ast.**Name**(*id*, *ctx*)

A variable name. `id` holds the name as a string, and `ctx` is one of the following types.

class ast.**Load**

class ast.**Store**

class ast.**Del**

Variable references can be used to load the value of a variable, to assign a new value to it, or to delete it. Variable references are given a context to distinguish these cases.

```
>>> print(ast.dump(ast.parse('a'), indent=4))
Module(
    body=[
        Expr(
            value=Name(id='a', ctx=Load()))])

>>> print(ast.dump(ast.parse('a = 1'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='a', ctx=Store())],
            value=Constant(value=1))])

>>> print(ast.dump(ast.parse('del a'), indent=4))
Module(
    body=[
        Delete(
            targets=[
                Name(id='a', ctx=Del())])])
```

class ast.**Starred**(*value*, *ctx*)

A `*var` variable reference. `value` holds the variable, typically a *Name* node. This type must be used when building a *Call* node with `*args`.

```
>>> print(ast.dump(ast.parse('a, *b = it'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Tuple(
                    elts=[
                        Name(id='a', ctx=Store()),
```

(continues on next page)

(continued from previous page)

```
                        Starred(
                            value=Name(id='b', ctx=Store()),
                            ctx=Store())],
                    ctx=Store())],
            value=Name(id='it', ctx=Load()))])
```

Expressions

class ast.**Expr**(*value*)

When an expression, such as a function call, appears as a statement by itself with its return value not used or stored, it is wrapped in this container. `value` holds one of the other nodes in this section, a *Constant*, a *Name*, a *Lambda*, a *Yield* or *YieldFrom* node.

```
>>> print(ast.dump(ast.parse('-a'), indent=4))
Module(
    body=[
        Expr(
            value=UnaryOp(
                op=USub(),
                operand=Name(id='a', ctx=Load()))))])
```

class ast.**UnaryOp**(*op*, *operand*)

A unary operation. `op` is the operator, and `operand` any expression node.

class ast.**UAdd**

class ast.**USub**

class ast.**Not**

class ast.**Invert**

Unary operator tokens. *Not* is the `not` keyword, *Invert* is the ~ operator.

```
>>> print(ast.dump(ast.parse('not x', mode='eval'), indent=4))
Expression(
    body=UnaryOp(
        op=Not(),
        operand=Name(id='x', ctx=Load())))
```

class ast.**BinOp**(*left*, *op*, *right*)

A binary operation (like addition or division). `op` is the operator, and `left` and `right` are any expression nodes.

```
>>> print(ast.dump(ast.parse('x + y', mode='eval'), indent=4))
Expression(
    body=BinOp(
        left=Name(id='x', ctx=Load()),
        op=Add(),
        right=Name(id='y', ctx=Load())))
```

class ast.**Add**

class ast.**Sub**

class ast.**Mult**

class ast.**Div**

class ast.**FloorDiv**

class ast.**Mod**

class ast.**Pow**

class ast.**LShift**

class ast.**RShift**

class ast.**BitOr**

class ast.**BitXor**

class ast.**BitAnd**

class ast.**MatMult**

Binary operator tokens.

class ast.**BoolOp**(*op*, *values*)

A boolean operation, 'or' or 'and'. `op` is *Or* or *And*. `values` are the values involved. Consecutive operations with the same operator, such as `a or b or c`, are collapsed into one node with several values.

This doesn't include `not`, which is a *UnaryOp*.

```
>>> print(ast.dump(ast.parse('x or y', mode='eval'), indent=4))
Expression(
    body=BoolOp(
        op=Or(),
        values=[
            Name(id='x', ctx=Load()),
            Name(id='y', ctx=Load())]))
```

class ast.**And**

class ast.**Or**

Boolean operator tokens.

class ast.**Compare**(*left*, *ops*, *comparators*)

A comparison of two or more values. `left` is the first value in the comparison, `ops` the list of operators, and `comparators` the list of values after the first element in the comparison.

```
>>> print(ast.dump(ast.parse('1 <= a < 10', mode='eval'), indent=4))
Expression(
    body=Compare(
        left=Constant(value=1),
        ops=[
            LtE(),
            Lt()],
        comparators=[
            Name(id='a', ctx=Load()),
            Constant(value=10)]))
```

class ast.**Eq**

class ast.**NotEq**

class ast.**Lt**

class ast.**LtE**

class ast.**Gt**

class ast.**GtE**

class ast.**Is**

class ast.**IsNot**

class ast.**In**

class ast.**NotIn**

Comparison operator tokens.

class ast.**Call**(*func*, *args*, *keywords*)

A function call. `func` is the function, which will often be a *Name* or *Attribute* object. Of the arguments:

- `args` holds a list of the arguments passed by position.
- `keywords` holds a list of *keyword* objects representing arguments passed by keyword.

The `args` and `keywords` arguments are optional and default to empty lists.

```
>>> print(ast.dump(ast.parse('func(a, b=c, *d, **e)', mode='eval'), indent=4))
Expression(
    body=Call(
        func=Name(id='func', ctx=Load()),
        args=[
            Name(id='a', ctx=Load()),
            Starred(
                value=Name(id='d', ctx=Load()),
                ctx=Load())],
        keywords=[
            keyword(
                arg='b',
                value=Name(id='c', ctx=Load())),
            keyword(
                value=Name(id='e', ctx=Load()))]))
```

class ast.**keyword**(*arg*, *value*)

A keyword argument to a function call or class definition. `arg` is a raw string of the parameter name, `value` is a node to pass in.

class ast.**IfExp**(*test*, *body*, *orelse*)

An expression such as `a if b else c`. Each field holds a single node, so in the following example, all three are *Name* nodes.

```
>>> print(ast.dump(ast.parse('a if b else c', mode='eval'), indent=4))
Expression(
    body=IfExp(
        test=Name(id='b', ctx=Load()),
        body=Name(id='a', ctx=Load()),
        orelse=Name(id='c', ctx=Load())))
```

class ast.**Attribute**(*value*, *attr*, *ctx*)

Attribute access, e.g. `d.keys`. `value` is a node, typically a *Name*. `attr` is a bare string giving the name of the attribute, and `ctx` is *Load*, *Store* or *Del* according to how the attribute is acted on.

```
>>> print(ast.dump(ast.parse('snake.colour', mode='eval'), indent=4))
Expression(
    body=Attribute(
        value=Name(id='snake', ctx=Load()),
        attr='colour',
        ctx=Load()))
```

class ast.**NamedExpr**(*target*, *value*)

A named expression. This AST node is produced by the assignment expressions operator (also known as the walrus operator). As opposed to the *Assign* node in which the first argument can be multiple nodes, in this case both `target` and `value` must be single nodes.

```
>>> print(ast.dump(ast.parse('(x := 4)', mode='eval'), indent=4))
Expression(
    body=NamedExpr(
        target=Name(id='x', ctx=Store()),
        value=Constant(value=4)))
```

Added in version 3.8.

Subscripting

class ast.**Subscript**(*value*, *slice*, *ctx*)

A subscript, such as `l[1]`. `value` is the subscripted object (usually sequence or mapping). `slice` is an index, slice or key. It can be a *Tuple* and contain a *Slice*. `ctx` is *Load*, *Store* or *Del* according to the action performed with the subscript.

```
>>> print(ast.dump(ast.parse('l[1:2, 3]', mode='eval'), indent=4))
Expression(
    body=Subscript(
        value=Name(id='l', ctx=Load()),
        slice=Tuple(
            elts=[
                Slice(
                    lower=Constant(value=1),
                    upper=Constant(value=2)),
                Constant(value=3)],
            ctx=Load()),
        ctx=Load()))
```

class ast.**Slice**(*lower*, *upper*, *step*)

Regular slicing (on the form `lower:upper` or `lower:upper:step`). Can occur only inside the *slice* field of *Subscript*, either directly or as an element of *Tuple*.

```
>>> print(ast.dump(ast.parse('l[1:2]', mode='eval'), indent=4))
Expression(
    body=Subscript(
        value=Name(id='l', ctx=Load()),
        slice=Slice(
            lower=Constant(value=1),
            upper=Constant(value=2)),
        ctx=Load()))
```

Comprehensions

class ast.**ListComp**(*elt*, *generators*)

class ast.**SetComp**(*elt*, *generators*)

class ast.**GeneratorExp**(*elt*, *generators*)

class ast.**DictComp**(*key*, *value*, *generators*)

List and set comprehensions, generator expressions, and dictionary comprehensions. `elt` (or `key` and `value`) is a single node representing the part that will be evaluated for each item.

`generators` is a list of *comprehension* nodes.

```
>>> print(ast.dump(
...     ast.parse('[x for x in numbers]', mode='eval'),
...     indent=4,
... ))
Expression(
    body=ListComp(
        elt=Name(id='x', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                is_async=0)]))
>>> print(ast.dump(
...     ast.parse('{x: x**2 for x in numbers}', mode='eval'),
...     indent=4,
... ))
Expression(
    body=DictComp(
        key=Name(id='x', ctx=Load()),
        value=BinOp(
            left=Name(id='x', ctx=Load()),
            op=Pow(),
            right=Constant(value=2)),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                is_async=0)]))
>>> print(ast.dump(
...     ast.parse('{x for x in numbers}', mode='eval'),
...     indent=4,
... ))
Expression(
    body=SetComp(
        elt=Name(id='x', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                is_async=0)]))
```

class ast.**comprehension**(*target*, *iter*, *ifs*, *is_async*)

One `for` clause in a comprehension. `target` is the reference to use for each element - typically a *`Name`* or *`Tuple`* node. `iter` is the object to iterate over. `ifs` is a list of test expressions: each `for` clause can have multiple `ifs`.

`is_async` indicates a comprehension is asynchronous (using an `async for` instead of `for`). The value is an integer (0 or 1).

```
>>> print(ast.dump(ast.parse('[ord(c) for line in file for c in line]', mode='eval
↪'),
...                indent=4)) # Multiple comprehensions in one.
Expression(
    body=ListComp(
        elt=Call(
            func=Name(id='ord', ctx=Load()),
            args=[
                Name(id='c', ctx=Load())]),
```

(continues on next page)

(continued from previous page)

```
        generators=[
            comprehension(
                target=Name(id='line', ctx=Store()),
                iter=Name(id='file', ctx=Load()),
                is_async=0),
            comprehension(
                target=Name(id='c', ctx=Store()),
                iter=Name(id='line', ctx=Load()),
                is_async=0)]))

>>> print(ast.dump(ast.parse('(n**2 for n in it if n>5 if n<10)', mode='eval'),
...                indent=4)) # generator comprehension
Expression(
    body=GeneratorExp(
        elt=BinOp(
            left=Name(id='n', ctx=Load()),
            op=Pow(),
            right=Constant(value=2)),
        generators=[
            comprehension(
                target=Name(id='n', ctx=Store()),
                iter=Name(id='it', ctx=Load()),
                ifs=[
                    Compare(
                        left=Name(id='n', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=5)]),
                    Compare(
                        left=Name(id='n', ctx=Load()),
                        ops=[
                            Lt()],
                        comparators=[
                            Constant(value=10)])],
                is_async=0)]))

>>> print(ast.dump(ast.parse('[i async for i in soc]', mode='eval'),
...                indent=4)) # Async comprehension
Expression(
    body=ListComp(
        elt=Name(id='i', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='i', ctx=Store()),
                iter=Name(id='soc', ctx=Load()),
                is_async=1)]))
```

Statements

class ast.**Assign**(*targets*, *value*, *type_comment*)

An assignment. `targets` is a list of nodes, and `value` is a single node.

Multiple nodes in `targets` represents assigning the same value to each. Unpacking is represented by putting a *Tuple* or *List* within `targets`.

type_comment

`type_comment` is an optional string with the type annotation as a comment.

```
>>> print(ast.dump(ast.parse('a = b = 1'), indent=4)) # Multiple assignment
Module(
    body=[
        Assign(
            targets=[
                Name(id='a', ctx=Store()),
                Name(id='b', ctx=Store())],
            value=Constant(value=1))])

>>> print(ast.dump(ast.parse('a,b = c'), indent=4)) # Unpacking
Module(
    body=[
        Assign(
            targets=[
                Tuple(
                    elts=[
                        Name(id='a', ctx=Store()),
                        Name(id='b', ctx=Store())],
                    ctx=Store())],
            value=Name(id='c', ctx=Load()))])
```

class ast.**AnnAssign**(*target*, *annotation*, *value*, *simple*)

An assignment with a type annotation. `target` is a single node and can be a *Name*, an *Attribute* or a *Subscript*. `annotation` is the annotation, such as a *Constant* or *Name* node. `value` is a single optional node.

`simple` is always either 0 (indicating a "complex" target) or 1 (indicating a "simple" target). A "simple" target consists solely of a *Name* node that does not appear between parentheses; all other targets are considered complex. Only simple targets appear in the `__annotations__` dictionary of modules and classes.

```
>>> print(ast.dump(ast.parse('c: int'), indent=4))
Module(
    body=[
        AnnAssign(
            target=Name(id='c', ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=1)])

>>> print(ast.dump(ast.parse('(a): int = 1'), indent=4)) # Annotation with␣
↪parenthesis
Module(
    body=[
        AnnAssign(
            target=Name(id='a', ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            value=Constant(value=1),
```

(continues on next page)

(continued from previous page)

```
            simple=0)])

>>> print(ast.dump(ast.parse('a.b: int'), indent=4)) # Attribute annotation
Module(
    body=[
        AnnAssign(
            target=Attribute(
                value=Name(id='a', ctx=Load()),
                attr='b',
                ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=0)])

>>> print(ast.dump(ast.parse('a[1]: int'), indent=4)) # Subscript annotation
Module(
    body=[
        AnnAssign(
            target=Subscript(
                value=Name(id='a', ctx=Load()),
                slice=Constant(value=1),
                ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=0)])
```

class ast.**AugAssign**(*target*, *op*, *value*)

Augmented assignment, such as `a += 1`. In the following example, `target` is a *Name* node for `x` (with the *Store* context), `op` is *Add*, and `value` is a *Constant* with value for 1.

The `target` attribute cannot be of class *Tuple* or *List*, unlike the targets of *Assign*.

```
>>> print(ast.dump(ast.parse('x += 2'), indent=4))
Module(
    body=[
        AugAssign(
            target=Name(id='x', ctx=Store()),
            op=Add(),
            value=Constant(value=2))])
```

class ast.**Raise**(*exc*, *cause*)

A `raise` statement. `exc` is the exception object to be raised, normally a *Call* or *Name*, or `None` for a standalone `raise`. `cause` is the optional part for `y` in `raise x from y`.

```
>>> print(ast.dump(ast.parse('raise x from y'), indent=4))
Module(
    body=[
        Raise(
            exc=Name(id='x', ctx=Load()),
            cause=Name(id='y', ctx=Load()))])
```

class ast.**Assert**(*test*, *msg*)

An assertion. `test` holds the condition, such as a *Compare* node. `msg` holds the failure message.

```
>>> print(ast.dump(ast.parse('assert x,y'), indent=4))
Module(
    body=[
        Assert(
```

(continues on next page)

(continued from previous page)

```
            test=Name(id='x', ctx=Load()),
            msg=Name(id='y', ctx=Load()))])
```

class ast.**Delete**(*targets*)

Represents a `del` statement. `targets` is a list of nodes, such as *Name*, *Attribute* or *Subscript* nodes.

```
>>> print(ast.dump(ast.parse('del x,y,z'), indent=4))
Module(
    body=[
        Delete(
            targets=[
                Name(id='x', ctx=Del()),
                Name(id='y', ctx=Del()),
                Name(id='z', ctx=Del())])])
```

class ast.**Pass**

A `pass` statement.

```
>>> print(ast.dump(ast.parse('pass'), indent=4))
Module(
    body=[
        Pass()])
```

class ast.**TypeAlias**(*name*, *type_params*, *value*)

A *type alias* created through the `type` statement. `name` is the name of the alias, `type_params` is a list of *type parameters*, and `value` is the value of the type alias.

```
>>> print(ast.dump(ast.parse('type Alias = int'), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            value=Name(id='int', ctx=Load()))])
```

Added in version 3.12.

Other statements which are only applicable inside functions or loops are described in other sections.

Imports

class ast.**Import**(*names*)

An import statement. `names` is a list of *alias* nodes.

```
>>> print(ast.dump(ast.parse('import x,y,z'), indent=4))
Module(
    body=[
        Import(
            names=[
                alias(name='x'),
                alias(name='y'),
                alias(name='z')])])
```

class ast.**ImportFrom**(*module*, *names*, *level*)

Represents `from x import y`. `module` is a raw string of the 'from' name, without any leading dots, or `None`

for statements such as `from . import foo`. `level` is an integer holding the level of the relative import (0 means absolute import).

```
>>> print(ast.dump(ast.parse('from y import x,y,z'), indent=4))
Module(
    body=[
        ImportFrom(
            module='y',
            names=[
                alias(name='x'),
                alias(name='y'),
                alias(name='z')],
            level=0)])
```

class `ast.`**alias**(*name*, *asname*)

Both parameters are raw strings of the names. `asname` can be `None` if the regular name is to be used.

```
>>> print(ast.dump(ast.parse('from ..foo.bar import a as b, c'), indent=4))
Module(
    body=[
        ImportFrom(
            module='foo.bar',
            names=[
                alias(name='a', asname='b'),
                alias(name='c')],
            level=2)])
```

Control flow

> **Note**
>
> Optional clauses such as `else` are stored as an empty list if they're not present.

class `ast.`**If**(*test*, *body*, *orelse*)

An `if` statement. `test` holds a single node, such as a *`Compare`* node. `body` and `orelse` each hold a list of nodes.

`elif` clauses don't have a special representation in the AST, but rather appear as extra *`If`* nodes within the `orelse` section of the previous one.

```
>>> print(ast.dump(ast.parse("""
... if x:
...     ...
... elif y:
...     ...
... else:
...     ...
... """), indent=4))
Module(
    body=[
        If(
            test=Name(id='x', ctx=Load()),
            body=[
                Expr(
```

(continues on next page)

(continued from previous page)

```
                    value=Constant(value=Ellipsis))],
            orelse=[
                If(
                    test=Name(id='y', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))],
                    orelse=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

class ast.**For**(*target*, *iter*, *body*, *orelse*, *type_comment*)

A `for` loop. `target` holds the variable(s) the loop assigns to, as a single *Name*, *Tuple*, *List*, *Attribute* or *Subscript* node. `iter` holds the item to be looped over, again as a single node. `body` and `orelse` contain lists of nodes to execute. Those in `orelse` are executed if the loop finishes normally, rather than via a `break` statement.

type_comment

`type_comment` is an optional string with the type annotation as a comment.

```
>>> print(ast.dump(ast.parse("""
... for x in y:
...     ...
... else:
...     ...
... """), indent=4))
Module(
    body=[
        For(
            target=Name(id='x', ctx=Store()),
            iter=Name(id='y', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.**While**(*test*, *body*, *orelse*)

A `while` loop. `test` holds the condition, such as a *Compare* node.

```
>> print(ast.dump(ast.parse("""
... while x:
...     ...
... else:
...     ...
... """), indent=4))
Module(
    body=[
        While(
            test=Name(id='x', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
```

(continues on next page)

(continued from previous page)

```
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.**Break**

class ast.**Continue**

The `break` and `continue` statements.

```
>>> print(ast.dump(ast.parse("""\
... for a in b:
...     if a > 5:
...         break
...     else:
...         continue
...
... """), indent=4))
Module(
    body=[
        For(
            target=Name(id='a', ctx=Store()),
            iter=Name(id='b', ctx=Load()),
            body=[
                If(
                    test=Compare(
                        left=Name(id='a', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=5)]),
                    body=[
                        Break()],
                    orelse=[
                        Continue()])])])
```

class ast.**Try**(*body*, *handlers*, *orelse*, *finalbody*)

`try` blocks. All attributes are list of nodes to execute, except for `handlers`, which is a list of *`ExceptHandler`* nodes.

```
>>> print(ast.dump(ast.parse("""
... try:
...     ...
... except Exception:
...     ...
... except OtherException as e:
...     ...
... else:
...     ...
... finally:
...     ...
... """), indent=4))
Module(
    body=[
        Try(
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            handlers=[
```

(continues on next page)

(continued from previous page)

```
                ExceptHandler(
                    type=Name(id='Exception', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                ExceptHandler(
                    type=Name(id='OtherException', ctx=Load()),
                    name='e',
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))],
            finalbody=[
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.**TryStar**(*body*, *handlers*, *orelse*, *finalbody*)

`try` blocks which are followed by `except*` clauses. The attributes are the same as for *Try* but the *ExceptHandler* nodes in `handlers` are interpreted as `except*` blocks rather then `except`.

```
>>> print(ast.dump(ast.parse("""
... try:
...     ...
... except* Exception:
...     ...
... """), indent=4))
Module(
    body=[
        TryStar(
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            handlers=[
                ExceptHandler(
                    type=Name(id='Exception', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.11.

class ast.**ExceptHandler**(*type*, *name*, *body*)

A single `except` clause. `type` is the exception type it will match, typically a *Name* node (or `None` for a catch-all `except:` clause). `name` is a raw string for the name to hold the exception, or `None` if the clause doesn't have `as foo`. `body` is a list of nodes.

```
>>> print(ast.dump(ast.parse("""\
... try:
...     a + 1
... except TypeError:
...     pass
... """), indent=4))
Module(
    body=[
```

(continues on next page)

(continued from previous page)

```
        Try(
            body=[
                Expr(
                    value=BinOp(
                        left=Name(id='a', ctx=Load()),
                        op=Add(),
                        right=Constant(value=1)))],
            handlers=[
                ExceptHandler(
                    type=Name(id='TypeError', ctx=Load()),
                    body=[
                        Pass()])])])
```

class ast.**With**(*items*, *body*, *type_comment*)

A `with` block. `items` is a list of *withitem* nodes representing the context managers, and `body` is the indented block inside the context.

type_comment

`type_comment` is an optional string with the type annotation as a comment.

class ast.**withitem**(*context_expr*, *optional_vars*)

A single context manager in a `with` block. `context_expr` is the context manager, often a *Call* node. `optional_vars` is a *Name*, *Tuple* or *List* for the `as foo` part, or `None` if that isn't used.

```
>>> print(ast.dump(ast.parse("""\
... with a as b, c as d:
...    something(b, d)
... """), indent=4))
Module(
    body=[
        With(
            items=[
                withitem(
                    context_expr=Name(id='a', ctx=Load()),
                    optional_vars=Name(id='b', ctx=Store())),
                withitem(
                    context_expr=Name(id='c', ctx=Load()),
                    optional_vars=Name(id='d', ctx=Store()))],
            body=[
                Expr(
                    value=Call(
                        func=Name(id='something', ctx=Load()),
                        args=[
                            Name(id='b', ctx=Load()),
                            Name(id='d', ctx=Load())])])])
```

Pattern matching

class ast.**Match**(*subject*, *cases*)

A `match` statement. `subject` holds the subject of the match (the object that is being matched against the cases) and `cases` contains an iterable of *match_case* nodes with the different cases.

Added in version 3.10.

class ast.**match_case**(*pattern*, *guard*, *body*)

A single case pattern in a `match` statement. `pattern` contains the match pattern that the subject will be matched against. Note that the *AST* nodes produced for patterns differ from those produced for expressions, even when they share the same syntax.

The `guard` attribute contains an expression that will be evaluated if the pattern matches the subject.

`body` contains a list of nodes to execute if the pattern matches and the result of evaluating the guard expression is true.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] if x>0:
...         ...
...     case tuple():
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchAs(name='x')]),
                    guard=Compare(
                        left=Name(id='x', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=0)]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='tuple', ctx=Load())),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchValue**(*value*)

A match literal or value pattern that compares by equality. `value` is an expression node. Permitted value nodes are restricted as described in the match statement documentation. This pattern succeeds if the match subject is equal to the evaluated value.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case "Relevant":
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchValue(
                        value=Constant(value='Relevant')),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchSingleton**(*value*)

A match literal pattern that compares by identity. `value` is the singleton to be compared against: `None`, `True`, or `False`. This pattern succeeds if the match subject is the given constant.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case None:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSingleton(value=None),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchSequence**(*patterns*)

A match sequence pattern. `patterns` contains the patterns to be matched against the subject elements if the subject is a sequence. Matches a variable length sequence if one of the subpatterns is a `MatchStar` node, otherwise matches a fixed length sequence.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [1, 2]:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
```

(continues on next page)

(continued from previous page)

```
                    pattern=MatchSequence(
                        patterns=[
                            MatchValue(
                                value=Constant(value=1)),
                            MatchValue(
                                value=Constant(value=2))]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])]))
```

Added in version 3.10.

class ast.**MatchStar**(*name*)

Matches the rest of the sequence in a variable length match sequence pattern. If `name` is not `None`, a list containing the remaining sequence elements is bound to that name if the overall sequence pattern is successful.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [1, 2, *rest]:
...         ...
...     case [*_]:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchValue(
                                value=Constant(value=1)),
                            MatchValue(
                                value=Constant(value=2)),
                            MatchStar(name='rest')]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchStar()]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchMapping**(*keys*, *patterns*, *rest*)

A match mapping pattern. `keys` is a sequence of expression nodes. `patterns` is a corresponding sequence of pattern nodes. `rest` is an optional name that can be specified to capture the remaining mapping elements. Permitted key expressions are restricted as described in the match statement documentation.

This pattern succeeds if the subject is a mapping, all evaluated key expressions are present in the mapping, and the value corresponding to each key matches the corresponding subpattern. If `rest` is not `None`, a dict containing the remaining mapping elements is bound to that name if the overall mapping pattern is successful.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case {1: _, 2: _}:
...         ...
...     case {**rest}:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchMapping(
                        keys=[
                            Constant(value=1),
                            Constant(value=2)],
                        patterns=[
                            MatchAs(),
                            MatchAs()]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchMapping(rest='rest'),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchClass**(*cls*, *patterns*, *kwd_attrs*, *kwd_patterns*)

A match class pattern. `cls` is an expression giving the nominal class to be matched. `patterns` is a sequence of pattern nodes to be matched against the class defined sequence of pattern matching attributes. `kwd_attrs` is a sequence of additional attributes to be matched (specified as keyword arguments in the class pattern), `kwd_patterns` are the corresponding patterns (specified as keyword values in the class pattern).

This pattern succeeds if the subject is an instance of the nominated class, all positional patterns match the corresponding class-defined attributes, and any specified keyword attributes match their corresponding pattern.

Note: classes may define a property that returns self in order to match a pattern node against the instance being matched. Several builtin types are also matched that way, as described in the match statement documentation.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case Point2D(0, 0):
...         ...
...     case Point3D(x=0, y=0, z=0):
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='Point2D', ctx=Load()),
```

(continues on next page)

(continued from previous page)

```
                        patterns=[
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0))]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='Point3D', ctx=Load()),
                        kwd_attrs=[
                            'x',
                            'y',
                            'z'],
                        kwd_patterns=[
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0))]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchAs**(*pattern*, *name*)

A match "as-pattern", capture pattern or wildcard pattern. `pattern` contains the match pattern that the subject will be matched against. If the pattern is `None`, the node represents a capture pattern (i.e a bare name) and will always succeed.

The `name` attribute contains the name that will be bound if the pattern is successful. If `name` is `None`, `pattern` must also be `None` and the node represents the wildcard pattern.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] as y:
...         ...
...     case _:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchAs(
                        pattern=MatchSequence(
                            patterns=[
                                MatchAs(name='x')]),
                        name='y'),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
```

(continues on next page)

(continued from previous page)

```
                match_case(
                    pattern=MatchAs(),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchOr**(*patterns*)

A match "or-pattern". An or-pattern matches each of its subpatterns in turn to the subject, until one succeeds. The or-pattern is then deemed to succeed. If none of the subpatterns succeed the or-pattern fails. The `patterns` attribute contains a list of match pattern nodes that will be matched against the subject.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] | (y):
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchOr(
                        patterns=[
                            MatchSequence(
                                patterns=[
                                    MatchAs(name='x')]),
                            MatchAs(name='y')]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

Type parameters

Type parameters can exist on classes, functions, and type aliases.

class ast.**TypeVar**(*name*, *bound*, *default_value*)

A *typing.TypeVar*. *name* is the name of the type variable. *bound* is the bound or constraints, if any. If *bound* is a *Tuple*, it represents constraints; otherwise it represents the bound. *default_value* is the default value; if the `TypeVar` has no default, this attribute will be set to `None`.

```
>>> print(ast.dump(ast.parse("type Alias[T: int = bool] = list[T]"), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                TypeVar(
                    name='T',
                    bound=Name(id='int', ctx=Load()),
                    default_value=Name(id='bool', ctx=Load()))],
```

(continues on next page)

(continued from previous page)

```
            value=Subscript(
                value=Name(id='list', ctx=Load()),
                slice=Name(id='T', ctx=Load()),
                ctx=Load())])
```

Added in version 3.12.

Changed in version 3.13: Added the *default_value* parameter.

class ast.**ParamSpec**(*name*, *default_value*)

A *typing.ParamSpec*. *name* is the name of the parameter specification. *default_value* is the default value; if the `ParamSpec` has no default, this attribute will be set to `None`.

```
>>> print(ast.dump(ast.parse("type Alias[**P = (int, str)] = Callable[P, int]"),␣
↪indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                ParamSpec(
                    name='P',
                    default_value=Tuple(
                        elts=[
                            Name(id='int', ctx=Load()),
                            Name(id='str', ctx=Load())],
                        ctx=Load()))],
            value=Subscript(
                value=Name(id='Callable', ctx=Load()),
                slice=Tuple(
                    elts=[
                        Name(id='P', ctx=Load()),
                        Name(id='int', ctx=Load())],
                    ctx=Load()),
                ctx=Load()))])
```

Added in version 3.12.

Changed in version 3.13: Added the *default_value* parameter.

class ast.**TypeVarTuple**(*name*, *default_value*)

A *typing.TypeVarTuple*. *name* is the name of the type variable tuple. *default_value* is the default value; if the `TypeVarTuple` has no default, this attribute will be set to `None`.

```
>>> print(ast.dump(ast.parse("type Alias[*Ts = ()] = tuple[*Ts]"), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                TypeVarTuple(
                    name='Ts',
                    default_value=Tuple(ctx=Load()))],
            value=Subscript(
                value=Name(id='tuple', ctx=Load()),
                slice=Tuple(
                    elts=[
```

(continues on next page)

(continued from previous page)

```
                        Starred(
                            value=Name(id='Ts', ctx=Load()),
                            ctx=Load())],
                    ctx=Load()),
                ctx=Load()))])
```

Added in version 3.12.

Changed in version 3.13: Added the *default_value* parameter.

Function and class definitions

class ast.**FunctionDef**(*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*, *type_params*)

A function definition.

- `name` is a raw string of the function name.
- `args` is an *arguments* node.
- `body` is the list of nodes inside the function.
- `decorator_list` is the list of decorators to be applied, stored outermost first (i.e. the first in the list will be applied last).
- `returns` is the return annotation.
- `type_params` is a list of *type parameters*.

type_comment

`type_comment` is an optional string with the type annotation as a comment.

Changed in version 3.12: Added `type_params`.

class ast.**Lambda**(*args*, *body*)

`lambda` is a minimal function definition that can be used inside an expression. Unlike *FunctionDef*, `body` holds a single node.

```
>>> print(ast.dump(ast.parse('lambda x,y: ...'), indent=4))
Module(
    body=[
        Expr(
            value=Lambda(
                args=arguments(
                    args=[
                        arg(arg='x'),
                        arg(arg='y')]),
                body=Constant(value=Ellipsis)))])
```

class ast.**arguments**(*posonlyargs*, *args*, *vararg*, *kwonlyargs*, *kw_defaults*, *kwarg*, *defaults*)

The arguments for a function.

- `posonlyargs`, `args` and `kwonlyargs` are lists of *arg* nodes.
- `vararg` and `kwarg` are single *arg* nodes, referring to the `*args, **kwargs` parameters.
- `kw_defaults` is a list of default values for keyword-only arguments. If one is `None`, the corresponding argument is required.
- `defaults` is a list of default values for arguments that can be passed positionally. If there are fewer defaults, they correspond to the last n arguments.

class ast.**arg**(*arg*, *annotation*, *type_comment*)

A single argument in a list. `arg` is a raw string of the argument name; `annotation` is its annotation, such as a *Name* node.

type_comment

`type_comment` is an optional string with the type annotation as a comment

```
>>> print(ast.dump(ast.parse("""\
... @decorator1
... @decorator2
... def f(a: 'annotation', b=1, c=2, *d, e, f=3, **g) -> 'return annotation':
...     pass
... """), indent=4))
Module(
    body=[
        FunctionDef(
            name='f',
            args=arguments(
                args=[
                    arg(
                        arg='a',
                        annotation=Constant(value='annotation')),
                    arg(arg='b'),
                    arg(arg='c')],
                vararg=arg(arg='d'),
                kwonlyargs=[
                    arg(arg='e'),
                    arg(arg='f')],
                kw_defaults=[
                    None,
                    Constant(value=3)],
                kwarg=arg(arg='g'),
                defaults=[
                    Constant(value=1),
                    Constant(value=2)]),
            body=[
                Pass()],
            decorator_list=[
                Name(id='decorator1', ctx=Load()),
                Name(id='decorator2', ctx=Load())],
            returns=Constant(value='return annotation'))])
```

class ast.**Return**(*value*)

A `return` statement.

```
>>> print(ast.dump(ast.parse('return 4'), indent=4))
Module(
    body=[
        Return(
            value=Constant(value=4))])
```

class ast.**Yield**(*value*)

class ast.**YieldFrom**(*value*)

A `yield` or `yield from` expression. Because these are expressions, they must be wrapped in an *Expr* node if the value sent back is not used.

```
>>> print(ast.dump(ast.parse('yield x'), indent=4))
Module(
    body=[
        Expr(
            value=Yield(
                value=Name(id='x', ctx=Load()))))])

>>> print(ast.dump(ast.parse('yield from x'), indent=4))
Module(
    body=[
        Expr(
            value=YieldFrom(
                value=Name(id='x', ctx=Load()))))])
```

class ast.**Global**(*names*)

class ast.**Nonlocal**(*names*)

`global` and `nonlocal` statements. `names` is a list of raw strings.

```
>>> print(ast.dump(ast.parse('global x,y,z'), indent=4))
Module(
    body=[
        Global(
            names=[
                'x',
                'y',
                'z'])])

>>> print(ast.dump(ast.parse('nonlocal x,y,z'), indent=4))
Module(
    body=[
        Nonlocal(
            names=[
                'x',
                'y',
                'z'])])
```

class ast.**ClassDef**(*name*, *bases*, *keywords*, *body*, *decorator_list*, *type_params*)

A class definition.

- `name` is a raw string for the class name
- `bases` is a list of nodes for explicitly specified base classes.
- `keywords` is a list of *`keyword`* nodes, principally for 'metaclass'. Other keywords will be passed to the metaclass, as per PEP-3115.
- `body` is a list of nodes representing the code within the class definition.
- `decorator_list` is a list of nodes, as in *`FunctionDef`*.
- `type_params` is a list of *type parameters*.

```
>>> print(ast.dump(ast.parse("""\
... @decorator1
... @decorator2
... class Foo(base1, base2, metaclass=meta):
...     pass
... """), indent=4))
```

(continues on next page)

(continued from previous page)

```
Module(
    body=[
        ClassDef(
            name='Foo',
            bases=[
                Name(id='base1', ctx=Load()),
                Name(id='base2', ctx=Load())],
            keywords=[
                keyword(
                    arg='metaclass',
                    value=Name(id='meta', ctx=Load()))],
            body=[
                Pass()],
            decorator_list=[
                Name(id='decorator1', ctx=Load()),
                Name(id='decorator2', ctx=Load())])])
```

Changed in version 3.12: Added `type_params`.

Async and await

class `ast.`**AsyncFunctionDef**(*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*, *type_params*)

An `async def` function definition. Has the same fields as *`FunctionDef`*.

Changed in version 3.12: Added `type_params`.

class `ast.`**Await**(*value*)

An `await` expression. `value` is what it waits for. Only valid in the body of an *`AsyncFunctionDef`*.

```
>>> print(ast.dump(ast.parse("""\
... async def f():
...     await other_func()
... """), indent=4))
Module(
    body=[
        AsyncFunctionDef(
            name='f',
            args=arguments(),
            body=[
                Expr(
                    value=Await(
                        value=Call(
                            func=Name(id='other_func', ctx=Load())))))])])
```

class `ast.`**AsyncFor**(*target*, *iter*, *body*, *orelse*, *type_comment*)

class `ast.`**AsyncWith**(*items*, *body*, *type_comment*)

`async for` loops and `async with` context managers. They have the same fields as *`For`* and *`With`*, respectively. Only valid in the body of an *`AsyncFunctionDef`*.

Note

When a string is parsed by *`ast.parse()`*, operator nodes (subclasses of `ast.operator`, `ast.unaryop`, `ast.cmpop`, `ast.boolop` and `ast.expr_context`) on the returned tree will be singletons. Changes to one will be reflected in all other occurrences of the same value (e.g. *`ast.Add`*).

32.1.3 ast Helpers

Apart from the node classes, the *ast* module defines these utility functions and classes for traversing abstract syntax trees:

ast.**parse**(*source*, *filename='<unknown>'*, *mode='exec'*, *, *type_comments=False*, *feature_version=None*, *optimize=-1*)

Parse the source into an AST node. Equivalent to `compile(source, filename, mode, flags=FLAGS_VALUE, optimize=optimize)`, where `FLAGS_VALUE` is `ast.PyCF_ONLY_AST` if `optimize <= 0` and `ast.PyCF_OPTIMIZED_AST` otherwise.

If `type_comments=True` is given, the parser is modified to check and return type comments as specified by **PEP 484** and **PEP 526**. This is equivalent to adding *ast.PyCF_TYPE_COMMENTS* to the flags passed to *compile()*. This will report syntax errors for misplaced type comments. Without this flag, type comments will be ignored, and the `type_comment` field on selected AST nodes will always be `None`. In addition, the locations of `# type: ignore` comments will be returned as the `type_ignores` attribute of *Module* (otherwise it is always an empty list).

In addition, if `mode` is `'func_type'`, the input syntax is modified to correspond to **PEP 484** "signature type comments", e.g. `(str, int) -> List[str]`.

Setting `feature_version` to a tuple `(major, minor)` will result in a "best-effort" attempt to parse using that Python version's grammar. For example, setting `feature_version=(3, 9)` will attempt to disallow parsing of `match` statements. Currently `major` must equal to `3`. The lowest supported version is `(3, 7)` (and this may increase in future Python versions); the highest is `sys.version_info[0:2]`. "Best-effort" attempt means there is no guarantee that the parse (or success of the parse) is the same as when run on the Python version corresponding to `feature_version`.

If source contains a null character (`\0`), *ValueError* is raised.

> **Warning**
>
> Note that successfully parsing source code into an AST object doesn't guarantee that the source code provided is valid Python code that can be executed as the compilation step can raise further *SyntaxError* exceptions. For instance, the source `return 42` generates a valid AST node for a return statement, but it cannot be compiled alone (it needs to be inside a function node).
>
> In particular, *ast.parse()* won't do any scoping checks, which the compilation step does.

> **Warning**
>
> It is possible to crash the Python interpreter with a sufficiently large/complex string due to stack depth limitations in Python's AST compiler.

Changed in version 3.8: Added `type_comments`, `mode='func_type'` and `feature_version`.

Changed in version 3.13: The minimum supported version for `feature_version` is now `(3, 7)`. The `optimize` argument was added.

ast.**unparse**(*ast_obj*)

Unparse an *ast.AST* object and generate a string with code that would produce an equivalent *ast.AST* object if parsed back with *ast.parse()*.

> **Warning**
>
> The produced code string will not necessarily be equal to the original code that generated the *ast.AST* object (without any compiler optimizations, such as constant tuples/frozensets).

> **Warning**
>
> Trying to unparse a highly complex expression would result with *RecursionError*.

Added in version 3.9.

ast.**literal_eval**(*node_or_string*)

Evaluate an expression node or a string containing only a Python literal or container display. The string or node provided may only consist of the following Python literal structures: strings, bytes, numbers, tuples, lists, dicts, sets, booleans, `None` and `Ellipsis`.

This can be used for evaluating strings containing Python values without the need to parse the values oneself. It is not capable of evaluating arbitrarily complex expressions, for example involving operators or indexing.

This function had been documented as "safe" in the past without defining what that meant. That was misleading. This is specifically designed not to execute Python code, unlike the more general *eval()*. There is no namespace, no name lookups, or ability to call out. But it is not free from attack: A relatively small input can lead to memory exhaustion or to C stack exhaustion, crashing the process. There is also the possibility for excessive CPU consumption denial of service on some inputs. Calling it on untrusted data is thus not recommended.

> **Warning**
>
> It is possible to crash the Python interpreter due to stack depth limitations in Python's AST compiler.
>
> It can raise *ValueError*, *TypeError*, *SyntaxError*, *MemoryError* and *RecursionError* depending on the malformed input.

Changed in version 3.2: Now allows bytes and set literals.

Changed in version 3.9: Now supports creating empty sets with `'set()'`.

Changed in version 3.10: For string inputs, leading spaces and tabs are now stripped.

ast.**get_docstring**(*node*, *clean=True*)

Return the docstring of the given *node* (which must be a *FunctionDef*, *AsyncFunctionDef*, *ClassDef*, or *Module* node), or `None` if it has no docstring. If *clean* is true, clean up the docstring's indentation with *inspect.cleandoc()*.

Changed in version 3.5: *AsyncFunctionDef* is now supported.

ast.**get_source_segment**(*source*, *node*, *, *padded=False*)

Get source code segment of the *source* that generated *node*. If some location information (*lineno*, *end_lineno*, *col_offset*, or *end_col_offset*) is missing, return `None`.

If *padded* is `True`, the first line of a multi-line statement will be padded with spaces to match its original position.

Added in version 3.8.

ast.**fix_missing_locations**(*node*)

When you compile a node tree with *compile()*, the compiler expects *lineno* and *col_offset* attributes

for every node that supports them. This is rather tedious to fill in for generated nodes, so this helper adds these attributes recursively where not already set, by setting them to the values of the parent node. It works recursively starting at *node*.

`ast.`**`increment_lineno`**(*node*, *n=1*)

Increment the line number and end line number of each node in the tree starting at *node* by *n*. This is useful to "move code" to a different location in a file.

`ast.`**`copy_location`**(*new_node*, *old_node*)

Copy source location (`lineno`, `col_offset`, `end_lineno`, and `end_col_offset`) from *old_node* to *new_node* if possible, and return *new_node*.

`ast.`**`iter_fields`**(*node*)

Yield a tuple of `(fieldname, value)` for each field in `node._fields` that is present on *node*.

`ast.`**`iter_child_nodes`**(*node*)

Yield all direct child nodes of *node*, that is, all fields that are nodes and all items of fields that are lists of nodes.

`ast.`**`walk`**(*node*)

Recursively yield all descendant nodes in the tree starting at *node* (including *node* itself), in no specified order. This is useful if you only want to modify nodes in place and don't care about the context.

class `ast.`**`NodeVisitor`**

A node visitor base class that walks the abstract syntax tree and calls a visitor function for every node found. This function may return a value which is forwarded by the `visit()` method.

This class is meant to be subclassed, with the subclass adding visitor methods.

`visit`(*node*)

Visit a node. The default implementation calls the method called `self.visit_`*classname* where *classname* is the name of the node class, or `generic_visit()` if that method doesn't exist.

`generic_visit`(*node*)

This visitor calls `visit()` on all children of the node.

Note that child nodes of nodes that have a custom visitor method won't be visited unless the visitor calls `generic_visit()` or visits them itself.

`visit_Constant`(*node*)

Handles all constant nodes.

Don't use the `NodeVisitor` if you want to apply changes to nodes during traversal. For this a special visitor exists (`NodeTransformer`) that allows modifications.

Deprecated since version 3.8: Methods `visit_Num()`, `visit_Str()`, `visit_Bytes()`, `visit_NameConstant()` and `visit_Ellipsis()` are deprecated now and will not be called in future Python versions. Add the `visit_Constant()` method to handle all constant nodes.

class `ast.`**`NodeTransformer`**

A `NodeVisitor` subclass that walks the abstract syntax tree and allows modification of nodes.

The `NodeTransformer` will walk the AST and use the return value of the visitor methods to replace or remove the old node. If the return value of the visitor method is `None`, the node will be removed from its location, otherwise it is replaced with the return value. The return value may be the original node in which case no replacement takes place.

Here is an example transformer that rewrites all occurrences of name lookups (`foo`) to `data['foo']`:

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Constant(value=node.id),
            ctx=node.ctx
        )
```

Keep in mind that if the node you're operating on has child nodes you must either transform the child nodes yourself or call the *generic_visit()* method for the node first.

For nodes that were part of a collection of statements (that applies to all statement nodes), the visitor may also return a list of nodes rather than just a single node.

If *NodeTransformer* introduces new nodes (that weren't part of original tree) without giving them location information (such as *lineno*), *fix_missing_locations()* should be called with the new sub-tree to recalculate the location information:

```
tree = ast.parse('foo', mode='eval')
new_tree = fix_missing_locations(RewriteName().visit(tree))
```

Usually you use the transformer like this:

```
node = YourTransformer().visit(node)
```

ast.**dump**(*node*, *annotate_fields=True*, *include_attributes=False*, ***, *indent=None*, *show_empty=False*)

Return a formatted dump of the tree in *node*. This is mainly useful for debugging purposes. If *annotate_fields* is true (by default), the returned string will show the names and the values for fields. If *annotate_fields* is false, the result string will be more compact by omitting unambiguous field names. Attributes such as line numbers and column offsets are not dumped by default. If this is wanted, *include_attributes* can be set to true.

If *indent* is a non-negative integer or string, then the tree will be pretty-printed with that indent level. An indent level of 0, negative, or `""` will only insert newlines. `None` (the default) selects the single line representation. Using a positive integer indent indents that many spaces per level. If *indent* is a string (such as `"\t"`), that string is used to indent each level.

If *show_empty* is `False` (the default), empty lists and fields that are `None` will be omitted from the output.

Changed in version 3.9: Added the *indent* option.

Changed in version 3.13: Added the *show_empty* option.

```
>>> print(ast.dump(ast.parse("""\
... async def f():
...     await other_func()
... """), indent=4, show_empty=True))
Module(
    body=[
        AsyncFunctionDef(
            name='f',
            args=arguments(
                posonlyargs=[],
                args=[],
                kwonlyargs=[],
                kw_defaults=[],
                defaults=[]),
            body=[
```

(continues on next page)

(continued from previous page)

```
                Expr(
                    value=Await(
                        value=Call(
                            func=Name(id='other_func', ctx=Load()),
                            args=[],
                            keywords=[]))),
            decorator_list=[],
            type_params=[])],
    type_ignores=[])
```

32.1.4 Compiler Flags

The following flags may be passed to *compile()* in order to change effects on the compilation of a program:

ast.**PyCF_ALLOW_TOP_LEVEL_AWAIT**

Enables support for top-level `await`, `async for`, `async with` and async comprehensions.

Added in version 3.8.

ast.**PyCF_ONLY_AST**

Generates and returns an abstract syntax tree instead of returning a compiled code object.

ast.**PyCF_OPTIMIZED_AST**

The returned AST is optimized according to the *optimize* argument in *compile()* or *ast.parse()*.

Added in version 3.13.

ast.**PyCF_TYPE_COMMENTS**

Enables support for PEP 484 and PEP 526 style type comments (`# type: <type>`, `# type: ignore <stuff>`).

Added in version 3.8.

32.1.5 Command-Line Usage

Added in version 3.9.

The *ast* module can be executed as a script from the command line. It is as simple as:

```
python -m ast [-m <mode>] [-a] [infile]
```

The following options are accepted:

-h, --help

Show the help message and exit.

-m <mode>

--mode <mode>

Specify what kind of code must be compiled, like the *mode* argument in *parse()*.

--no-type-comments

Don't parse type comments.

-a, --include-attributes

Include attributes such as line numbers and column offsets.

-i <indent>

--indent <indent>

Indentation of nodes in AST (number of spaces).

If `infile` is specified its contents are parsed to AST and dumped to stdout. Otherwise, the content is read from stdin.

See also

Green Tree Snakes, an external documentation resource, has good details on working with Python ASTs.

ASTTokens annotates Python ASTs with the positions of tokens and text in the source code that generated them. This is helpful for tools that make source code transformations.

leoAst.py unifies the token-based and parse-tree-based views of python programs by inserting two-way links between tokens and ast nodes.

LibCST parses code as a Concrete Syntax Tree that looks like an ast tree and keeps all formatting details. It's useful for building automated refactoring (codemod) applications and linters.

Parso is a Python parser that supports error recovery and round-trip parsing for different Python versions (in multiple Python versions). Parso is also able to list multiple syntax errors in your Python file.

32.2 `symtable` — Access to the compiler's symbol tables

Source code: Lib/symtable.py

Symbol tables are generated by the compiler from AST just before bytecode is generated. The symbol table is responsible for calculating the scope of every identifier in the code. *symtable* provides an interface to examine these tables.

32.2.1 Generating Symbol Tables

symtable.**symtable**(*code*, *filename*, *compile_type*)

Return the toplevel *SymbolTable* for the Python source *code*. *filename* is the name of the file containing the code. *compile_type* is like the *mode* argument to *compile()*.

32.2.2 Examining Symbol Tables

class symtable.**SymbolTableType**

An enumeration indicating the type of a *SymbolTable* object.

MODULE = "module"

Used for the symbol table of a module.

FUNCTION = "function"

Used for the symbol table of a function.

CLASS = "class"

Used for the symbol table of a class.

The following members refer to different flavors of annotation scopes.

ANNOTATION = "annotation"

Used for annotations if `from __future__ import annotations` is active.

TYPE_ALIAS = "type alias"

Used for the symbol table of `type` constructions.

TYPE_PARAMETERS = "type parameters"

Used for the symbol table of generic functions or generic classes.

TYPE_VARIABLE = "type variable"

Used for the symbol table of the bound, the constraint tuple or the default value of a single type variable in the formal sense, i.e., a TypeVar, a TypeVarTuple or a ParamSpec object (the latter two do not support a bound or a constraint tuple).

Added in version 3.13.

class symtable.**SymbolTable**

A namespace table for a block. The constructor is not public.

get_type()

Return the type of the symbol table. Possible values are members of the *SymbolTableType* enumeration.

Changed in version 3.12: Added `'annotation'`, `'TypeVar bound'`, `'type alias'`, and `'type parameter'` as possible return values.

Changed in version 3.13: Return values are members of the *SymbolTableType* enumeration.

The exact values of the returned string may change in the future, and thus, it is recommended to use *SymbolTableType* members instead of hard-coded strings.

get_id()

Return the table's identifier.

get_name()

Return the table's name. This is the name of the class if the table is for a class, the name of the function if the table is for a function, or `'top'` if the table is global (*get_type()* returns `'module'`). For type parameter scopes (which are used for generic classes, functions, and type aliases), it is the name of the underlying class, function, or type alias. For type alias scopes, it is the name of the type alias. For *TypeVar* bound scopes, it is the name of the `TypeVar`.

get_lineno()

Return the number of the first line in the block this table represents.

is_optimized()

Return `True` if the locals in this table can be optimized.

is_nested()

Return `True` if the block is a nested class or function.

has_children()

Return `True` if the block has nested namespaces within it. These can be obtained with *get_children()*.

get_identifiers()

Return a view object containing the names of symbols in the table. See the *documentation of view objects*.

lookup(*name*)

Lookup *name* in the table and return a *Symbol* instance.

get_symbols()

Return a list of *Symbol* instances for names in the table.

get_children()

Return a list of the nested symbol tables.

class symtable.**Function**

A namespace for a function or method. This class inherits from *SymbolTable*.

get_parameters()

Return a tuple containing names of parameters to this function.

get_locals()

Return a tuple containing names of locals in this function.

get_globals()

Return a tuple containing names of globals in this function.

get_nonlocals()

Return a tuple containing names of nonlocals in this function.

get_frees()

Return a tuple containing names of free variables in this function.

class symtable.**Class**

A namespace of a class. This class inherits from *SymbolTable*.

get_methods()

Return a tuple containing the names of method-like functions declared in the class.

Here, the term 'method' designates *any* function defined in the class body via `def` or `async def`.

Functions defined in a deeper scope (e.g., in an inner class) are not picked up by *get_methods()*.

For example:

```
>>> import symtable
>>> st = symtable.symtable('''
... def outer(): pass
...
... class A:
...     def f():
...         def w(): pass
...
...     def g(self): pass
...
...     @classmethod
...     async def h(cls): pass
...
...     global outer
...     def outer(self): pass
... ''', 'test', 'exec')
>>> class_A = st.get_children()[1]
>>> class_A.get_methods()
('f', 'g', 'h')
```

Although `A().f()` raises *TypeError* at runtime, `A.f` is still considered as a method-like function.

class symtable.**Symbol**

An entry in a *SymbolTable* corresponding to an identifier in the source. The constructor is not public.

get_name()

Return the symbol's name.

is_referenced()

Return `True` if the symbol is used in its block.

is_imported()

Return `True` if the symbol is created from an import statement.

is_parameter()

Return `True` if the symbol is a parameter.

is_global()

Return `True` if the symbol is global.

is_nonlocal()

Return `True` if the symbol is nonlocal.

is_declared_global()

Return `True` if the symbol is declared global with a global statement.

is_local()

Return `True` if the symbol is local to its block.

is_annotated()

Return `True` if the symbol is annotated.

Added in version 3.6.

is_free()

Return `True` if the symbol is referenced in its block, but not assigned to.

is_assigned()

Return `True` if the symbol is assigned to in its block.

is_namespace()

Return `True` if name binding introduces new namespace.

If the name is used as the target of a function or class statement, this will be true.

For example:

```
>>> table = symtable.symtable("def some_func(): pass", "string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

Note that a single name can be bound to multiple objects. If the result is `True`, the name may also be bound to other objects, like an int or list, that does not introduce a new namespace.

get_namespaces()

Return a list of namespaces bound to this name.

get_namespace()

Return the namespace bound to this name. If more than one or no namespace is bound to this name, a *ValueError* is raised.

32.2.3 Command-Line Usage

Added in version 3.13.

The *symtable* module can be executed as a script from the command line.

```
python -m symtable [infile...]
```

Symbol tables are generated for the specified Python source files and dumped to stdout. If no input file is specified, the content is read from stdin.

32.3 token — Constants used with Python parse trees

Source code: Lib/token.py

This module provides constants which represent the numeric values of leaf nodes of the parse tree (terminal tokens). Refer to the file `Grammar/Tokens` in the Python distribution for the definitions of the names in the context of the language grammar. The specific numeric values which the names map to may change between Python versions.

The module also provides a mapping from numeric codes to names and some functions. The functions mirror definitions in the Python C header files.

`token.`**`tok_name`**

Dictionary mapping the numeric values of the constants defined in this module back to name strings, allowing more human-readable representation of parse trees to be generated.

`token.`**`ISTERMINAL`**(*x*)

Return `True` for terminal token values.

`token.`**`ISNONTERMINAL`**(*x*)

Return `True` for non-terminal token values.

`token.`**`ISEOF`**(*x*)

Return `True` if *x* is the marker indicating the end of input.

The token constants are:

`token.`**`ENDMARKER`**

`token.`**`NAME`**

`token.`**`NUMBER`**

`token.`**`STRING`**

`token.`**`NEWLINE`**

`token.`**`INDENT`**

`token.`**`DEDENT`**

`token.`**`LPAR`**

Token value for `"("`.

`token.`**`RPAR`**

Token value for `")"`.

token.**LSQB**
Token value for "[".

token.**RSQB**
Token value for "]".

token.**COLON**
Token value for ":".

token.**COMMA**
Token value for ",".

token.**SEMI**
Token value for ";".

token.**PLUS**
Token value for "+".

token.**MINUS**
Token value for "-".

token.**STAR**
Token value for "*".

token.**SLASH**
Token value for "/".

token.**VBAR**
Token value for "|".

token.**AMPER**
Token value for "&".

token.**LESS**
Token value for "<".

token.**GREATER**
Token value for ">".

token.**EQUAL**
Token value for "=".

token.**DOT**
Token value for ".".

token.**PERCENT**
Token value for "%".

token.**LBRACE**
Token value for "{".

token.**RBRACE**
Token value for "}".

token.**EQEQUAL**
Token value for "==".

token.**NOTEQUAL**
Token value for "!=".

token.**LESSEQUAL**
Token value for "<=".

token.**GREATEREQUAL**
Token value for ">=".

token.**TILDE**
Token value for "~".

token.**CIRCUMFLEX**
Token value for "^".

token.**LEFTSHIFT**
Token value for "<<".

token.**RIGHTSHIFT**
Token value for ">>".

token.**DOUBLESTAR**
Token value for "**".

token.**PLUSEQUAL**
Token value for "+=".

token.**MINEQUAL**
Token value for "-=".

token.**STAREQUAL**
Token value for "*=".

token.**SLASHEQUAL**
Token value for "/=".

token.**PERCENTEQUAL**
Token value for "%=".

token.**AMPEREQUAL**
Token value for "&=".

token.**VBAREQUAL**
Token value for "|=".

token.**CIRCUMFLEXEQUAL**
Token value for "^=".

token.**LEFTSHIFTEQUAL**
Token value for "<<=".

token.**RIGHTSHIFTEQUAL**
Token value for ">>=".

token.**DOUBLESTAREQUAL**
Token value for "**=".

token.**DOUBLESLASH**
Token value for "//".

token.**DOUBLESLASHEQUAL**
Token value for "//=".

token.**AT**
: Token value for `"@"`.

token.**ATEQUAL**
: Token value for `"@="`.

token.**RARROW**
: Token value for `"->"`.

token.**ELLIPSIS**
: Token value for `"..."`.

token.**COLONEQUAL**
: Token value for `":="`.

token.**EXCLAMATION**
: Token value for `"!"`.

token.**OP**

token.**TYPE_IGNORE**

token.**TYPE_COMMENT**

token.**SOFT_KEYWORD**

token.**FSTRING_START**

token.**FSTRING_MIDDLE**

token.**FSTRING_END**

token.**COMMENT**

token.**NL**

token.**ERRORTOKEN**

token.**N_TOKENS**

token.**NT_OFFSET**

The following token type values aren't used by the C tokenizer but are needed for the *tokenize* module.

token.**COMMENT**
: Token value used to indicate a comment.

token.**NL**
: Token value used to indicate a non-terminating newline. The *NEWLINE* token indicates the end of a logical line of Python code; `NL` tokens are generated when a logical line of code is continued over multiple physical lines.

token.**ENCODING**
: Token value that indicates the encoding used to decode the source bytes into text. The first token returned by *tokenize.tokenize()* will always be an `ENCODING` token.

token.**TYPE_COMMENT**
: Token value indicating that a type comment was recognized. Such tokens are only produced when *ast.parse()* is invoked with `type_comments=True`.

Changed in version 3.5: Added AWAIT and ASYNC tokens.

Changed in version 3.7: Added *COMMENT*, *NL* and *ENCODING* tokens.

Changed in version 3.7: Removed AWAIT and ASYNC tokens. "async" and "await" are now tokenized as *NAME* tokens.

Changed in version 3.8: Added *TYPE_COMMENT*, *TYPE_IGNORE*, *COLONEQUAL*. Added AWAIT and ASYNC tokens back (they're needed to support parsing older Python versions for *ast.parse()* with feature_version set to 6 or lower).

Changed in version 3.13: Removed AWAIT and ASYNC tokens again.

32.4 keyword — Testing for Python keywords

Source code: Lib/keyword.py

This module allows a Python program to determine if a string is a keyword or soft keyword.

keyword.**iskeyword**(*s*)

Return True if *s* is a Python keyword.

keyword.**kwlist**

Sequence containing all the keywords defined for the interpreter. If any keywords are defined to only be active when particular *__future__* statements are in effect, these will be included as well.

keyword.**issoftkeyword**(*s*)

Return True if *s* is a Python soft keyword.

Added in version 3.9.

keyword.**softkwlist**

Sequence containing all the soft keywords defined for the interpreter. If any soft keywords are defined to only be active when particular *__future__* statements are in effect, these will be included as well.

Added in version 3.9.

32.5 tokenize — Tokenizer for Python source

Source code: Lib/tokenize.py

The *tokenize* module provides a lexical scanner for Python source code, implemented in Python. The scanner in this module returns comments as tokens as well, making it useful for implementing "pretty-printers", including colorizers for on-screen displays.

To simplify token stream handling, all operator and delimiter tokens and *Ellipsis* are returned using the generic *OP* token type. The exact type can be determined by checking the exact_type property on the *named tuple* returned from *tokenize.tokenize()*.

Warning

Note that the functions in this module are only designed to parse syntactically valid Python code (code that does not raise when parsed using *ast.parse()*). The behavior of the functions in this module is **undefined** when providing invalid Python code and it can change at any point.

32.5.1 Tokenizing Input

The primary entry point is a *generator*:

tokenize.**tokenize**(*readline*)

The *tokenize()* generator requires one argument, *readline*, which must be a callable object which provides the same interface as the *io.IOBase.readline()* method of file objects. Each call to the function should return one line of input as bytes.

The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple `(srow, scol)` of ints specifying the row and column where the token begins in the source; a 2-tuple `(erow, ecol)` of ints specifying the row and column where the token ends in the source; and the line on which the token was found. The line passed (the last tuple item) is the *physical* line. The 5 tuple is returned as a *named tuple* with the field names: `type string start end line`.

The returned *named tuple* has an additional property named `exact_type` that contains the exact operator type for *OP* tokens. For all other token types `exact_type` equals the named tuple `type` field.

Changed in version 3.1: Added support for named tuples.

Changed in version 3.3: Added support for `exact_type`.

tokenize() determines the source encoding of the file by looking for a UTF-8 BOM or encoding cookie, according to **PEP 263**.

tokenize.**generate_tokens**(*readline*)

Tokenize a source reading unicode strings instead of bytes.

Like *tokenize()*, the *readline* argument is a callable returning a single line of input. However, *generate_tokens()* expects *readline* to return a str object rather than bytes.

The result is an iterator yielding named tuples, exactly like *tokenize()*. It does not yield an *ENCODING* token.

All constants from the *token* module are also exported from *tokenize*.

Another function is provided to reverse the tokenization process. This is useful for creating tools that tokenize a script, modify the token stream, and write back the modified script.

tokenize.**untokenize**(*iterable*)

Converts tokens back into Python source code. The *iterable* must return sequences with at least two elements, the token type and the token string. Any additional sequence elements are ignored.

The reconstructed script is returned as a single string. The result is guaranteed to tokenize back to match the input so that the conversion is lossless and round-trips are assured. The guarantee applies only to the token type and token string as the spacing between tokens (column positions) may change.

It returns bytes, encoded using the *ENCODING* token, which is the first token sequence output by *tokenize()*. If there is no encoding token in the input, it returns a str instead.

tokenize() needs to detect the encoding of source files it tokenizes. The function it uses to do this is available:

tokenize.**detect_encoding**(*readline*)

The *detect_encoding()* function is used to detect the encoding that should be used to decode a Python source file. It requires one argument, readline, in the same way as the *tokenize()* generator.

It will call readline a maximum of twice, and return the encoding used (as a string) and a list of any lines (not decoded from bytes) it has read in.

It detects the encoding from the presence of a UTF-8 BOM or an encoding cookie as specified in **PEP 263**. If both a BOM and a cookie are present, but disagree, a *SyntaxError* will be raised. Note that if the BOM is found, `'utf-8-sig'` will be returned as an encoding.

If no encoding is specified, then the default of `'utf-8'` will be returned.

Use *open()* to open Python source files: it uses *detect_encoding()* to detect the file encoding.

`tokenize.`**open**(*filename*)

Open a file in read only mode using the encoding detected by *detect_encoding()*.

Added in version 3.2.

exception `tokenize.`**TokenError**

Raised when either a docstring or expression that may be split over several lines is not completed anywhere in the file, for example:

```
"""Beginning of
docstring
```

or:

```
[1,
 2,
 3
```

32.5.2 Command-Line Usage

Added in version 3.3.

The *tokenize* module can be executed as a script from the command line. It is as simple as:

```
python -m tokenize [-e] [filename.py]
```

The following options are accepted:

-h, --help

show this help message and exit

-e, --exact

display token names using the exact type

If `filename.py` is specified its contents are tokenized to stdout. Otherwise, tokenization is performed on stdin.

32.5.3 Examples

Example of a script rewriter that transforms float literals into Decimal objects:

```
from tokenize import tokenize, untokenize, NUMBER, STRING, NAME, OP
from io import BytesIO

def decistmt(s):
    """Substitute Decimals for floats in a string of statements.

    >>> from decimal import Decimal
    >>> s = 'print(+21.3e-5*-.1234/81.7)'
    >>> decistmt(s)
    "print (+Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7'))"

    The format of the exponent is inherited from the platform C library.
```

(continues on next page)

(continued from previous page)

```
    Known cases are "e-007" (Windows) and "e-07" (not Windows).  Since
    we're only showing 12 digits, and the 13th isn't close to 5, the
    rest of the output should be platform-independent.

    >>> exec(s)  #doctest: +ELLIPSIS
    -3.21716034272e-0...7

    Output from calculations with Decimal should be identical across all
    platforms.

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
    """
    result = []
    g = tokenize(BytesIO(s.encode('utf-8')).readline)  # tokenize the string
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval:  # replace NUMBER tokens
            result.extend([
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
        else:
            result.append((toknum, tokval))
    return untokenize(result).decode('utf-8')
```

Example of tokenizing from the command line. The script:

```
def say_hello():
    print("Hello, World!")

say_hello()
```

will be tokenized to the following output where the first column is the range of the line/column coordinates where the token is found, the second column is the name of the token, and the final column is the value of the token (if any)

```
$ python -m tokenize hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          OP             '('
1,14-1,15:          OP             ')'
1,15-1,16:          OP             ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           OP             '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          OP             ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           OP             '('
4,10-4,11:          OP             ')'
4,11-4,12:          NEWLINE        '\n'
```

(continues on next page)

(continued from previous page)

```
5,0-5,0:            ENDMARKER      ''
```

The exact token type names can be displayed using the `-e` option:

```
$ python -m tokenize -e hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          LPAR           '('
1,14-1,15:          RPAR           ')'
1,15-1,16:          COLON          ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           LPAR           '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          RPAR           ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           LPAR           '('
4,10-4,11:          RPAR           ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

Example of tokenizing a file programmatically, reading unicode strings instead of bytes with *generate_tokens()*:

```
import tokenize

with tokenize.open('hello.py') as f:
    tokens = tokenize.generate_tokens(f.readline)
    for token in tokens:
        print(token)
```

Or reading bytes directly with *tokenize()*:

```
import tokenize

with open('hello.py', 'rb') as f:
    tokens = tokenize.tokenize(f.readline)
    for token in tokens:
        print(token)
```

32.6 tabnanny — Detection of ambiguous indentation

Source code: Lib/tabnanny.py

For the time being this module is intended to be called as a script. However it is possible to import it into an IDE and use the function *check()* described below.

Note

The API provided by this module is likely to change in future releases; such changes may not be backward compatible.

`tabnanny.`**`check`**(*file_or_dir*)

If *file_or_dir* is a directory and not a symbolic link, then recursively descend the directory tree named by *file_or_dir*, checking all `.py` files along the way. If *file_or_dir* is an ordinary Python source file, it is checked for whitespace related problems. The diagnostic messages are written to standard output using the *print()* function.

`tabnanny.`**`verbose`**

Flag indicating whether to print verbose messages. This is incremented by the `-v` option if called as a script.

`tabnanny.`**`filename_only`**

Flag indicating whether to print only the filenames of files containing whitespace related problems. This is set to true by the `-q` option if called as a script.

exception `tabnanny.`**`NannyNag`**

Raised by *process_tokens()* if detecting an ambiguous indent. Captured and handled in *check()*.

`tabnanny.`**`process_tokens`**(*tokens*)

This function is used by *check()* to process tokens generated by the *tokenize* module.

See also

Module *tokenize*
Lexical scanner for Python source code.

32.7 `pyclbr` — Python module browser support

Source code: Lib/pyclbr.py

The *pyclbr* module provides limited information about the functions, classes, and methods defined in a Python-coded module. The information is sufficient to implement a module browser. The information is extracted from the Python source code rather than by importing the module, so this module is safe to use with untrusted code. This restriction makes it impossible to use this module with modules not implemented in Python, including all standard and optional extension modules.

`pyclbr.`**`readmodule`**(*module*, *path=None*)

Return a dictionary mapping module-level class names to class descriptors. If possible, descriptors for imported base classes are included. Parameter *module* is a string with the name of the module to read; it may be the name of a module within a package. If given, *path* is a sequence of directory paths prepended to `sys.path`, which is used to locate the module source code.

This function is the original interface and is only kept for back compatibility. It returns a filtered version of the following.

`pyclbr.`**`readmodule_ex`**(*module*, *path=None*)

Return a dictionary-based tree containing a function or class descriptors for each function and class defined in the module with a `def` or `class` statement. The returned dictionary maps module-level function and class names to their descriptors. Nested objects are entered into the children dictionary of their parent. As with readmodule,

module names the module to be read and *path* is prepended to sys.path. If the module being read is a package, the returned dictionary has a key `'__path__'` whose value is a list containing the package search path.

Added in version 3.7: Descriptors for nested definitions. They are accessed through the new children attribute. Each has a new parent attribute.

The descriptors returned by these functions are instances of Function and Class classes. Users are not expected to create instances of these classes.

32.7.1 Function Objects

class pyclbr.**Function**

Class `Function` instances describe functions defined by def statements. They have the following attributes:

file

Name of the file in which the function is defined.

module

The name of the module defining the function described.

name

The name of the function.

lineno

The line number in the file where the definition starts.

parent

For top-level functions, `None`. For nested functions, the parent.

Added in version 3.7.

children

A *dictionary* mapping names to descriptors for nested functions and classes.

Added in version 3.7.

is_async

`True` for functions that are defined with the `async` prefix, `False` otherwise.

Added in version 3.10.

32.7.2 Class Objects

class pyclbr.**Class**

Class `Class` instances describe classes defined by class statements. They have the same attributes as *Functions* and two more.

file

Name of the file in which the class is defined.

module

The name of the module defining the class described.

name

The name of the class.

lineno

The line number in the file where the definition starts.

parent

For top-level classes, `None`. For nested classes, the parent.

Added in version 3.7.

children

A dictionary mapping names to descriptors for nested functions and classes.

Added in version 3.7.

super

A list of `Class` objects which describe the immediate base classes of the class being described. Classes which are named as superclasses but which are not discoverable by *readmodule_ex()* are listed as a string with the class name instead of as `Class` objects.

methods

A *dictionary* mapping method names to line numbers. This can be derived from the newer *children* dictionary, but remains for back-compatibility.

32.8 py_compile — Compile Python source files

Source code: Lib/py_compile.py

The *py_compile* module provides a function to generate a byte-code file from a source file, and another function used when the module source file is invoked as a script.

Though not often needed, this function can be useful when installing modules for shared use, especially if some of the users may not have permission to write the byte-code cache files in the directory containing the source code.

exception py_compile.**PyCompileError**

Exception raised when an error occurs while attempting to compile the file.

py_compile.**compile**(*file*, *cfile=None*, *dfile=None*, *doraise=False*, *optimize=-1*, *invalidation_mode=PycInvalidationMode.TIMESTAMP*, *quiet=0*)

Compile a source file to byte-code and write out the byte-code cache file. The source code is loaded from the file named *file*. The byte-code is written to *cfile*, which defaults to the **PEP 3147/PEP 488** path, ending in `.pyc`. For example, if *file* is `/foo/bar/baz.py` *cfile* will default to `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. If *dfile* is specified, it is used instead of *file* as the name of the source file from which source lines are obtained for display in exception tracebacks. If *doraise* is true, a *PyCompileError* is raised when an error is encountered while compiling *file*. If *doraise* is false (the default), an error string is written to `sys.stderr`, but no exception is raised. This function returns the path to byte-compiled file, i.e. whatever *cfile* value was used.

The *doraise* and *quiet* arguments determine how errors are handled while compiling file. If *quiet* is 0 or 1, and *doraise* is false, the default behaviour is enabled: an error string is written to `sys.stderr`, and the function returns `None` instead of a path. If *doraise* is true, a *PyCompileError* is raised instead. However if *quiet* is 2, no message is written, and *doraise* has no effect.

If the path that *cfile* becomes (either explicitly specified or computed) is a symlink or non-regular file, *FileExistsError* will be raised. This is to act as a warning that import will turn those paths into regular files if it is allowed to write byte-compiled files to those paths. This is a side-effect of import using file renaming to place the final byte-compiled file into place to prevent concurrent file writing issues.

optimize controls the optimization level and is passed to the built-in *compile()* function. The default of -1 selects the optimization level of the current interpreter.

invalidation_mode should be a member of the *PycInvalidationMode* enum and controls how the generated bytecode cache is invalidated at runtime. The default is *PycInvalidationMode.CHECKED_HASH* if the SOURCE_DATE_EPOCH environment variable is set, otherwise the default is *PycInvalidationMode.TIMESTAMP*.

Changed in version 3.2: Changed default value of *cfile* to be **PEP 3147**-compliant. Previous default was *file* + 'c' ('o' if optimization was enabled). Also added the *optimize* parameter.

Changed in version 3.4: Changed code to use *importlib* for the byte-code cache file writing. This means file creation/writing semantics now match what *importlib* does, e.g. permissions, write-and-move semantics, etc. Also added the caveat that *FileExistsError* is raised if *cfile* is a symlink or non-regular file.

Changed in version 3.7: The *invalidation_mode* parameter was added as specified in **PEP 552**. If the SOURCE_DATE_EPOCH environment variable is set, *invalidation_mode* will be forced to *PycInvalidationMode.CHECKED_HASH*.

Changed in version 3.7.2: The SOURCE_DATE_EPOCH environment variable no longer overrides the value of the *invalidation_mode* argument, and determines its default value instead.

Changed in version 3.8: The *quiet* parameter was added.

class py_compile.**PycInvalidationMode**

An enumeration of possible methods the interpreter can use to determine whether a bytecode file is up to date with a source file. The .pyc file indicates the desired invalidation mode in its header. See pyc-invalidation for more information on how Python invalidates .pyc files at runtime.

Added in version 3.7.

TIMESTAMP

The .pyc file includes the timestamp and size of the source file, which Python will compare against the metadata of the source file at runtime to determine if the .pyc file needs to be regenerated.

CHECKED_HASH

The .pyc file includes a hash of the source file content, which Python will compare against the source at runtime to determine if the .pyc file needs to be regenerated.

UNCHECKED_HASH

Like *CHECKED_HASH*, the .pyc file includes a hash of the source file content. However, Python will at runtime assume the .pyc file is up to date and not validate the .pyc against the source file at all.

This option is useful when the .pycs are kept up to date by some system external to Python like a build system.

32.8.1 Command-Line Interface

This module can be invoked as a script to compile several source files. The files named in *filenames* are compiled and the resulting bytecode is cached in the normal manner. This program does not search a directory structure to locate source files; it only compiles files named explicitly. The exit status is nonzero if one of the files could not be compiled.

<file> ... <fileN>

-

Positional arguments are files to compile. If − is the only parameter, the list of files is taken from standard input.

-q, --quiet

Suppress errors output.

Changed in version 3.2: Added support for -.

Changed in version 3.10: Added support for -q.

See also

Module compileall
Utilities to compile all Python source files in a directory tree.

32.9 compileall — Byte-compile Python libraries

Source code: Lib/compileall.py

This module provides some utility functions to support installing Python libraries. These functions compile Python source files in a directory tree. This module can be used to create the cached byte-code files at library installation time, which makes them available for use even by users who don't have write permission to the library directories.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly platforms* for more information.

32.9.1 Command-line use

This module can work as a script (using **python -m compileall**) to compile Python sources.

directory ...

file ...

Positional arguments are files to compile or directories that contain source files, traversed recursively. If no argument is given, behave as if the command line was -l <*directories from sys.path*>.

-l

Do not recurse into subdirectories, only compile source code files directly contained in the named or implied directories.

-f

Force rebuild even if timestamps are up-to-date.

-q

Do not print the list of files compiled. If passed once, error messages will still be printed. If passed twice (-qq), all output is suppressed.

-d destdir

Directory prepended to the path to each file being compiled. This will appear in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

-s strip_prefix

-p prepend_prefix

Remove (-s) or append (-p) the given prefix of paths recorded in the .pyc files. Cannot be combined with -d.

-x regex

regex is used to search the full path to each file considered for compilation, and if the regex produces a match, the file is skipped.

-i list

Read the file list and add each line that it contains to the list of files and directories to compile. If list is -, read lines from stdin.

-b

Write the byte-code files to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** locations and names, which allows byte-code files from multiple versions of Python to coexist.

-r

Control the maximum recursion level for subdirectories. If this is given, then -l option will not be taken into account. **python -m compileall <directory> -r 0** is equivalent to **python -m compileall <directory> -l**.

-j N

Use *N* workers to compile the files within the given directory. If 0 is used, then the result of *os.process_cpu_count()* will be used.

--invalidation-mode [timestamp|checked-hash|unchecked-hash]

Control how the generated byte-code files are invalidated at runtime. The timestamp value, means that .pyc files with the source timestamp and size embedded will be generated. The checked-hash and unchecked-hash values cause hash-based pycs to be generated. Hash-based pycs embed a hash of the source file contents rather than a timestamp. See pyc-invalidation for more information on how Python validates bytecode cache files at runtime. The default is timestamp if the SOURCE_DATE_EPOCH environment variable is not set, and checked-hash if the SOURCE_DATE_EPOCH environment variable is set.

-o level

Compile with the given optimization level. May be used multiple times to compile for multiple levels at a time (for example, compileall -o 1 -o 2).

-e dir

Ignore symlinks pointing outside the given directory.

--hardlink-dupes

If two .pyc files with different optimization level have the same content, use hard links to consolidate duplicate files.

Changed in version 3.2: Added the -i, -b and -h options.

Changed in version 3.5: Added the -j, -r, and -qq options. -q option was changed to a multilevel value. -b will always produce a byte-code file ending in .pyc, never .pyo.

Changed in version 3.7: Added the --invalidation-mode option.

Changed in version 3.9: Added the -s, -p, -e and --hardlink-dupes options. Raised the default recursion limit from 10 to *sys.getrecursionlimit()*. Added the possibility to specify the -o option multiple times.

There is no command-line option to control the optimization level used by the *compile()* function, because the Python interpreter itself already provides the option: **python -O -m compileall**.

Similarly, the *compile()* function respects the *sys.pycache_prefix* setting. The generated bytecode cache will only be useful if *compile()* is run with the same *sys.pycache_prefix* (if any) that will be used at runtime.

32.9.2 Public functions

compileall.**compile_dir**(*dir*, *maxlevels=sys.getrecursionlimit()*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *workers=1*, *invalidation_mode=None*, ***, *stripdir=None*, *prependdir=None*, *limit_sl_dest=None*, *hardlink_dupes=False*)

Recursively descend the directory tree named by *dir*, compiling all `.py` files along the way. Return a true value if all the files compiled successfully, and a false value otherwise.

The *maxlevels* parameter is used to limit the depth of the recursion; it defaults to `sys.getrecursionlimit()`.

If *ddir* is given, it is prepended to the path to each file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If *force* is true, modules are re-compiled even if the timestamps are up to date.

If *rx* is given, its `search` method is called on the complete path to each file considered for compilation, and if it returns a true value, the file is skipped. This can be used to exclude files matching a regular expression, given as a *re.Pattern* object.

If *quiet* is `False` or `0` (the default), the filenames and other information are printed to standard out. Set to `1`, only errors are printed. Set to `2`, all output is suppressed.

If *legacy* is true, byte-code files are written to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** locations and names, which allows byte-code files from multiple versions of Python to coexist.

optimize specifies the optimization level for the compiler. It is passed to the built-in *compile()* function. Accepts also a sequence of optimization levels which lead to multiple compilations of one `.py` file in one call.

The argument *workers* specifies how many workers are used to compile files in parallel. The default is to not use multiple workers. If the platform can't use multiple workers and *workers* argument is given, then sequential compilation will be used as a fallback. If *workers* is 0, the number of cores in the system is used. If *workers* is lower than `0`, a *ValueError* will be raised.

invalidation_mode should be a member of the *py_compile.PycInvalidationMode* enum and controls how the generated pycs are invalidated at runtime.

The *stripdir*, *prependdir* and *limit_sl_dest* arguments correspond to the `-s`, `-p` and `-e` options described above. They may be specified as `str` or *os.PathLike*.

If *hardlink_dupes* is true and two `.pyc` files with different optimization level have the same content, use hard links to consolidate duplicate files.

Changed in version 3.2: Added the *legacy* and *optimize* parameter.

Changed in version 3.5: Added the *workers* parameter.

Changed in version 3.5: *quiet* parameter was changed to a multilevel value.

Changed in version 3.5: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

Changed in version 3.6: Accepts a *path-like object*.

Changed in version 3.7: The *invalidation_mode* parameter was added.

Changed in version 3.7.2: The *invalidation_mode* parameter's default value is updated to `None`.

Changed in version 3.8: Setting *workers* to 0 now chooses the optimal number of cores.

Changed in version 3.9: Added *stripdir*, *prependdir*, *limit_sl_dest* and *hardlink_dupes* arguments. Default value of *maxlevels* was changed from `10` to `sys.getrecursionlimit()`

compileall.**compile_file**(*fullname*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=None*, ***, *stripdir=None*, *prependdir=None*, *limit_sl_dest=None*, *hardlink_dupes=False*)

Compile the file with path *fullname*. Return a true value if the file compiled successfully, and a false value otherwise.

If *ddir* is given, it is prepended to the path to the file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If *rx* is given, its `search` method is passed the full path name to the file being compiled, and if it returns a true value, the file is not compiled and `True` is returned. This can be used to exclude files matching a regular expression, given as a *re.Pattern* object.

If *quiet* is `False` or `0` (the default), the filenames and other information are printed to standard out. Set to `1`, only errors are printed. Set to `2`, all output is suppressed.

If *legacy* is true, byte-code files are written to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** locations and names, which allows byte-code files from multiple versions of Python to coexist.

optimize specifies the optimization level for the compiler. It is passed to the built-in *compile()* function. Accepts also a sequence of optimization levels which lead to multiple compilations of one `.py` file in one call.

invalidation_mode should be a member of the *py_compile.PycInvalidationMode* enum and controls how the generated pycs are invalidated at runtime.

The *stripdir*, *prependdir* and *limit_sl_dest* arguments correspond to the `-s`, `-p` and `-e` options described above. They may be specified as `str` or *os.PathLike*.

If *hardlink_dupes* is true and two `.pyc` files with different optimization level have the same content, use hard links to consolidate duplicate files.

Added in version 3.2.

Changed in version 3.5: *quiet* parameter was changed to a multilevel value.

Changed in version 3.5: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

Changed in version 3.7: The *invalidation_mode* parameter was added.

Changed in version 3.7.2: The *invalidation_mode* parameter's default value is updated to `None`.

Changed in version 3.9: Added *stripdir*, *prependdir*, *limit_sl_dest* and *hardlink_dupes* arguments.

compileall.**compile_path**(*skip_curdir=True*, *maxlevels=0*, *force=False*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=None*)

Byte-compile all the `.py` files found along `sys.path`. Return a true value if all the files compiled successfully, and a false value otherwise.

If *skip_curdir* is true (the default), the current directory is not included in the search. All other parameters are passed to the *compile_dir()* function. Note that unlike the other compile functions, `maxlevels` defaults to `0`.

Changed in version 3.2: Added the *legacy* and *optimize* parameter.

Changed in version 3.5: *quiet* parameter was changed to a multilevel value.

Changed in version 3.5: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

Changed in version 3.7: The *invalidation_mode* parameter was added.

Changed in version 3.7.2: The *invalidation_mode* parameter's default value is updated to `None`.

To force a recompile of all the `.py` files in the `Lib/` subdirectory and all its subdirectories:

```
import compileall

compileall.compile_dir('Lib/', force=True)

# Perform same compilation, excluding files in .svn directories.
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\\][.]svn'), force=True)

# pathlib.Path objects can also be used.
import pathlib
compileall.compile_dir(pathlib.Path('Lib/'), force=True)
```

See also

Module ***py_compile***
Byte-compile a single source file.

32.10 `dis` — Disassembler for Python bytecode

Source code: Lib/dis.py

The *dis* module supports the analysis of CPython *bytecode* by disassembling it. The CPython bytecode which this module takes as an input is defined in the file `Include/opcode.h` and used by the compiler and the interpreter.

CPython implementation detail: Bytecode is an implementation detail of the CPython interpreter. No guarantees are made that bytecode will not be added, removed, or changed between versions of Python. Use of this module should not be considered to work across Python VMs or Python releases.

Changed in version 3.6: Use 2 bytes for each instruction. Previously the number of bytes varied by instruction.

Changed in version 3.10: The argument of jump, exception handling and loop instructions is now the instruction offset rather than the byte offset.

Changed in version 3.11: Some instructions are accompanied by one or more inline cache entries, which take the form of *CACHE* instructions. These instructions are hidden by default, but can be shown by passing `show_caches=True` to any *dis* utility. Furthermore, the interpreter now adapts the bytecode to specialize it for different runtime conditions. The adaptive bytecode can be shown by passing `adaptive=True`.

Changed in version 3.12: The argument of a jump is the offset of the target instruction relative to the instruction that appears immediately after the jump instruction's *CACHE* entries.

As a consequence, the presence of the *CACHE* instructions is transparent for forward jumps but needs to be taken into account when reasoning about backward jumps.

Changed in version 3.13: The output shows logical labels rather than instruction offsets for jump targets and exception handlers. The `-O` command line option and the `show_offsets` argument were added.

Example: Given the function `myfunc()`:

```
def myfunc(alist):
    return len(alist)
```

the following command can be used to display the disassembly of `myfunc()`:

```
>>> dis.dis(myfunc)
  2           RESUME                   0

  3           LOAD_GLOBAL              1 (len + NULL)
              LOAD_FAST                0 (alist)
              CALL                     1
              RETURN_VALUE
```

(The "2" is a line number).

32.10.1 Command-line interface

The *dis* module can be invoked as a script from the command line:

```
python -m dis [-h] [-C] [-O] [infile]
```

The following options are accepted:

-h, --help

Display usage and exit.

-C, --show-caches

Show inline caches.

-O, --show-offsets

Show offsets of instructions.

If `infile` is specified, its disassembled code will be written to stdout. Otherwise, disassembly is performed on compiled source code received from stdin.

32.10.2 Bytecode analysis

Added in version 3.4.

The bytecode analysis API allows pieces of Python code to be wrapped in a *Bytecode* object that provides easy access to details of the compiled code.

class dis.**Bytecode**(*x*, ***, *first_line=None*, *current_offset=None*, *show_caches=False*, *adaptive=False*, *show_offsets=False*)

Analyse the bytecode corresponding to a function, generator, asynchronous generator, coroutine, method, string of source code, or a code object (as returned by *compile()*).

This is a convenience wrapper around many of the functions listed below, most notably *get_instructions()*, as iterating over a *Bytecode* instance yields the bytecode operations as *Instruction* instances.

If *first_line* is not `None`, it indicates the line number that should be reported for the first source line in the disassembled code. Otherwise, the source line information (if any) is taken directly from the disassembled code object.

If *current_offset* is not `None`, it refers to an instruction offset in the disassembled code. Setting this means *dis()* will display a "current instruction" marker against the specified opcode.

If *show_caches* is `True`, *dis()* will display inline cache entries used by the interpreter to specialize the bytecode.

If *adaptive* is `True`, *dis()* will display specialized bytecode that may be different from the original bytecode.

If *show_offsets* is `True`, *dis()* will include instruction offsets in the output.

classmethod from_traceback(*tb*, *, *show_caches=False*)

Construct a *Bytecode* instance from the given traceback, setting *current_offset* to the instruction responsible for the exception.

codeobj

The compiled code object.

first_line

The first source line of the code object (if available)

dis()

Return a formatted view of the bytecode operations (the same as printed by *dis.dis()*, but returned as a multi-line string).

info()

Return a formatted multi-line string with detailed information about the code object, like *code_info()*.

Changed in version 3.7: This can now handle coroutine and asynchronous generator objects.

Changed in version 3.11: Added the *show_caches* and *adaptive* parameters.

Example:

```
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...     print(instr.opname)
...
RESUME
LOAD_GLOBAL
LOAD_FAST
CALL
RETURN_VALUE
```

32.10.3 Analysis functions

The *dis* module also defines the following analysis functions that convert the input directly to the desired output. They can be useful if only a single operation is being performed, so the intermediate analysis object isn't useful:

dis.**code_info**(*x*)

Return a formatted multi-line string with detailed code object information for the supplied function, generator, asynchronous generator, coroutine, method, source code string or code object.

Note that the exact contents of code info strings are highly implementation dependent and they may change arbitrarily across Python VMs or Python releases.

Added in version 3.2.

Changed in version 3.7: This can now handle coroutine and asynchronous generator objects.

dis.**show_code**(*x*, *, *file=None*)

Print detailed code object information for the supplied function, method, source code string or code object to *file* (or `sys.stdout` if *file* is not specified).

This is a convenient shorthand for `print(code_info(x), file=file)`, intended for interactive exploration at the interpreter prompt.

Added in version 3.2.

Changed in version 3.4: Added *file* parameter.

dis.**dis**(*x=None*, *, *file=None*, *depth=None*, *show_caches=False*, *adaptive=False*)

Disassemble the *x* object. *x* can denote either a module, a class, a method, a function, a generator, an asynchronous generator, a coroutine, a code object, a string of source code or a byte sequence of raw bytecode. For a module, it disassembles all functions. For a class, it disassembles all methods (including class and static methods). For a code object or sequence of raw bytecode, it prints one line per bytecode instruction. It also recursively disassembles nested code objects. These can include generator expressions, nested functions, the bodies of nested classes, and the code objects used for annotation scopes. Strings are first compiled to code objects with the *compile()* built-in function before being disassembled. If no object is provided, this function disassembles the last traceback.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

The maximal depth of recursion is limited by *depth* unless it is `None`. `depth=0` means no recursion.

If *show_caches* is `True`, this function will display inline cache entries used by the interpreter to specialize the bytecode.

If *adaptive* is `True`, this function will display specialized bytecode that may be different from the original bytecode.

Changed in version 3.4: Added *file* parameter.

Changed in version 3.7: Implemented recursive disassembling and added *depth* parameter.

Changed in version 3.7: This can now handle coroutine and asynchronous generator objects.

Changed in version 3.11: Added the *show_caches* and *adaptive* parameters.

distb(tb=None, *, file=None, show_caches=False, adaptive=False, show_offset=False)

Disassemble the top-of-stack function of a traceback, using the last traceback if none was passed. The instruction causing the exception is indicated.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

Changed in version 3.4: Added *file* parameter.

Changed in version 3.11: Added the *show_caches* and *adaptive* parameters.

Changed in version 3.13: Added the *show_offsets* parameter.

dis.**disassemble**(*code*, *lasti=-1*, *, *file=None*, *show_caches=False*, *adaptive=False*)

disco(code, lasti=-1, *, file=None, show_caches=False, adaptive=False, show_offsets=False)

Disassemble a code object, indicating the last instruction if *lasti* was provided. The output is divided in the following columns:

1. the line number, for the first instruction of each line
2. the current instruction, indicated as `-->`,
3. a labelled instruction, indicated with `>>`,
4. the address of the instruction,
5. the operation code name,
6. operation parameters, and
7. interpretation of the parameters in parentheses.

The parameter interpretation recognizes local and global variable names, constant values, branch targets, and compare operators.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

Changed in version 3.4: Added *file* parameter.

Changed in version 3.11: Added the *show_caches* and *adaptive* parameters.

Changed in version 3.13: Added the *show_offsets* parameter.

`dis.`**`get_instructions`**(*x*, ***, *first_line=None*, *show_caches=False*, *adaptive=False*)

Return an iterator over the instructions in the supplied function, method, source code string or code object.

The iterator generates a series of *`Instruction`* named tuples giving the details of each operation in the supplied code.

If *first_line* is not `None`, it indicates the line number that should be reported for the first source line in the disassembled code. Otherwise, the source line information (if any) is taken directly from the disassembled code object.

The *adaptive* parameter works as it does in *`dis()`*.

Added in version 3.4.

Changed in version 3.11: Added the *show_caches* and *adaptive* parameters.

Changed in version 3.13: The *show_caches* parameter is deprecated and has no effect. The iterator generates the *`Instruction`* instances with the *cache_info* field populated (regardless of the value of *show_caches*) and it no longer generates separate items for the cache entries.

`dis.`**`findlinestarts`**(*code*)

This generator function uses the `co_lines()` method of the code object *code* to find the offsets which are starts of lines in the source code. They are generated as `(offset, lineno)` pairs.

Changed in version 3.6: Line numbers can be decreasing. Before, they were always increasing.

Changed in version 3.10: The **PEP 626** `co_lines()` method is used instead of the `co_firstlineno` and `co_lnotab` attributes of the code object.

Changed in version 3.13: Line numbers can be `None` for bytecode that does not map to source lines.

`dis.`**`findlabels`**(*code*)

Detect all offsets in the raw compiled bytecode string *code* which are jump targets, and return a list of these offsets.

`dis.`**`stack_effect`**(*opcode*, *oparg=None*, ***, *jump=None*)

Compute the stack effect of *opcode* with argument *oparg*.

If the code has a jump target and *jump* is `True`, *`stack_effect()`* will return the stack effect of jumping. If *jump* is `False`, it will return the stack effect of not jumping. And if *jump* is `None` (default), it will return the maximal stack effect of both cases.

Added in version 3.4.

Changed in version 3.8: Added *jump* parameter.

Changed in version 3.13: If `oparg` is omitted (or `None`), the stack effect is now returned for `oparg=0`. Previously this was an error for opcodes that use their arg. It is also no longer an error to pass an integer `oparg` when the `opcode` does not use it; the `oparg` in this case is ignored.

32.10.4 Python Bytecode Instructions

The *get_instructions()* function and *Bytecode* class provide details of bytecode instructions as *Instruction* instances:

class dis.**Instruction**

Details for a bytecode operation

opcode

numeric code for operation, corresponding to the opcode values listed below and the bytecode values in the *Opcode collections*.

opname

human readable name for operation

baseopcode

numeric code for the base operation if operation is specialized; otherwise equal to *opcode*

baseopname

human readable name for the base operation if operation is specialized; otherwise equal to *opname*

arg

numeric argument to operation (if any), otherwise `None`

oparg

alias for *arg*

argval

resolved arg value (if any), otherwise `None`

argrepr

human readable description of operation argument (if any), otherwise an empty string.

offset

start index of operation within bytecode sequence

start_offset

start index of operation within bytecode sequence, including prefixed `EXTENDED_ARG` operations if present; otherwise equal to *offset*

cache_offset

start index of the cache entries following the operation

end_offset

end index of the cache entries following the operation

starts_line

`True` if this opcode starts a source line, otherwise `False`

line_number

source line number associated with this opcode (if any), otherwise `None`

is_jump_target

`True` if other code jumps to here, otherwise `False`

jump_target

bytecode index of the jump target if this is a jump operation, otherwise `None`

positions

dis.Positions object holding the start and end locations that are covered by this instruction.

Added in version 3.4.

Changed in version 3.11: Field `positions` is added.

Changed in version 3.13: Changed field `starts_line`.

Added fields `start_offset`, `cache_offset`, `end_offset`, `baseopname`, `baseopcode`, `jump_target`, `oparg`, `line_number` and `cache_info`.

class dis.**Positions**

In case the information is not available, some fields might be `None`.

lineno

end_lineno

col_offset

end_col_offset

Added in version 3.11.

The Python compiler currently generates the following bytecode instructions.

General instructions

In the following, We will refer to the interpreter stack as `STACK` and describe operations on it as if it was a Python list. The top of the stack corresponds to `STACK[-1]` in this language.

NOP

Do nothing code. Used as a placeholder by the bytecode optimizer, and to generate line tracing events.

POP_TOP

Removes the top-of-stack item:

```
STACK.pop()
```

END_FOR

Removes the top-of-stack item. Equivalent to `POP_TOP`. Used to clean up at the end of loops, hence the name.

Added in version 3.12.

END_SEND

Implements `del STACK[-2]`. Used to clean up when a generator exits.

Added in version 3.12.

COPY (*i*)

Push the i-th item to the top of the stack without removing it from its original location:

```
assert i > 0
STACK.append(STACK[-i])
```

Added in version 3.11.

SWAP (*i*)

Swap the top of the stack with the i-th element:

```
STACK[-i], STACK[-1] = STACK[-1], STACK[-i]
```

Added in version 3.11.

CACHE

Rather than being an actual instruction, this opcode is used to mark extra space for the interpreter to cache useful data directly in the bytecode itself. It is automatically hidden by all `dis` utilities, but can be viewed with `show_caches=True`.

Logically, this space is part of the preceding instruction. Many opcodes expect to be followed by an exact number of caches, and will instruct the interpreter to skip over them at runtime.

Populated caches can look like arbitrary instructions, so great care should be taken when reading or modifying raw, adaptive bytecode containing quickened data.

Added in version 3.11.

Unary operations

Unary operations take the top of the stack, apply the operation, and push the result back on the stack.

UNARY_NEGATIVE

Implements `STACK[-1] = -STACK[-1]`.

UNARY_NOT

Implements `STACK[-1] = not STACK[-1]`.

Changed in version 3.13: This instruction now requires an exact *bool* operand.

UNARY_INVERT

Implements `STACK[-1] = ~STACK[-1]`.

GET_ITER

Implements `STACK[-1] = iter(STACK[-1])`.

GET_YIELD_FROM_ITER

If `STACK[-1]` is a *generator iterator* or *coroutine* object it is left as is. Otherwise, implements `STACK[-1] = iter(STACK[-1])`.

Added in version 3.5.

TO_BOOL

Implements `STACK[-1] = bool(STACK[-1])`.

Added in version 3.13.

Binary and in-place operations

Binary operations remove the top two items from the stack (`STACK[-1]` and `STACK[-2]`). They perform the operation, then put the result back on the stack.

In-place operations are like binary operations, but the operation is done in-place when `STACK[-2]` supports it, and the resulting `STACK[-1]` may be (but does not have to be) the original `STACK[-2]`.

BINARY_OP (*op*)

Implements the binary and in-place operators (depending on the value of *op*):

```
rhs = STACK.pop()
lhs = STACK.pop()
STACK.append(lhs op rhs)
```

Added in version 3.11.

BINARY_SUBSCR

Implements:

```
key = STACK.pop()
container = STACK.pop()
STACK.append(container[key])
```

STORE_SUBSCR

Implements:

```
key = STACK.pop()
container = STACK.pop()
value = STACK.pop()
container[key] = value
```

DELETE_SUBSCR

Implements:

```
key = STACK.pop()
container = STACK.pop()
del container[key]
```

BINARY_SLICE

Implements:

```
end = STACK.pop()
start = STACK.pop()
container = STACK.pop()
STACK.append(container[start:end])
```

Added in version 3.12.

STORE_SLICE

Implements:

```
end = STACK.pop()
start = STACK.pop()
container = STACK.pop()
values = STACK.pop()
container[start:end] = value
```

Added in version 3.12.

Coroutine opcodes

GET_AWAITABLE (*where*)

Implements `STACK[-1] = get_awaitable(STACK[-1])`, where `get_awaitable(o)` returns `o` if `o` is a coroutine object or a generator object with the *CO_ITERABLE_COROUTINE* flag, or resolves `o.__await__`.

If the `where` operand is nonzero, it indicates where the instruction occurs:

- `1`: After a call to `__aenter__`
- `2`: After a call to `__aexit__`

Added in version 3.5.

Changed in version 3.11: Previously, this instruction did not have an oparg.

GET_AITER

Implements `STACK[-1] = STACK[-1].__aiter__()`.

Added in version 3.5.

Changed in version 3.7: Returning awaitable objects from `__aiter__` is no longer supported.

GET_ANEXT

Implement `STACK.append(get_awaitable(STACK[-1].__anext__()))` to the stack. See `GET_AWAITABLE` for details about `get_awaitable`.

Added in version 3.5.

END_ASYNC_FOR

Terminates an `async for` loop. Handles an exception raised when awaiting a next item. The stack contains the async iterable in `STACK[-2]` and the raised exception in `STACK[-1]`. Both are popped. If the exception is not *StopAsyncIteration*, it is re-raised.

Added in version 3.8.

Changed in version 3.11: Exception representation on the stack now consist of one, not three, items.

CLEANUP_THROW

Handles an exception raised during a `throw()` or `close()` call through the current frame. If `STACK[-1]` is an instance of *StopIteration*, pop three values from the stack and push its `value` member. Otherwise, re-raise `STACK[-1]`.

Added in version 3.12.

BEFORE_ASYNC_WITH

Resolves `__aenter__` and `__aexit__` from `STACK[-1]`. Pushes `__aexit__` and result of `__aenter__()` to the stack:

```
STACK.extend((__aexit__, __aenter__())
```

Added in version 3.5.

Miscellaneous opcodes

SET_ADD (*i*)

Implements:

```
item = STACK.pop()
set.add(STACK[-i], item)
```

Used to implement set comprehensions.

LIST_APPEND (*i*)

Implements:

```
item = STACK.pop()
list.append(STACK[-i], item)
```

Used to implement list comprehensions.

MAP_ADD (*i*)

Implements:

```
value = STACK.pop()
key = STACK.pop()
dict.__setitem__(STACK[-i], key, value)
```

Used to implement dict comprehensions.

Added in version 3.1.

Changed in version 3.8: Map value is `STACK[-1]` and map key is `STACK[-2]`. Before, those were reversed.

For all of the *SET_ADD*, *LIST_APPEND* and *MAP_ADD* instructions, while the added value or key/value pair is popped off, the container object remains on the stack so that it is available for further iterations of the loop.

RETURN_VALUE

Returns with `STACK[-1]` to the caller of the function.

RETURN_CONST (*consti*)

Returns with `co_consts[consti]` to the caller of the function.

Added in version 3.12.

YIELD_VALUE

Yields `STACK.pop()` from a *generator*.

Changed in version 3.11: oparg set to be the stack depth.

Changed in version 3.12: oparg set to be the exception block depth, for efficient closing of generators.

Changed in version 3.13: oparg is `1` if this instruction is part of a yield-from or await, and `0` otherwise.

SETUP_ANNOTATIONS

Checks whether `__annotations__` is defined in `locals()`, if not it is set up to an empty `dict`. This opcode is only emitted if a class or module body contains *variable annotations* statically.

Added in version 3.6.

POP_EXCEPT

Pops a value from the stack, which is used to restore the exception state.

Changed in version 3.11: Exception representation on the stack now consist of one, not three, items.

RERAISE

Re-raises the exception currently on top of the stack. If oparg is non-zero, pops an additional value from the stack which is used to set `f_lasti` of the current frame.

Added in version 3.9.

Changed in version 3.11: Exception representation on the stack now consist of one, not three, items.

PUSH_EXC_INFO

Pops a value from the stack. Pushes the current exception to the top of the stack. Pushes the value originally popped back to the stack. Used in exception handlers.

Added in version 3.11.

CHECK_EXC_MATCH

Performs exception matching for `except`. Tests whether the `STACK[-2]` is an exception matching `STACK[-1]`. Pops `STACK[-1]` and pushes the boolean result of the test.

Added in version 3.11.

CHECK_EG_MATCH

Performs exception matching for `except*`. Applies `split(STACK[-1])` on the exception group representing `STACK[-2]`.

In case of a match, pops two items from the stack and pushes the non-matching subgroup (`None` in case of full match) followed by the matching subgroup. When there is no match, pops one item (the match type) and pushes `None`.

Added in version 3.11.

WITH_EXCEPT_START

Calls the function in position 4 on the stack with arguments (type, val, tb) representing the exception at the top of the stack. Used to implement the call `context_manager.__exit__(*exc_info())` when an exception has occurred in a `with` statement.

Added in version 3.9.

Changed in version 3.11: The `__exit__` function is in position 4 of the stack rather than 7. Exception representation on the stack now consist of one, not three, items.

LOAD_ASSERTION_ERROR

Pushes *AssertionError* onto the stack. Used by the `assert` statement.

Added in version 3.9.

LOAD_BUILD_CLASS

Pushes `builtins.__build_class__()` onto the stack. It is later called to construct a class.

BEFORE_WITH

This opcode performs several operations before a with block starts. First, it loads `__exit__()` from the context manager and pushes it onto the stack for later use by *WITH_EXCEPT_START*. Then, `__enter__()` is called. Finally, the result of calling the `__enter__()` method is pushed onto the stack.

Added in version 3.11.

GET_LEN

Perform `STACK.append(len(STACK[-1]))`.

Added in version 3.10.

MATCH_MAPPING

If `STACK[-1]` is an instance of *collections.abc.Mapping* (or, more technically: if it has the `Py_TPFLAGS_MAPPING` flag set in its `tp_flags`), push `True` onto the stack. Otherwise, push `False`.

Added in version 3.10.

MATCH_SEQUENCE

If `STACK[-1]` is an instance of *collections.abc.Sequence* and is *not* an instance of *str*/*bytes*/*bytearray* (or, more technically: if it has the `Py_TPFLAGS_SEQUENCE` flag set in its `tp_flags`), push `True` onto the stack. Otherwise, push `False`.

Added in version 3.10.

MATCH_KEYS

`STACK[-1]` is a tuple of mapping keys, and `STACK[-2]` is the match subject. If `STACK[-2]` contains all of the keys in `STACK[-1]`, push a *tuple* containing the corresponding values. Otherwise, push `None`.

Added in version 3.10.

Changed in version 3.11: Previously, this instruction also pushed a boolean value indicating success (`True`) or failure (`False`).

STORE_NAME (*namei*)

Implements `name = STACK.pop()`. *namei* is the index of *name* in the attribute `co_names` of the code object. The compiler tries to use *STORE_FAST* or *STORE_GLOBAL* if possible.

DELETE_NAME (*namei*)

Implements `del name`, where *namei* is the index into `co_names` attribute of the code object.

UNPACK_SEQUENCE (*count*)

Unpacks `STACK[-1]` into *count* individual values, which are put onto the stack right-to-left. Require there to be exactly *count* values.:

```
assert(len(STACK[-1]) == count)
STACK.extend(STACK.pop()[:-count-1:-1])
```

UNPACK_EX (*counts*)

Implements assignment with a starred target: Unpacks an iterable in `STACK[-1]` into individual values, where the total number of values can be smaller than the number of items in the iterable: one of the new values will be a list of all leftover items.

The number of values before and after the list value is limited to 255.

The number of values before the list value is encoded in the argument of the opcode. The number of values after the list if any is encoded using an `EXTENDED_ARG`. As a consequence, the argument can be seen as a two bytes values where the low byte of *counts* is the number of values before the list value, the high byte of *counts* the number of values after it.

The extracted values are put onto the stack right-to-left, i.e. `a, *b, c = d` will be stored after execution as `STACK.extend((a, b, c))`.

STORE_ATTR (*namei*)

Implements:

```
obj = STACK.pop()
value = STACK.pop()
obj.name = value
```

where *namei* is the index of name in `co_names` of the code object.

DELETE_ATTR (*namei*)

Implements:

```
obj = STACK.pop()
del obj.name
```

where *namei* is the index of name into `co_names` of the code object.

STORE_GLOBAL (*namei*)

Works as *STORE_NAME*, but stores the name as a global.

DELETE_GLOBAL (*namei*)

Works as *DELETE_NAME*, but deletes a global name.

LOAD_CONST (*consti*)

Pushes `co_consts[consti]` onto the stack.

LOAD_NAME (*namei*)

Pushes the value associated with `co_names[namei]` onto the stack. The name is looked up within the locals, then the globals, then the builtins.

LOAD_LOCALS

Pushes a reference to the locals dictionary onto the stack. This is used to prepare namespace dictionaries for *LOAD_FROM_DICT_OR_DEREF* and *LOAD_FROM_DICT_OR_GLOBALS*.

Added in version 3.12.

LOAD_FROM_DICT_OR_GLOBALS (*i*)

Pops a mapping off the stack and looks up the value for `co_names[namei]`. If the name is not found there, looks it up in the globals and then the builtins, similar to *LOAD_GLOBAL*. This is used for loading global variables in annotation scopes within class bodies.

Added in version 3.12.

BUILD_TUPLE (*count*)

Creates a tuple consuming *count* items from the stack, and pushes the resulting tuple onto the stack.:

```
assert count > 0
STACK, values = STACK[:-count], STACK[-count:]
STACK.append(tuple(values))
```

BUILD_LIST (*count*)

Works as *BUILD_TUPLE*, but creates a list.

BUILD_SET (*count*)

Works as *BUILD_TUPLE*, but creates a set.

BUILD_MAP (*count*)

Pushes a new dictionary object onto the stack. Pops `2 * count` items so that the dictionary holds *count* entries: `{..., STACK[-4]: STACK[-3], STACK[-2]: STACK[-1]}`.

Changed in version 3.5: The dictionary is created from stack items instead of creating an empty dictionary pre-sized to hold *count* items.

BUILD_CONST_KEY_MAP (*count*)

The version of *BUILD_MAP* specialized for constant keys. Pops the top element on the stack which contains a tuple of keys, then starting from `STACK[-2]`, pops *count* values to form values in the built dictionary.

Added in version 3.6.

BUILD_STRING (*count*)

Concatenates *count* strings from the stack and pushes the resulting string onto the stack.

Added in version 3.6.

LIST_EXTEND (*i*)

Implements:

```
seq = STACK.pop()
list.extend(STACK[-i], seq)
```

Used to build lists.

Added in version 3.9.

SET_UPDATE (*i*)

Implements:

```
seq = STACK.pop()
set.update(STACK[-i], seq)
```

Used to build sets.

Added in version 3.9.

DICT_UPDATE (*i*)

Implements:

```
map = STACK.pop()
dict.update(STACK[-i], map)
```

Used to build dicts.

Added in version 3.9.

DICT_MERGE (*i*)

Like *DICT_UPDATE* but raises an exception for duplicate keys.

Added in version 3.9.

LOAD_ATTR (*namei*)

If the low bit of `namei` is not set, this replaces `STACK[-1]` with `getattr(STACK[-1], co_names[namei>>1])`.

If the low bit of `namei` is set, this will attempt to load a method named `co_names[namei>>1]` from the `STACK[-1]` object. `STACK[-1]` is popped. This bytecode distinguishes two cases: if `STACK[-1]` has a method with the correct name, the bytecode pushes the unbound method and `STACK[-1]`. `STACK[-1]` will be used as the first argument (`self`) by *CALL* or *CALL_KW* when calling the unbound method. Otherwise, `NULL` and the object returned by the attribute lookup are pushed.

Changed in version 3.12: If the low bit of `namei` is set, then a `NULL` or `self` is pushed to the stack before the attribute or unbound method respectively.

LOAD_SUPER_ATTR (*namei*)

This opcode implements *super()*, both in its zero-argument and two-argument forms (e.g. `super().method()`, `super().attr` and `super(cls, self).method()`, `super(cls, self).attr`).

It pops three values from the stack (from top of stack down):

- `self`: the first argument to the current method
- `cls`: the class within which the current method was defined
- the global `super`

With respect to its argument, it works similarly to *LOAD_ATTR*, except that `namei` is shifted left by 2 bits instead of 1.

The low bit of `namei` signals to attempt a method load, as with *LOAD_ATTR*, which results in pushing `NULL` and the loaded method. When it is unset a single value is pushed to the stack.

The second-low bit of `namei`, if set, means that this was a two-argument call to *super()* (unset means zero-argument).

Added in version 3.12.

COMPARE_OP (*opname*)

Performs a Boolean operation. The operation name can be found in `cmp_op[opname >> 5]`. If the fifth-lowest bit of `opname` is set (`opname & 16`), the result should be coerced to `bool`.

Changed in version 3.13: The fifth-lowest bit of the oparg now indicates a forced conversion to *bool*.

IS_OP (*invert*)

Performs `is` comparison, or `is not` if `invert` is 1.

Added in version 3.9.

CONTAINS_OP (*invert*)

Performs `in` comparison, or `not in` if `invert` is 1.

Added in version 3.9.

IMPORT_NAME (*namei*)

Imports the module `co_names[namei]`. `STACK[-1]` and `STACK[-2]` are popped and provide the *fromlist* and *level* arguments of *__import__()*. The module object is pushed onto the stack. The current namespace is not affected: for a proper import statement, a subsequent *STORE_FAST* instruction modifies the namespace.

IMPORT_FROM (*namei*)

Loads the attribute `co_names[namei]` from the module found in `STACK[-1]`. The resulting object is pushed onto the stack, to be subsequently stored by a *STORE_FAST* instruction.

JUMP_FORWARD (*delta*)

Increments bytecode counter by *delta*.

JUMP_BACKWARD (*delta*)

Decrements bytecode counter by *delta*. Checks for interrupts.

Added in version 3.11.

JUMP_BACKWARD_NO_INTERRUPT (*delta*)

Decrements bytecode counter by *delta*. Does not check for interrupts.

Added in version 3.11.

POP_JUMP_IF_TRUE (*delta*)

If `STACK[-1]` is true, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

Changed in version 3.11: The oparg is now a relative delta rather than an absolute target. This opcode is a pseudo-instruction, replaced in final bytecode by the directed versions (forward/backward).

Changed in version 3.12: This is no longer a pseudo-instruction.

Changed in version 3.13: This instruction now requires an exact *bool* operand.

POP_JUMP_IF_FALSE (*delta*)

If `STACK[-1]` is false, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

Changed in version 3.11: The oparg is now a relative delta rather than an absolute target. This opcode is a pseudo-instruction, replaced in final bytecode by the directed versions (forward/backward).

Changed in version 3.12: This is no longer a pseudo-instruction.

Changed in version 3.13: This instruction now requires an exact *bool* operand.

POP_JUMP_IF_NOT_NONE (*delta*)

If `STACK[-1]` is not `None`, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

This opcode is a pseudo-instruction, replaced in final bytecode by the directed versions (forward/backward).

Added in version 3.11.

Changed in version 3.12: This is no longer a pseudo-instruction.

POP_JUMP_IF_NONE (*delta*)

If `STACK[-1]` is `None`, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

This opcode is a pseudo-instruction, replaced in final bytecode by the directed versions (forward/backward).

Added in version 3.11.

Changed in version 3.12: This is no longer a pseudo-instruction.

FOR_ITER (*delta*)

STACK[-1] is an *iterator*. Call its *__next__()* method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted then the byte code counter is incremented by *delta*.

Changed in version 3.12: Up until 3.11 the iterator was popped when it was exhausted.

LOAD_GLOBAL (*namei*)

Loads the global named co_names[namei>>1] onto the stack.

Changed in version 3.11: If the low bit of namei is set, then a NULL is pushed to the stack before the global variable.

LOAD_FAST (*var_num*)

Pushes a reference to the local co_varnames[var_num] onto the stack.

Changed in version 3.12: This opcode is now only used in situations where the local variable is guaranteed to be initialized. It cannot raise *UnboundLocalError*.

LOAD_FAST_CHECK (*var_num*)

Pushes a reference to the local co_varnames[var_num] onto the stack, raising an *UnboundLocalError* if the local variable has not been initialized.

Added in version 3.12.

LOAD_FAST_AND_CLEAR (*var_num*)

Pushes a reference to the local co_varnames[var_num] onto the stack (or pushes NULL onto the stack if the local variable has not been initialized) and sets co_varnames[var_num] to NULL.

Added in version 3.12.

STORE_FAST (*var_num*)

Stores STACK.pop() into the local co_varnames[var_num].

DELETE_FAST (*var_num*)

Deletes local co_varnames[var_num].

MAKE_CELL (*i*)

Creates a new cell in slot i. If that slot is nonempty then that value is stored into the new cell.

Added in version 3.11.

LOAD_DEREF (*i*)

Loads the cell contained in slot i of the "fast locals" storage. Pushes a reference to the object the cell contains on the stack.

Changed in version 3.11: i is no longer offset by the length of co_varnames.

LOAD_FROM_DICT_OR_DEREF (*i*)

Pops a mapping off the stack and looks up the name associated with slot i of the "fast locals" storage in this mapping. If the name is not found there, loads it from the cell contained in slot i, similar to *LOAD_DEREF*. This is used for loading free variables in class bodies (which previously used LOAD_CLASSDEREF) and in annotation scopes within class bodies.

Added in version 3.12.

STORE_DEREF (*i*)

Stores STACK.pop() into the cell contained in slot i of the "fast locals" storage.

Changed in version 3.11: i is no longer offset by the length of co_varnames.

DELETE_DEREF (*i*)

Empties the cell contained in slot `i` of the "fast locals" storage. Used by the `del` statement.

Added in version 3.2.

Changed in version 3.11: `i` is no longer offset by the length of `co_varnames`.

COPY_FREE_VARS (*n*)

Copies the `n` free variables from the closure into the frame. Removes the need for special code on the caller's side when calling closures.

Added in version 3.11.

RAISE_VARARGS (*argc*)

Raises an exception using one of the 3 forms of the `raise` statement, depending on the value of *argc*:

- 0: `raise` (re-raise previous exception)
- 1: `raise STACK[-1]` (raise exception instance or type at `STACK[-1]`)
- 2: `raise STACK[-2] from STACK[-1]` (raise exception instance or type at `STACK[-2]` with `__cause__` set to `STACK[-1]`)

CALL (*argc*)

Calls a callable object with the number of arguments specified by `argc`. On the stack are (in ascending order):

- The callable
- `self` or `NULL`
- The remaining positional arguments

`argc` is the total of the positional arguments, excluding `self`.

`CALL` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Added in version 3.11.

Changed in version 3.13: The callable now always appears at the same position on the stack.

Changed in version 3.13: Calls with keyword arguments are now handled by *CALL_KW*.

CALL_KW (*argc*)

Calls a callable object with the number of arguments specified by `argc`, including one or more named arguments. On the stack are (in ascending order):

- The callable
- `self` or `NULL`
- The remaining positional arguments
- The named arguments
- A *tuple* of keyword argument names

`argc` is the total of the positional and named arguments, excluding `self`. The length of the tuple of keyword argument names is the number of named arguments.

`CALL_KW` pops all arguments, the keyword names, and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Added in version 3.13.

CALL_FUNCTION_EX (*flags*)

Calls a callable object with variable set of positional and keyword arguments. If the lowest bit of *flags* is set, the top of the stack contains a mapping object containing additional keyword arguments. Before the callable is called, the mapping object and iterable object are each "unpacked" and their contents passed in as keyword and positional arguments respectively. `CALL_FUNCTION_EX` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Added in version 3.6.

PUSH_NULL

Pushes a `NULL` to the stack. Used in the call sequence to match the `NULL` pushed by *LOAD_METHOD* for non-method calls.

Added in version 3.11.

MAKE_FUNCTION

Pushes a new function object on the stack built from the code object at `STACK[1]`.

Changed in version 3.10: Flag value `0x04` is a tuple of strings instead of dictionary

Changed in version 3.11: Qualified name at `STACK[-1]` was removed.

Changed in version 3.13: Extra function attributes on the stack, signaled by oparg flags, were removed. They now use *SET_FUNCTION_ATTRIBUTE*.

SET_FUNCTION_ATTRIBUTE (*flag*)

Sets an attribute on a function object. Expects the function at `STACK[-1]` and the attribute value to set at `STACK[-2]`; consumes both and leaves the function at `STACK[-1]`. The flag determines which attribute to set:

- `0x01` a tuple of default values for positional-only and positional-or-keyword parameters in positional order
- `0x02` a dictionary of keyword-only parameters' default values
- `0x04` a tuple of strings containing parameters' annotations
- `0x08` a tuple containing cells for free variables, making a closure

Added in version 3.13.

BUILD_SLICE (*argc*)

Pushes a slice object on the stack. *argc* must be 2 or 3. If it is 2, implements:

```
end = STACK.pop()
start = STACK.pop()
STACK.append(slice(start, stop))
```

if it is 3, implements:

```
step = STACK.pop()
end = STACK.pop()
start = STACK.pop()
STACK.append(slice(start, end, step))
```

See the *slice()* built-in function for more information.

EXTENDED_ARG (*ext*)

Prefixes any opcode which has an argument too big to fit into the default one byte. *ext* holds an additional byte which act as higher bits in the argument. For each opcode, at most three prefixal `EXTENDED_ARG` are allowed, forming an argument from two-byte to four-byte.

CONVERT_VALUE (*oparg*)

Convert value to a string, depending on `oparg`:

```
value = STACK.pop()
result = func(value)
STACK.append(result)
```

- `oparg == 1`: call *str()* on *value*
- `oparg == 2`: call *repr()* on *value*
- `oparg == 3`: call *ascii()* on *value*

Used for implementing formatted literal strings (f-strings).

Added in version 3.13.

FORMAT_SIMPLE

Formats the value on top of stack:

```
value = STACK.pop()
result = value.__format__("")
STACK.append(result)
```

Used for implementing formatted literal strings (f-strings).

Added in version 3.13.

FORMAT_SPEC

Formats the given value with the given format spec:

```
spec = STACK.pop()
value = STACK.pop()
result = value.__format__(spec)
STACK.append(result)
```

Used for implementing formatted literal strings (f-strings).

Added in version 3.13.

MATCH_CLASS (*count*)

`STACK[-1]` is a tuple of keyword attribute names, `STACK[-2]` is the class being matched against, and `STACK[-3]` is the match subject. *count* is the number of positional sub-patterns.

Pop `STACK[-1]`, `STACK[-2]`, and `STACK[-3]`. If `STACK[-3]` is an instance of `STACK[-2]` and has the positional and keyword attributes required by *count* and `STACK[-1]`, push a tuple of extracted attributes. Otherwise, push `None`.

Added in version 3.10.

Changed in version 3.11: Previously, this instruction also pushed a boolean value indicating success (`True`) or failure (`False`).

RESUME (*context*)

A no-op. Performs internal tracing, debugging and optimization checks.

The `context` oparand consists of two parts. The lowest two bits indicate where the `RESUME` occurs:

- `0` The start of a function, which is neither a generator, coroutine nor an async generator
- `1` After a `yield` expression

- 2 After a `yield from` expression
- 3 After an `await` expression

The next bit is 1 if the RESUME is at except-depth 1, and 0 otherwise.

Added in version 3.11.

Changed in version 3.13: The oparg value changed to include information about except-depth

RETURN_GENERATOR

Create a generator, coroutine, or async generator from the current frame. Used as first opcode of in code object for the above mentioned callables. Clear the current frame and return the newly created generator.

Added in version 3.11.

SEND (*delta*)

Equivalent to `STACK[-1] = STACK[-2].send(STACK[-1])`. Used in `yield from` and `await` statements.

If the call raises *StopIteration*, pop the top value from the stack, push the exception's `value` attribute, and increment the bytecode counter by *delta*.

Added in version 3.11.

HAVE_ARGUMENT

This is not really an opcode. It identifies the dividing line between opcodes in the range [0,255] which don't use their argument and those that do (`< HAVE_ARGUMENT` and `>= HAVE_ARGUMENT`, respectively).

If your application uses pseudo instructions or specialized instructions, use the *hasarg* collection instead.

Changed in version 3.6: Now every instruction has an argument, but opcodes `< HAVE_ARGUMENT` ignore it. Before, only opcodes `>= HAVE_ARGUMENT` had an argument.

Changed in version 3.12: Pseudo instructions were added to the *dis* module, and for them it is not true that comparison with `HAVE_ARGUMENT` indicates whether they use their arg.

Deprecated since version 3.13: Use *hasarg* instead.

CALL_INTRINSIC_1

Calls an intrinsic function with one argument. Passes `STACK[-1]` as the argument and sets `STACK[-1]` to the result. Used to implement functionality that is not performance critical.

The operand determines which intrinsic function is called:

Operand	Description
INTRIN-SIC_1_INVALID	Not valid
INTRINSIC_PRINT	Prints the argument to standard out. Used in the REPL.
INTRIN-SIC_IMPORT_STAR	Performs `import *` for the named module.
INTRIN-SIC_STOPITERATION	Extracts the return value from a `StopIteration` exception.
INTRIN-SIC_ASYNC_GEN_WRA	Wraps an async generator value
INTRIN-SIC_UNARY_POSITIV	Performs the unary + operation
INTRIN-SIC_LIST_TO_TUPLE	Converts a list to a tuple
INTRIN-SIC_TYPEVAR	Creates a *typing.TypeVar*
INTRIN-SIC_PARAMSPEC	Creates a *typing.ParamSpec*
INTRIN-SIC_TYPEVARTUPLE	Creates a *typing.TypeVarTuple*
INTRIN-SIC_SUBSCRIPT_GEN	Returns *typing.Generic* subscripted with the argument
INTRIN-SIC_TYPEALIAS	Creates a *typing.TypeAliasType*; used in the `type` statement. The argument is a tuple of the type alias's name, type parameters, and value.

Added in version 3.12.

CALL_INTRINSIC_2

Calls an intrinsic function with two arguments. Used to implement functionality that is not performance critical:

```
arg2 = STACK.pop()
arg1 = STACK.pop()
result = intrinsic2(arg1, arg2)
STACK.append(result)
```

The operand determines which intrinsic function is called:

Operand	Description
INTRINSIC_2_INVALID	Not valid
INTRINSIC_PREP_RERAISE_STAR	Calculates the *ExceptionGroup* to raise from a `try-except*`.
INTRINSIC_TYPEVAR_WITH_BOUND	Creates a *typing.TypeVar* with a bound.
INTRINSIC_TYPEVAR_WITH_CONSTRAI	Creates a *typing.TypeVar* with constraints.
INTRINSIC_SET_FUNCTION_TYPE_PAR	Sets the `__type_params__` attribute of a function.

Added in version 3.12.

Pseudo-instructions

These opcodes do not appear in Python bytecode. They are used by the compiler but are replaced by real opcodes or removed before bytecode is generated.

SETUP_FINALLY (*target*)

Set up an exception handler for the following code block. If an exception occurs, the value stack level is restored to its current state and control is transferred to the exception handler at `target`.

SETUP_CLEANUP (*target*)

Like `SETUP_FINALLY`, but in case of an exception also pushes the last instruction (`lasti`) to the stack so that `RERAISE` can restore it. If an exception occurs, the value stack level and the last instruction on the frame are restored to their current state, and control is transferred to the exception handler at `target`.

SETUP_WITH (*target*)

Like `SETUP_CLEANUP`, but in case of an exception one more item is popped from the stack before control is transferred to the exception handler at `target`.

This variant is used in `with` and `async with` constructs, which push the return value of the context manager's `__enter__()` or `__aenter__()` to the stack.

POP_BLOCK

Marks the end of the code block associated with the last `SETUP_FINALLY`, `SETUP_CLEANUP` or `SETUP_WITH`.

JUMP

JUMP_NO_INTERRUPT

Undirected relative jump instructions which are replaced by their directed (forward/backward) counterparts by the assembler.

LOAD_CLOSURE (*i*)

Pushes a reference to the cell contained in slot `i` of the "fast locals" storage.

Note that `LOAD_CLOSURE` is replaced with `LOAD_FAST` in the assembler.

Changed in version 3.13: This opcode is now a pseudo-instruction.

LOAD_METHOD

Optimized unbound method lookup. Emitted as a `LOAD_ATTR` opcode with a flag set in the arg.

32.10.5 Opcode collections

These collections are provided for automatic introspection of bytecode instructions:

Changed in version 3.12: The collections now contain pseudo instructions and instrumented instructions as well. These are opcodes with values `>= MIN_PSEUDO_OPCODE` and `>= MIN_INSTRUMENTED_OPCODE`.

`dis.`**opname**

Sequence of operation names, indexable using the bytecode.

`dis.`**opmap**

Dictionary mapping operation names to bytecodes.

`dis.`**cmp_op**

Sequence of all compare operation names.

`dis.`**hasarg**

Sequence of bytecodes that use their argument.

Added in version 3.12.

dis.**hasconst**

Sequence of bytecodes that access a constant.

dis.**hasfree**

Sequence of bytecodes that access a free variable. 'free' in this context refers to names in the current scope that are referenced by inner scopes or names in outer scopes that are referenced from this scope. It does *not* include references to global or builtin scopes.

dis.**hasname**

Sequence of bytecodes that access an attribute by name.

dis.**hasjump**

Sequence of bytecodes that have a jump target. All jumps are relative.

Added in version 3.13.

dis.**haslocal**

Sequence of bytecodes that access a local variable.

dis.**hascompare**

Sequence of bytecodes of Boolean operations.

dis.**hasexc**

Sequence of bytecodes that set an exception handler.

Added in version 3.12.

dis.**hasjrel**

Sequence of bytecodes that have a relative jump target.

Deprecated since version 3.13: All jumps are now relative. Use *hasjump*.

dis.**hasjabs**

Sequence of bytecodes that have an absolute jump target.

Deprecated since version 3.13: All jumps are now relative. This list is empty.

32.11 pickletools — Tools for pickle developers

Source code: Lib/pickletools.py

This module contains various constants relating to the intimate details of the *pickle* module, some lengthy comments about the implementation, and a few useful functions for analyzing pickled data. The contents of this module are useful for Python core developers who are working on the *pickle*; ordinary users of the *pickle* module probably won't find the *pickletools* module relevant.

32.11.1 Command line usage

Added in version 3.2.

When invoked from the command line, `python -m pickletools` will disassemble the contents of one or more pickle files. Note that if you want to see the Python object stored in the pickle rather than the details of pickle format, you may want to use `-m pickle` instead. However, when the pickle file that you want to examine comes from an untrusted source, `-m pickletools` is a safer option because it does not execute pickle bytecode.

For example, with a tuple `(1, 2)` pickled in file `x.pickle`:

```
$ python -m pickle x.pickle
(1, 2)

$ python -m pickletools x.pickle
    0: \x80 PROTO      3
    2: K    BININT1    1
    4: K    BININT1    2
    6: \x86 TUPLE2
    7: q    BINPUT     0
    9: .    STOP
highest protocol among opcodes = 2
```

Command line options

-a, --annotate

Annotate each line with a short opcode description.

-o, --output=<file>

Name of a file where the output should be written.

-l, --indentlevel=<num>

The number of blanks by which to indent a new MARK level.

-m, --memo

When multiple objects are disassembled, preserve memo between disassemblies.

-p, --preamble=<preamble>

When more than one pickle file are specified, print given preamble before each disassembly.

32.11.2 Programmatic Interface

`pickletools.`**dis**(*pickle*, *out=None*, *memo=None*, *indentlevel=4*, *annotate=0*)

Outputs a symbolic disassembly of the pickle to the file-like object *out*, defaulting to `sys.stdout`. *pickle* can be a string or a file-like object. *memo* can be a Python dictionary that will be used as the pickle's memo; it can be used to perform disassemblies across multiple pickles created by the same pickler. Successive levels, indicated by `MARK` opcodes in the stream, are indented by *indentlevel* spaces. If a nonzero value is given to *annotate*, each opcode in the output is annotated with a short description. The value of *annotate* is used as a hint for the column where annotation should start.

Changed in version 3.2: Added the *annotate* parameter.

pickletools.**genops**(*pickle*)

Provides an *iterator* over all of the opcodes in a pickle, returning a sequence of `(opcode, arg, pos)` triples. *opcode* is an instance of an `OpcodeInfo` class; *arg* is the decoded value, as a Python object, of the opcode's argument; *pos* is the position at which this opcode is located. *pickle* can be a string or a file-like object.

pickletools.**optimize**(*picklestring*)

Returns a new equivalent pickle string after eliminating unused `PUT` opcodes. The optimized pickle is shorter, takes less transmission time, requires less storage space, and unpickles more efficiently.

CHAPTER

THIRTYTHREE

MS WINDOWS SPECIFIC SERVICES

This chapter describes modules that are only available on MS Windows platforms.

33.1 `msvcrt` — Useful routines from the MS VC++ runtime

These functions provide access to some useful capabilities on Windows platforms. Some higher-level modules use these functions to build the Windows implementations of their services. For example, the *getpass* module uses this in the implementation of the *getpass()* function.

Further documentation on these functions can be found in the Platform API documentation.

The module implements both the normal and wide char variants of the console I/O api. The normal API deals only with ASCII characters and is of limited use for internationalized applications. The wide char API should be used where ever possible.

Changed in version 3.3: Operations in this module now raise *OSError* where *IOError* was raised.

33.1.1 File Operations

msvcrt.**locking**(*fd*, *mode*, *nbytes*)

Lock part of a file based on file descriptor *fd* from the C runtime. Raises *OSError* on failure. The locked region of the file extends from the current file position for *nbytes* bytes, and may continue beyond the end of the file. *mode* must be one of the `LK_*` constants listed below. Multiple regions in a file may be locked at the same time, but may not overlap. Adjacent regions are not merged; they must be unlocked individually.

Raises an *auditing event* `msvcrt.locking` with arguments `fd`, `mode`, `nbytes`.

msvcrt.**LK_LOCK**

msvcrt.**LK_RLCK**

Locks the specified bytes. If the bytes cannot be locked, the program immediately tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, *OSError* is raised.

msvcrt.**LK_NBLCK**

msvcrt.**LK_NBRLCK**

Locks the specified bytes. If the bytes cannot be locked, *OSError* is raised.

msvcrt.**LK_UNLCK**

Unlocks the specified bytes, which must have been previously locked.

msvcrt.**setmode**(*fd*, *flags*)

Set the line-end translation mode for the file descriptor *fd*. To set it to text mode, *flags* should be *os.O_TEXT*; for binary, it should be *os.O_BINARY*.

msvcrt.**open_osfhandle**(*handle*, *flags*)

Create a C runtime file descriptor from the file handle *handle*. The *flags* parameter should be a bitwise OR of *os.O_APPEND*, *os.O_RDONLY*, *os.O_TEXT* and *os.O_NOINHERIT*. The returned file descriptor may be used as a parameter to *os.fdopen()* to create a file object.

The file descriptor is inheritable by default. Pass *os.O_NOINHERIT* flag to make it non inheritable.

Raises an *auditing event* `msvcrt.open_osfhandle` with arguments `handle`, `flags`.

msvcrt.**get_osfhandle**(*fd*)

Return the file handle for the file descriptor *fd*. Raises *OSError* if *fd* is not recognized.

Raises an *auditing event* `msvcrt.get_osfhandle` with argument `fd`.

33.1.2 Console I/O

msvcrt.**kbhit**()

Returns a nonzero value if a keypress is waiting to be read. Otherwise, return 0.

msvcrt.**getch**()

Read a keypress and return the resulting character as a byte string. Nothing is echoed to the console. This call will block if a keypress is not already available, but will not wait for `Enter` to be pressed. If the pressed key was a special function key, this will return `'\000'` or `'\xe0'`; the next call will return the keycode. The `Control-C` keypress cannot be read with this function.

msvcrt.**getwch**()

Wide char variant of *getch()*, returning a Unicode value.

msvcrt.**getche**()

Similar to *getch()*, but the keypress will be echoed if it represents a printable character.

msvcrt.**getwche**()

Wide char variant of *getche()*, returning a Unicode value.

msvcrt.**putch**(*char*)

Print the byte string *char* to the console without buffering.

msvcrt.**putwch**(*unicode_char*)

Wide char variant of *putch()*, accepting a Unicode value.

msvcrt.**ungetch**(*char*)

Cause the byte string *char* to be "pushed back" into the console buffer; it will be the next character read by *getch()* or *getche()*.

msvcrt.**ungetwch**(*unicode_char*)

Wide char variant of *ungetch()*, accepting a Unicode value.

33.1.3 Other Functions

msvcrt.**heapmin**()

Force the `malloc()` heap to clean itself up and return unused blocks to the operating system. On failure, this raises *OSError*.

msvcrt.**set_error_mode**(*mode*)

Changes the location where the C runtime writes an error message for an error that might end the program. *mode* must be one of the `OUT_*` constants listed below or *REPORT_ERRMODE*. Returns the old setting or -1 if an error occurs. Only available in debug build of Python.

msvcrt.**OUT_TO_DEFAULT**

Error sink is determined by the app's type. Only available in debug build of Python.

msvcrt.**OUT_TO_STDERR**

Error sink is a standard error. Only available in debug build of Python.

msvcrt.**OUT_TO_MSGBOX**

Error sink is a message box. Only available in debug build of Python.

msvcrt.**REPORT_ERRMODE**

Report the current error mode value. Only available in debug build of Python.

msvcrt.**CrtSetReportMode**(*type*, *mode*)

Specifies the destination or destinations for a specific report type generated by `_CrtDbgReport()` in the MS VC++ runtime. *type* must be one of the `CRT_*` constants listed below. *mode* must be one of the `CRTDBG_*` constants listed below. Only available in debug build of Python.

msvcrt.**CrtSetReportFile**(*type*, *file*)

After you use *CrtSetReportMode()* to specify *CRTDBG_MODE_FILE*, you can specify the file handle to receive the message text. *type* must be one of the `CRT_*` constants listed below. *file* should be the file handle your want specified. Only available in debug build of Python.

msvcrt.**CRT_WARN**

Warnings, messages, and information that doesn't need immediate attention.

msvcrt.**CRT_ERROR**

Errors, unrecoverable problems, and issues that require immediate attention.

msvcrt.**CRT_ASSERT**

Assertion failures.

msvcrt.**CRTDBG_MODE_DEBUG**

Writes the message to the debugger's output window.

msvcrt.**CRTDBG_MODE_FILE**

Writes the message to a user-supplied file handle. *CrtSetReportFile()* should be called to define the specific file or stream to use as the destination.

msvcrt.**CRTDBG_MODE_WNDW**

Creates a message box to display the message along with the `Abort`, `Retry`, and `Ignore` buttons.

msvcrt.**CRTDBG_REPORT_MODE**

Returns current *mode* for the specified *type*.

msvcrt.**CRT_ASSEMBLY_VERSION**

The CRT Assembly version, from the `crtassem.h` header file.

msvcrt.**VC_ASSEMBLY_PUBLICKEYTOKEN**

The VC Assembly public key token, from the `crtassem.h` header file.

msvcrt.**LIBRARIES_ASSEMBLY_NAME_PREFIX**

The Libraries Assembly name prefix, from the `crtassem.h` header file.

33.2 winreg — Windows registry access

These functions expose the Windows registry API to Python. Instead of using an integer as the registry handle, a *handle object* is used to ensure that the handles are closed correctly, even if the programmer neglects to explicitly close them.

Changed in version 3.3: Several functions in this module used to raise a *WindowsError*, which is now an alias of *OSError*.

33.2.1 Functions

This module offers the following functions:

winreg.**CloseKey**(*hkey*)

Closes a previously opened registry key. The *hkey* argument specifies a previously opened key.

> **Note**
>
> If *hkey* is not closed using this method (or via *hkey.Close()*), it is closed when the *hkey* object is destroyed by Python.

winreg.**ConnectRegistry**(*computer_name*, *key*)

Establishes a connection to a predefined registry handle on another computer, and returns a *handle object*.

computer_name is the name of the remote computer, of the form `r"\\computername"`. If `None`, the local computer is used.

key is the predefined handle to connect to.

The return value is the handle of the opened key. If the function fails, an *OSError* exception is raised.

Raises an *auditing event* `winreg.ConnectRegistry` with arguments `computer_name`, `key`.

Changed in version 3.3: See *above*.

winreg.**CreateKey**(*key*, *sub_key*)

Creates or opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the key this method opens or creates.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an *OSError* exception is raised.

Raises an *auditing event* `winreg.CreateKey` with arguments `key`, `sub_key`, `access`.

Raises an *auditing event* `winreg.OpenKey/result` with argument `key`.

Changed in version 3.3: See *above*.

`winreg.`**`CreateKeyEx`**(*key*, *sub_key*, *reserved=0*, *access=KEY_WRITE*)

Creates or opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the key this method opens or creates.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is *KEY_WRITE*. See *Access Rights* for other allowed values.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an *OSError* exception is raised.

Raises an *auditing event* `winreg.CreateKey` with arguments `key`, `sub_key`, `access`.

Raises an *auditing event* `winreg.OpenKey/result` with argument `key`.

Added in version 3.2.

Changed in version 3.3: See *above*.

`winreg.`**`DeleteKey`**(*key*, *sub_key*)

Deletes the specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be `None`, and the key may not have subkeys.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an *OSError* exception is raised.

Raises an *auditing event* `winreg.DeleteKey` with arguments `key`, `sub_key`, `access`.

Changed in version 3.3: See *above*.

`winreg.`**`DeleteKeyEx`**(*key*, *sub_key*, *access=KEY_WOW64_64KEY*, *reserved=0*)

Deletes the specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be `None`, and the key may not have subkeys.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is *KEY_WOW64_64KEY*. On 32-bit Windows, the WOW64 constants are ignored. See *Access Rights* for other allowed values.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an *OSError* exception is raised.

On unsupported Windows versions, *NotImplementedError* is raised.

Raises an *auditing event* `winreg.DeleteKey` with arguments `key`, `sub_key`, `access`.

Added in version 3.2.

Changed in version 3.3: See *above*.

`winreg.`**`DeleteValue`**(*key*, *value*)

Removes a named value from a registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value is a string that identifies the value to remove.

Raises an *auditing event* `winreg.DeleteValue` with arguments `key`, `value`.

`winreg.`**`EnumKey`**(*key*, *index*)

Enumerates subkeys of an open registry key, returning a string.

key is an already open key, or one of the predefined *HKEY_* constants*.

index is an integer that identifies the index of the key to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly until an *OSError* exception is raised, indicating, no more values are available.

Raises an *auditing event* `winreg.EnumKey` with arguments `key`, `index`.

Changed in version 3.3: See *above*.

`winreg.`**`EnumValue`**(*key*, *index*)

Enumerates values of an open registry key, returning a tuple.

key is an already open key, or one of the predefined *HKEY_* constants*.

index is an integer that identifies the index of the value to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly, until an *OSError* exception is raised, indicating no more values.

The result is a tuple of 3 items:

Index	Meaning
0	A string that identifies the value name
1	An object that holds the value data, and whose type depends on the underlying registry type
2	An integer that identifies the type of the value data (see table in docs for *SetValueEx()*)

Raises an *auditing event* `winreg.EnumValue` with arguments `key`, `index`.

Changed in version 3.3: See *above*.

`winreg.`**`ExpandEnvironmentStrings`**(*str*)

Expands environment variable placeholders `%NAME%` in strings like *REG_EXPAND_SZ*:

```
>>> ExpandEnvironmentStrings('%windir%')
'C:\\Windows'
```

Raises an *auditing event* `winreg.ExpandEnvironmentStrings` with argument `str`.

winreg.**FlushKey**(*key*)

Writes all the attributes of a key to the registry.

key is an already open key, or one of the predefined *HKEY_* constants*.

It is not necessary to call `FlushKey()` to change a key. Registry changes are flushed to disk by the registry using its lazy flusher. Registry changes are also flushed to disk at system shutdown. Unlike `CloseKey()`, the `FlushKey()` method returns only when all the data has been written to the registry. An application should only call `FlushKey()` if it requires absolute certainty that registry changes are on disk.

> **Note**
>
> If you don't know whether a `FlushKey()` call is required, it probably isn't.

winreg.**LoadKey**(*key*, *sub_key*, *file_name*)

Creates a subkey under the specified key and stores registration information from a specified file into that subkey.

key is a handle returned by `ConnectRegistry()` or one of the constants `HKEY_USERS` or `HKEY_LOCAL_MACHINE`.

sub_key is a string that identifies the subkey to load.

file_name is the name of the file to load registry data from. This file must have been created with the `SaveKey()` function. Under the file allocation table (FAT) file system, the filename may not have an extension.

A call to `LoadKey()` fails if the calling process does not have the `SE_RESTORE_PRIVILEGE` privilege. Note that privileges are different from permissions – see the RegLoadKey documentation for more details.

If *key* is a handle returned by `ConnectRegistry()`, then the path specified in *file_name* is relative to the remote computer.

Raises an *auditing event* `winreg.LoadKey` with arguments `key`, `sub_key`, `file_name`.

winreg.**OpenKey**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)

winreg.**OpenKeyEx**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)

Opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that identifies the sub_key to open.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is `KEY_READ`. See *Access Rights* for other allowed values.

The result is a new handle to the specified key.

If the function fails, `OSError` is raised.

Raises an *auditing event* `winreg.OpenKey` with arguments `key`, `sub_key`, `access`.

Raises an *auditing event* `winreg.OpenKey/result` with argument `key`.

Changed in version 3.2: Allow the use of named arguments.

Changed in version 3.3: See *above*.

winreg.**QueryInfoKey**(*key*)

Returns information about a key, as a tuple.

key is an already open key, or one of the predefined *HKEY_* constants*.

The result is a tuple of 3 items:

Index	Meaning
0	An integer giving the number of sub keys this key has.
1	An integer giving the number of values this key has.
2	An integer giving when the key was last modified (if available) as 100's of nanoseconds since Jan 1, 1601.

Raises an *auditing event* `winreg.QueryInfoKey` with argument `key`.

`winreg.`**`QueryValue`**(*key*, *sub_key*)

Retrieves the unnamed value for a key, as a string.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that holds the name of the subkey with which the value is associated. If this parameter is `None` or empty, the function retrieves the value set by the *`SetValue()`* method for the key identified by *key*.

Values in the registry have name, type, and data components. This method retrieves the data for a key's first value that has a `NULL` name. But the underlying API call doesn't return the type, so always use *`QueryValueEx()`* if possible.

Raises an *auditing event* `winreg.QueryValue` with arguments `key`, `sub_key`, `value_name`.

`winreg.`**`QueryValueEx`**(*key*, *value_name*)

Retrieves the type and data for a specified value name associated with an open registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value_name is a string indicating the value to query.

The result is a tuple of 2 items:

Index	Meaning
0	The value of the registry item.
1	An integer giving the registry type for this value (see table in docs for *`SetValueEx()`*)

Raises an *auditing event* `winreg.QueryValue` with arguments `key`, `sub_key`, `value_name`.

`winreg.`**`SaveKey`**(*key*, *file_name*)

Saves the specified key, and all its subkeys to the specified file.

key is an already open key, or one of the predefined *HKEY_* constants*.

file_name is the name of the file to save registry data to. This file cannot already exist. If this filename includes an extension, it cannot be used on file allocation table (FAT) file systems by the *`LoadKey()`* method.

If *key* represents a key on a remote computer, the path described by *file_name* is relative to the remote computer. The caller of this method must possess the **SeBackupPrivilege** security privilege. Note that privileges are different than permissions – see the Conflicts Between User Rights and Permissions documentation for more details.

This function passes `NULL` for *security_attributes* to the API.

Raises an *auditing event* `winreg.SaveKey` with arguments `key`, `file_name`.

winreg.**SetValue**(*key*, *sub_key*, *type*, *value*)

Associates a value with a specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the subkey with which the value is associated.

type is an integer that specifies the type of the data. Currently this must be *REG_SZ*, meaning only strings are supported. Use the *SetValueEx()* function for support for other data types.

value is a string that specifies the new value.

If the key specified by the *sub_key* parameter does not exist, the SetValue function creates it.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

The key identified by the *key* parameter must have been opened with *KEY_SET_VALUE* access.

Raises an *auditing event* `winreg.SetValue` with arguments `key`, `sub_key`, `type`, `value`.

winreg.**SetValueEx**(*key*, *value_name*, *reserved*, *type*, *value*)

Stores data in the value field of an open registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value_name is a string that names the subkey with which the value is associated.

reserved can be anything – zero is always passed to the API.

type is an integer that specifies the type of the data. See *Value Types* for the available types.

value is a string that specifies the new value.

This method can also set additional value and type information for the specified key. The key identified by the key parameter must have been opened with *KEY_SET_VALUE* access.

To open the key, use the *CreateKey()* or *OpenKey()* methods.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

Raises an *auditing event* `winreg.SetValue` with arguments `key`, `sub_key`, `type`, `value`.

winreg.**DisableReflectionKey**(*key*)

Disables registry reflection for 32-bit processes running on a 64-bit operating system.

key is an already open key, or one of the predefined *HKEY_* constants*.

Will generally raise *NotImplementedError* if executed on a 32-bit operating system.

If the key is not on the reflection list, the function succeeds but has no effect. Disabling reflection for a key does not affect reflection of any subkeys.

Raises an *auditing event* `winreg.DisableReflectionKey` with argument `key`.

winreg.**EnableReflectionKey**(*key*)

Restores registry reflection for the specified disabled key.

key is an already open key, or one of the predefined *HKEY_* constants*.

Will generally raise *NotImplementedError* if executed on a 32-bit operating system.

Restoring reflection for a key does not affect reflection of any subkeys.

Raises an *auditing event* `winreg.EnableReflectionKey` with argument `key`.

winreg.**QueryReflectionKey**(*key*)

Determines the reflection state for the specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

Returns `True` if reflection is disabled.

Will generally raise `NotImplementedError` if executed on a 32-bit operating system.

Raises an *auditing event* `winreg.QueryReflectionKey` with argument `key`.

33.2.2 Constants

The following constants are defined for use in many `winreg` functions.

HKEY_* Constants

winreg.**HKEY_CLASSES_ROOT**

Registry entries subordinate to this key define types (or classes) of documents and the properties associated with those types. Shell and COM applications use the information stored under this key.

winreg.**HKEY_CURRENT_USER**

Registry entries subordinate to this key define the preferences of the current user. These preferences include the settings of environment variables, data about program groups, colors, printers, network connections, and application preferences.

winreg.**HKEY_LOCAL_MACHINE**

Registry entries subordinate to this key define the physical state of the computer, including data about the bus type, system memory, and installed hardware and software.

winreg.**HKEY_USERS**

Registry entries subordinate to this key define the default user configuration for new users on the local computer and the user configuration for the current user.

winreg.**HKEY_PERFORMANCE_DATA**

Registry entries subordinate to this key allow you to access performance data. The data is not actually stored in the registry; the registry functions cause the system to collect the data from its source.

winreg.**HKEY_CURRENT_CONFIG**

Contains information about the current hardware profile of the local computer system.

winreg.**HKEY_DYN_DATA**

This key is not used in versions of Windows after 98.

Access Rights

For more information, see Registry Key Security and Access.

winreg.**KEY_ALL_ACCESS**

Combines the STANDARD_RIGHTS_REQUIRED, *KEY_QUERY_VALUE*, *KEY_SET_VALUE*, *KEY_CREATE_SUB_KEY*, *KEY_ENUMERATE_SUB_KEYS*, *KEY_NOTIFY*, and *KEY_CREATE_LINK* access rights.

winreg.**KEY_WRITE**

Combines the STANDARD_RIGHTS_WRITE, *KEY_SET_VALUE*, and *KEY_CREATE_SUB_KEY* access rights.

winreg.**KEY_READ**

Combines the STANDARD_RIGHTS_READ, *KEY_QUERY_VALUE*, *KEY_ENUMERATE_SUB_KEYS*, and *KEY_NOTIFY* values.

winreg.**KEY_EXECUTE**

Equivalent to *KEY_READ*.

winreg.**KEY_QUERY_VALUE**

Required to query the values of a registry key.

winreg.**KEY_SET_VALUE**

Required to create, delete, or set a registry value.

winreg.**KEY_CREATE_SUB_KEY**

Required to create a subkey of a registry key.

winreg.**KEY_ENUMERATE_SUB_KEYS**

Required to enumerate the subkeys of a registry key.

winreg.**KEY_NOTIFY**

Required to request change notifications for a registry key or for subkeys of a registry key.

winreg.**KEY_CREATE_LINK**

Reserved for system use.

64-bit Specific

For more information, see Accessing an Alternate Registry View.

winreg.**KEY_WOW64_64KEY**

Indicates that an application on 64-bit Windows should operate on the 64-bit registry view. On 32-bit Windows, this constant is ignored.

winreg.**KEY_WOW64_32KEY**

Indicates that an application on 64-bit Windows should operate on the 32-bit registry view. On 32-bit Windows, this constant is ignored.

Value Types

For more information, see Registry Value Types.

winreg.**REG_BINARY**

Binary data in any form.

winreg.**REG_DWORD**

32-bit number.

winreg.**REG_DWORD_LITTLE_ENDIAN**

A 32-bit number in little-endian format. Equivalent to *REG_DWORD*.

winreg.**REG_DWORD_BIG_ENDIAN**
: A 32-bit number in big-endian format.

winreg.**REG_EXPAND_SZ**
: Null-terminated string containing references to environment variables (`%PATH%`).

winreg.**REG_LINK**
: A Unicode symbolic link.

winreg.**REG_MULTI_SZ**
: A sequence of null-terminated strings, terminated by two null characters. (Python handles this termination automatically.)

winreg.**REG_NONE**
: No defined value type.

winreg.**REG_QWORD**
: A 64-bit number.

  Added in version 3.6.

winreg.**REG_QWORD_LITTLE_ENDIAN**
: A 64-bit number in little-endian format. Equivalent to *REG_QWORD*.

  Added in version 3.6.

winreg.**REG_RESOURCE_LIST**
: A device-driver resource list.

winreg.**REG_FULL_RESOURCE_DESCRIPTOR**
: A hardware setting.

winreg.**REG_RESOURCE_REQUIREMENTS_LIST**
: A hardware resource list.

winreg.**REG_SZ**
: A null-terminated string.

33.2.3 Registry Handle Objects

This object wraps a Windows HKEY object, automatically closing it when the object is destroyed. To guarantee cleanup, you can call either the *Close()* method on the object, or the *CloseKey()* function.

All registry functions in this module return one of these objects.

All registry functions in this module which accept a handle object also accept an integer, however, use of the handle object is encouraged.

Handle objects provide semantics for `__bool__()` – thus

```
if handle:
    print("Yes")
```

will print `Yes` if the handle is currently valid (has not been closed or detached).

The object also support comparison semantics, so handle objects will compare true if they both reference the same underlying Windows handle value.

Handle objects can be converted to an integer (e.g., using the built-in *int()* function), in which case the underlying Windows handle value is returned. You can also use the *Detach()* method to return the integer handle, and also disconnect the Windows handle from the handle object.

PyHKEY.**Close**()

Closes the underlying Windows handle.

If the handle is already closed, no error is raised.

PyHKEY.**Detach**()

Detaches the Windows handle from the handle object.

The result is an integer that holds the value of the handle before it is detached. If the handle is already detached or closed, this will return zero.

After calling this function, the handle is effectively invalidated, but the handle is not closed. You would call this function when you need the underlying Win32 handle to exist beyond the lifetime of the handle object.

Raises an *auditing event* `winreg.PyHKEY.Detach` with argument `key`.

PyHKEY.**__enter__**()

PyHKEY.**__exit__**(**exc_info*)

The HKEY object implements `__enter__()` and `__exit__()` and thus supports the context protocol for the `with` statement:

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ...  # work with key
```

will automatically close *key* when control leaves the `with` block.

33.3 winsound — Sound-playing interface for Windows

The *winsound* module provides access to the basic sound-playing machinery provided by Windows platforms. It includes functions and several constants.

winsound.**Beep**(*frequency*, *duration*)

Beep the PC's speaker. The *frequency* parameter specifies frequency, in hertz, of the sound, and must be in the range 37 through 32,767. The *duration* parameter specifies the number of milliseconds the sound should last. If the system is not able to beep the speaker, *RuntimeError* is raised.

winsound.**PlaySound**(*sound*, *flags*)

Call the underlying `PlaySound()` function from the Platform API. The *sound* parameter may be a filename, a system sound alias, audio data as a *bytes-like object*, or `None`. Its interpretation depends on the value of *flags*, which can be a bitwise ORed combination of the constants described below. If the *sound* parameter is `None`, any currently playing waveform sound is stopped. If the system indicates an error, *RuntimeError* is raised.

winsound.**MessageBeep**(*type=MB_OK*)

Call the underlying `MessageBeep()` function from the Platform API. This plays a sound as specified in the registry. The *type* argument specifies which sound to play; possible values are `-1`, `MB_ICONASTERISK`, `MB_ICONEXCLAMATION`, `MB_ICONHAND`, `MB_ICONQUESTION`, and `MB_OK`, all described below. The value `-1` produces a "simple beep"; this is the final fallback if a sound cannot be played otherwise. If the system indicates an error, *RuntimeError* is raised.

winsound.**SND_FILENAME**

The *sound* parameter is the name of a WAV file. Do not use with *SND_ALIAS*.

winsound.**SND_ALIAS**

The *sound* parameter is a sound association name from the registry. If the registry contains no such name, play the system default sound unless *SND_NODEFAULT* is also specified. If no default sound is registered, raise *RuntimeError*. Do not use with *SND_FILENAME*.

All Win32 systems support at least the following; most systems support many more:

PlaySound() *name*	Corresponding Control Panel Sound name
`'SystemAsterisk'`	Asterisk
`'SystemExclamation'`	Exclamation
`'SystemExit'`	Exit Windows
`'SystemHand'`	Critical Stop
`'SystemQuestion'`	Question

For example:

```
import winsound
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered (because
# "*" probably isn't the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

winsound.**SND_LOOP**

Play the sound repeatedly. The *SND_ASYNC* flag must also be used to avoid blocking. Cannot be used with *SND_MEMORY*.

winsound.**SND_MEMORY**

The *sound* parameter to *PlaySound()* is a memory image of a WAV file, as a *bytes-like object*.

> **Note**
>
> This module does not support playing from a memory image asynchronously, so a combination of this flag and *SND_ASYNC* will raise *RuntimeError*.

winsound.**SND_PURGE**

Stop playing all instances of the specified sound.

> **Note**
>
> This flag is not supported on modern Windows platforms.

winsound.**SND_ASYNC**

Return immediately, allowing sounds to play asynchronously.

winsound.**SND_NODEFAULT**

If the specified sound cannot be found, do not play the system default sound.

winsound.**SND_NOSTOP**
: Do not interrupt sounds currently playing.

winsound.**SND_NOWAIT**
: Return immediately if the sound driver is busy.

> **Note**
>
> This flag is not supported on modern Windows platforms.

winsound.**MB_ICONASTERISK**
: Play the `SystemDefault` sound.

winsound.**MB_ICONEXCLAMATION**
: Play the `SystemExclamation` sound.

winsound.**MB_ICONHAND**
: Play the `SystemHand` sound.

winsound.**MB_ICONQUESTION**
: Play the `SystemQuestion` sound.

winsound.**MB_OK**
: Play the `SystemDefault` sound.

CHAPTER

THIRTYFOUR

UNIX SPECIFIC SERVICES

The modules described in this chapter provide interfaces to features that are unique to the Unix operating system, or in some cases to some or many variants of it. Here's an overview:

34.1 `posix` — The most common POSIX system calls

This module provides access to operating system functionality that is standardized by the C Standard and the POSIX standard (a thinly disguised Unix interface).

Availability: Unix.

Do not import this module directly. Instead, import the module *os*, which provides a *portable* version of this interface. On Unix, the *os* module provides a superset of the *posix* interface. On non-Unix operating systems the *posix* module is not available, but a subset is always available through the *os* interface. Once *os* is imported, there is *no* performance penalty in using it instead of *posix*. In addition, *os* provides some additional functionality, such as automatically calling *putenv()* when an entry in `os.environ` is changed.

Errors are reported as exceptions; the usual exceptions are given for type errors, while errors reported by the system calls raise *OSError*.

34.1.1 Large File Support

Several operating systems (including AIX and Solaris) provide support for files that are larger than 2 GiB from a C programming model where `int` and `long` are 32-bit values. This is typically accomplished by defining the relevant size and offset types as 64-bit values. Such files are sometimes referred to as *large files*.

Large file support is enabled in Python when the size of an `off_t` is larger than a `long` and the `long long` is at least as large as an `off_t`. It may be necessary to configure and compile Python with certain compiler flags to enable this mode. For example, with Solaris 2.6 and 2.7 you need to do something like:

```
CFLAGS="`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS" \
        ./configure
```

On large-file-capable Linux systems, this might work:

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -O2 $CFLAGS" \
        ./configure
```

34.1.2 Notable Module Contents

In addition to many functions described in the *os* module documentation, *posix* defines the following data item:

posix.**environ**

A dictionary representing the string environment at the time the interpreter was started. Keys and values are bytes on Unix and str on Windows. For example, `environ[b'HOME']` (`environ['HOME']` on Windows) is the pathname of your home directory, equivalent to `getenv("HOME")` in C.

Modifying this dictionary does not affect the string environment passed on by *execv()*, *popen()* or *system()*; if you need to change the environment, pass `environ` to *execve()* or add variable assignments and export statements to the command string for *system()* or *popen()*.

Changed in version 3.2: On Unix, keys and values are bytes.

Note

The *os* module provides an alternate implementation of `environ` which updates the environment on modification. Note also that updating *os.environ* will render this dictionary obsolete. Use of the *os* module version of this is recommended over direct access to the *posix* module.

34.2 pwd — The password database

This module provides access to the Unix user account and password database. It is available on all Unix versions.

Availability: Unix, not WASI, not iOS.

Password database entries are reported as a tuple-like object, whose attributes correspond to the members of the `passwd` structure (Attribute field below, see `<pwd.h>`):

Index	Attribute	Meaning
0	`pw_name`	Login name
1	`pw_passwd`	Optional encrypted password
2	`pw_uid`	Numerical user ID
3	`pw_gid`	Numerical group ID
4	`pw_gecos`	User name or comment field
5	`pw_dir`	User home directory
6	`pw_shell`	User command interpreter

The uid and gid items are integers, all others are strings. *KeyError* is raised if the entry asked for cannot be found.

Note

In traditional Unix the field `pw_passwd` usually contains a password encrypted with a DES derived algorithm. However most modern unices use a so-called *shadow password* system. On those unices the *pw_passwd* field only contains an asterisk (`'*'`) or the letter `'x'` where the encrypted password is stored in a file `/etc/shadow` which is not world readable. Whether the *pw_passwd* field contains anything useful is system-dependent.

It defines the following items:

pwd.**getpwuid**(*uid*)

Return the password database entry for the given numeric user ID.

pwd.**getpwnam**(*name*)

Return the password database entry for the given user name.

pwd.**getpwall**()

Return a list of all available password database entries, in arbitrary order.

See also

Module *grp*
An interface to the group database, similar to this.

34.3 grp — The group database

This module provides access to the Unix group database. It is available on all Unix versions.

Availability: Unix, not WASI, not iOS.

Group database entries are reported as a tuple-like object, whose attributes correspond to the members of the `group` structure (Attribute field below, see `<grp.h>`):

Index	Attribute	Meaning
0	gr_name	the name of the group
1	gr_passwd	the (encrypted) group password; often empty
2	gr_gid	the numerical group ID
3	gr_mem	all the group member's user names

The gid is an integer, name and password are strings, and the member list is a list of strings. (Note that most users are not explicitly listed as members of the group they are in according to the password database. Check both databases to get complete membership information. Also note that a `gr_name` that starts with a + or − is likely to be a YP/NIS reference and may not be accessible via *getgrnam()* or *getgrgid()*.)

It defines the following items:

grp.**getgrgid**(*id*)

Return the group database entry for the given numeric group ID. *KeyError* is raised if the entry asked for cannot be found.

Changed in version 3.10: *TypeError* is raised for non-integer arguments like floats or strings.

grp.**getgrnam**(*name*)

Return the group database entry for the given group name. *KeyError* is raised if the entry asked for cannot be found.

grp.**getgrall**()

Return a list of all available group entries, in arbitrary order.

See also

Module *pwd*
An interface to the user database, similar to this.

34.4 termios — POSIX style tty control

This module provides an interface to the POSIX calls for tty I/O control. For a complete description of these calls, see *termios(3)* Unix manual page. It is only available for those Unix versions that support POSIX *termios* style tty I/O control configured during installation.

Availability: Unix.

All functions in this module take a file descriptor *fd* as their first argument. This can be an integer file descriptor, such as returned by `sys.stdin.fileno()`, or a *file object*, such as `sys.stdin` itself.

This module also defines all the constants needed to work with the functions provided here; these have the same name as their counterparts in C. Please refer to your system documentation for more information on using these terminal control interfaces.

The module defines the following functions:

`termios.`**`tcgetattr`**(*fd*)

Return a list containing the tty attributes for file descriptor *fd*, as follows: `[iflag, oflag, cflag, lflag, ispeed, ospeed, cc]` where *cc* is a list of the tty special characters (each a string of length 1, except the items with indices `VMIN` and `VTIME`, which are integers when these fields are defined). The interpretation of the flags and the speeds as well as the indexing in the *cc* array must be done using the symbolic constants defined in the *termios* module.

`termios.`**`tcsetattr`**(*fd*, *when*, *attributes*)

Set the tty attributes for file descriptor *fd* from the *attributes*, which is a list like the one returned by *tcgetattr()*. The *when* argument determines when the attributes are changed:

`termios.`**`TCSANOW`**

Change attributes immediately.

`termios.`**`TCSADRAIN`**

Change attributes after transmitting all queued output.

`termios.`**`TCSAFLUSH`**

Change attributes after transmitting all queued output and discarding all queued input.

`termios.`**`tcsendbreak`**(*fd*, *duration*)

Send a break on file descriptor *fd*. A zero *duration* sends a break for 0.25–0.5 seconds; a nonzero *duration* has a system dependent meaning.

`termios.`**`tcdrain`**(*fd*)

Wait until all output written to file descriptor *fd* has been transmitted.

`termios.`**`tcflush`**(*fd*, *queue*)

Discard queued data on file descriptor *fd*. The *queue* selector specifies which queue: `TCIFLUSH` for the input queue, `TCOFLUSH` for the output queue, or `TCIOFLUSH` for both queues.

termios.**tcflow**(*fd*, *action*)

Suspend or resume input or output on file descriptor *fd*. The *action* argument can be TCOOFF to suspend output, TCOON to restart output, TCIOFF to suspend input, or TCION to restart input.

termios.**tcgetwinsize**(*fd*)

Return a tuple (ws_row, ws_col) containing the tty window size for file descriptor *fd*. Requires termios.TIOCGWINSZ or termios.TIOCGSIZE.

Added in version 3.11.

termios.**tcsetwinsize**(*fd*, *winsize*)

Set the tty window size for file descriptor *fd* from *winsize*, which is a two-item tuple (ws_row, ws_col) like the one returned by *tcgetwinsize()*. Requires at least one of the pairs (termios.TIOCGWINSZ, termios.TIOCSWINSZ); (termios.TIOCGSIZE, termios.TIOCSSIZE) to be defined.

Added in version 3.11.

See also

Module *tty*
Convenience functions for common terminal control operations.

34.4.1 Example

Here's a function that prompts for a password with echoing turned off. Note the technique using a separate *tcgetattr()* call and a try … finally statement to ensure that the old tty attributes are restored exactly no matter what happens:

```
def getpass(prompt="Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~termios.ECHO          # lflags
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

34.5 tty — Terminal control functions

Source code: Lib/tty.py

The *tty* module defines functions for putting the tty into cbreak and raw modes.

Availability: Unix.

Because it requires the *termios* module, it will work only on Unix.

The *tty* module defines the following functions:

tty.**cfmakeraw**(*mode*)

Convert the tty attribute list *mode*, which is a list like the one returned by `termios.tcgetattr()`, to that of a tty in raw mode.

Added in version 3.12.

tty.**cfmakecbreak**(*mode*)

Convert the tty attribute list *mode*, which is a list like the one returned by `termios.tcgetattr()`, to that of a tty in cbreak mode.

This clears the `ECHO` and `ICANON` local mode flags in *mode* as well as setting the minimum input to 1 byte with no delay.

Added in version 3.12.

Changed in version 3.12.2: The `ICRNL` flag is no longer cleared. This matches Linux and macOS `stty cbreak` behavior and what `setcbreak()` historically did.

tty.**setraw**(*fd*, *when=termios.TCSAFLUSH*)

Change the mode of the file descriptor *fd* to raw. If *when* is omitted, it defaults to `termios.TCSAFLUSH`, and is passed to `termios.tcsetattr()`. The return value of `termios.tcgetattr()` is saved before setting *fd* to raw mode; this value is returned.

Changed in version 3.12: The return value is now the original tty attributes, instead of `None`.

tty.**setcbreak**(*fd*, *when=termios.TCSAFLUSH*)

Change the mode of file descriptor *fd* to cbreak. If *when* is omitted, it defaults to `termios.TCSAFLUSH`, and is passed to `termios.tcsetattr()`. The return value of `termios.tcgetattr()` is saved before setting *fd* to cbreak mode; this value is returned.

This clears the `ECHO` and `ICANON` local mode flags as well as setting the minimum input to 1 byte with no delay.

Changed in version 3.12: The return value is now the original tty attributes, instead of `None`.

Changed in version 3.12.2: The `ICRNL` flag is no longer cleared. This restores the behavior of Python 3.11 and earlier as well as matching what Linux, macOS, & BSDs describe in their `stty(1)` man pages regarding cbreak mode.

See also

Module ***termios***
Low-level terminal control interface.

34.6 `pty` — Pseudo-terminal utilities

Source code: Lib/pty.py

The `pty` module defines operations for handling the pseudo-terminal concept: starting another process and being able to write to and read from its controlling terminal programmatically.

Availability: Unix.

Pseudo-terminal handling is highly platform dependent. This code is mainly tested on Linux, FreeBSD, and macOS (it is supposed to work on other POSIX platforms but it's not been thoroughly tested).

The `pty` module defines the following functions:

pty.**fork**()

Fork. Connect the child's controlling terminal to a pseudo-terminal. Return value is `(pid, fd)`. Note that the child gets *pid* 0, and the *fd* is *invalid*. The parent's return value is the *pid* of the child, and *fd* is a file descriptor connected to the child's controlling terminal (and also to the child's standard input and output).

> **Warning**
>
> On macOS the use of this function is unsafe when mixed with using higher-level system APIs, and that includes using *urllib.request*.

pty.**openpty**()

Open a new pseudo-terminal pair, using *os.openpty()* if possible, or emulation code for generic Unix systems. Return a pair of file descriptors `(master, slave)`, for the master and the slave end, respectively.

pty.**spawn**(*argv*[, *master_read*[, *stdin_read*]])

Spawn a process, and connect its controlling terminal with the current process's standard io. This is often used to baffle programs which insist on reading from the controlling terminal. It is expected that the process spawned behind the pty will eventually terminate, and when it does *spawn* will return.

A loop copies STDIN of the current process to the child and data received from the child to STDOUT of the current process. It is not signaled to the child if STDIN of the current process closes down.

The functions *master_read* and *stdin_read* are passed a file descriptor which they should read from, and they should always return a byte string. In order to force spawn to return before the child process exits an empty byte array should be returned to signal end of file.

The default implementation for both functions will read and return up to 1024 bytes each time the function is called. The *master_read* callback is passed the pseudoterminal's master file descriptor to read output from the child process, and *stdin_read* is passed file descriptor 0, to read from the parent process's standard input.

Returning an empty byte string from either callback is interpreted as an end-of-file (EOF) condition, and that callback will not be called after that. If *stdin_read* signals EOF the controlling terminal can no longer communicate with the parent process OR the child process. Unless the child process will quit without any input, *spawn* will then loop forever. If *master_read* signals EOF the same behavior results (on linux at least).

Return the exit status value from *os.waitpid()* on the child process.

os.waitstatus_to_exitcode() can be used to convert the exit status into an exit code.

Raises an *auditing event* `pty.spawn` with argument `argv`.

Changed in version 3.4: *spawn()* now returns the status value from *os.waitpid()* on the child process.

34.6.1 Example

The following program acts like the Unix command *script(1)*, using a pseudo-terminal to record all input and output of a terminal session in a "typescript".

```
import argparse
import os
import pty
import sys
import time

parser = argparse.ArgumentParser()
parser.add_argument('-a', dest='append', action='store_true')
```

(continues on next page)

(continued from previous page)

```
parser.add_argument('-p', dest='use_python', action='store_true')
parser.add_argument('filename', nargs='?', default='typescript')
options = parser.parse_args()

shell = sys.executable if options.use_python else os.environ.get('SHELL', 'sh')
filename = options.filename
mode = 'ab' if options.append else 'wb'

with open(filename, mode) as script:
    def read(fd):
        data = os.read(fd, 1024)
        script.write(data)
        return data

    print('Script started, file is', filename)
    script.write(('Script started on %s\n' % time.asctime()).encode())

    pty.spawn(shell, read)

    script.write(('Script done on %s\n' % time.asctime()).encode())
    print('Script done, file is', filename)
```

34.7 `fcntl` — The `fcntl` and `ioctl` system calls

This module performs file and I/O control on file descriptors. It is an interface to the `fcntl()` and `ioctl()` Unix routines. See the *fcntl(2)* and *ioctl(2)* Unix manual pages for full details.

Availability: Unix, not WASI.

All functions in this module take a file descriptor *fd* as their first argument. This can be an integer file descriptor, such as returned by `sys.stdin.fileno()`, or an *io.IOBase* object, such as `sys.stdin` itself, which provides a *fileno()* that returns a genuine file descriptor.

Changed in version 3.3: Operations in this module used to raise an *IOError* where they now raise an *OSError*.

Changed in version 3.8: The `fcntl` module now contains `F_ADD_SEALS`, `F_GET_SEALS`, and `F_SEAL_*` constants for sealing of *os.memfd_create()* file descriptors.

Changed in version 3.9: On macOS, the `fcntl` module exposes the `F_GETPATH` constant, which obtains the path of a file from a file descriptor. On Linux(>=3.15), the `fcntl` module exposes the `F_OFD_GETLK`, `F_OFD_SETLK` and `F_OFD_SETLKW` constants, which are used when working with open file description locks.

Changed in version 3.10: On Linux >= 2.6.11, the `fcntl` module exposes the `F_GETPIPE_SZ` and `F_SETPIPE_SZ` constants, which allow to check and modify a pipe's size respectively.

Changed in version 3.11: On FreeBSD, the `fcntl` module exposes the `F_DUP2FD` and `F_DUP2FD_CLOEXEC` constants, which allow to duplicate a file descriptor, the latter setting `FD_CLOEXEC` flag in addition.

Changed in version 3.12: On Linux >= 4.5, the *fcntl* module exposes the `FICLONE` and `FICLONERANGE` constants, which allow to share some data of one file with another file by reflinking on some filesystems (e.g., btrfs, OCFS2, and XFS). This behavior is commonly referred to as "copy-on-write".

Changed in version 3.13: On Linux >= 2.6.32, the `fcntl` module exposes the `F_GETOWN_EX`, `F_SETOWN_EX`, `F_OWNER_TID`, `F_OWNER_PID`, `F_OWNER_PGRP` constants, which allow to direct I/O availability signals to a specific thread, process, or process group. On Linux >= 4.13, the `fcntl` module exposes the `F_GET_RW_HINT`,

F_SET_RW_HINT, F_GET_FILE_RW_HINT, F_SET_FILE_RW_HINT, and RWH_WRITE_LIFE_* constants, which allow to inform the kernel about the relative expected lifetime of writes on a given inode or via a particular open file description. On Linux >= 5.1 and NetBSD, the fcntl module exposes the F_SEAL_FUTURE_WRITE constant for use with F_ADD_SEALS and F_GET_SEALS operations. On FreeBSD, the fcntl module exposes the F_READAHEAD, F_ISUNIONSTACK, and F_KINFO constants. On macOS and FreeBSD, the fcntl module exposes the F_RDAHEAD constant. On NetBSD and AIX, the fcntl module exposes the F_CLOSEM constant. On NetBSD, the fcntl module exposes the F_MAXFD constant. On macOS and NetBSD, the fcntl module exposes the F_GETNOSIGPIPE and F_SETNOSIGPIPE constant.

The module defines the following functions:

fcntl.**fcntl**(*fd*, *cmd*, *arg=0*)

Perform the operation *cmd* on file descriptor *fd* (file objects providing a *fileno()* method are accepted as well). The values used for *cmd* are operating system dependent, and are available as constants in the *fcntl* module, using the same names as used in the relevant C header files. The argument *arg* can either be an integer value, or a *bytes* object. With an integer value, the return value of this function is the integer return value of the C fcntl() call. When the argument is bytes it represents a binary structure, e.g. created by *struct.pack()*. The binary data is copied to a buffer whose address is passed to the C fcntl() call. The return value after a successful call is the contents of the buffer, converted to a *bytes* object. The length of the returned object will be the same as the length of the *arg* argument. This is limited to 1024 bytes. If the information returned in the buffer by the operating system is larger than 1024 bytes, this is most likely to result in a segmentation violation or a more subtle data corruption.

If the fcntl() call fails, an *OSError* is raised.

Raises an *auditing event* fcntl.fcntl with arguments fd, cmd, arg.

fcntl.**ioctl**(*fd*, *request*, *arg=0*, *mutate_flag=True*)

This function is identical to the *fcntl()* function, except that the argument handling is even more complicated.

The *request* parameter is limited to values that can fit in 32-bits. Additional constants of interest for use as the *request* argument can be found in the *termios* module, under the same names as used in the relevant C header files.

The parameter *arg* can be one of an integer, an object supporting the read-only buffer interface (like *bytes*) or an object supporting the read-write buffer interface (like *bytearray*).

In all but the last case, behaviour is as for the *fcntl()* function.

If a mutable buffer is passed, then the behaviour is determined by the value of the *mutate_flag* parameter.

If it is false, the buffer's mutability is ignored and behaviour is as for a read-only buffer, except that the 1024 byte limit mentioned above is avoided – so long as the buffer you pass is at least as long as what the operating system wants to put there, things should work.

If *mutate_flag* is true (the default), then the buffer is (in effect) passed to the underlying *ioctl()* system call, the latter's return code is passed back to the calling Python, and the buffer's new contents reflect the action of the *ioctl()*. This is a slight simplification, because if the supplied buffer is less than 1024 bytes long it is first copied into a static buffer 1024 bytes long which is then passed to *ioctl()* and copied back into the supplied buffer.

If the ioctl() call fails, an *OSError* exception is raised.

An example:

```
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP, "  "))[0]
13341
>>> buf = array.array('h', [0])
```

(continues on next page)

(continued from previous page)

```
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, 1)
0
>>> buf
array('h', [13341])
```

Raises an *auditing event* `fcntl.ioctl` with arguments `fd`, `request`, `arg`.

`fcntl.`**`flock`**(*fd*, *operation*)

Perform the lock operation *operation* on file descriptor *fd* (file objects providing a *fileno()* method are accepted as well). See the Unix manual *flock(2)* for details. (On some systems, this function is emulated using `fcntl()`.)

If the `flock()` call fails, an *OSError* exception is raised.

Raises an *auditing event* `fcntl.flock` with arguments `fd`, `operation`.

`fcntl.`**`lockf`**(*fd*, *cmd*, *len=0*, *start=0*, *whence=0*)

This is essentially a wrapper around the *fcntl()* locking calls. *fd* is the file descriptor (file objects providing a *fileno()* method are accepted as well) of the file to lock or unlock, and *cmd* is one of the following values:

`fcntl.`**`LOCK_UN`**

Release an existing lock.

`fcntl.`**`LOCK_SH`**

Acquire a shared lock.

`fcntl.`**`LOCK_EX`**

Acquire an exclusive lock.

`fcntl.`**`LOCK_NB`**

Bitwise OR with any of the other three `LOCK_*` constants to make the request non-blocking.

If `LOCK_NB` is used and the lock cannot be acquired, an *OSError* will be raised and the exception will have an *errno* attribute set to *EACCES* or *EAGAIN* (depending on the operating system; for portability, check for both values). On at least some systems, `LOCK_EX` can only be used if the file descriptor refers to a file opened for writing.

len is the number of bytes to lock, *start* is the byte offset at which the lock starts, relative to *whence*, and *whence* is as with *io.IOBase.seek()*, specifically:

- `0` – relative to the start of the file (*os.SEEK_SET*)
- `1` – relative to the current buffer position (*os.SEEK_CUR*)
- `2` – relative to the end of the file (*os.SEEK_END*)

The default for *start* is 0, which means to start at the beginning of the file. The default for *len* is 0 which means to lock to the end of the file. The default for *whence* is also 0.

Raises an *auditing event* `fcntl.lockf` with arguments `fd`, `cmd`, `len`, `start`, `whence`.

Examples (all on a SVR4 compliant system):

```
import struct, fcntl, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

Note that in the first example the return value variable *rv* will hold an integer value; in the second example it will hold a `bytes` object. The structure lay-out for the *lockdata* variable is system dependent — therefore using the `flock()` call may be better.

See also

Module `os`
If the locking flags `O_SHLOCK` and `O_EXLOCK` are present in the `os` module (on BSD only), the `os.open()` function provides an alternative to the `lockf()` and `flock()` functions.

34.8 `resource` — Resource usage information

This module provides basic mechanisms for measuring and controlling system resources utilized by a program.

Availability: Unix, not WASI.

Symbolic constants are used to specify particular system resources and to request usage information about either the current process or its children.

An `OSError` is raised on syscall failure.

exception `resource.`**error**

A deprecated alias of `OSError`.

Changed in version 3.3: Following **PEP 3151**, this class was made an alias of `OSError`.

34.8.1 Resource Limits

Resources usage can be limited using the `setrlimit()` function described below. Each resource is controlled by a pair of limits: a soft limit and a hard limit. The soft limit is the current limit, and may be lowered or raised by a process over time. The soft limit can never exceed the hard limit. The hard limit can be lowered to any value greater than the soft limit, but not raised. (Only processes with the effective UID of the super-user can raise a hard limit.)

The specific resources that can be limited are system dependent. They are described in the `getrlimit(2)` man page. The resources listed below are supported when the underlying operating system supports them; resources which cannot be checked or controlled by the operating system are not defined in this module for those platforms.

`resource.`**RLIM_INFINITY**

Constant used to represent the limit for an unlimited resource.

`resource.`**getrlimit**(*resource*)

Returns a tuple `(soft, hard)` with the current soft and hard limits of *resource*. Raises `ValueError` if an invalid resource is specified, or `error` if the underlying system call fails unexpectedly.

`resource.`**setrlimit**(*resource*, *limits*)

Sets new limits of consumption of *resource*. The *limits* argument must be a tuple `(soft, hard)` of two integers describing the new limits. A value of `RLIM_INFINITY` can be used to request a limit that is unlimited.

Raises `ValueError` if an invalid resource is specified, if the new soft limit exceeds the hard limit, or if a process tries to raise its hard limit. Specifying a limit of `RLIM_INFINITY` when the hard or system limit for that resource is not unlimited will result in a `ValueError`. A process with the effective UID of super-user can request any valid

limit value, including unlimited, but *ValueError* will still be raised if the requested limit exceeds the system imposed limit.

`setrlimit` may also raise *error* if the underlying system call fails.

VxWorks only supports setting *RLIMIT_NOFILE*.

Raises an *auditing event* `resource.setrlimit` with arguments `resource`, `limits`.

`resource.`**`prlimit`**(*pid*, *resource*[, *limits*])

Combines *setrlimit()* and *getrlimit()* in one function and supports to get and set the resources limits of an arbitrary process. If *pid* is 0, then the call applies to the current process. *resource* and *limits* have the same meaning as in *setrlimit()*, except that *limits* is optional.

When *limits* is not given the function returns the *resource* limit of the process *pid*. When *limits* is given the *resource* limit of the process is set and the former resource limit is returned.

Raises *ProcessLookupError* when *pid* can't be found and *PermissionError* when the user doesn't have `CAP_SYS_RESOURCE` for the process.

Raises an *auditing event* `resource.prlimit` with arguments `pid`, `resource`, `limits`.

Availability: Linux >= 2.6.36 with glibc >= 2.13.

Added in version 3.4.

These symbols define resources whose consumption can be controlled using the *setrlimit()* and *getrlimit()* functions described below. The values of these symbols are exactly the constants used by C programs.

The Unix man page for *getrlimit(2)* lists the available resources. Note that not all systems use the same symbol or same value to denote the same resource. This module does not attempt to mask platform differences — symbols not defined for a platform will not be available from this module on that platform.

`resource.`**`RLIMIT_CORE`**

The maximum size (in bytes) of a core file that the current process can create. This may result in the creation of a partial core file if a larger core would be required to contain the entire process image.

`resource.`**`RLIMIT_CPU`**

The maximum amount of processor time (in seconds) that a process can use. If this limit is exceeded, a `SIGXCPU` signal is sent to the process. (See the *signal* module documentation for information about how to catch this signal and do something useful, e.g. flush open files to disk.)

`resource.`**`RLIMIT_FSIZE`**

The maximum size of a file which the process may create.

`resource.`**`RLIMIT_DATA`**

The maximum size (in bytes) of the process's heap.

`resource.`**`RLIMIT_STACK`**

The maximum size (in bytes) of the call stack for the current process. This only affects the stack of the main thread in a multi-threaded process.

`resource.`**`RLIMIT_RSS`**

The maximum resident set size that should be made available to the process.

`resource.`**`RLIMIT_NPROC`**

The maximum number of processes the current process may create.

`resource.`**`RLIMIT_NOFILE`**

The maximum number of open file descriptors for the current process.

resource.**RLIMIT_OFILE**

The BSD name for *RLIMIT_NOFILE*.

resource.**RLIMIT_MEMLOCK**

The maximum address space which may be locked in memory.

resource.**RLIMIT_VMEM**

The largest area of mapped memory which the process may occupy.

Availability: FreeBSD >= 11.

resource.**RLIMIT_AS**

The maximum area (in bytes) of address space which may be taken by the process.

resource.**RLIMIT_MSGQUEUE**

The number of bytes that can be allocated for POSIX message queues.

Availability: Linux >= 2.6.8.

Added in version 3.4.

resource.**RLIMIT_NICE**

The ceiling for the process's nice level (calculated as 20 - rlim_cur).

Availability: Linux >= 2.6.12.

Added in version 3.4.

resource.**RLIMIT_RTPRIO**

The ceiling of the real-time priority.

Availability: Linux >= 2.6.12.

Added in version 3.4.

resource.**RLIMIT_RTTIME**

The time limit (in microseconds) on CPU time that a process can spend under real-time scheduling without making a blocking syscall.

Availability: Linux >= 2.6.25.

Added in version 3.4.

resource.**RLIMIT_SIGPENDING**

The number of signals which the process may queue.

Availability: Linux >= 2.6.8.

Added in version 3.4.

resource.**RLIMIT_SBSIZE**

The maximum size (in bytes) of socket buffer usage for this user. This limits the amount of network memory, and hence the amount of mbufs, that this user may hold at any time.

Availability: FreeBSD.

Added in version 3.4.

resource.**RLIMIT_SWAP**

The maximum size (in bytes) of the swap space that may be reserved or used by all of this user id's processes. This limit is enforced only if bit 1 of the vm.overcommit sysctl is set. Please see tuning(7) for a complete description of this sysctl.

Availability: FreeBSD.

Added in version 3.4.

resource.**RLIMIT_NPTS**

The maximum number of pseudo-terminals created by this user id.

Availability: FreeBSD.

Added in version 3.4.

resource.**RLIMIT_KQUEUES**

The maximum number of kqueues this user id is allowed to create.

Availability: FreeBSD >= 11.

Added in version 3.10.

34.8.2 Resource Usage

These functions are used to retrieve resource usage information:

resource.**getrusage**(*who*)

This function returns an object that describes the resources consumed by either the current process or its children, as specified by the *who* parameter. The *who* parameter should be specified using one of the RUSAGE_* constants described below.

A simple example:

```
from resource import *
import time

# a non CPU-bound task
time.sleep(3)
print(getrusage(RUSAGE_SELF))

# a CPU-bound task
for i in range(10 ** 8):
    _ = 1 + 1
print(getrusage(RUSAGE_SELF))
```

The fields of the return value each describe how a particular system resource has been used, e.g. amount of time spent running is user mode or number of times the process was swapped out of main memory. Some values are dependent on the clock tick internal, e.g. the amount of memory the process is using.

For backward compatibility, the return value is also accessible as a tuple of 16 elements.

The fields ru_utime and ru_stime of the return value are floating-point values representing the amount of time spent executing in user mode and the amount of time spent executing in system mode, respectively. The remaining values are integers. Consult the *getrusage(2)* man page for detailed information about these values. A brief summary is presented here:

Index	Field	Resource
0	ru_utime	time in user mode (float seconds)
1	ru_stime	time in system mode (float seconds)
2	ru_maxrss	maximum resident set size
3	ru_ixrss	shared memory size
4	ru_idrss	unshared memory size
5	ru_isrss	unshared stack size
6	ru_minflt	page faults not requiring I/O
7	ru_majflt	page faults requiring I/O
8	ru_nswap	number of swap outs
9	ru_inblock	block input operations
10	ru_oublock	block output operations
11	ru_msgsnd	messages sent
12	ru_msgrcv	messages received
13	ru_nsignals	signals received
14	ru_nvcsw	voluntary context switches
15	ru_nivcsw	involuntary context switches

This function will raise a *ValueError* if an invalid *who* parameter is specified. It may also raise *error* exception in unusual circumstances.

resource.**getpagesize**()

Returns the number of bytes in a system page. (This need not be the same as the hardware page size.)

The following RUSAGE_* symbols are passed to the *getrusage()* function to specify which processes information should be provided for.

resource.**RUSAGE_SELF**

Pass to *getrusage()* to request resources consumed by the calling process, which is the sum of resources used by all threads in the process.

resource.**RUSAGE_CHILDREN**

Pass to *getrusage()* to request resources consumed by child processes of the calling process which have been terminated and waited for.

resource.**RUSAGE_BOTH**

Pass to *getrusage()* to request resources consumed by both the current process and child processes. May not be available on all systems.

resource.**RUSAGE_THREAD**

Pass to *getrusage()* to request resources consumed by the current thread. May not be available on all systems.

Added in version 3.2.

34.9 syslog — Unix syslog library routines

This module provides an interface to the Unix syslog library routines. Refer to the Unix manual pages for a detailed description of the syslog facility.

Availability: Unix, not WASI, not iOS.

This module wraps the system `syslog` family of routines. A pure Python library that can speak to a syslog server is available in the *`logging.handlers`* module as *`SysLogHandler`*.

The module defines the following functions:

`syslog.`**`syslog`**(*message*)

`syslog.`**`syslog`**(*priority*, *message*)

Send the string *message* to the system logger. A trailing newline is added if necessary. Each message is tagged with a priority composed of a *facility* and a *level*. The optional *priority* argument, which defaults to *`LOG_INFO`*, determines the message priority. If the facility is not encoded in *priority* using logical-or (`LOG_INFO | LOG_USER`), the value given in the *`openlog()`* call is used.

If *`openlog()`* has not been called prior to the call to *`syslog()`*, *`openlog()`* will be called with no arguments.

Raises an *auditing event* `syslog.syslog` with arguments `priority`, `message`.

Changed in version 3.2: In previous versions, *`openlog()`* would not be called automatically if it wasn't called prior to the call to *`syslog()`*, deferring to the syslog implementation to call `openlog()`.

Changed in version 3.12: This function is restricted in subinterpreters. (Only code that runs in multiple interpreters is affected and the restriction is not relevant for most users.) *`openlog()`* must be called in the main interpreter before *`syslog()`* may be used in a subinterpreter. Otherwise it will raise *`RuntimeError`*.

`syslog.`**`openlog`**([*ident*[, *logoption*[, *facility*]]])

Logging options of subsequent *`syslog()`* calls can be set by calling *`openlog()`*. *`syslog()`* will call *`openlog()`* with no arguments if the log is not currently open.

The optional *ident* keyword argument is a string which is prepended to every message, and defaults to `sys.argv[0]` with leading path components stripped. The optional *logoption* keyword argument (default is 0) is a bit field – see below for possible values to combine. The optional *facility* keyword argument (default is *`LOG_USER`*) sets the default facility for messages which do not have a facility explicitly encoded.

Raises an *auditing event* `syslog.openlog` with arguments `ident`, `logoption`, `facility`.

Changed in version 3.2: In previous versions, keyword arguments were not allowed, and *ident* was required.

Changed in version 3.12: This function is restricted in subinterpreters. (Only code that runs in multiple interpreters is affected and the restriction is not relevant for most users.) This may only be called in the main interpreter. It will raise *`RuntimeError`* if called in a subinterpreter.

`syslog.`**`closelog`**()

Reset the syslog module values and call the system library `closelog()`.

This causes the module to behave as it does when initially imported. For example, *`openlog()`* will be called on the first *`syslog()`* call (if *`openlog()`* hasn't already been called), and *ident* and other *`openlog()`* parameters are reset to defaults.

Raises an *auditing event* `syslog.closelog` with no arguments.

Changed in version 3.12: This function is restricted in subinterpreters. (Only code that runs in multiple interpreters is affected and the restriction is not relevant for most users.) This may only be called in the main interpreter. It will raise *`RuntimeError`* if called in a subinterpreter.

`syslog.`**`setlogmask`**(*maskpri*)

Set the priority mask to *maskpri* and return the previous mask value. Calls to *`syslog()`* with a priority level not set in *maskpri* are ignored. The default is to log all priorities. The function `LOG_MASK(pri)` calculates the mask for the individual priority *pri*. The function `LOG_UPTO(pri)` calculates the mask for all priorities up to and including *pri*.

Raises an *auditing event* `syslog.setlogmask` with argument `maskpri`.

The module defines the following constants:

syslog.**LOG_EMERG**
syslog.**LOG_ALERT**
syslog.**LOG_CRIT**
syslog.**LOG_ERR**
syslog.**LOG_WARNING**
syslog.**LOG_NOTICE**
syslog.**LOG_INFO**
syslog.**LOG_DEBUG**

Priority levels (high to low).

syslog.**LOG_AUTH**
syslog.**LOG_AUTHPRIV**
syslog.**LOG_CRON**
syslog.**LOG_DAEMON**
syslog.**LOG_FTP**
syslog.**LOG_INSTALL**
syslog.**LOG_KERN**
syslog.**LOG_LAUNCHD**
syslog.**LOG_LPR**
syslog.**LOG_MAIL**
syslog.**LOG_NETINFO**
syslog.**LOG_NEWS**
syslog.**LOG_RAS**
syslog.**LOG_REMOTEAUTH**
syslog.**LOG_SYSLOG**
syslog.**LOG_USER**
syslog.**LOG_UUCP**
syslog.**LOG_LOCAL0**
syslog.**LOG_LOCAL1**
syslog.**LOG_LOCAL2**
syslog.**LOG_LOCAL3**
syslog.**LOG_LOCAL4**
syslog.**LOG_LOCAL5**
syslog.**LOG_LOCAL6**
syslog.**LOG_LOCAL7**

Facilities, depending on availability in <syslog.h> for *LOG_AUTHPRIV*, *LOG_FTP*, *LOG_NETINFO*, *LOG_REMOTEAUTH*, *LOG_INSTALL* and *LOG_RAS*.

Changed in version 3.13: Added *LOG_FTP*, *LOG_NETINFO*, *LOG_REMOTEAUTH*, *LOG_INSTALL*, *LOG_RAS*, and *LOG_LAUNCHD*.

syslog.**LOG_PID**
syslog.**LOG_CONS**
syslog.**LOG_NDELAY**
syslog.**LOG_ODELAY**
syslog.**LOG_NOWAIT**
syslog.**LOG_PERROR**

Log options, depending on availability in <syslog.h> for *LOG_ODELAY*, *LOG_NOWAIT* and *LOG_PERROR*.

34.9.1 Examples

Simple example

A simple set of examples:

```
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

An example of setting some log options, these would include the process ID in logged messages, and write the messages to the destination facility used for mail logging:

```
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```

CHAPTER

THIRTYFIVE

MODULES COMMAND-LINE INTERFACE (CLI)

The following modules have a command-line interface.

- *ast*
- *asyncio*
- `base64`
- *calendar*
- `code`
- *compileall*
- `cProfile`: see *profile*
- *difflib*
- *dis*
- `doctest`
- `encodings.rot_13`
- `ensurepip`
- `filecmp`
- `fileinput`
- `ftplib`
- *gzip*
- *http.server*
- `idlelib`
- *inspect*
- *json.tool*
- `mimetypes`
- `pdb`
- `pickle`
- *pickletools*
- `platform`
- `poplib`

- *profile*
- `pstats`
- *py_compile*
- `pyclbr`
- `pydoc`
- `quopri`
- *random*
- `runpy`
- *site*
- *sqlite3*
- *sysconfig*
- `tabnanny`
- *tarfile*
- `this`
- *timeit*
- *tokenize*
- *trace*
- `turtledemo`
- *unittest*
- *uuid*
- `venv`
- `webbrowser`
- *zipapp*
- *zipfile*

See also the Python command-line interface.

CHAPTER

THIRTYSIX

SUPERSEDED MODULES

The modules described in this chapter are deprecated or *soft deprecated* and only kept for backwards compatibility. They have been superseded by other modules.

36.1 `getopt` — C-style parser for command line options

Source code: Lib/getopt.py

Deprecated since version 3.13: The `getopt` module is *soft deprecated* and will not be developed further; development will continue with the `argparse` module.

> **Note**
>
> The `getopt` module is a parser for command line options whose API is designed to be familiar to users of the C `getopt()` function. Users who are unfamiliar with the C `getopt()` function or who would like to write less code and get better help and error messages should consider using the `argparse` module instead.

This module helps scripts to parse the command line arguments in `sys.argv`. It supports the same conventions as the Unix `getopt()` function (including the special meanings of arguments of the form '-' and '--'). Long options similar to those supported by GNU software may be used as well via an optional third argument.

This module provides two functions and an exception:

`getopt.`**`getopt`**(*args*, *shortopts*, *longopts=[]*)

Parses command line options and parameter list. *args* is the argument list to be parsed, without the leading reference to the running program. Typically, this means `sys.argv[1:]`. *shortopts* is the string of option letters that the script wants to recognize, with options that require an argument followed by a colon (`':'`; i.e., the same format that Unix `getopt()` uses).

> **Note**
>
> Unlike GNU `getopt()`, after a non-option argument, all further arguments are considered also non-options. This is similar to the way non-GNU Unix systems work.

longopts, if specified, must be a list of strings with the names of the long options which should be supported. The leading `'--'` characters should not be included in the option name. Long options which require an argument should be followed by an equal sign (`'='`). Optional arguments are not supported. To accept only long options, *shortopts* should be an empty string. Long options on the command line can be recognized so long as they provide

a prefix of the option name that matches exactly one of the accepted options. For example, if *longopts* is `['foo', 'frob']`, the option `--fo` will match as `--foo`, but `--f` will not match uniquely, so *GetoptError* will be raised.

The return value consists of two elements: the first is a list of `(option, value)` pairs; the second is the list of program arguments left after the option list was stripped (this is a trailing slice of *args*). Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., `'-x'`) or two hyphens for long options (e.g., `'--long-option'`), and the option argument as its second element, or an empty string if the option has no argument. The options occur in the list in the same order in which they were found, thus allowing multiple occurrences. Long and short options may be mixed.

getopt.**gnu_getopt**(*args*, *shortopts*, *longopts=[]*)

This function works like *getopt()*, except that GNU style scanning mode is used by default. This means that option and non-option arguments may be intermixed. The *getopt()* function stops processing options as soon as a non-option argument is encountered.

If the first character of the option string is `'+'`, or if the environment variable `POSIXLY_CORRECT` is set, then option processing stops as soon as a non-option argument is encountered.

exception getopt.**GetoptError**

This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none. The argument to the exception is a string indicating the cause of the error. For long options, an argument given to an option which does not require one will also cause this exception to be raised. The attributes `msg` and `opt` give the error message and related option; if there is no specific option to which the exception relates, `opt` is an empty string.

exception getopt.**error**

Alias for *GetoptError*; for backward compatibility.

An example using only Unix style options:

```
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

Using long option names is equally easy:

```
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc.def'), ('-x', '')]
>>> args
['a1', 'a2']
```

In a script, typical usage is something like this:

```
import getopt, sys
```

(continues on next page)

(continued from previous page)

```
def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
    except getopt.GetoptError as err:
        # print help information and exit:
        print(err)  # will print something like "option -a not recognized"
        usage()
        sys.exit(2)
    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        elif o in ("-h", "--help"):
            usage()
            sys.exit()
        elif o in ("-o", "--output"):
            output = a
        else:
            assert False, "unhandled option"
    # ...

if __name__ == "__main__":
    main()
```

Note that an equivalent command line interface could be produced with less code and more informative help and error messages by using the *argparse* module:

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    args = parser.parse_args()
    # ... do something with args.output ...
    # ... do something with args.verbose ..
```

See also

Module *argparse*
Alternative command line option and argument parsing library.

36.2 optparse — Parser for command line options

Source code: Lib/optparse.py

Deprecated since version 3.2: The *optparse* module is *soft deprecated* and will not be developed further; development will continue with the *argparse* module.

---

optparse is a more convenient, flexible, and powerful library for parsing command-line options than the old *getopt* module. *optparse* uses a more declarative style of command-line parsing: you create an instance of *OptionParser*,

populate it with options, and parse the command line. *optparse* allows users to specify options in the conventional GNU/POSIX syntax, and additionally generates usage and help messages for you.

Here's an example of using *optparse* in a simple script:

```
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

With these few lines of code, users of your script can now do the "usual thing" on the command-line, for example:

```
<yourscript> --file=outfile -q
```

As it parses the command line, *optparse* sets attributes of the `options` object returned by *parse_args()* based on user-supplied command-line values. When *parse_args()* returns from parsing this command line, `options.filename` will be `"outfile"` and `options.verbose` will be `False`. *optparse* supports both long and short options, allows short options to be merged together, and allows options to be associated with their arguments in a variety of ways. Thus, the following command lines are all equivalent to the above example:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

Additionally, users can run one of the following

```
<yourscript> -h
<yourscript> --help
```

and *optparse* will print out a brief summary of your script's options:

```
Usage: <yourscript> [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet           don't print status messages to stdout
```

where the value of *yourscript* is determined at runtime (normally from `sys.argv[0]`).

36.2.1 Background

optparse was explicitly designed to encourage the creation of programs with straightforward, conventional command-line interfaces. To that end, it supports only the most common command-line syntax and semantics conventionally used under Unix. If you are unfamiliar with these conventions, read this section to acquaint yourself with them.

Terminology

argument
a string entered on the command-line, and passed by the shell to `execl()` or `execv()`. In Python, arguments are elements of `sys.argv[1:]` (`sys.argv[0]` is the name of the program being executed). Unix shells also use the term "word".

It is occasionally desirable to substitute an argument list other than `sys.argv[1:]`, so you should read "argument" as "an element of `sys.argv[1:]`, or of some other list provided as a substitute for `sys.argv[1:]`".

option
an argument used to supply extra information to guide or customize the execution of a program. There are many different syntaxes for options; the traditional Unix syntax is a hyphen ("-") followed by a single letter, e.g. `-x` or `-F`. Also, traditional Unix syntax allows multiple options to be merged into a single argument, e.g. `-x -F` is equivalent to `-xF`. The GNU project introduced `--` followed by a series of hyphen-separated words, e.g. `--file` or `--dry-run`. These are the only two option syntaxes provided by *optparse*.

Some other option syntaxes that the world has seen include:

- a hyphen followed by a few letters, e.g. `-pf` (this is *not* the same as multiple options merged into a single argument)
- a hyphen followed by a whole word, e.g. `-file` (this is technically equivalent to the previous syntax, but they aren't usually seen in the same program)
- a plus sign followed by a single letter, or a few letters, or a word, e.g. `+f`, `+rgb`
- a slash followed by a letter, or a few letters, or a word, e.g. `/f`, `/file`

These option syntaxes are not supported by *optparse*, and they never will be. This is deliberate: the first three are non-standard on any environment, and the last only makes sense if you're exclusively targeting Windows or certain legacy platforms (e.g. VMS, MS-DOS).

option argument
an argument that follows an option, is closely associated with that option, and is consumed from the argument list when that option is. With *optparse*, option arguments may either be in a separate argument from their option:

```
-f foo
--file foo
```

or included in the same argument:

```
-ffoo
--file=foo
```

Typically, a given option either takes an argument or it doesn't. Lots of people want an "optional option arguments" feature, meaning that some options will take an argument if they see it, and won't if they don't. This is somewhat controversial, because it makes parsing ambiguous: if `-a` takes an optional argument and `-b` is another option entirely, how do we interpret `-ab`? Because of this ambiguity, *optparse* does not support this feature.

positional argument
something leftover in the argument list after options have been parsed, i.e. after options and their arguments have been parsed and removed from the argument list.

required option
: an option that must be supplied on the command-line; note that the phrase "required option" is self-contradictory in English. *optparse* doesn't prevent you from implementing required options, but doesn't give you much help at it either.

For example, consider this hypothetical command-line:

```
prog -v --report report.txt foo bar
```

`-v` and `--report` are both options. Assuming that `--report` takes one argument, `report.txt` is an option argument. `foo` and `bar` are positional arguments.

What are options for?

Options are used to provide extra information to tune or customize the execution of a program. In case it wasn't clear, options are usually *optional*. A program should be able to run just fine with no options whatsoever. (Pick a random program from the Unix or GNU toolsets. Can it run without any options at all and still make sense? The main exceptions are `find`, `tar`, and `dd`—all of which are mutant oddballs that have been rightly criticized for their non-standard syntax and confusing interfaces.)

Lots of people want their programs to have "required options". Think about it. If it's required, then it's *not optional*! If there is a piece of information that your program absolutely requires in order to run successfully, that's what positional arguments are for.

As an example of good command-line interface design, consider the humble `cp` utility, for copying files. It doesn't make much sense to try to copy files without supplying a destination and at least one source. Hence, `cp` fails if you run it with no arguments. However, it has a flexible, useful syntax that does not require any options at all:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

You can get pretty far with just that. Most `cp` implementations provide a bunch of options to tweak exactly how the files are copied: you can preserve mode and modification time, avoid following symlinks, ask before clobbering existing files, etc. But none of this distracts from the core mission of `cp`, which is to copy either one file to another, or several files to another directory.

What are positional arguments for?

Positional arguments are for those pieces of information that your program absolutely, positively requires to run.

A good user interface should have as few absolute requirements as possible. If your program requires 17 distinct pieces of information in order to run successfully, it doesn't much matter *how* you get that information from the user—most people will give up and walk away before they successfully run the program. This applies whether the user interface is a command-line, a configuration file, or a GUI: if you make that many demands on your users, most of them will simply give up.

In short, try to minimize the amount of information that users are absolutely required to supply—use sensible defaults whenever possible. Of course, you also want to make your programs reasonably flexible. That's what options are for. Again, it doesn't matter if they are entries in a config file, widgets in the "Preferences" dialog of a GUI, or command-line options—the more options you implement, the more flexible your program is, and the more complicated its implementation becomes. Too much flexibility has drawbacks as well, of course; too many options can overwhelm users and make your code much harder to maintain.

36.2.2 Tutorial

While *optparse* is quite flexible and powerful, it's also straightforward to use in most cases. This section covers the code patterns that are common to any *optparse*-based program.

First, you need to import the OptionParser class; then, early in the main program, create an OptionParser instance:

```
from optparse import OptionParser
...
parser = OptionParser()
```

Then you can start defining options. The basic syntax is:

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

Each option has one or more option strings, such as `-f` or `--file`, and several option attributes that tell *optparse* what to expect and what to do when it encounters that option on the command line.

Typically, each option will have one short option string and one long option string, e.g.:

```
parser.add_option("-f", "--file", ...)
```

You're free to define as many short option strings and as many long option strings as you like (including zero), as long as there is at least one option string overall.

The option strings passed to *OptionParser.add_option()* are effectively labels for the option defined by that call. For brevity, we will frequently refer to *encountering an option* on the command line; in reality, *optparse* encounters *option strings* and looks up options from them.

Once all of your options are defined, instruct *optparse* to parse your program's command line:

```
(options, args) = parser.parse_args()
```

(If you like, you can pass a custom argument list to *parse_args()*, but that's rarely necessary: by default it uses `sys.argv[1:]`.)

parse_args() returns two values:

- `options`, an object containing values for all of your options—e.g. if `--file` takes a single string argument, then `options.file` will be the filename supplied by the user, or `None` if the user did not supply that option
- `args`, the list of positional arguments leftover after parsing options

This tutorial section only covers the four most important option attributes: *action*, *type*, *dest* (destination), and *help*. Of these, *action* is the most fundamental.

Understanding option actions

Actions tell *optparse* what to do when it encounters an option on the command line. There is a fixed set of actions hard-coded into *optparse*; adding new actions is an advanced topic covered in section *Extending optparse*. Most actions tell *optparse* to store a value in some variable—for example, take a string from the command line and store it in an attribute of `options`.

If you don't specify an option action, *optparse* defaults to `store`.

The store action

The most common option action is `store`, which tells *optparse* to take the next argument (or the remainder of the current argument), ensure that it is of the correct type, and store it to your chosen destination.

For example:

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

Now let's make up a fake command line and ask *optparse* to parse it:

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

When *optparse* sees the option string `-f`, it consumes the next argument, `foo.txt`, and stores it in `options.filename`. So, after this call to *parse_args()*, `options.filename` is `"foo.txt"`.

Some other option types supported by *optparse* are `int` and `float`. Here's an option that expects an integer argument:

```
parser.add_option("-n", type="int", dest="num")
```

Note that this option has no long option string, which is perfectly acceptable. Also, there's no explicit action, since the default is `store`.

Let's parse another fake command-line. This time, we'll jam the option argument right up against the option: since `-n42` (one argument) is equivalent to `-n 42` (two arguments), the code

```
(options, args) = parser.parse_args(["-n42"])
print(options.num)
```

will print `42`.

If you don't specify a type, *optparse* assumes `string`. Combined with the fact that the default action is `store`, that means our first example can be a lot shorter:

```
parser.add_option("-f", "--file", dest="filename")
```

If you don't supply a destination, *optparse* figures out a sensible default from the option strings: if the first long option string is `--foo-bar`, then the default destination is `foo_bar`. If there are no long option strings, *optparse* looks at the first short option string: the default destination for `-f` is `f`.

optparse also includes the built-in `complex` type. Adding types is covered in section *Extending optparse*.

Handling boolean (flag) options

Flag options—set a variable to true or false when a particular option is seen—are quite common. *optparse* supports them with two separate actions, `store_true` and `store_false`. For example, you might have a `verbose` flag that is turned on with `-v` and off with `-q`:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

Here we have two different options with the same destination, which is perfectly OK. (It just means you have to be a bit careful when setting default values—see below.)

When *optparse* encounters `-v` on the command line, it sets `options.verbose` to `True`; when it encounters `-q`, `options.verbose` is set to `False`.

Other actions

Some other actions supported by *optparse* are:

"store_const"
store a constant value, pre-set via *Option.const*

"append"
append this option's argument to a list

"count"
increment a counter by one

"callback"
call a specified function

These are covered in section *Reference Guide*, and section *Option Callbacks*.

Default values

All of the above examples involve setting some variable (the "destination") when certain command-line options are seen. What happens if those options are never seen? Since we didn't supply any defaults, they are all set to `None`. This is usually fine, but sometimes you want more control. *optparse* lets you supply a default value for each destination, which is assigned before the command line is parsed.

First, consider the verbose/quiet example. If we want *optparse* to set `verbose` to `True` unless `-q` is seen, then we can do this:

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

Since default values apply to the *destination* rather than to any particular option, and these two options happen to have the same destination, this is exactly equivalent:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Consider this:

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Again, the default value for `verbose` will be `True`: the last default value supplied for any particular destination is the one that counts.

A clearer way to specify default values is the `set_defaults()` method of OptionParser, which you can call at any time before calling *parse_args()*:

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

As before, the last value specified for a given option destination is the one that counts. For clarity, try to use one method or the other of setting default values, not both.

Generating help

optparse's ability to generate help and usage text automatically is useful for creating user-friendly command-line interfaces. All you have to do is supply a *help* value for each option, and optionally a short usage message for your whole program. Here's an OptionParser populated with user-friendly (documented) options:

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE")
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate, "
                       "or expert [default: %default]")
```

If *optparse* encounters either `-h` or `--help` on the command-line, or if you just call `parser.print_help()`, it prints the following to standard output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]
```

(If the help output is triggered by a help option, *optparse* exits after printing the help text.)

There's a lot going on here to help *optparse* generate the best possible help message:

- the script defines its own usage message:

  ```
  usage = "usage: %prog [options] arg1 arg2"
  ```

  optparse expands `%prog` in the usage string to the name of the current program, i.e. `os.path.basename(sys.argv[0])`. The expanded string is then printed before the detailed option help.

  If you don't supply a usage string, *optparse* uses a bland but sensible default: `"Usage: %prog [options]"`, which is fine if your script doesn't take any positional arguments.

- every option defines a help string, and doesn't worry about line-wrapping—*optparse* takes care of wrapping lines and making the help output look good.

- options that take a value indicate this fact in their automatically generated help message, e.g. for the "mode" option:

  ```
  -m MODE, --mode=MODE
  ```

  Here, "MODE" is called the meta-variable: it stands for the argument that the user is expected to supply to `-m`/`--mode`. By default, *optparse* converts the destination variable name to uppercase and uses that for the meta-variable. Sometimes, that's not what you want—for example, the `--filename` option explicitly sets `metavar="FILE"`, resulting in this automatically generated option description:

```
-f FILE, --filename=FILE
```

This is important for more than just saving space, though: the manually written help text uses the meta-variable `FILE` to clue the user in that there's a connection between the semi-formal syntax `-f FILE` and the informal semantic description "write output to FILE". This is a simple but effective way to make your help text a lot clearer and more useful for end users.

- options that have a default value can include `%default` in the help string—*optparse* will replace it with *str()* of the option's default value. If an option has no default value (or the default value is `None`), `%default` expands to `none`.

Grouping Options

When dealing with many options, it is convenient to group these options for better help output. An *OptionParser* can contain several option groups, each of which can contain several options.

An option group is obtained using the class *OptionGroup*:

class optparse.**OptionGroup**(*parser*, *title*, *description=None*)

where

- parser is the *OptionParser* instance the group will be inserted in to
- title is the group title
- description, optional, is a long description of the group

OptionGroup inherits from `OptionContainer` (like *OptionParser*) and so the `add_option()` method can be used to add an option to the group.

Once all the options are declared, using the *OptionParser* method `add_option_group()` the group is added to the previously defined parser.

Continuing with the parser defined in the previous section, adding an *OptionGroup* to a parser is easy:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

This would result in the following help output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.
```

(continues on next page)

(continued from previous page)

```
    -g                  Group option.
```

A bit more complete example might involve using more than one group: still extending the previous example:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
                 help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
                 help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
parser.add_option_group(group)
```

that results in the following output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or expert
                        [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.

  Debug Options:
    -d, --debug         Print debug information
    -s, --sql           Print all SQL statements executed
    -e                  Print every action done
```

Another interesting method, in particular when working programmatically with option groups is:

`OptionParser.`**`get_option_group`**`(`*opt_str*`)`

Return the *`OptionGroup`* to which the short or long option string *opt_str* (e.g. `'-o'` or `'--option'`) belongs. If there's no such *`OptionGroup`*, return `None`.

Printing a version string

Similar to the brief usage string, *optparse* can also print a version string for your program. You have to supply the string as the `version` argument to OptionParser:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

`%prog` is expanded just like it is in `usage`. Apart from that, `version` can contain anything you like. When you supply it, *optparse* automatically adds a `--version` option to your parser. If it encounters this option on the command line, it expands your `version` string (by replacing `%prog`), prints it to stdout, and exits.

For example, if your script is called `/usr/bin/foo`:

```
$ /usr/bin/foo --version
foo 1.0
```

The following two methods can be used to print and get the `version` string:

`OptionParser.`**`print_version`**`(`*file=None*`)`

Print the version message for the current program (`self.version`) to *file* (default stdout). As with *print_usage()*, any occurrence of `%prog` in `self.version` is replaced with the name of the current program. Does nothing if `self.version` is empty or undefined.

`OptionParser.`**`get_version`**`()`

Same as *print_version()* but returns the version string instead of printing it.

How `optparse` handles errors

There are two broad classes of errors that *optparse* has to worry about: programmer errors and user errors. Programmer errors are usually erroneous calls to *OptionParser.add_option()*, e.g. invalid option strings, unknown option attributes, missing option attributes, etc. These are dealt with in the usual way: raise an exception (either *optparse.OptionError* or *TypeError*) and let the program crash.

Handling user errors is much more important, since they are guaranteed to happen no matter how stable your code is. *optparse* can automatically detect some user errors, such as bad option arguments (passing `-n 4x` where `-n` takes an integer argument), missing arguments (`-n` at the end of the command line, where `-n` takes an argument of any type). Also, you can call `OptionParser.error()` to signal an application-defined error condition:

```
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

In either case, *optparse* handles the error the same way: it prints the program's usage message and an error message to standard error and exits with error status 2.

Consider the first example above, where the user passes `4x` to an option that takes an integer:

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

Or, where the user fails to pass a value at all:

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

optparse-generated error messages take care always to mention the option involved in the error; be sure to do the same when calling `OptionParser.error()` from your application code.

If *optparse*'s default error-handling behaviour does not suit your needs, you'll need to subclass OptionParser and override its `exit()` and/or `error()` methods.

Putting it all together

Here's what *optparse*-based scripts usually look like:

```
from optparse import OptionParser
...
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    ...
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print("reading %s..." % options.filename)
    ...

if __name__ == "__main__":
    main()
```

36.2.3 Reference Guide

Creating the parser

The first step in using *optparse* is to create an OptionParser instance.

class optparse.**OptionParser**(...)

The OptionParser constructor has no required arguments, but a number of optional keyword arguments. You should always pass them as keyword arguments, i.e. do not rely on the order in which the arguments are declared.

usage (**default: `"%prog [options]"`**)

The usage summary to print when your program is run incorrectly or with a help option. When *optparse* prints the usage string, it expands `%prog` to `os.path.basename(sys.argv[0])` (or to `prog` if you passed that keyword argument). To suppress a usage message, pass the special value `optparse.SUPPRESS_USAGE`.

option_list (**default: `[]`**)

A list of Option objects to populate the parser with. The options in `option_list` are added after any

options in `standard_option_list` (a class attribute that may be set by OptionParser subclasses), but before any version or help options. Deprecated; use *`add_option()`* after creating the parser instead.

`option_class` (default: optparse.Option)
Class to use when adding options to the parser in *`add_option()`*.

`version` (default: `None`)
A version string to print when the user supplies a version option. If you supply a true value for `version`, *`optparse`* automatically adds a version option with the single option string `--version`. The substring `%prog` is expanded the same as for `usage`.

`conflict_handler` (default: `"error"`)
Specifies what to do when options with conflicting option strings are added to the parser; see section *Conflicts between options*.

`description` (default: `None`)
A paragraph of text giving a brief overview of your program. *`optparse`* reformats this paragraph to fit the current terminal width and prints it when the user requests help (after `usage`, but before the list of options).

`formatter` (default: a new `IndentedHelpFormatter`)
An instance of optparse.HelpFormatter that will be used for printing help text. *`optparse`* provides two concrete classes for this purpose: IndentedHelpFormatter and TitledHelpFormatter.

`add_help_option` (default: `True`)
If true, *`optparse`* will add a help option (with option strings `-h` and `--help`) to the parser.

`prog`
The string to use when expanding `%prog` in `usage` and `version` instead of `os.path.basename(sys.argv[0])`.

`epilog` (default: `None`)
A paragraph of help text to print after the option help.

Populating the parser

There are several ways to populate the parser with options. The preferred way is by using *`OptionParser.add_option()`*, as shown in section *Tutorial*. `add_option()` can be called in one of two ways:

- pass it an Option instance (as returned by `make_option()`)
- pass it any combination of positional and keyword arguments that are acceptable to `make_option()` (i.e., to the Option constructor), and it will create the Option instance for you

The other alternative is to pass a list of pre-constructed Option instances to the OptionParser constructor, as in:

```
option_list = [
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose"),
    ]
parser = OptionParser(option_list=option_list)
```

(`make_option()` is a factory function for creating Option instances; currently it is an alias for the Option constructor. A future version of *`optparse`* may split Option into several classes, and `make_option()` will pick the right class to instantiate. Do not instantiate Option directly.)

Defining options

Each Option instance represents a set of synonymous command-line option strings, e.g. `-f` and `--file`. You can specify any number of short or long option strings, but you must specify at least one overall option string.

The canonical way to create an *Option* instance is with the `add_option()` method of *OptionParser*.

`OptionParser.`**`add_option`**`(`*option*`)`

`OptionParser.`**`add_option`**`(`**opt_str*, *attr*=*value*, ...`)`

To define an option with only a short option string:

```
parser.add_option("-f", attr=value, ...)
```

And to define an option with only a long option string:

```
parser.add_option("--foo", attr=value, ...)
```

The keyword arguments define attributes of the new Option object. The most important option attribute is *action*, and it largely determines which other attributes are relevant or required. If you pass irrelevant option attributes, or fail to pass required ones, *optparse* raises an *OptionError* exception explaining your mistake.

An option's *action* determines what *optparse* does when it encounters this option on the command-line. The standard option actions hard-coded into *optparse* are:

`"store"`
store this option's argument (default)

`"store_const"`
store a constant value, pre-set via *Option.const*

`"store_true"`
store `True`

`"store_false"`
store `False`

`"append"`
append this option's argument to a list

`"append_const"`
append a constant value to a list, pre-set via *Option.const*

`"count"`
increment a counter by one

`"callback"`
call a specified function

`"help"`
print a usage message including all options and the documentation for them

(If you don't supply an action, the default is `"store"`. For this action, you may also supply *type* and *dest* option attributes; see *Standard option actions*.)

As you can see, most actions involve storing or updating a value somewhere. *optparse* always creates a special object for this, conventionally called `options`, which is an instance of *optparse.Values*.

`class` optparse.**`Values`**

An object holding parsed argument names and values as attributes. Normally created by calling when calling *OptionParser.parse_args()*, and can be overridden by a custom subclass passed to the *values* argument of *OptionParser.parse_args()* (as described in *Parsing arguments*).

Option arguments (and various other values) are stored as attributes of this object, according to the *dest* (destination) option attribute.

For example, when you call

```
parser.parse_args()
```

one of the first things *optparse* does is create the `options` object:

```
options = Values()
```

If one of the options in this parser is defined with

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

and the command-line being parsed includes any of the following:

```
-ffoo
-f foo
--file=foo
--file foo
```

then *optparse*, on seeing this option, will do the equivalent of

```
options.filename = "foo"
```

The *type* and *dest* option attributes are almost as important as *action*, but *action* is the only one that makes sense for *all* options.

Option attributes

class optparse.**Option**

A single command line argument, with various attributes passed by keyword to the constructor. Normally created with *OptionParser.add_option()* rather than directly, and can be overridden by a custom class via the *option_class* argument to *OptionParser*.

The following option attributes may be passed as keyword arguments to *OptionParser.add_option()*. If you pass an option attribute that is not relevant to a particular option, or fail to pass a required option attribute, *optparse* raises *OptionError*.

Option.**action**

(default: `"store"`)

Determines *optparse*'s behaviour when this option is seen on the command line; the available options are documented *here*.

Option.**type**

(default: `"string"`)

The argument type expected by this option (e.g., `"string"` or `"int"`); the available option types are documented *here*.

Option.**dest**

(default: derived from option strings)

If the option's action implies writing or modifying a value somewhere, this tells *optparse* where to write it: *dest* names an attribute of the `options` object that *optparse* builds as it parses the command line.

Option.**default**

The value to use for this option's destination if the option is not seen on the command line. See also *OptionParser.set_defaults()*.

Option.**nargs**

(default: 1)

How many arguments of type *type* should be consumed when this option is seen. If > 1, *optparse* will store a tuple of values to *dest*.

Option.**const**

For actions that store a constant value, the constant value to store.

Option.**choices**

For options of type "choice", the list of strings the user may choose from.

Option.**callback**

For options with action "callback", the callable to call when this option is seen. See section *Option Callbacks* for detail on the arguments passed to the callable.

Option.**callback_args**

Option.**callback_kwargs**

Additional positional and keyword arguments to pass to callback after the four standard callback arguments.

Option.**help**

Help text to print for this option when listing all available options after the user supplies a *help* option (such as --help). If no help text is supplied, the option will be listed without help text. To hide this option, use the special value optparse.SUPPRESS_HELP.

Option.**metavar**

(default: derived from option strings)

Stand-in for the option argument(s) to use when printing help text. See section *Tutorial* for an example.

Standard option actions

The various option actions all have slightly different requirements and effects. Most actions have several relevant option attributes which you may specify to guide *optparse*'s behaviour; a few have required attributes, which you must specify for any option using that action.

- "store" [relevant: *type*, *dest*, *nargs*, *choices*]

  The option must be followed by an argument, which is converted to a value according to *type* and stored in *dest*. If *nargs* > 1, multiple arguments will be consumed from the command line; all will be converted according to *type* and stored to *dest* as a tuple. See the *Standard option types* section.

  If *choices* is supplied (a list or tuple of strings), the type defaults to "choice".

  If *type* is not supplied, it defaults to "string".

  If *dest* is not supplied, *optparse* derives a destination from the first long option string (e.g., --foo-bar implies foo_bar). If there are no long option strings, *optparse* derives a destination from the first short option string (e.g., -f implies f).

  Example:

```
parser.add_option("-f")
parser.add_option("-p", type="float", nargs=3, dest="point")
```

As it parses the command line

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

optparse will set

```
options.f = "foo.txt"
options.point = (1.0, -3.5, 4.0)
options.f = "bar.txt"
```

- "store_const" [required: *const*; relevant: *dest*]

  The value *const* is stored in *dest*.

  Example:

```
parser.add_option("-q", "--quiet",
                  action="store_const", const=0, dest="verbose")
parser.add_option("-v", "--verbose",
                  action="store_const", const=1, dest="verbose")
parser.add_option("--noisy",
                  action="store_const", const=2, dest="verbose")
```

  If --noisy is seen, *optparse* will set

```
options.verbose = 2
```

- "store_true" [relevant: *dest*]

  A special case of "store_const" that stores True to *dest*.

- "store_false" [relevant: *dest*]

  Like "store_true", but stores False.

  Example:

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- "append" [relevant: *type*, *dest*, *nargs*, *choices*]

  The option must be followed by an argument, which is appended to the list in *dest*. If no default value for *dest* is supplied, an empty list is automatically created when *optparse* first encounters this option on the command-line. If *nargs* > 1, multiple arguments are consumed, and a tuple of length *nargs* is appended to *dest*.

  The defaults for *type* and *dest* are the same as for the "store" action.

  Example:

```
parser.add_option("-t", "--tracks", action="append", type="int")
```

  If -t3 is seen on the command-line, *optparse* does the equivalent of:

```
options.tracks = []
options.tracks.append(int("3"))
```

  If, a little later on, --tracks=4 is seen, it does:

```
options.tracks.append(int("4"))
```

The `append` action calls the `append` method on the current value of the option. This means that any default value specified must have an `append` method. It also means that if the default value is non-empty, the default elements will be present in the parsed value for the option, with any values from the command line appended after those default values:

```
>>> parser.add_option("--files", action="append", default=['~/.mypkg/defaults'])
>>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
>>> opts.files
['~/.mypkg/defaults', 'overrides.mypkg']
```

- `"append_const"` [required: *const*; relevant: *dest*]

  Like `"store_const"`, but the value *const* is appended to *dest*; as with `"append"`, *dest* defaults to `None`, and an empty list is automatically created the first time the option is encountered.

- `"count"` [relevant: *dest*]

  Increment the integer stored at *dest*. If no default value is supplied, *dest* is set to zero before being incremented the first time.

  Example:

  ```
  parser.add_option("-v", action="count", dest="verbosity")
  ```

  The first time `-v` is seen on the command line, *optparse* does the equivalent of:

  ```
  options.verbosity = 0
  options.verbosity += 1
  ```

  Every subsequent occurrence of `-v` results in

  ```
  options.verbosity += 1
  ```

- `"callback"` [required: *callback*; relevant: *type*, *nargs*, *callback_args*, *callback_kwargs*]

  Call the function specified by *callback*, which is called as

  ```
  func(option, opt_str, value, parser, *args, **kwargs)
  ```

  See section *Option Callbacks* for more detail.

- `"help"`

  Prints a complete help message for all the options in the current option parser. The help message is constructed from the `usage` string passed to OptionParser's constructor and the *help* string passed to every option.

  If no *help* string is supplied for an option, it will still be listed in the help message. To omit an option entirely, use the special value `optparse.SUPPRESS_HELP`.

  optparse automatically adds a *help* option to all OptionParsers, so you do not normally need to create one.

  Example:

  ```
  from optparse import OptionParser, SUPPRESS_HELP

  # usually, a help option is added automatically, but that can
  # be suppressed using the add_help_option argument
  parser = OptionParser(add_help_option=False)

  parser.add_option("-h", "--help", action="help")
  parser.add_option("-v", action="store_true", dest="verbose",
  ```

(continues on next page)

(continued from previous page)

```
                    help="Be moderately verbose")
parser.add_option("--file", dest="filename",
                    help="Input file to read data from")
parser.add_option("--secret", help=SUPPRESS_HELP)
```

If *optparse* sees either `-h` or `--help` on the command line, it will print something like the following help message to stdout (assuming `sys.argv[0]` is `"foo.py"`):

```
Usage: foo.py [options]

Options:
  -h, --help        Show this help message and exit
  -v                Be moderately verbose
  --file=FILENAME   Input file to read data from
```

After printing the help message, *optparse* terminates your process with `sys.exit(0)`.

- `"version"`

  Prints the version number supplied to the OptionParser to stdout and exits. The version number is actually formatted and printed by the `print_version()` method of OptionParser. Generally only relevant if the `version` argument is supplied to the OptionParser constructor. As with *help* options, you will rarely create `version` options, since *optparse* automatically adds them when needed.

Standard option types

optparse has five built-in option types: `"string"`, `"int"`, `"choice"`, `"float"` and `"complex"`. If you need to add new option types, see section *Extending optparse*.

Arguments to string options are not checked or converted in any way: the text on the command line is stored in the destination (or passed to the callback) as-is.

Integer arguments (type `"int"`) are parsed as follows:

- if the number starts with `0x`, it is parsed as a hexadecimal number
- if the number starts with `0`, it is parsed as an octal number
- if the number starts with `0b`, it is parsed as a binary number
- otherwise, the number is parsed as a decimal number

The conversion is done by calling *int()* with the appropriate base (2, 8, 10, or 16). If this fails, so will *optparse*, although with a more useful error message.

`"float"` and `"complex"` option arguments are converted directly with *float()* and *complex()*, with similar error-handling.

`"choice"` options are a subtype of `"string"` options. The *choices* option attribute (a sequence of strings) defines the set of allowed option arguments. `optparse.check_choice()` compares user-supplied option arguments against this master list and raises *OptionValueError* if an invalid string is given.

Parsing arguments

The whole point of creating and populating an OptionParser is to call its *parse_args()* method.

OptionParser.**parse_args**(*args=None*, *values=None*)

Parse the command-line options found in *args*.

The input parameters are

args
: the list of arguments to process (default: `sys.argv[1:]`)

values
: a *Values* object to store option arguments in (default: a new instance of *Values*) – if you give an existing object, the option defaults will not be initialized on it

and the return value is a pair `(options, args)` where

options
: the same object that was passed in as *values*, or the `optparse.Values` instance created by *optparse*

args
: the leftover positional arguments after all options have been processed

The most common usage is to supply neither keyword argument. If you supply `values`, it will be modified with repeated *setattr()* calls (roughly one for every option argument stored to an option destination) and returned by *parse_args()*.

If *parse_args()* encounters any errors in the argument list, it calls the OptionParser's `error()` method with an appropriate end-user error message. This ultimately terminates your process with an exit status of 2 (the traditional Unix exit status for command-line errors).

Querying and manipulating your option parser

The default behavior of the option parser can be customized slightly, and you can also poke around your option parser and see what's there. OptionParser provides several methods to help you out:

OptionParser.**disable_interspersed_args**()

Set parsing to stop on the first non-option. For example, if `-a` and `-b` are both simple options that take no arguments, *optparse* normally accepts this syntax:

```
prog -a arg1 -b arg2
```

and treats it as equivalent to

```
prog -a -b arg1 arg2
```

To disable this feature, call *disable_interspersed_args()*. This restores traditional Unix syntax, where option parsing stops with the first non-option argument.

Use this if you have a command processor which runs another command which has options of its own and you want to make sure these options don't get confused. For example, each command might have a different set of options.

OptionParser.**enable_interspersed_args**()

Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior.

OptionParser.**get_option**(*opt_str*)

Returns the Option instance with the option string *opt_str*, or `None` if no options have that option string.

OptionParser.**has_option**(*opt_str*)

Return `True` if the OptionParser has an option with option string *opt_str* (e.g., `-q` or `--verbose`).

OptionParser.**remove_option**(*opt_str*)

If the *OptionParser* has an option corresponding to *opt_str*, that option is removed. If that option provided any other option strings, all of those option strings become invalid. If *opt_str* does not occur in any option belonging to this *OptionParser*, raises *ValueError*.

Conflicts between options

If you're not careful, it's easy to define options with conflicting option strings:

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(This is particularly true if you've defined your own OptionParser subclass with some standard options.)

Every time you add an option, *optparse* checks for conflicts with existing options. If it finds any, it invokes the current conflict-handling mechanism. You can set the conflict-handling mechanism either in the constructor:

```
parser = OptionParser(..., conflict_handler=handler)
```

or with a separate call:

```
parser.set_conflict_handler(handler)
```

The available conflict handlers are:

"error" (default)
assume option conflicts are a programming error and raise *OptionConflictError*

"resolve"
resolve option conflicts intelligently (see below)

As an example, let's define an *OptionParser* that resolves conflicts intelligently and add conflicting options to it:

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

At this point, *optparse* detects that a previously added option is already using the `-n` option string. Since `conflict_handler` is `"resolve"`, it resolves the situation by removing `-n` from the earlier option's list of option strings. Now `--dry-run` is the only way for the user to activate that option. If the user asks for help, the help message will reflect that:

```
Options:
  --dry-run     do no harm
  ...
  -n, --noisy   be noisy
```

It's possible to whittle away the option strings for a previously added option until there are none left, and the user has no way of invoking that option from the command-line. In that case, *optparse* removes that option completely, so it doesn't show up in help text or anywhere else. Carrying on with our existing OptionParser:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

At this point, the original -n/--dry-run option is no longer accessible, so *optparse* removes it, leaving this help text:

```
Options:
  ...
  -n, --noisy   be noisy
  --dry-run     new dry-run option
```

Cleanup

OptionParser instances have several cyclic references. This should not be a problem for Python's garbage collector, but you may wish to break the cyclic references explicitly by calling `destroy()` on your OptionParser once you are done with it. This is particularly useful in long-running applications where large object graphs are reachable from your OptionParser.

Other methods

OptionParser supports several other public methods:

`OptionParser.`**`set_usage`**(*usage*)

Set the usage string according to the rules described above for the `usage` constructor keyword argument. Passing `None` sets the default usage string; use `optparse.SUPPRESS_USAGE` to suppress a usage message.

`OptionParser.`**`print_usage`**(*file=None*)

Print the usage message for the current program (`self.usage`) to *file* (default stdout). Any occurrence of the string `%prog` in `self.usage` is replaced with the name of the current program. Does nothing if `self.usage` is empty or not defined.

`OptionParser.`**`get_usage`**()

Same as *print_usage()* but returns the usage string instead of printing it.

`OptionParser.`**`set_defaults`**(*dest=value*, ...)

Set default values for several option destinations at once. Using *set_defaults()* is the preferred way to set default values for options, since multiple options can share the same destination. For example, if several "mode" options all set the same destination, any one of them can set the default, and the last one wins:

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")    # overridden below
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")  # overrides above setting
```

To avoid this confusion, use *set_defaults()*:

```
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice")
```

36.2.4 Option Callbacks

When *optparse*'s built-in actions and types aren't quite enough for your needs, you have two choices: extend *optparse* or define a callback option. Extending *optparse* is more general, but overkill for a lot of simple cases. Quite often a simple callback is all you need.

There are two steps to defining a callback option:

- define the option itself using the `"callback"` action
- write the callback; this is a function (or method) that takes at least four arguments, as described below

Defining a callback option

As always, the easiest way to define a callback option is by using the *OptionParser.add_option()* method. Apart from *action*, the only option attribute you must specify is `callback`, the function to call:

```
parser.add_option("-c", action="callback", callback=my_callback)
```

`callback` is a function (or other callable object), so you must have already defined `my_callback()` when you create this callback option. In this simple case, *optparse* doesn't even know if `-c` takes any arguments, which usually means that the option takes no arguments—the mere presence of `-c` on the command-line is all it needs to know. In some circumstances, though, you might want your callback to consume an arbitrary number of command-line arguments. This is where writing callbacks gets tricky; it's covered later in this section.

optparse always passes four particular arguments to your callback, and it will only pass additional arguments if you specify them via *callback_args* and *callback_kwargs*. Thus, the minimal callback function signature is:

```
def my_callback(option, opt, value, parser):
```

The four arguments to a callback are described below.

There are several other option attributes that you can supply when you define a callback option:

type
: has its usual meaning: as with the `"store"` or `"append"` actions, it instructs *optparse* to consume one argument and convert it to *type*. Rather than storing the converted value(s) anywhere, though, *optparse* passes it to your callback function.

nargs
: also has its usual meaning: if it is supplied and > 1, *optparse* will consume *nargs* arguments, each of which must be convertible to *type*. It then passes a tuple of converted values to your callback.

callback_args
: a tuple of extra positional arguments to pass to the callback

callback_kwargs
: a dictionary of extra keyword arguments to pass to the callback

How callbacks are called

All callbacks are called as follows:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

where

option
: is the Option instance that's calling the callback

opt_str
: is the option string seen on the command-line that's triggering the callback. (If an abbreviated long option was used, `opt_str` will be the full, canonical option string—e.g. if the user puts `--foo` on the command-line as an abbreviation for `--foobar`, then `opt_str` will be `"--foobar"`.)

value
: is the argument to this option seen on the command-line. *optparse* will only expect an argument if *type* is set; the type of `value` will be the type implied by the option's type. If *type* for this option is `None` (no argument expected), then `value` will be `None`. If *nargs* > 1, `value` will be a tuple of values of the appropriate type.

parser
: is the OptionParser instance driving the whole thing, mainly useful because you can access some other interesting data through its instance attributes:

  parser.largs
  : the current list of leftover arguments, ie. arguments that have been consumed but are neither options nor option arguments. Feel free to modify `parser.largs`, e.g. by adding more arguments to it. (This list will become `args`, the second return value of *parse_args()*.)

  parser.rargs
  : the current list of remaining arguments, ie. with `opt_str` and `value` (if applicable) removed, and only the arguments following them still there. Feel free to modify `parser.rargs`, e.g. by consuming more arguments.

  parser.values
  : the object where option values are by default stored (an instance of optparse.OptionValues). This lets callbacks use the same mechanism as the rest of *optparse* for storing option values; you don't need to mess around with globals or closures. You can also access or modify the value(s) of any options already encountered on the command-line.

args
: is a tuple of arbitrary positional arguments supplied via the *callback_args* option attribute.

kwargs
: is a dictionary of arbitrary keyword arguments supplied via *callback_kwargs*.

Raising errors in a callback

The callback function should raise *OptionValueError* if there are any problems with the option or its argument(s). *optparse* catches this and terminates the program, printing the error message you supply to stderr. Your message should be clear, concise, accurate, and mention the option at fault. Otherwise, the user will have a hard time figuring out what they did wrong.

Callback example 1: trivial callback

Here's an example of a callback option that takes no arguments, and simply records that the option was seen:

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

Of course, you could do that with the `"store_true"` action.

Callback example 2: check option order

Here's a slightly more interesting example: record the fact that `-a` is seen, but blow up if it comes after `-b` in the command-line.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
...
parser.add_option("-a", action="callback", callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

Callback example 3: check option order (generalized)

If you want to reuse this callback for several similar options (set a flag, but blow up if `-b` has already been seen), it needs a bit of work: the error message and the flag that it sets must be generalized.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("-a", action="callback", callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest='c')
```

Callback example 4: check arbitrary condition

Of course, you could put any condition in there—you're not limited to checking the values of already-defined options. For example, if you have options that should not be called when the moon is full, all you have to do is this:

```
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full"
                               % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

(The definition of `is_moon_full()` is left as an exercise for the reader.)

Callback example 5: fixed arguments

Things get slightly more interesting when you define callback options that take a fixed number of arguments. Specifying that a callback option takes arguments is similar to defining a "store" or "append" option: if you define *type*, then the option takes one argument that must be convertible to that type; if you further define *nargs*, then the option takes *nargs* arguments.

Here's an example that just emulates the standard "store" action:

```
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

Note that *optparse* takes care of consuming 3 arguments and converting them to integers for you; all you have to do is store them. (Or whatever; obviously you don't need a callback for this example.)

Callback example 6: variable arguments

Things get hairy when you want an option to take a variable number of arguments. For this case, you must write a callback, as *optparse* doesn't provide any built-in capabilities for it. And you have to deal with certain intricacies of conventional Unix command-line parsing that *optparse* normally handles for you. In particular, callbacks should implement the conventional rules for bare -- and - arguments:

- either -- or - can be option arguments
- bare -- (if not the argument to some option): halt command-line processing and discard the --
- bare - (if not the argument to some option): halt command-line processing but keep the - (append it to parser.largs)

If you want an option that takes a variable number of arguments, there are several subtle, tricky issues to worry about. The exact implementation you choose will be based on which trade-offs you're willing to make for your application (which is why *optparse* doesn't support this sort of thing directly).

Nevertheless, here's a stab at a callback for an option with variable arguments:

```
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []

    def floatable(str):
        try:
            float(str)
            return True
        except ValueError:
            return False

    for arg in parser.rargs:
        # stop on --foo like options
        if arg[:2] == "--" and len(arg) > 2:
            break
        # stop on -a, but not on -3 or -3.0
        if arg[:1] == "-" and len(arg) > 1 and not floatable(arg):
            break
        value.append(arg)
```

(continues on next page)

(continued from previous page)

```
    del parser.rargs[:len(value)]
    setattr(parser.values, option.dest, value)

...
parser.add_option("-c", "--callback", dest="vararg_attr",
                  action="callback", callback=vararg_callback)
```

36.2.5 Extending `optparse`

Since the two major controlling factors in how *optparse* interprets command-line options are the action and type of each option, the most likely direction of extension is to add new actions and new types.

Adding new types

To add new types, you need to define your own subclass of *optparse*'s *Option* class. This class has a couple of attributes that define *optparse*'s types: *TYPES* and *TYPE_CHECKER*.

`Option.`**`TYPES`**

A tuple of type names; in your subclass, simply define a new tuple *TYPES* that builds on the standard one.

`Option.`**`TYPE_CHECKER`**

A dictionary mapping type names to type-checking functions. A type-checking function has the following signature:

```
def check_mytype(option, opt, value)
```

where `option` is an *Option* instance, `opt` is an option string (e.g., `-f`), and `value` is the string from the command line that must be checked and converted to your desired type. `check_mytype()` should return an object of the hypothetical type `mytype`. The value returned by a type-checking function will wind up in the OptionValues instance returned by *OptionParser.parse_args()*, or be passed to a callback as the `value` parameter.

Your type-checking function should raise *OptionValueError* if it encounters any problems. *OptionValueError* takes a single string argument, which is passed as-is to *OptionParser*'s `error()` method, which in turn prepends the program name and the string `"error:"` and prints everything to stderr before terminating the process.

Here's a silly example that demonstrates adding a `"complex"` option type to parse Python-style complex numbers on the command line. (This is even sillier than it used to be, because *optparse* 1.3 added built-in support for complex numbers, but never mind.)

First, the necessary imports:

```
from copy import copy
from optparse import Option, OptionValueError
```

You need to define your type-checker first, since it's referred to later (in the *TYPE_CHECKER* class attribute of your Option subclass):

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
```

(continues on next page)

(continued from previous page)

```
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

Finally, the Option subclass:

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(If we didn't make a *copy()* of *Option.TYPE_CHECKER*, we would end up modifying the *TYPE_CHECKER* attribute of *optparse*'s Option class. This being Python, nothing stops you from doing that except good manners and common sense.)

That's it! Now you can write a script that uses the new option type just like any other *optparse*-based script, except you have to instruct your OptionParser to use MyOption instead of Option:

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

Alternately, you can build your own option list and pass it to OptionParser; if you don't use `add_option()` in the above way, you don't need to tell OptionParser which option class to use:

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

Adding new actions

Adding new actions is a bit trickier, because you have to understand that *optparse* has a couple of classifications for actions:

"store" actions
: actions that result in *optparse* storing a value to an attribute of the current OptionValues instance; these options require a *dest* attribute to be supplied to the Option constructor.

"typed" actions
: actions that take a value from the command line and expect it to be of a certain type; or rather, a string that can be converted to a certain type. These options require a *type* attribute to the Option constructor.

These are overlapping sets: some default "store" actions are `"store"`, `"store_const"`, `"append"`, and `"count"`, while the default "typed" actions are `"store"`, `"append"`, and `"callback"`.

When you add an action, you need to categorize it by listing it in at least one of the following class attributes of Option (all are lists of strings):

`Option.`**`ACTIONS`**
: All actions must be listed in ACTIONS.

`Option.`**`STORE_ACTIONS`**
: "store" actions are additionally listed here.

`Option.`**`TYPED_ACTIONS`**
: "typed" actions are additionally listed here.

Option.**ALWAYS_TYPED_ACTIONS**

Actions that always take a type (i.e. whose options always take a value) are additionally listed here. The only effect of this is that *optparse* assigns the default type, "string", to options with no explicit type whose action is listed in *ALWAYS_TYPED_ACTIONS*.

In order to actually implement your new action, you must override Option's take_action() method and add a case that recognizes your action.

For example, let's add an "extend" action. This is similar to the standard "append" action, but instead of taking a single value from the command-line and appending it to an existing list, "extend" will take multiple values in a single comma-delimited string, and extend an existing list with them. That is, if --names is an "extend" option of type "string", the command line

```
--names=foo,bar --names blah --names ding,dong
```

would result in a list

```
["foo", "bar", "blah", "ding", "dong"]
```

Again we define a subclass of Option:

```
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

Features of note:

- "extend" both expects a value on the command-line and stores that value somewhere, so it goes in both *STORE_ACTIONS* and *TYPED_ACTIONS*.
- to ensure that *optparse* assigns the default type of "string" to "extend" actions, we put the "extend" action in *ALWAYS_TYPED_ACTIONS* as well.
- MyOption.take_action() implements just this one new action, and passes control back to Option.take_action() for the standard *optparse* actions.
- values is an instance of the optparse_parser.Values class, which provides the very useful ensure_value() method. ensure_value() is essentially *getattr()* with a safety valve; it is called as

  ```
  values.ensure_value(attr, value)
  ```

  If the attr attribute of values doesn't exist or is None, then ensure_value() first sets it to value, and then returns value. This is very handy for actions like "extend", "append", and "count", all of which accumulate data in a variable and expect that variable to be of a certain type (a list for the first two, an integer for the latter). Using ensure_value() means that scripts using your action don't have to worry about setting a default value for the option destinations in question; they can just leave the default as None and ensure_value() will take care of getting it right when it's needed.

36.2.6 Exceptions

exception optparse.**OptionError**

Raised if an *Option* instance is created with invalid or inconsistent arguments.

exception optparse.**OptionConflictError**

Raised if conflicting options are added to an *OptionParser*.

exception optparse.**OptionValueError**

Raised if an invalid option value is encountered on the command line.

exception optparse.**BadOptionError**

Raised if an invalid option is passed on the command line.

exception optparse.**AmbiguousOptionError**

Raised if an ambiguous option is passed on the command line.

CHAPTER

THIRTYSEVEN

SECURITY CONSIDERATIONS

The following modules have specific security considerations:

- `base64`: *base64 security considerations* in **RFC 4648**
- `hashlib`: *all constructors take a "usedforsecurity" keyword-only argument disabling known insecure and blocked algorithms*
- `http.server` is not suitable for production use, only implementing basic security checks. See the *security considerations*.
- `logging`: *Logging configuration uses eval()*
- `multiprocessing`: *Connection.recv() uses pickle*
- `pickle`: *Restricting globals in pickle*
- `random` shouldn't be used for security purposes, use `secrets` instead
- `shelve`: *shelve is based on pickle and thus unsuitable for dealing with untrusted sources*
- `ssl`: *SSL/TLS security considerations*
- `subprocess`: *Subprocess security considerations*
- `tempfile`: *mktemp is deprecated due to vulnerability to race conditions*
- `xml`: *XML vulnerabilities*
- `zipfile`: *maliciously prepared .zip files can cause disk volume exhaustion*

The `-I` command line option can be used to run Python in isolated mode. When it cannot be used, the `-P` option or the `PYTHONSAFEPATH` environment variable can be used to not prepend a potentially unsafe path to `sys.path` such as the current directory, the script's directory or an empty string.

APPENDIX

A

GLOSSARY

>>>
: The default Python prompt of the *interactive* shell. Often seen for code examples which can be executed interactively in the interpreter.

...
: Can refer to:

  - The default Python prompt of the *interactive* shell when entering the code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.
  - The `Ellipsis` built-in constant.

abstract base class
: Abstract base classes complement *duck-typing* by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections.abc` module), numbers (in the `numbers` module), streams (in the `io` module), import finders and loaders (in the `importlib.abc` module). You can create your own ABCs with the `abc` module.

annotation
: A label associated with a variable, a class attribute or a function parameter or return value, used by convention as a *type hint*.

  Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions are stored in the `__annotations__` special attribute of modules, classes, and functions, respectively.

  See *variable annotation*, *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

argument
: A value passed to a *function* (or *method*) when calling the function. There are two kinds of argument:

  - *keyword argument*: an argument preceded by an identifier (e.g. `name=`) in a function call or passed as a value in a dictionary preceded by `**`. For example, `3` and `5` are both keyword arguments in the following calls to `complex()`:

    ```
    complex(real=3, imag=5)
    complex(**{'real': 3, 'imag': 5})
    ```

  - *positional argument*: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an *iterable* preceded by `*`. For example, `3` and `5` are both positional arguments in the following calls:

```
complex(3, 5)
complex(*(3, 5))
```

Arguments are assigned to the named local variables in a function body. See the calls section for the rules governing this assignment. Syntactically, any expression can be used to represent an argument; the evaluated value is assigned to the local variable.

See also the *parameter* glossary entry, the FAQ question on the difference between arguments and parameters, and **PEP 362**.

asynchronous context manager
An object which controls the environment seen in an `async with` statement by defining `__aenter__()` and `__aexit__()` methods. Introduced by **PEP 492**.

asynchronous generator
A function which returns an *asynchronous generator iterator*. It looks like a coroutine function defined with `async def` except that it contains `yield` expressions for producing a series of values usable in an `async for` loop.

Usually refers to an asynchronous generator function, but may refer to an *asynchronous generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

An asynchronous generator function may contain `await` expressions as well as `async for`, and `async with` statements.

asynchronous generator iterator
An object created by a *asynchronous generator* function.

This is an *asynchronous iterator* which when called using the `__anext__()` method returns an awaitable object which will execute the body of the asynchronous generator function until the next `yield` expression.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See **PEP 492** and **PEP 525**.

asynchronous iterable
An object, that can be used in an `async for` statement. Must return an *asynchronous iterator* from its `__aiter__()` method. Introduced by **PEP 492**.

asynchronous iterator
An object that implements the `__aiter__()` and `__anext__()` methods. `__anext__()` must return an *awaitable* object. `async for` resolves the awaitables returned by an asynchronous iterator's `__anext__()` method until it raises a *`StopAsyncIteration`* exception. Introduced by **PEP 492**.

attribute
A value associated with an object which is usually referenced by name using dotted expressions. For example, if an object *o* has an attribute *a* it would be referenced as *o.a*.

It is possible to give an object an attribute whose name is not an identifier as defined by identifiers, for example using *`setattr()`*, if the object allows it. Such an attribute will not be accessible using a dotted expression, and would instead need to be retrieved with *`getattr()`*.

awaitable
An object that can be used in an `await` expression. Can be a *coroutine* or an object with an `__await__()` method. See also **PEP 492**.

BDFL
Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python's creator.

binary file
A *file object* able to read and write *bytes-like objects*. Examples of binary files are files opened in binary mode

('rb', 'wb' or 'rb+'), *sys.stdin.buffer*, *sys.stdout.buffer*, and instances of *io.BytesIO* and *gzip.GzipFile*.

See also *text file* for a file object able to read and write *str* objects.

borrowed reference
In Python's C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

Calling `Py_INCREF()` on the *borrowed reference* is recommended to convert it to a *strong reference* in-place, except when the object cannot be destroyed before the last usage of the borrowed reference. The `Py_NewRef()` function can be used to create a new *strong reference*.

bytes-like object
An object that supports the bufferobjects and can export a C-*contiguous* buffer. This includes all *bytes*, *bytearray*, and *array.array* objects, as well as many common *memoryview* objects. Bytes-like objects can be used for various operations that work with binary data; these include compression, saving to a binary file, and sending over a socket.

Some operations need the binary data to be mutable. The documentation often refers to these as "read-write bytes-like objects". Example mutable buffer objects include *bytearray* and a *memoryview* of a *bytearray*. Other operations require the binary data to be stored in immutable objects ("read-only bytes-like objects"); examples of these include *bytes* and a *memoryview* of a *bytes* object.

bytecode
Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This "intermediate language" is said to run on a *virtual machine* that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

A list of bytecode instructions can be found in the documentation for *the dis module*.

callable
A callable is an object that can be called, possibly with a set of arguments (see *argument*), with the following syntax:

```
callable(argument1, argument2, argumentN)
```

A *function*, and by extension a *method*, is a callable. An instance of a class that implements the `__call__()` method is also a callable.

callback
A subroutine function which is passed as an argument to be executed at some point in the future.

class
A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

class variable
A variable defined in a class and intended to be modified only at class level (i.e., not in an instance of the class).

complex number
An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of `-1`), often written `i` in mathematics or `j` in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a `j` suffix, e.g., `3+1j`. To get access to complex equivalents of the *math* module, use *cmath*. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

context manager
An object which controls the environment seen in a `with` statement by defining `__enter__()` and `__exit__()` methods. See **PEP 343**.

context variable
A variable which can have different values depending on its context. This is similar to Thread-Local Storage in which each execution thread may have a different value for a variable. However, with context variables, there may be several contexts in one execution thread and the main usage for context variables is to keep track of variables in concurrent asynchronous tasks. See *contextvars*.

contiguous
A buffer is considered contiguous exactly if it is either *C-contiguous* or *Fortran contiguous*. Zero-dimensional buffers are C and Fortran contiguous. In one-dimensional arrays, the items must be laid out in memory next to each other, in order of increasing indexes starting from zero. In multidimensional C-contiguous arrays, the last index varies the fastest when visiting items in order of memory address. However, in Fortran contiguous arrays, the first index varies the fastest.

coroutine
Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the `async def` statement. See also **PEP 492**.

coroutine function
A function which returns a *coroutine* object. A coroutine function may be defined with the `async def` statement, and may contain `await`, `async for`, and `async with` keywords. These were introduced by **PEP 492**.

CPython
The canonical implementation of the Python programming language, as distributed on python.org. The term "CPython" is used when necessary to distinguish this implementation from others such as Jython or IronPython.

decorator
A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for decorators are *classmethod()* and *staticmethod()*.

The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for function definitions and class definitions for more about decorators.

descriptor
Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

For more information about descriptors' methods, see descriptors or the Descriptor How To Guide.

dictionary
An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

dictionary comprehension
A compact way to process all or part of the elements in an iterable and return a dictionary with the results. `results = {n: n ** 2 for n in range(10)}` generates a dictionary containing key `n` mapped to value `n ** 2`. See comprehensions.

dictionary view
The objects returned from *`dict.keys()`*, *`dict.values()`*, and *`dict.items()`* are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See *Dictionary view objects*.

docstring
A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

duck-typing
A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using *`type()`* or *`isinstance()`*. (Note, however, that duck-typing can be complemented with *abstract base classes*.) Instead, it typically employs *`hasattr()`* tests or *EAFP* programming.

EAFP
Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the *LBYL* style common to many other languages such as C.

expression
A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statement*s which cannot be used as expressions, such as `while`. Assignments are also statements, not expressions.

extension module
A module written in C or C++, using Python's C API to interact with the core and with user code.

f-string
String literals prefixed with `'f'` or `'F'` are commonly called "f-strings" which is short for formatted string literals. See also **PEP 498**.

file object
An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

There are actually three categories of file objects: raw *binary files*, buffered *binary files* and *text files*. Their interfaces are defined in the *`io`* module. The canonical way to create a file object is by using the *`open()`* function.

file-like object
A synonym for *file object*.

filesystem encoding and error handler
Encoding and error handler used by Python to decode bytes from the operating system and encode Unicode to the operating system.

The filesystem encoding must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise *UnicodeError*.

The *sys.getfilesystemencoding()* and *sys.getfilesystemencodeerrors()* functions can be used to get the filesystem encoding and error handler.

The *filesystem encoding and error handler* are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

See also the *locale encoding*.

finder
An object that tries to find the *loader* for a module that is being imported.

There are two types of finder: *meta path finders* for use with *sys.meta_path*, and *path entry finders* for use with *sys.path_hooks*.

See importsystem and *importlib* for much more detail.

floor division
Mathematical division that rounds down to nearest integer. The floor division operator is `//`. For example, the expression `11 // 4` evaluates to `2` in contrast to the `2.75` returned by float true division. Note that `(-11) // 4` is `-3` because that is `-2.75` rounded *downward*. See **PEP 238**.

free threading
A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See **PEP 703**.

function
A series of statements which returns some value to a caller. It can also be passed zero or more *arguments* which may be used in the execution of the body. See also *parameter*, *method*, and the function section.

function annotation
An *annotation* of a function parameter or return value.

Function annotations are usually used for *type hints*: for example, this function is expected to take two *int* arguments and is also expected to have an *int* return value:

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

Function annotation syntax is explained in section function.

See *variable annotation* and **PEP 484**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

__future__
A future statement, `from __future__ import <feature>`, directs the compiler to compile the current module using syntax or semantics that will become standard in a future release of Python. The *__future__* module documents the possible values of *feature*. By importing this module and evaluating its variables, you can see when a new feature was first added to the language and when it will (or did) become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection
The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles. The garbage collector can be controlled using the *gc* module.

generator
A function which returns a *generator iterator*. It looks like a normal function except that it contains `yield` expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the *`next()`* function.

Usually refers to a generator function, but may refer to a *generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

generator iterator
An object created by a *generator* function.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

generator expression
An *expression* that returns an *iterator*. It looks like a normal expression followed by a `for` clause defining a loop variable, range, and an optional `if` clause. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

generic function
A function composed of multiple functions implementing the same operation for different types. Which implementation should be used during a call is determined by the dispatch algorithm.

See also the *single dispatch* glossary entry, the *`functools.singledispatch()`* decorator, and **PEP 443**.

generic type
A *type* that can be parameterized; typically a container class such as *`list`* or *`dict`*. Used for *type hints* and *annotations*.

For more details, see *generic alias types*, **PEP 483**, **PEP 484**, **PEP 585**, and the *`typing`* module.

GIL
See *global interpreter lock*.

global interpreter lock
The mechanism used by the *CPython* interpreter to assure that only one thread executes Python *bytecode* at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as *`dict`*) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil 0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see **PEP 703**.

hash-based pyc
A bytecode cache file that uses the hash rather than the last-modified time of the corresponding source file to determine its validity. See pyc-invalidation.

hashable
An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

Most of Python's immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not; immutable containers (such as tuples and frozensets) are only hashable if their elements are hashable. Objects which are instances of user-defined classes are hashable by default. They all compare unequal (except with themselves), and their hash value is derived from their `id()`.

IDLE
An Integrated Development and Learning Environment for Python. *IDLE* is a basic editor and interpreter environment which ships with the standard distribution of Python.

immortal
If an object is immortal, its reference count is never modified, and therefore it is never deallocated.

Built-in strings and singletons are immortal objects. For example, `True` and `None` singletons are immmortal.

See PEP 683 – Immortal Objects, Using a Fixed Refcount for more information.

immutable
An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

import path
A list of locations (or *path entries*) that are searched by the *path based finder* for modules to import. During import, this list of locations usually comes from `sys.path`, but for subpackages it may also come from the parent package's `__path__` attribute.

importing
The process by which Python code in one module is made available to Python code in another module.

importer
An object that both finds and loads a module; both a *finder* and *loader* object.

interactive
Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`). For more on interactive mode, see tut-interac.

interpreted
Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also *interactive*.

interpreter shutdown
When asked to shut down, the Python interpreter enters a special phase where it gradually releases all allocated resources, such as modules and various critical internal structures. It also makes several calls to the *garbage collector*. This can trigger the execution of code in user-defined destructors or weakref callbacks. Code executed during the shutdown phase can encounter various exceptions as the resources it relies on may not function anymore (common examples are library modules or the warnings machinery).

The main reason for interpreter shutdown is that the `__main__` module or the script being run has finished executing.

iterable
An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, *file objects*, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *sequence* semantics.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (*zip()*, *map()*, ...). When an iterable object is passed as an argument to the built-in function *iter()*, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call *iter()* or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterator
An object representing a stream of data. Repeated calls to the iterator's *__next__()* method (or passing it to the built-in function *next()*) return successive items in the stream. When no more data are available a *StopIteration* exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise *StopIteration* again. Iterators are required to have an *__iter__()* method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a *list*) produces a fresh new iterator each time you pass it to the *iter()* function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

More information can be found in *Iterator Types*.

CPython implementation detail: CPython does not consistently apply the requirement that an iterator define *__iter__()*. And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

key function
A key function or collation function is a callable that returns a value used for sorting or ordering. For example, *locale.strxfrm()* is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include *min()*, *max()*, *sorted()*, *list.sort()*, *heapq.merge()*, *heapq.nsmallest()*, *heapq.nlargest()*, and *itertools.groupby()*.

There are several ways to create a key function. For example. the *str.lower()* method can serve as a key function for case insensitive sorts. Alternatively, a key function can be built from a `lambda` expression such as `lambda r: (r[0], r[2])`. Also, *operator.attrgetter()*, *operator.itemgetter()*, and *operator.methodcaller()* are three key function constructors. See the Sorting HOW TO for examples of how to create and use key functions.

keyword argument
See *argument*.

lambda
An anonymous inline function consisting of a single *expression* which is evaluated when the function is called. The syntax to create a lambda function is `lambda [parameters]: expression`

LBYL
Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the *EAFP* approach and is characterized by the presence of many `if` statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between "the looking" and "the leaping". For example, the code, `if key in mapping: return mapping[key]` can fail if another thread removes *key* from *mapping* after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

list
A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is *O*(1).

list comprehension
A compact way to process all or part of the elements in a sequence and return a list with the results. `result =`

`['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

loader
An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See **PEP 302** for details and *importlib.abc.Loader* for an *abstract base class*.

locale encoding
On Unix, it is the encoding of the LC_CTYPE locale. It can be set with *locale.setlocale(locale.LC_CTYPE, new_locale)*.

On Windows, it is the ANSI code page (ex: `"cp1252"`).

On Android and VxWorks, Python uses `"utf-8"` as the locale encoding.

locale.getencoding() can be used to get the locale encoding.

See also the *filesystem encoding and error handler*.

magic method
An informal synonym for *special method*.

mapping
A container object that supports arbitrary key lookups and implements the methods specified in the *collections.abc.Mapping* or *collections.abc.MutableMapping* *abstract base classes*. Examples include *dict*, *collections.defaultdict*, *collections.OrderedDict* and *collections.Counter*.

meta path finder
A *finder* returned by a search of *sys.meta_path*. Meta path finders are related to, but different from *path entry finders*.

See *importlib.abc.MetaPathFinder* for the methods that meta path finders implement.

metaclass
The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in metaclasses.

method
A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first *argument* (which is usually called `self`). See *function* and *nested scope*.

method resolution order
Method Resolution Order is the order in which base classes are searched for a member during lookup. See python_2.3_mro for details of the algorithm used by the Python interpreter since the 2.3 release.

module
An object that serves as an organizational unit of Python code. Modules have a namespace containing arbitrary Python objects. Modules are loaded into Python by the process of *importing*.

See also *package*.

module spec
A namespace containing the import-related information used to load a module. An instance of *importlib.machinery.ModuleSpec*.

MRO
See *method resolution order*.

mutable
Mutable objects can change their value but keep their `id()`. See also *immutable*.

named tuple
The term "named tuple" applies to any type or class that inherits from tuple and whose indexable elements are also accessible using named attributes. The type or class may have other features as well.

Several built-in types are named tuples, including the values returned by `time.localtime()` and `os.stat()`. Another example is `sys.float_info`:

```
>>> sys.float_info[1]                   # indexed access
1024
>>> sys.float_info.max_exp              # named field access
1024
>>> isinstance(sys.float_info, tuple)   # kind of tuple
True
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand, or it can be created by inheriting `typing.NamedTuple`, or with the factory function `collections.namedtuple()`. The latter techniques also add some extra methods that may not be found in hand-written or built-in named tuples.

namespace
The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `builtins.open` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.islice()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

namespace package
A **PEP 420** *package* which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a *regular package* because they have no `__init__.py` file.

See also *module*.

nested scope
The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes by default work only for reference and not for assignment. Local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace. The `nonlocal` allows writing to outer scopes.

new-style class
Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()`, class methods, and static methods.

object
Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any *new-style class*.

optimized scope
A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations

are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

package
A Python *module* which can contain submodules or recursively, subpackages. Technically, a package is a Python module with a `__path__` attribute.

See also *regular package* and *namespace package*.

parameter
A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are five kinds of parameter:

- *positional-or-keyword*: specifies an argument that can be passed either *positionally* or as a *keyword argument*. This is the default kind of parameter, for example *foo* and *bar* in the following:

```
def func(foo, bar=None): ...
```

- *positional-only*: specifies an argument that can be supplied only by position. Positional-only parameters can be defined by including a `/` character in the parameter list of the function definition after them, for example *posonly1* and *posonly2* in the following:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare `*` in the parameter list of the function definition before them, for example *kw_only1* and *kw_only2* in the following:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `*`, for example *args* in the following:

```
def func(*args, **kwargs): ...
```

- *var-keyword*: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `**`, for example *kwargs* in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

See also the *argument* glossary entry, the FAQ question on the difference between arguments and parameters, the `inspect.Parameter` class, the function section, and **PEP 362**.

path entry
A single location on the *import path* which the *path based finder* consults to find modules for importing.

path entry finder
A *finder* returned by a callable on `sys.path_hooks` (i.e. a *path entry hook*) which knows how to locate modules given a *path entry*.

See `importlib.abc.PathEntryFinder` for the methods that path entry finders implement.

path entry hook
A callable on the `sys.path_hooks` list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

path based finder
One of the default *meta path finders* which searches an *import path* for modules.

path-like object
An object representing a file system path. A path-like object is either a *str* or *bytes* object representing a path, or an object implementing the *os.PathLike* protocol. An object that supports the *os.PathLike* protocol can be converted to a *str* or *bytes* file system path by calling the *os.fspath()* function; *os.fsdecode()* and *os.fsencode()* can be used to guarantee a *str* or *bytes* result instead, respectively. Introduced by **PEP 519**.

PEP
Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. PEPs should provide a concise technical specification and a rationale for proposed features.

PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

See **PEP 1**.

portion
A set of files in a single directory (possibly stored in a zip file) that contribute to a namespace package, as defined in **PEP 420**.

positional argument
See *argument*.

provisional API
A provisional API is one which has been deliberately excluded from the standard library's backwards compatibility guarantees. While major changes to such interfaces are not expected, as long as they are marked provisional, backwards incompatible changes (up to and including removal of the interface) may occur if deemed necessary by core developers. Such changes will not be made gratuitously – they will occur only if serious fundamental flaws are uncovered that were missed prior to the inclusion of the API.

Even for provisional APIs, backwards incompatible changes are seen as a "solution of last resort" - every attempt will still be made to find a backwards compatible resolution to any identified problems.

This process allows the standard library to continue to evolve over time, without locking in problematic design errors for extended periods of time. See **PEP 411** for more details.

provisional package
See *provisional API*.

Python 3000
Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated "Py3k".

Pythonic
An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a `for` statement. Many other languages don't have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
for i in range(len(food)):
    print(food[i])
```

As opposed to the cleaner, Pythonic method:

```
for piece in food:
    print(piece)
```

qualified name
A dotted name showing the "path" from a module's global scope to a class, function or method defined in that module, as defined in **PEP 3155**. For top-level functions and classes, the qualified name is the same as the object's name:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

When used to refer to modules, the *fully qualified name* means the entire dotted path to the module, including any parent packages, e.g. `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count
The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Some objects are *immortal* and have reference counts that are never modified, and therefore the objects are never deallocated. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. Programmers can call the `sys.getrefcount()` function to return the reference count for a particular object.

regular package
A traditional *package*, such as a directory containing an `__init__.py` file.

See also *namespace package*.

REPL
An acronym for the "read–eval–print loop", another name for the *interactive* interpreter shell.

__slots__
A declaration inside a class that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequence
An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `bytes`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`. For more documentation on sequence methods generally, see *Common Sequence Operations*.

set comprehension
A compact way to process all or part of the elements in an iterable and return a set with the results. `results = {c for c in 'abracadabra' if c not in 'abc'}` generates the set of strings `{'r', 'd'}`. See comprehensions.

single dispatch
A form of *generic function* dispatch where the implementation is chosen based on the type of a single argument.

slice
An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses *`slice`* objects internally.

soft deprecated
A soft deprecation can be used when using an API which should no longer be used to write new code, but it remains safe to continue using it in existing code. The API remains documented and tested, but will not be developed further (no enhancement).

The main difference between a "soft" and a (regular) "hard" deprecation is that the soft deprecation does not imply scheduling the removal of the deprecated API.

Another difference is that a soft deprecation does not issue a warning.

See PEP 387: Soft Deprecation.

special method
A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in specialnames.

statement
A statement is part of a suite (a "block" of code). A statement is either an *expression* or one of several constructs with a keyword, such as `if`, `while` or `for`.

static type checker
An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also *type hints* and the *`typing`* module.

strong reference
In Python's C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling `Py_INCREF()` when the reference is created and released with `Py_DECREF()` when the reference is deleted.

The `Py_NewRef()` function can be used to create a strong reference to an object. Usually, the `Py_DECREF()` function must be called on the strong reference before exiting the scope of the strong reference, to avoid leaking one reference.

See also *borrowed reference*.

text encoding
A string in Python is a sequence of Unicode code points (in range `U+0000`–`U+10FFFF`). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as "encoding", and recreating the string from the sequence of bytes is known as "decoding".

There are a variety of different text serialization *codecs*, which are collectively referred to as "text encodings".

text file
A *file object* able to read and write *`str`* objects. Often, a text file actually accesses a byte-oriented datastream and handles the *text encoding* automatically. Examples of text files are files opened in text mode (`'r'` or `'w'`), *`sys.stdin`*, *`sys.stdout`*, and instances of *`io.StringIO`*.

See also *binary file* for a file object able to read and write *bytes-like objects*.

triple-quoted string
A string which is bound by three instances of either a quotation mark (") or an apostrophe ('). While they don't provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They

allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

type
The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its *__class__* attribute or can be retrieved with `type(obj)`.

type alias
A synonym for a type, created by assigning the type to an identifier.

Type aliases are useful for simplifying *type hints*. For example:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

could be made more readable like this:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

See *typing* and **PEP 484**, which describe this functionality.

type hint
An *annotation* that specifies the expected type for a variable, a class attribute, or a function parameter or return value.

Type hints are optional and are not enforced by Python but they are useful to *static type checkers*. They can also aid IDEs with code completion and refactoring.

Type hints of global variables, class attributes, and functions, but not local variables, can be accessed using *typing.get_type_hints()*.

See *typing* and **PEP 484**, which describe this functionality.

universal newlines
A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See **PEP 278** and **PEP 3116**, as well as *bytes.splitlines()* for an additional use.

variable annotation
An *annotation* of a variable or a class attribute.

When annotating a variable or a class attribute, assignment is optional:

```
class C:
    field: 'annotation'
```

Variable annotations are usually used for *type hints*: for example this variable is expected to take *int* values:

```
count: int = 0
```

Variable annotation syntax is explained in section annassign.

See *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

virtual environment
A cooperatively isolated runtime environment that allows Python users and applications to install and upgrade

Python distribution packages without interfering with the behaviour of other Python applications running on the same system.

See also *venv*.

virtual machine
A computer defined entirely in software. Python's virtual machine executes the *bytecode* emitted by the bytecode compiler.

Zen of Python
Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing "`import this`" at the interactive prompt.

APPENDIX

B

ABOUT THESE DOCUMENTS

These documents are generated from reStructuredText sources by Sphinx, a document processor specifically written for the Python documentation.

Development of the documentation and its toolchain is an entirely volunteer effort, just like Python itself. If you want to contribute, please take a look at the reporting-bugs page for information on how to do so. New volunteers are always welcome!

Many thanks go to:

- Fred L. Drake, Jr., the creator of the original Python documentation toolset and writer of much of the content;
- the Docutils project for creating reStructuredText and the Docutils suite;
- Fredrik Lundh for his Alternative Python Reference project from which Sphinx got many good ideas.

B.1 Contributors to the Python Documentation

Many people have contributed to the Python language, the Python standard library, and the Python documentation. See Misc/ACKS in the Python source distribution for a partial list of contributors.

It is only with the input and contributions of the Python community that Python has such wonderful documentation – Thank You!

APPENDIX

C

HISTORY AND LICENSE

C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see https://www.zope.org/). In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 and above	2.1.1	2001-now	PSF	yes

Note

GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

C.2 Terms and conditions for accessing or otherwise using Python

Python software and documentation are licensed under the *PSF License Agreement*.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Agreement and the *Zero-Clause BSD license*.

Some software incorporated into Python is under different licenses. The licenses are listed with code falling under that license. See *Licenses and Acknowledgements for Incorporated Software* for an incomplete list of these licenses.

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.13.0rc1

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   3.13.0rc1 software in source or binary form and its associated␣
↪documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 3.13.0rc1 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.13.0rc1 alone or in any derivative␣
↪version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.13.0rc1 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python
   3.13.0rc1.

4. PSF is making Python 3.13.0rc1 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION␣
↪OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT␣
↪THE
   USE OF PYTHON 3.13.0rc1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.13.0rc1
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.13.0rc1, OR ANY␣
↪DERIVATIVE
```

```
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach␣
↪of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This␣
↪License
   Agreement does not grant permission to use PSF trademarks or trade name in␣
↪a
   trademark sense to endorse or promote products or services of Licensee, or␣
↪any
   third party.

8. By copying, installing or otherwise using Python 3.13.0rc1, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
```

(continues on next page)

(continued from previous page)

```
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
```

(continues on next page)

(continued from previous page)

```
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0rc1 DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The `_random` C extension underlying the *random* module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Sockets

The *socket* module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, https://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Asynchronous socket services

The `test.support.asynchat` and `test.support.asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Cookie management

The *http.cookies* module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Execution tracing

The *trace* module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode and UUdecode functions

The `uu` codec contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML Remote Procedure Calls

The *xmlrpc.client* module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
```

(continues on next page)

(continued from previous page)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

The `test.test_epoll` module contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

The *select* module contains the following notice for the kqueue interface:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
```

(continues on next page)

(continued from previous page)

```
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

The file `Python/pyhash.c` contains Marek Majkowski' implementation of Dan Bernstein's SipHash24 algorithm. It contains the following note:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod and dtoa

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
```

(continues on next page)

(continued from previous page)

```
 *
 ************************************************************/
```

C.3.12 OpenSSL

The modules *hashlib*, *posix* and *ssl* use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
```

(continues on next page)

(continued from previous page)

```
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
```

(continues on next page)

(continued from previous page)

```
      excluding those notices that do not pertain to any part of
      the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
      distribution, then any Derivative Works that You distribute must
      include a readable copy of the attribution notices contained
      within such NOTICE file, excluding those notices that do not
      pertain to any part of the Derivative Works, in at least one
      of the following places: within a NOTICE text file distributed
      as part of the Derivative Works; within the Source form or
      documentation, if provided along with the Derivative Works; or,
      within a display generated by the Derivative Works, if and
      wherever such third-party notices normally appear. The contents
      of the NOTICE file are for informational purposes only and
      do not modify the License. You may add Your own attribution
      notices within Derivative Works that You distribute, alongside
      or as an addendum to the NOTICE text from the Work, provided
      that such additional attribution notices cannot be construed
      as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
```

(continues on next page)

(continued from previous page)

```
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

The *pyexpat* extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

The _ctypes C extension underlying the *ctypes* module is built using an included copy of the libffi sources unless the build is configured --with-system-libffi:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

The *zlib* extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

The implementation of the hash table used by the *tracemalloc* is based on the cfuhash project:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

The `_decimal` C extension underlying the *decimal* module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
```

(continues on next page)

(continued from previous page)

```
THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N test suite

The C14N 2.0 test suite in the *test* package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at https://www.w3.org/TR/xml-c14n2-testcases/ and is distributed under the 3-clause BSD license:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

MIT License:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

C.3.20 asyncio

Parts of the *asyncio* module are incorporated from uvloop 0.16, which is distributed under the MIT license:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's "Global Unbounded Sequences" safe memory reclamation scheme in subr_smr.c. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
   disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

APPENDIX

D

COPYRIGHT

Python and this documentation is:

Copyright © 2001-2024 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

See *History and License* for complete license and permissions information.

BIBLIOGRAPHY

[Frie09] Friedl, Jeffrey. Mastering Regular Expressions. 3rd ed., O'Reilly Media, 2009. The third edition of the book no longer covers Python at all, but the first edition covered writing good regular expression patterns in great detail.

[C99] ISO/IEC 9899:1999. "Programming languages – C." A public draft of this standard is available at https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

PYTHON MODULE INDEX

_

__future__, 2085
__main__, 2030
_thread, 1062
_tkinter, 1666

a

abc, 2070
argparse, 787
array, 293
ast, 2165
asyncio, 1065
atexit, 2075

b

base64, 1365
bdb, 1927
binascii, 1369
bisect, 290
builtins, 2030
bz2, 595

c

calendar, 254
cmath, 355
cmd, 1651
code, 2115
codecs, 191
codeop, 2117
collections, 261
collections.abc, 280
colorsys, 1592
compileall, 2221
concurrent.futures, 1023
configparser, 645
contextlib, 2054
contextvars, 1058
copy, 312
copyreg, 541
cProfile, 1947
csv, 637
ctypes, 920
curses (*Unix*), 873
curses.ascii, 903
curses.panel, 907
curses.textpad, 901

d

dataclasses, 2043
datetime, 209
dbm, 546
dbm.dumb, 551
dbm.gnu (*Unix*), 549
dbm.ndbm (*Unix*), 550
dbm.sqlite3 (*All*), 548
decimal, 359
difflib, 156
dis, 2225
doctest, 1779

e

email, 1271
email.charset, 1324
email.contentmanager, 1301
email.encoders, 1326
email.errors, 1294
email.generator, 1284
email.header, 1321
email.headerregistry, 1295
email.iterators, 1329
email.message, 1272
email.mime, 1319
email.mime.application, 1320
email.mime.audio, 1320
email.mime.base, 1319
email.mime.image, 1320
email.mime.message, 1321
email.mime.multipart, 1319
email.mime.nonmultipart, 1319
email.mime.text, 1321
email.parser, 1280
email.policy, 1287
email.utils, 1327
encodings.idna, 207

encodings.mbcs, 208
encodings.utf_8_sig, 208
ensurepip, 1973
enum, 323
errno, 913

f

faulthandler, 1933
fcntl (*Unix*), 2274
filecmp, 498
fileinput, 489
fnmatch, 508
fractions, 388
ftplib, 1500
functools, 436

g

gc, 2086
getopt, 2287
getpass, 872
gettext, 1593
glob, 506
graphlib, 339
grp (*Unix*), 2269
gzip, 591

h

hashlib, 671
heapq, 286
hmac, 683
html, 1373
html.entities, 1378
html.parser, 1373
http, 1488
http.client, 1492
http.cookiejar, 1549
http.cookies, 1545
http.server, 1538

i

idlelib, 1718
imaplib, 1510
importlib, 2128
importlib.abc, 2131
importlib.machinery, 2138
importlib.metadata, 2155
importlib.resources, 2150
importlib.resources.abc, 2153
importlib.util, 2144
inspect, 2090
io, 760
ipaddress, 1572
itertools, 419

j

json, 1331
json.tool, 1340

k

keyword, 2212

l

linecache, 510
locale, 1601
logging, 823
logging.config, 844
logging.handlers, 857
lzma, 600

m

mailbox, 1341
marshal, 544
math, 346
mimetypes, 1362
mmap, 1263
modulefinder, 2125
msvcrt (*Windows*), 2251
multiprocessing, 970
multiprocessing.connection, 1003
multiprocessing.dummy, 1007
multiprocessing.managers, 993
multiprocessing.pool, 1000
multiprocessing.shared_memory, 1017
multiprocessing.sharedctypes, 991

n

netrc, 665
numbers, 343

o

operator, 446
optparse, 2289
os, 689
os.path, 482

p

pathlib, 455
pdb, 1935
pickle, 523
pickletools, 2248
pkgutil, 2121
platform, 908
plistlib, 667
poplib, 1507
posix (*Unix*), 2267
pprint, 314
profile, 1947

pstats, 1949
pty (*Unix*), 2272
pwd (*Unix*), 2268
py_compile, 2219
pyclbr, 2217
pydoc, 1774

q

queue, 1054
quopri, 1371

r

random, 392
re, 132
readline (*Unix*), 177
reprlib, 320
resource (*Unix*), 2277
rlcompleter, 181
runpy, 2126

s

sched, 1052
secrets, 685
select, 1242
selectors, 1249
shelve, 541
shlex, 1656
shutil, 510
signal, 1253
site, 2111
sitecustomize, 2112
smtplib, 1517
socket, 1175
socketserver, 1529
sqlite3, 553
ssl, 1204
stat, 491
statistics, 402
string, 121
stringprep, 175
struct, 183
subprocess, 1030
symtable, 2204
sys, 1991
sys.monitoring, 2017
sysconfig, 2023
syslog (*Unix*), 2281

t

tabnanny, 2216
tarfile, 618
tempfile, 500
termios (*Unix*), 2270
test, 1900
test.regrtest, 1902
test.support, 1903
test.support.bytecode_helper, 1915
test.support.import_helper, 1918
test.support.os_helper, 1916
test.support.script_helper, 1913
test.support.socket_helper, 1913
test.support.threading_helper, 1915
test.support.warnings_helper, 1919
textwrap, 169
threading, 955
time, 774
timeit, 1953
tkinter, 1663
tkinter.colorchooser (*Tk*), 1677
tkinter.commondialog (*Tk*), 1682
tkinter.dnd (*Tk*), 1685
tkinter.filedialog (*Tk*), 1679
tkinter.font (*Tk*), 1677
tkinter.messagebox (*Tk*), 1682
tkinter.scrolledtext (*Tk*), 1684
tkinter.simpledialog (*Tk*), 1679
tkinter.ttk, 1686
token, 2208
tokenize, 2212
tomllib, 664
trace, 1959
traceback, 2076
tracemalloc, 1961
tty (*Unix*), 2271
turtle, 1611
turtledemo, 1649
types, 305
typing, 1719

u

unicodedata, 173
unittest, 1803
unittest.mock, 1835
urllib, 1457
urllib.error, 1486
urllib.parse, 1477
urllib.request, 1457
urllib.response, 1476
urllib.robotparser, 1487
usercustomize, 2113
uuid, 1524

v

venv, 1975

w

warnings, 2036
wave, 1589

weakref, 297
webbrowser, 1443
winreg (*Windows*), 2254
winsound (*Windows*), 2263
wsgiref, 1446
wsgiref.handlers, 1451
wsgiref.headers, 1448
wsgiref.simple_server, 1449
wsgiref.types, 1455
wsgiref.util, 1446
wsgiref.validate, 1450

X

xml, 1379
xml.dom, 1401
xml.dom.minidom, 1412
xml.dom.pulldom, 1417
xml.etree.ElementInclude, 1393
xml.etree.ElementTree, 1381
xml.parsers.expat, 1432
xml.parsers.expat.errors, 1440
xml.parsers.expat.model, 1439
xml.sax, 1419
xml.sax.handler, 1421
xml.sax.saxutils, 1426
xml.sax.xmlreader, 1428
xmlrpc.client, 1558
xmlrpc.server, 1566

Z

zipapp, 1985
zipfile, 606
zipimport, 2119
zlib, 587
zoneinfo, 248

INDEX

Non-alphabetical

??
    in regular expressions, 134
..
    in pathnames, 757
..., **2321**
    ellipsis literal, 33, 103
    in doctests, 1786
    interpreter prompt, 1783, 2010
    placeholder, 173, 315, 320
{} (*curly brackets*)
    in regular expressions, 134
    in string formatting, 123
. (*dot*)
    in glob-style wildcards, 506
    in pathnames, 757, 758
    in printf-style formatting, 61, 77
    in regular expressions, 133
    in string formatting, 123
! (*exclamation*)
    in a command interpreter, 1652
    in curses module, 906
    in glob-style wildcards, 506, 508
    in string formatting, 123
    in struct format strings, 184
- (*minus*)
    binary operator, 37
    in doctests, 1788
    in glob-style wildcards, 506, 508
    in printf-style formatting, 61, 78
    in regular expressions, 135
    in string formatting, 125
    unary operator, 37
! (*pdb command*), 1943
? (*question mark*)
    in a command interpreter, 1652
    in argparse module, 802
    in AST grammar, 2168
    in glob-style wildcards, 506, 508
    in regular expressions, 134
    in SQL statements, 571
    in struct format strings, 186, 187
    replacement character, 194
(*hash*)
    comment, 2111
    in doctests, 1788
    in printf-style formatting, 61, 78
    in regular expressions, 142
    in string formatting, 125
$ (*dollar*)
    environment variables expansion, 484
    in regular expressions, 133
    in template strings, 130
    interpolation in configuration files, 650
% (*percent*)
    datetime format, 243, 779, 781
    environment variables expansion (*Windows*), 484, 2256
    interpolation in configuration files, 650
    operator, 37
    printf-style formatting, 60, 77
& (*ampersand*)
    operator, 38
(?
    in regular expressions, 135
(?!
    in regular expressions, 137
(?#
    in regular expressions, 137
(?(
    in regular expressions, 138
() (*parentheses*)
    in printf-style formatting, 61, 77
    in regular expressions, 135
(?:
    in regular expressions, 136
(?<!
    in regular expressions, 138
(?<=
    in regular expressions, 137
(?=
    in regular expressions, 137

(?P<
    in regular expressions, 137
(?P=
    in regular expressions, 137
*?
    in regular expressions, 134
* (*asterisk*)
    in argparse module, 803
    in AST grammar, 2168
    in glob-style wildcards, 506, 508
    in printf-style formatting, 61, 77
    in regular expressions, 134
    operator, 37
**
    in glob-style wildcards, 506
    operator, 37
*+
    in regular expressions, 134
+?
    in regular expressions, 134
?+
    in regular expressions, 134
+ (*plus*)
    binary operator, 37
    in argparse module, 803
    in doctests, 1788
    in printf-style formatting, 61, 78
    in regular expressions, 134
    in string formatting, 125
    unary operator, 37
++
    in regular expressions, 134
, (*comma*)
    in string formatting, 126
-
    python--m-py_compile command line option, 2220
/ (*slash*)
    in pathnames, 757
    operator, 37
//
    operator, 37
2-digit years, 775
: (*colon*)
    in SQL statements, 571
    in string formatting, 123
    path separator (*POSIX*), 758
; (*semicolon*), 758
< (*less*)
    in string formatting, 125
    in struct format strings, 184
    operator, 36
<<
    operator, 38
<=
    operator, 36
<BLANKLINE>, 1786
<file>
    python--m-py_compile command line option, 2220
!=
    operator, 36
= (*equals*)
    in string formatting, 125
    in struct format strings, 184
==
    operator, 36
> (*greater*)
    in string formatting, 125
    in struct format strings, 184
    operator, 36
>=
    operator, 36
>>
    operator, 38
>>>, **2321**
    interpreter prompt, 1783, 2010
@ (*at*)
    in struct format strings, 184
[] (*square brackets*)
    in glob-style wildcards, 506, 508
    in regular expressions, 135
    in string formatting, 123
\ (*backslash*)
    escape sequence, 194
    in pathnames (*Windows*), 757
    in regular expressions, 134, 135, 138
\\
    in regular expressions, 139
\A
    in regular expressions, 138
\a
    in regular expressions, 139
\B
    in regular expressions, 138
\b
    in regular expressions, 138, 139
\D
    in regular expressions, 139
\d
    in regular expressions, 138
\f
    in regular expressions, 139
\g
    in regular expressions, 144
\N
    escape sequence, 194
    in regular expressions, 139

\n
    in regular expressions, 139
\r
    in regular expressions, 139
\S
    in regular expressions, 139
\s
    in regular expressions, 139
\t
    in regular expressions, 139
\U
    escape sequence, 194
    in regular expressions, 139
\u
    escape sequence, 194
    in regular expressions, 139
\v
    in regular expressions, 139
\W
    in regular expressions, 139
\w
    in regular expressions, 139
\x
    escape sequence, 194
    in regular expressions, 139
\Z
    in regular expressions, 139
^ (*caret*)
    in curses module, 906
    in regular expressions, 133, 135
    in string formatting, 125
    marker, 1785, 2077
    operator, 38
_ (*underscore*)
    gettext, 1593
    in string formatting, 126
__abs__() (*in module operator*), 447
__add__() (*in module operator*), 447
__and__() (*enum.Flag method*), 332
__and__() (*in module operator*), 447
__args__ (*genericalias attribute*), 98
__bases__ (*class attribute*), 103
__bound__ (*typing.TypeVar attribute*), 1745
__breakpointhook__ (*in module sys*), 1996
__bytes__() (*email.message.EmailMessage method*), 1273
__bytes__() (*email.message.Message method*), 1312
__call__() (*email.headerregistry.HeaderRegistry method*), 1300
__call__() (*enum.EnumType method*), 325
__call__() (*in module operator*), 449
__call__() (*weakref.finalize method*), 300
__callback__ (*weakref.ref attribute*), 298
__cause__ (*BaseException attribute*), 107
__cause__ (*exception attribute*), 107
__cause__ (*traceback.TracebackException attribute*), 2079
__ceil__() (*fractions.Fraction method*), 391
__class__ (*instance attribute*), 103
__class__ (*unittest.mock.Mock attribute*), 1846
__code__ (*function object attribute*), 102
__concat__() (*in module operator*), 448
__constraints__ (*typing.TypeVar attribute*), 1745
__contains__() (*email.message.EmailMessage method*), 1274
__contains__() (*email.message.Message method*), 1313
__contains__() (*enum.EnumType method*), 326
__contains__() (*enum.Flag method*), 331
__contains__() (*in module operator*), 448
__contains__() (*mailbox.Mailbox method*), 1343
__context__ (*BaseException attribute*), 107
__context__ (*exception attribute*), 107
__context__ (*traceback.TracebackException attribute*), 2079
__contravariant__ (*typing.TypeVar attribute*), 1745
__copy__() (*copy protocol*), 313
__covariant__ (*typing.TypeVar attribute*), 1745
__debug__ (*built-in variable*), 33
__deepcopy__() (*copy protocol*), 313
__default__ (*typing.ParamSpec attribute*), 1748
__default__ (*typing.TypeVar attribute*), 1745
__default__ (*typing.TypeVarTuple attribute*), 1747
__del__() (*io.IOBase method*), 765
__delitem__() (*email.message.EmailMessage method*), 1274
__delitem__() (*email.message.Message method*), 1314
__delitem__() (*in module operator*), 449
__delitem__() (*mailbox.Mailbox method*), 1342
__delitem__() (*mailbox.MH method*), 1349
__dict__ (*object attribute*), 103
__dir__() (*enum.Enum method*), 328
__dir__() (*enum.EnumType method*), 326
__dir__() (*unittest.mock.Mock method*), 1842
__displayhook__ (*in module sys*), 1996
__doc__ (*types.ModuleType attribute*), 308
__enter__() (*contextmanager method*), 94
__enter__() (*winreg.PyHKEY method*), 2263
__eq__() (*email.charset.Charset method*), 1325
__eq__() (*email.header.Header method*), 1323
__eq__() (*in module operator*), 446
__eq__() (*instance method*), 36
__eq__() (*memoryview method*), 80
__excepthook__ (*in module sys*), 1996
__excepthook__ (*in module threading*), 956
__exit__() (*contextmanager method*), 94
__exit__() (*winreg.PyHKEY method*), 2263

__floor__() (*fractions.Fraction method*), 390
__floordiv__() (*in module operator*), 447
__format__, 15
__format__() (*datetime.date method*), 219
__format__() (*datetime.datetime method*), 230
__format__() (*datetime.time method*), 235
__format__() (*enum.Enum method*), 330
__format__() (*fractions.Fraction method*), 391
__format__() (*ipaddress.IPv4Address method*), 1575
__format__() (*ipaddress.IPv6Address method*), 1577
__fspath__() (*os.PathLike method*), 692
__future__, **2326**
  module, 2085
__ge__() (*in module operator*), 446
__ge__() (*instance method*), 36
__getitem__() (*email.headerregistry.HeaderRegistry method*), 1300
__getitem__() (*email.message.EmailMessage method*), 1274
__getitem__() (*email.message.Message method*), 1314
__getitem__() (*enum.EnumType method*), 326
__getitem__() (*in module operator*), 449
__getitem__() (*mailbox.Mailbox method*), 1343
__getitem__() (*re.Match method*), 149
__getnewargs__() (*object method*), 530
__getnewargs_ex__() (*object method*), 530
__getstate__() (*copy protocol*), 535
__getstate__() (*object method*), 530
__gt__() (*in module operator*), 446
__gt__() (*instance method*), 36
__iadd__() (*in module operator*), 452
__iand__() (*in module operator*), 452
__iconcat__() (*in module operator*), 452
__ifloordiv__() (*in module operator*), 452
__ilshift__() (*in module operator*), 452
__imatmul__() (*in module operator*), 453
__imod__() (*in module operator*), 452
__import__()
  built-in function, 31
__import__() (*in module importlib*), 2130
__imul__() (*in module operator*), 452
__index__() (*in module operator*), 447
__infer_variance__ (*typing.TypeVar attribute*), 1745
__init__() (*asyncio.Future method*), 1162
__init__() (*asyncio.Task method*), 1162
__init__() (*difflib.HtmlDiff method*), 157
__init__() (*enum.Enum method*), 328
__init__() (*logging.Handler method*), 830
__init_subclass__() (*enum.Enum method*), 329
__interactivehook__ (*in module sys*), 2006
__inv__() (*in module operator*), 447
__invert__() (*in module operator*), 447
__ior__() (*in module operator*), 453
__ipow__() (*in module operator*), 453
__irshift__() (*in module operator*), 453
__isub__() (*in module operator*), 453
__iter__() (*container method*), 44
__iter__() (*enum.EnumType method*), 326
__iter__() (*iterator method*), 44
__iter__() (*mailbox.Mailbox method*), 1342
__iter__() (*unittest.TestSuite method*), 1825
__itruediv__() (*in module operator*), 453
__ixor__() (*in module operator*), 453
__le__() (*in module operator*), 446
__le__() (*instance method*), 36
__len__() (*email.message.EmailMessage method*), 1274
__len__() (*email.message.Message method*), 1313
__len__() (*enum.EnumType method*), 326
__len__() (*mailbox.Mailbox method*), 1343
__loader__ (*types.ModuleType attribute*), 308
__lshift__() (*in module operator*), 447
__lt__() (*in module operator*), 446
__lt__() (*instance method*), 36
__main__
  module, 2030, 2126, 2127
__matmul__() (*in module operator*), 448
__members__ (*enum.EnumType attribute*), 326
__missing__(), 90
__missing__() (*collections.defaultdict method*), 271
__mod__() (*in module operator*), 447
__module__ (*typing.NewType attribute*), 1752
__module__ (*typing.TypeAliasType attribute*), 1749
__mro__ (*class attribute*), 103
__mul__() (*in module operator*), 448
__mutable_keys__ (*typing.TypedDict attribute*), 1756
__name__ (*definition attribute*), 103
__name__ (*types.ModuleType attribute*), 308
__name__ (*typing.NewType attribute*), 1752
__name__ (*typing.ParamSpec attribute*), 1748
__name__ (*typing.TypeAliasType attribute*), 1749
__name__ (*typing.TypeVar attribute*), 1744
__name__ (*typing.TypeVarTuple attribute*), 1747
__ne__() (*email.charset.Charset method*), 1325
__ne__() (*email.header.Header method*), 1323
__ne__() (*in module operator*), 446
__ne__() (*instance method*), 36
__neg__() (*in module operator*), 448
__new__() (*enum.Enum method*), 329
__next__() (*csv.csvreader method*), 643
__next__() (*iterator method*), 44
__not__() (*in module operator*), 447
__notes__ (*BaseException attribute*), 108
__notes__ (*traceback.TracebackException attribute*), 2079
__optional_keys__ (*typing.TypedDict attribute*), 1756

__or__() (*enum.Flag method*), 332
__or__() (*in module operator*), 448
__origin__ (*genericalias attribute*), 98
__package__ (*types.ModuleType attribute*), 308
__parameters__ (*genericalias attribute*), 98
__pos__() (*in module operator*), 448
__post_init__() (*in module dataclasses*), 2050
__pow__() (*in module operator*), 448
__qualname__ (*definition attribute*), 103
__readonly_keys__ (*typing.TypedDict attribute*), 1756
__reduce__() (*object method*), 531
__reduce_ex__() (*object method*), 532
__replace__() (*replace protocol*), 313
__repr__() (*enum.Enum method*), 329
__repr__() (*multiprocessing.managers.BaseProxy method*), 999
__repr__() (*netrc.netrc method*), 666
__required_keys__ (*typing.TypedDict attribute*), 1756
__reversed__() (*enum.EnumType method*), 327
__round__() (*fractions.Fraction method*), 391
__rshift__() (*in module operator*), 448
__setitem__() (*email.message.EmailMessage method*), 1274
__setitem__() (*email.message.Message method*), 1314
__setitem__() (*in module operator*), 449
__setitem__() (*mailbox.Mailbox method*), 1342
__setitem__() (*mailbox.Maildir method*), 1347
__setstate__() (*copy protocol*), 535
__setstate__() (*object method*), 531
__slots__, **2334**
__spec__ (*types.ModuleType attribute*), 309
__static_attributes__ (*class attribute*), 104
__stderr__ (*in module sys*), 2014
__stdin__ (*in module sys*), 2014
__stdout__ (*in module sys*), 2014
__str__() (*datetime.date method*), 218
__str__() (*datetime.datetime method*), 229
__str__() (*datetime.time method*), 235
__str__() (*email.charset.Charset method*), 1325
__str__() (*email.header.Header method*), 1323
__str__() (*email.headerregistry.Address method*), 1300
__str__() (*email.headerregistry.Group method*), 1301
__str__() (*email.message.EmailMessage method*), 1273
__str__() (*email.message.Message method*), 1311
__str__() (*enum.Enum method*), 330
__str__() (*multiprocessing.managers.BaseProxy method*), 999
__sub__() (*in module operator*), 448
__subclasses__() (*class method*), 104
__subclasshook__() (*abc.ABCMeta method*), 2071
__supertype__ (*typing.NewType attribute*), 1752
__suppress_context__ (*BaseException attribute*), 107
__suppress_context__ (*exception attribute*), 107
__suppress_context__ (*traceback.TracebackException attribute*), 2079
__total__ (*typing.TypedDict attribute*), 1755
__traceback__ (*BaseException attribute*), 108
__truediv__() (*importlib.abc.Traversable method*), 2137
__truediv__() (*importlib.resources.abc.Traversable method*), 2154
__truediv__() (*in module operator*), 448
__type_params__ (*definition attribute*), 103
__type_params__ (*typing.TypeAliasType attribute*), 1750
__unpacked__ (*genericalias attribute*), 99
__unraisablehook__ (*in module sys*), 1996
__value__ (*typing.TypeAliasType attribute*), 1750
__version__ (*in module curses*), 888
__xor__() (*enum.Flag method*), 332
__xor__() (*in module operator*), 448
_add_alias_() (*enum.EnumType method*), 327
_add_value_alias_() (*enum.EnumType method*), 327
align (*ctypes.Structure attribute*), 953
anonymous (*ctypes.Structure attribute*), 953
_asdict() (*collections.somenamedtuple method*), 274
_b_base_ (*ctypes._CData attribute*), 949
_b_needsfree_ (*ctypes._CData attribute*), 949
_callmethod() (*multiprocessing.managers.BaseProxy method*), 999
_CData (*class in ctypes*), 948
_clear_internal_caches() (*in module sys*), 1993
_clear_type_cache() (*in module sys*), 1993
_current_exceptions() (*in module sys*), 1993
_current_frames() (*in module sys*), 1993
_debugmallocstats() (*in module sys*), 1994
_emscripten_info (*in module sys*), 1995
_enablelegacywindowsfsencoding() (*in module sys*), 2013
_enter_task() (*in module asyncio*), 1162
_exit() (*in module os*), 742
_Feature (*class in __future__*), 2085
_field_defaults (*collections.somenamedtuple attribute*), 275
_field_types (*ast.AST attribute*), 2169
_fields (*ast.AST attribute*), 2168
_fields (*collections.somenamedtuple attribute*), 275
fields (*ctypes.Structure attribute*), 952
_flush() (*wsgiref.handlers.BaseHandler method*), 1452
_FuncPtr (*class in ctypes*), 942
_generate_next_value_() (*enum.Enum method*), 328

_get_child_mock() (*unittest.mock.Mock method*), 1842
_get_preferred_schemes() (*in module sysconfig*), 2027
_getframe() (*in module sys*), 2003
_getframemodulename() (*in module sys*), 2003
_getvalue() (*multiprocessing.managers.BaseProxy method*), 999
_handle (*ctypes.PyDLL attribute*), 941
ignore (*enum.Enum attribute*), 327
_incompatible_extension_module_restrictions() (*in module importlib.util*), 2146
_is_gil_enabled() (*in module sys*), 2006
_is_interned() (*in module sys*), 2007
_leave_task() (*in module asyncio*), 1162
length (*ctypes.Array attribute*), 954
_locale
    module, 1601
_log (*logging.LoggerAdapter attribute*), 838
_make() (*collections.somenamedtuple class method*), 274
_makeResult() (*unittest.TextTestRunner method*), 1831
missing() (*enum.Enum method*), 329
_name (*ctypes.PyDLL attribute*), 941
name (*enum.Enum attribute*), 327
_numeric_repr_() (*enum.Flag method*), 333
_objects (*ctypes._CData attribute*), 949
order (*enum.Enum attribute*), 327
pack (*ctypes.Structure attribute*), 953
_parse() (*gettext.NullTranslations method*), 1595
_Pointer (*class in ctypes*), 954
_register_task() (*in module asyncio*), 1162
_replace() (*collections.somenamedtuple method*), 274
_setroot() (*xml.etree.ElementTree.ElementTree method*), 1396
_SimpleCData (*class in ctypes*), 949
_structure() (*in module email.iterators*), 1330
_thread
    module, 1062
_tkinter
    module, 1666
type (*ctypes._Pointer attribute*), 954
type (*ctypes.Array attribute*), 954
_unregister_task() (*in module asyncio*), 1162
value (*enum.Enum attribute*), 327
_write() (*wsgiref.handlers.BaseHandler method*), 1452
_xoptions (*in module sys*), 2017
| (*vertical bar*)
    in regular expressions, 135
    operator, 38
~ (*tilde*)
    home directory expansion, 483
    operator, 38

A

-a
    ast command line option, 2203
    pickletools command line option, 2249
A (*in module re*), 140
a2b_base64() (*in module binascii*), 1369
a2b_hex() (*in module binascii*), 1370
a2b_qp() (*in module binascii*), 1369
a2b_uu() (*in module binascii*), 1369
a85decode() (*in module base64*), 1367
a85encode() (*in module base64*), 1367
A_ALTCHARSET (*in module curses*), 889
A_ATTRIBUTES (*in module curses*), 890
A_BLINK (*in module curses*), 889
A_BOLD (*in module curses*), 889
A_CHARTEXT (*in module curses*), 890
A_COLOR (*in module curses*), 890
A_DIM (*in module curses*), 889
A_HORIZONTAL (*in module curses*), 889
A_INVIS (*in module curses*), 889
A_ITALIC (*in module curses*), 889
A_LEFT (*in module curses*), 889
A_LOW (*in module curses*), 889
A_NORMAL (*in module curses*), 889
A_PROTECT (*in module curses*), 889
A_REVERSE (*in module curses*), 889
A_RIGHT (*in module curses*), 889
A_STANDOUT (*in module curses*), 889
A_TOP (*in module curses*), 889
A_UNDERLINE (*in module curses*), 889
A_VERTICAL (*in module curses*), 889
abc
    module, 2070
ABC (*class in abc*), 2070
ABCMeta (*class in abc*), 2070
ABDAY_1 (*in module locale*), 1604
ABDAY_2 (*in module locale*), 1604
ABDAY_3 (*in module locale*), 1604
ABDAY_4 (*in module locale*), 1604
ABDAY_5 (*in module locale*), 1604
ABDAY_6 (*in module locale*), 1604
ABDAY_7 (*in module locale*), 1604
abiflags (*in module sys*), 1991
ABMON_1 (*in module locale*), 1605
ABMON_2 (*in module locale*), 1605
ABMON_3 (*in module locale*), 1605
ABMON_4 (*in module locale*), 1605
ABMON_5 (*in module locale*), 1605
ABMON_6 (*in module locale*), 1605
ABMON_7 (*in module locale*), 1605
ABMON_8 (*in module locale*), 1605
ABMON_9 (*in module locale*), 1605
ABMON_10 (*in module locale*), 1605
ABMON_11 (*in module locale*), 1605

ABMON_12 (*in module locale*), 1605
ABORT (*in module tkinter.messagebox*), 1683
abort() (*asyncio.Barrier method*), 1102
abort() (*asyncio.DatagramTransport method*), 1146
abort() (*asyncio.WriteTransport method*), 1145
abort() (*ftplib.FTP method*), 1502
abort() (*in module os*), 740
abort() (*threading.Barrier method*), 969
abort_clients() (*asyncio.Server method*), 1133
ABORTRETRYIGNORE (*in module tkinter.messagebox*), 1684
above() (*curses.panel.Panel method*), 907
ABOVE_NORMAL_PRIORITY_CLASS (*in module subprocess*), 1044
abs()
    built-in function, 5
abs() (*decimal.Context method*), 374
abs() (*in module operator*), 447
absolute() (*pathlib.Path method*), 469
AbsoluteLinkError, 620
AbsolutePathError, 620
abspath() (*in module os.path*), 482
abstract base class, **2321**
AbstractAsyncContextManager (*class in contextlib*), 2055
AbstractBasicAuthHandler (*class in urllib.request*), 1461
AbstractChildWatcher (*class in asyncio*), 1158
abstractclassmethod() (*in module abc*), 2073
AbstractContextManager (*class in contextlib*), 2055
AbstractDigestAuthHandler (*class in urllib.request*), 1461
AbstractEventLoop (*class in asyncio*), 1135
AbstractEventLoopPolicy (*class in asyncio*), 1157
abstractmethod() (*in module abc*), 2072
abstractproperty() (*in module abc*), 2073
AbstractSet (*class in typing*), 1769
abstractstaticmethod() (*in module abc*), 2073
accept() (*multiprocessing.connection.Listener method*), 1003
accept() (*socket.socket method*), 1192
access() (*in module os*), 713
accumulate() (*in module itertools*), 421
ACK (*in module curses.ascii*), 903
aclose() (*contextlib.AsyncExitStack method*), 2063
aclosing() (*in module contextlib*), 2057
acos() (*in module cmath*), 356
acos() (*in module math*), 352
acosh() (*in module cmath*), 357
acosh() (*in module math*), 353
acquire() (*_thread.lock method*), 1063
acquire() (*asyncio.Condition method*), 1099
acquire() (*asyncio.Lock method*), 1097
acquire() (*asyncio.Semaphore method*), 1101
acquire() (*logging.Handler method*), 830
acquire() (*multiprocessing.Lock method*), 988
acquire() (*multiprocessing.RLock method*), 989
acquire() (*threading.Condition method*), 964
acquire() (*threading.Lock method*), 961
acquire() (*threading.RLock method*), 962
acquire() (*threading.Semaphore method*), 966
ACS_BBSS (*in module curses*), 897
ACS_BLOCK (*in module curses*), 897
ACS_BOARD (*in module curses*), 897
ACS_BSBS (*in module curses*), 897
ACS_BSSB (*in module curses*), 897
ACS_BSSS (*in module curses*), 897
ACS_BTEE (*in module curses*), 897
ACS_BULLET (*in module curses*), 897
ACS_CKBOARD (*in module curses*), 897
ACS_DARROW (*in module curses*), 897
ACS_DEGREE (*in module curses*), 897
ACS_DIAMOND (*in module curses*), 897
ACS_GEQUAL (*in module curses*), 897
ACS_HLINE (*in module curses*), 897
ACS_LANTERN (*in module curses*), 898
ACS_LARROW (*in module curses*), 898
ACS_LEQUAL (*in module curses*), 898
ACS_LLCORNER (*in module curses*), 898
ACS_LRCORNER (*in module curses*), 898
ACS_LTEE (*in module curses*), 898
ACS_NEQUAL (*in module curses*), 898
ACS_PI (*in module curses*), 898
ACS_PLMINUS (*in module curses*), 898
ACS_PLUS (*in module curses*), 898
ACS_RARROW (*in module curses*), 898
ACS_RTEE (*in module curses*), 898
ACS_S1 (*in module curses*), 898
ACS_S3 (*in module curses*), 898
ACS_S7 (*in module curses*), 898
ACS_S9 (*in module curses*), 899
ACS_SBBS (*in module curses*), 899
ACS_SBSB (*in module curses*), 899
ACS_SBSS (*in module curses*), 899
ACS_SSBB (*in module curses*), 899
ACS_SSBS (*in module curses*), 899
ACS_SSSB (*in module curses*), 899
ACS_SSSS (*in module curses*), 899
ACS_STERLING (*in module curses*), 899
ACS_TTEE (*in module curses*), 899
ACS_UARROW (*in module curses*), 899
ACS_ULCORNER (*in module curses*), 899
ACS_URCORNER (*in module curses*), 899
ACS_VLINE (*in module curses*), 899
Action (*class in argparse*), 810
action (*optparse.Option attribute*), 2303
ACTIONS (*optparse.Option attribute*), 2316

activate_stack_trampoline() (*in module sys*), 2013
active_children() (*in module multiprocessing*), 984
active_count() (*in module threading*), 955
actual() (*tkinter.font.Font method*), 1678
Add (*class in ast*), 2174
add() (*decimal.Context method*), 374
add() (*frozenset method*), 88
add() (*graphlib.TopologicalSorter method*), 340
add() (*in module operator*), 447
add() (*mailbox.Mailbox method*), 1342
add() (*mailbox.Maildir method*), 1347
add() (*pstats.Stats method*), 1949
add() (*tarfile.TarFile method*), 625
add() (*tkinter.ttk.Notebook method*), 1693
add_alias() (*in module email.charset*), 1326
add_alternative() (*email.message.EmailMessage method*), 1279
add_argument() (*argparse.ArgumentParser method*), 799
add_argument_group() (*argparse.ArgumentParser method*), 818
add_attachment() (*email.message.EmailMessage method*), 1279
add_cgi_vars() (*wsgiref.handlers.BaseHandler method*), 1453
add_charset() (*in module email.charset*), 1325
add_child_handler() (*asyncio.AbstractChildWatcher method*), 1158
add_codec() (*in module email.charset*), 1326
add_cookie_header() (*http.cookiejar.CookieJar method*), 1551
add_dll_directory() (*in module os*), 740
add_done_callback() (*asyncio.Future method*), 1140
add_done_callback() (*asyncio.Task method*), 1085
add_done_callback() (*concurrent.futures.Future method*), 1028
add_fallback() (*gettext.NullTranslations method*), 1596
add_flag() (*mailbox.Maildir method*), 1346
add_flag() (*mailbox.MaildirMessage method*), 1353
add_flag() (*mailbox.mboxMessage method*), 1355
add_flag() (*mailbox.MMDFMessage method*), 1359
add_folder() (*mailbox.Maildir method*), 1345
add_folder() (*mailbox.MH method*), 1349
add_get_handler() (*email.contentmanager.ContentManager method*), 1302
add_handler() (*urllib.request.OpenerDirector method*), 1464
add_header() (*email.message.EmailMessage method*), 1275
add_header() (*email.message.Message method*), 1314
add_header() (*urllib.request.Request method*), 1463
add_header() (*wsgiref.headers.Headers method*), 1449
add_history() (*in module readline*), 179
add_label() (*mailbox.BabylMessage method*), 1357
add_mutually_exclusive_group() (*argparse.ArgumentParser method*), 819
add_note() (*BaseException method*), 108
add_option() (*optparse.OptionParser method*), 2302
add_parent() (*urllib.request.BaseHandler method*), 1465
add_password() (*urllib.request.HTTPPasswordMgr method*), 1468
add_password() (*urllib.request.HTTPPasswordMgrWithPriorAuth method*), 1468
add_reader() (*asyncio.loop method*), 1124
add_related() (*email.message.EmailMessage method*), 1279
add_section() (*configparser.ConfigParser method*), 659
add_section() (*configparser.RawConfigParser method*), 662
add_sequence() (*mailbox.MHMessage method*), 1356
add_set_handler() (*email.contentmanager.ContentManager method*), 1302
add_signal_handler() (*asyncio.loop method*), 1127
add_subparsers() (*argparse.ArgumentParser method*), 814
add_type() (*in module mimetypes*), 1363
add_type() (*mimetypes.MimeTypes method*), 1365
add_unredirected_header() (*urllib.request.Request method*), 1463
add_writer() (*asyncio.loop method*), 1124
addAsyncCleanup() (*unittest.IsolatedAsyncioTestCase method*), 1823
addaudithook() (*in module sys*), 1991
addch() (*curses.window method*), 881
addClassCleanup() (*unittest.TestCase class method*), 1822
addCleanup() (*unittest.TestCase method*), 1822
addcomponent() (*turtle.Shape method*), 1645
addDuration() (*unittest.TestResult method*), 1830
addError() (*unittest.TestResult method*), 1829
addExpectedFailure() (*unittest.TestResult method*), 1830
addFailure() (*unittest.TestResult method*), 1829
addfile() (*tarfile.TarFile method*), 625
addFilter() (*logging.Handler method*), 831
addFilter() (*logging.Logger method*), 828
addHandler() (*logging.Logger method*), 829
addinfourl (*class in urllib.response*), 1476

addLevelName() (*in module logging*), 840
addModuleCleanup() (*in module unittest*), 1834
addnstr() (*curses.window method*), 881
AddPackagePath() (*in module modulefinder*), 2125
addr_spec (*email.headerregistry.Address attribute*), 1300
Address (*class in email.headerregistry*), 1300
address (*email.headerregistry.SingleAddressHeader attribute*), 1298
address (*multiprocessing.connection.Listener attribute*), 1004
address (*multiprocessing.managers.BaseManager attribute*), 994
address_exclude() (*ipaddress.IPv4Network method*), 1581
address_exclude() (*ipaddress.IPv6Network method*), 1583
address_family (*socketserver.BaseServer attribute*), 1532
address_string() (*http.server.BaseHTTPRequestHandler method*), 1541
addresses (*email.headerregistry.AddressHeader attribute*), 1297
addresses (*email.headerregistry.Group attribute*), 1301
AddressHeader (*class in email.headerregistry*), 1297
addressof() (*in module ctypes*), 946
AddressValueError, 1587
addshape() (*in module turtle*), 1642
addsitedir() (*in module site*), 2113
addSkip() (*unittest.TestResult method*), 1830
addstr() (*curses.window method*), 881
addSubTest() (*unittest.TestResult method*), 1830
addSuccess() (*unittest.TestResult method*), 1830
addTest() (*unittest.TestSuite method*), 1824
addTests() (*unittest.TestSuite method*), 1825
addTypeEqualityFunc() (*unittest.TestCase method*), 1820
addUnexpectedSuccess() (*unittest.TestResult method*), 1830
adjust_int_max_str_digits() (*in module test.support*), 1912
adjusted() (*decimal.Decimal method*), 364
adler32() (*in module zlib*), 587
AF_ALG (*in module socket*), 1182
AF_CAN (*in module socket*), 1180
AF_DIVERT (*in module socket*), 1181
AF_HYPERV (*in module socket*), 1183
AF_INET (*in module socket*), 1179
AF_INET6 (*in module socket*), 1179
AF_LINK (*in module socket*), 1183
AF_PACKET (*in module socket*), 1182
AF_QIPCRTR (*in module socket*), 1183
AF_RDS (*in module socket*), 1182
AF_UNIX (*in module socket*), 1179
AF_UNSPEC (*in module socket*), 1179
AF_VSOCK (*in module socket*), 1182
aiter()
    built-in function, 6
alarm() (*in module signal*), 1257
ALERT_DESCRIPTION_HANDSHAKE_FAILURE (*in module ssl*), 1216
ALERT_DESCRIPTION_INTERNAL_ERROR (*in module ssl*), 1216
AlertDescription (*class in ssl*), 1216
algorithm (*sys.hash_info attribute*), 2005
algorithms_available (*in module hashlib*), 673
algorithms_guaranteed (*in module hashlib*), 673
Alias
    Generic, 95
alias (*class in ast*), 2183
alias (*pdb command*), 1943
alignment() (*in module ctypes*), 946
alive (*weakref.finalize attribute*), 300
all()
    built-in function, 6
ALL_COMPLETED (*in module asyncio*), 1081
ALL_COMPLETED (*in module concurrent.futures*), 1029
all_errors (*in module ftplib*), 1506
all_features (*in module xml.sax.handler*), 1422
all_frames (*tracemalloc.Filter attribute*), 1969
all_properties (*in module xml.sax.handler*), 1423
all_suffixes() (*in module importlib.machinery*), 2139
all_tasks() (*in module asyncio*), 1084
allocate_lock() (*in module _thread*), 1062
allow_reuse_address (*socketserver.BaseServer attribute*), 1532
allowed_domains() (*http.cookiejar.DefaultCookiePolicy method*), 1555
alt() (*in module curses.ascii*), 906
ALT_DIGITS (*in module locale*), 1606
altsep (*in module os*), 757
altzone (*in module time*), 786
ALWAYS_EQ (*in module test.support*), 1905
ALWAYS_TYPED_ACTIONS (*optparse.Option attribute*), 2316
AmbiguousOptionError, 2318
AMPER (*in module token*), 2209
AMPEREQUAL (*in module token*), 2210
Anchor (*class in importlib.resources*), 2150
anchor (*pathlib.PurePath attribute*), 461
and
    operator, 35
And (*class in ast*), 2175
and_() (*in module operator*), 447
android_ver() (*in module platform*), 912
anext()

built-in function, 6
AnnAssign (*class in ast*), 2180
--annotate
pickletools command line option, 2249
Annotated (*in module typing*), 1737
annotation, **2321**
type annotation; type hint, 95
annotation (*inspect.Parameter attribute*), 2099
ANNOTATION (*symtable.SymbolTableType attribute*), 2205
answer_challenge() (*in module multiprocessing.connection*), 1003
anticipate_failure() (*in module test.support*), 1908
Any (*in module typing*), 1731
ANY (*in module unittest.mock*), 1872
any()
built-in function, 6
ANY_CONTIGUOUS (*inspect.BufferFlags attribute*), 2110
AnyStr (*in module typing*), 1731
api_version (*in module sys*), 2016
apilevel (*in module sqlite3*), 558
apop() (*poplib.POP3 method*), 1508
append() (*array.array method*), 294
append() (*collections.deque method*), 268
append() (*email.header.Header method*), 1322
append() (*imaplib.IMAP4 method*), 1512
append() (*sequence method*), 47
append() (*xml.etree.ElementTree.Element method*), 1394
append_history_file() (*in module readline*), 178
appendChild() (*xml.dom.Node method*), 1405
appendleft() (*collections.deque method*), 268
AppleFrameworkLoader (*class in importlib.machinery*), 2143
application_uri() (*in module wsgiref.util*), 1447
apply() (*multiprocessing.pool.Pool method*), 1000
apply_async() (*multiprocessing.pool.Pool method*), 1000
apply_defaults() (*inspect.BoundArguments method*), 2101
APRIL (*in module calendar*), 259
architecture() (*in module platform*), 908
archive (*zipimport.zipimporter attribute*), 2121
AREGTYPE (*in module tarfile*), 621
aRepr (*in module reprlib*), 320
arg (*class in ast*), 2196
argparse
module, 787
args (*BaseException attribute*), 108
args (*functools.partial attribute*), 446
args (*inspect.BoundArguments attribute*), 2101
args (*pdb command*), 1941
args (*subprocess.CompletedProcess attribute*), 1032
args (*subprocess.Popen attribute*), 1042
args (*typing.ParamSpec attribute*), 1748
args_from_interpreter_flags() (*in module test.support*), 1907
argtypes (*ctypes._FuncPtr attribute*), 943
argument, **2321**
ArgumentDefaultsHelpFormatter (*class in argparse*), 794
ArgumentError, 823, 943
ArgumentParser (*class in argparse*), 790
arguments (*class in ast*), 2195
arguments (*inspect.BoundArguments attribute*), 2101
ArgumentTypeError, 823
argv (*in module sys*), 1992
arithmetic, 37
ArithmeticError, 109
array
module, 63, 293
array (*class in array*), 294
Array (*class in ctypes*), 954
ARRAY() (*in module ctypes*), 954
Array() (*in module multiprocessing*), 990
Array() (*in module multiprocessing.sharedctypes*), 991
Array() (*multiprocessing.managers.SyncManager method*), 995
arrays, 293
arraysize (*sqlite3.Cursor attribute*), 573
as_bytes() (*email.message.EmailMessage method*), 1273
as_bytes() (*email.message.Message method*), 1311
as_completed() (*in module asyncio*), 1081
as_completed() (*in module concurrent.futures*), 1029
as_file() (*in module importlib.resources*), 2150
as_integer_ratio() (*decimal.Decimal method*), 364
as_integer_ratio() (*float method*), 41
as_integer_ratio() (*fractions.Fraction method*), 390
as_integer_ratio() (*int method*), 41
as_posix() (*pathlib.PurePath method*), 462
as_string() (*email.message.EmailMessage method*), 1273
as_string() (*email.message.Message method*), 1311
as_tuple() (*decimal.Decimal method*), 365
as_uri() (*pathlib.Path method*), 468
ASCII (*in module re*), 140
ascii()
built-in function, 6
ascii() (*in module curses.ascii*), 906
ascii_letters (*in module string*), 121
ascii_lowercase (*in module string*), 121
ascii_uppercase (*in module string*), 121
asctime() (*in module time*), 776
asdict() (*in module dataclasses*), 2047

asin() (*in module cmath*), 356
asin() (*in module math*), 352
asinh() (*in module cmath*), 357
asinh() (*in module math*), 353
askcolor() (*in module tkinter.colorchooser*), 1677
askdirectory() (*in module tkinter.filedialog*), 1680
askfloat() (*in module tkinter.simpledialog*), 1679
askinteger() (*in module tkinter.simpledialog*), 1679
askokcancel() (*in module tkinter.messagebox*), 1683
askopenfile() (*in module tkinter.filedialog*), 1680
askopenfilename() (*in module tkinter.filedialog*), 1680
askopenfilenames() (*in module tkinter.filedialog*), 1680
askopenfiles() (*in module tkinter.filedialog*), 1680
askquestion() (*in module tkinter.messagebox*), 1683
askretrycancel() (*in module tkinter.messagebox*), 1683
asksaveasfile() (*in module tkinter.filedialog*), 1680
asksaveasfilename() (*in module tkinter.filedialog*), 1680
askstring() (*in module tkinter.simpledialog*), 1679
askyesno() (*in module tkinter.messagebox*), 1683
askyesnocancel() (*in module tkinter.messagebox*), 1683
assert statement, 109
Assert (*class in ast*), 2181
assert_any_await() (*unittest.mock.AsyncMock method*), 1850
assert_any_call() (*unittest.mock.Mock method*), 1840
assert_awaited() (*unittest.mock.AsyncMock method*), 1849
assert_awaited_once() (*unittest.mock.AsyncMock method*), 1849
assert_awaited_once_with() (*unittest.mock.AsyncMock method*), 1850
assert_awaited_with() (*unittest.mock.AsyncMock method*), 1849
assert_called() (*unittest.mock.Mock method*), 1839
assert_called_once() (*unittest.mock.Mock method*), 1839
assert_called_once_with() (*unittest.mock.Mock method*), 1840
assert_called_with() (*unittest.mock.Mock method*), 1839
assert_has_awaits() (*unittest.mock.AsyncMock method*), 1850
assert_has_calls() (*unittest.mock.Mock method*), 1840
assert_never() (*in module typing*), 1758
assert_not_awaited() (*unittest.mock.AsyncMock method*), 1851
assert_not_called() (*unittest.mock.Mock method*), 1840
assert_python_failure() (*in module test.support.script_helper*), 1914
assert_python_ok() (*in module test.support.script_helper*), 1914
assert_type() (*in module typing*), 1758
assertAlmostEqual() (*unittest.TestCase method*), 1819
assertCountEqual() (*unittest.TestCase method*), 1820
assertDictEqual() (*unittest.TestCase method*), 1821
assertEqual() (*unittest.TestCase method*), 1815
assertFalse() (*unittest.TestCase method*), 1815
assertGreater() (*unittest.TestCase method*), 1819
assertGreaterEqual() (*unittest.TestCase method*), 1819
assertIn() (*unittest.TestCase method*), 1816
assertInBytecode() (*test.support.bytecode_helper.BytecodeTestCase method*), 1915
AssertionError, 109
assertIs() (*unittest.TestCase method*), 1815
assertIsInstance() (*unittest.TestCase method*), 1816
assertIsNone() (*unittest.TestCase method*), 1815
assertIsNot() (*unittest.TestCase method*), 1815
assertIsNotNone() (*unittest.TestCase method*), 1815
assertLess() (*unittest.TestCase method*), 1819
assertLessEqual() (*unittest.TestCase method*), 1819
assertListEqual() (*unittest.TestCase method*), 1821
assertLogs() (*unittest.TestCase method*), 1818
assertMultiLineEqual() (*unittest.TestCase method*), 1820
assertNoLogs() (*unittest.TestCase method*), 1818
assertNotAlmostEqual() (*unittest.TestCase method*), 1819
assertNotEqual() (*unittest.TestCase method*), 1815
assertNotIn() (*unittest.TestCase method*), 1816
assertNotInBytecode() (*test.support.bytecode_helper.BytecodeTestCase method*), 1915
assertNotIsInstance() (*unittest.TestCase method*), 1816
assertNotRegex() (*unittest.TestCase method*), 1819
assertRaises() (*unittest.TestCase method*), 1816
assertRaisesRegex() (*unittest.TestCase method*), 1817
assertRegex() (*unittest.TestCase method*), 1819
assertSequenceEqual() (*unittest.TestCase method*), 1820
assertSetEqual() (*unittest.TestCase method*), 1821
assertTrue() (*unittest.TestCase method*), 1815
assertTupleEqual() (*unittest.TestCase method*),

1821
assertWarns() (*unittest.TestCase method*), 1817
assertWarnsRegex() (*unittest.TestCase method*), 1818
Assign (*class in ast*), 2180
assignment
    slice, 47
    subscript, 47
ast
    module, 2165
AST (*class in ast*), 2168
ast command line option
    -a, 2203
    -h, 2203
    --help, 2203
    -i, 2204
    --include-attributes, 2203
    --indent, 2204
    -m, 2203
    --mode, 2203
    --no-type-comments, 2203
astimezone() (*datetime.datetime method*), 226
astuple() (*in module dataclasses*), 2048
AsyncContextDecorator (*class in contextlib*), 2061
AsyncContextManager (*class in typing*), 1773
asynccontextmanager() (*in module contextlib*), 2056
AsyncExitStack (*class in contextlib*), 2063
AsyncFor (*class in ast*), 2198
AsyncFunctionDef (*class in ast*), 2198
AsyncGenerator (*class in collections.abc*), 285
AsyncGenerator (*class in typing*), 1771
AsyncGeneratorType (*in module types*), 307
asynchronous context manager, **2322**
asynchronous generator, **2322**
asynchronous generator iterator, **2322**
asynchronous iterable, **2322**
asynchronous iterator, **2322**
asyncio
    module, 1065
asyncio.subprocess.DEVNULL (*built-in variable*), 1105
asyncio.subprocess.PIPE (*built-in variable*), 1105
asyncio.subprocess.Process (*built-in class*), 1105
asyncio.subprocess.STDOUT (*built-in variable*), 1105
AsyncIterable (*class in collections.abc*), 284
AsyncIterable (*class in typing*), 1771
AsyncIterator (*class in collections.abc*), 284
AsyncIterator (*class in typing*), 1771
AsyncMock (*class in unittest.mock*), 1848
AsyncResult (*class in multiprocessing.pool*), 1002
asyncSetUp() (*unittest.IsolatedAsyncioTestCase method*), 1823
asyncTearDown() (*unittest.IsolatedAsyncioTestCase method*), 1823
AsyncWith (*class in ast*), 2198
AT (*in module token*), 2210
at_eof() (*asyncio.StreamReader method*), 1092
atan() (*in module cmath*), 356
atan() (*in module math*), 352
atan2() (*in module math*), 352
atanh() (*in module cmath*), 357
atanh() (*in module math*), 353
ATEQUAL (*in module token*), 2211
atexit
    module, 2075
atexit (*weakref.finalize attribute*), 300
atof() (*in module locale*), 1608
atoi() (*in module locale*), 1608
attach() (*email.message.Message method*), 1312
attach_loop() (*asyncio.AbstractChildWatcher method*), 1158
attach_mock() (*unittest.mock.Mock method*), 1841
attempted (*doctest.TestResults attribute*), 1797
AttlistDeclHandler() (*xml.parsers.expat.xmlparser method*), 1436
attrgetter() (*in module operator*), 449
attrib (*xml.etree.ElementTree.Element attribute*), 1394
attribute, **2322**
Attribute (*class in ast*), 2176
AttributeError, 109
attributes (*xml.dom.Node attribute*), 1404
AttributesImpl (*class in xml.sax.xmlreader*), 1428
AttributesNSImpl (*class in xml.sax.xmlreader*), 1428
attroff() (*curses.window method*), 882
attron() (*curses.window method*), 882
attrset() (*curses.window method*), 882
audit events, 1923
audit() (*in module sys*), 1992
auditing, 1992
AugAssign (*class in ast*), 2181
AUGUST (*in module calendar*), 259
auth() (*ftplib.FTP_TLS method*), 1506
auth() (*smtplib.SMTP method*), 1521
authenticate() (*imaplib.IMAP4 method*), 1512
AuthenticationError, 980
authenticators() (*netrc.netrc method*), 666
authkey (*multiprocessing.Process attribute*), 979
auto (*class in enum*), 337
autocommit (*sqlite3.Connection attribute*), 569
autorange() (*timeit.Timer method*), 1955
available_timezones() (*in module zoneinfo*), 253
avoids_symlink_attacks (*shutil.rmtree attribute*), 514
Await (*class in ast*), 2198

await_args (*unittest.mock.AsyncMock attribute*), 1851
await_args_list (*unittest.mock.AsyncMock attribute*), 1851
await_count (*unittest.mock.AsyncMock attribute*), 1851
awaitable, **2322**
Awaitable (*class in collections.abc*), 284
Awaitable (*class in typing*), 1772

B

-b
    compileall command line option, 2222
    unittest command line option, 1806
b2a_base64() (*in module binascii*), 1369
b2a_hex() (*in module binascii*), 1370
b2a_qp() (*in module binascii*), 1370
b2a_uu() (*in module binascii*), 1369
b16decode() (*in module base64*), 1367
b16encode() (*in module base64*), 1367
b32decode() (*in module base64*), 1366
b32encode() (*in module base64*), 1366
b32hexdecode() (*in module base64*), 1366
b32hexencode() (*in module base64*), 1366
b64decode() (*in module base64*), 1366
b64encode() (*in module base64*), 1365
b85decode() (*in module base64*), 1367
b85encode() (*in module base64*), 1367
Babyl (*class in mailbox*), 1350
BabylMessage (*class in mailbox*), 1357
back() (*in module turtle*), 1619
backend (*in module readline*), 177
backslashreplace
    error handler's name, 194
backslashreplace_errors() (*in module codecs*), 196
backup() (*sqlite3.Connection method*), 566
backward() (*in module turtle*), 1619
BadGzipFile, 591
BadOptionError, 2318
BadStatusLine, 1494
BadZipFile, 606
BadZipfile, 606
Barrier (*class in asyncio*), 1101
Barrier (*class in multiprocessing*), 988
Barrier (*class in threading*), 968
Barrier() (*multiprocessing.managers.SyncManager method*), 994
base64
    encoding, 1365
    module, 1365, 1369
base_exec_prefix (*in module sys*), 1992
base_prefix (*in module sys*), 1992
BaseCGIHandler (*class in wsgiref.handlers*), 1452
BaseCookie (*class in http.cookies*), 1545
BaseException, 108
BaseExceptionGroup, 117
BaseHandler (*class in urllib.request*), 1460
BaseHandler (*class in wsgiref.handlers*), 1452
BaseHeader (*class in email.headerregistry*), 1296
BaseHTTPRequestHandler (*class in http.server*), 1538
BaseManager (*class in multiprocessing.managers*), 993
basename() (*in module os.path*), 482
BaseProtocol (*class in asyncio*), 1147
BaseProxy (*class in multiprocessing.managers*), 998
BaseRequestHandler (*class in socketserver*), 1533
BaseRotatingHandler (*class in logging.handlers*), 860
BaseSelector (*class in selectors*), 1251
BaseServer (*class in socketserver*), 1531
BaseTransport (*class in asyncio*), 1142
basicConfig() (*in module logging*), 841
BasicContext (*class in decimal*), 372
BasicInterpolation (*class in configparser*), 650
batched() (*in module itertools*), 421
baudrate() (*in module curses*), 874
bbox() (*tkinter.ttk.Treeview method*), 1697
BDADDR_ANY (*in module socket*), 1183
BDADDR_LOCAL (*in module socket*), 1183
bdb
    module, 1927, 1935
Bdb (*class in bdb*), 1928
BdbQuit, 1927
BDFL, **2322**
beep() (*in module curses*), 874
Beep() (*in module winsound*), 2263
BEFORE_ASYNC_WITH (*opcode*), 2234
BEFORE_WITH (*opcode*), 2236
begin_fill() (*in module turtle*), 1630
begin_poly() (*in module turtle*), 1635
BEL (*in module curses.ascii*), 903
below() (*curses.panel.Panel method*), 907
BELOW_NORMAL_PRIORITY_CLASS (*in module subprocess*), 1044
Benchmarking, 1953
benchmarking, 778, 783
--best
    gzip command line option, 594
betavariate() (*in module random*), 395
bgcolor() (*in module turtle*), 1637
bgpic() (*in module turtle*), 1637
bidirectional() (*in module unicodedata*), 174
bigaddrspacetest() (*in module test.support*), 1910
BigEndianStructure (*class in ctypes*), 952
BigEndianUnion (*class in ctypes*), 952
bigmemtest() (*in module test.support*), 1910
bin()
    built-in function, 6
binary

data, packing, 183
literals, 36
Binary (*class in xmlrpc.client*), 1562
binary file, **2322**
binary mode, 23
binary semaphores, 1062
BINARY_OP (*opcode*), 2232
BINARY_SLICE (*opcode*), 2233
BINARY_SUBSCR (*opcode*), 2233
BinaryIO (*class in typing*), 1758
binascii
module, 1369
bind (*widgets*), 1675
bind() (*inspect.Signature method*), 2098
bind() (*socket.socket method*), 1192
bind_partial() (*inspect.Signature method*), 2098
bind_port() (*in module test.support.socket_helper*), 1913
bind_textdomain_codeset() (*in module locale*), 1610
bind_unix_socket() (*in module test.support.socket_helper*), 1913
bindtextdomain() (*in module gettext*), 1593
bindtextdomain() (*in module locale*), 1610
binomialvariate() (*in module random*), 395
BinOp (*class in ast*), 2174
bisect
module, 290
bisect() (*in module bisect*), 290
bisect_left() (*in module bisect*), 290
bisect_right() (*in module bisect*), 290
bit_count() (*int method*), 39
bit_length() (*int method*), 38
BitAnd (*class in ast*), 2174
BitOr (*class in ast*), 2174
bits_per_digit (*sys.int_info attribute*), 2006
bitwise
operations, 38
BitXor (*class in ast*), 2174
bk() (*in module turtle*), 1619
bkgd() (*curses.window method*), 882
bkgdset() (*curses.window method*), 882
blake2b() (*in module hashlib*), 676
blake2b, blake2s, 676
blake2b.MAX_DIGEST_SIZE (*in module hashlib*), 678
blake2b.MAX_KEY_SIZE (*in module hashlib*), 678
blake2b.PERSON_SIZE (*in module hashlib*), 677
blake2b.SALT_SIZE (*in module hashlib*), 677
blake2s() (*in module hashlib*), 676
blake2s.MAX_DIGEST_SIZE (*in module hashlib*), 678
blake2s.MAX_KEY_SIZE (*in module hashlib*), 678
blake2s.PERSON_SIZE (*in module hashlib*), 677
blake2s.SALT_SIZE (*in module hashlib*), 677
BLKTYPE (*in module tarfile*), 621
Blob (*class in sqlite3*), 574
blobopen() (*sqlite3.Connection method*), 560
block_on_close (*socketserver.ThreadingMixIn attribute*), 1530
block_size (*hmac.HMAC attribute*), 684
blocked_domains() (*http.cookiejar.DefaultCookiePolicy method*), 1555
BlockingIOError, 115, 762
blocksize (*http.client.HTTPConnection attribute*), 1496
body() (*tkinter.simpledialog.Dialog method*), 1679
body_encode() (*email.charset.Charset method*), 1325
body_encoding (*email.charset.Charset attribute*), 1324
body_line_iterator() (*in module email.iterators*), 1329
BOLD (*in module tkinter.font*), 1677
BOM (*in module codecs*), 193
BOM_BE (*in module codecs*), 193
BOM_LE (*in module codecs*), 193
BOM_UTF8 (*in module codecs*), 193
BOM_UTF16 (*in module codecs*), 193
BOM_UTF16_BE (*in module codecs*), 193
BOM_UTF16_LE (*in module codecs*), 193
BOM_UTF32 (*in module codecs*), 193
BOM_UTF32_BE (*in module codecs*), 193
BOM_UTF32_LE (*in module codecs*), 193
bool (*built-in class*), 7
Boolean
object, 36
operations, 35
type, 7
values, 43
BOOLEAN_STATES (*configparser.ConfigParser attribute*), 655
BoolOp (*class in ast*), 2175
bootstrap() (*in module ensurepip*), 1974
border() (*curses.window method*), 882
borrowed reference, **2323**
bottom() (*curses.panel.Panel method*), 907
bottom_panel() (*in module curses.panel*), 907
BoundArguments (*class in inspect*), 2101
BoundaryError, 1294
BoundedSemaphore (*class in asyncio*), 1101
BoundedSemaphore (*class in multiprocessing*), 988
BoundedSemaphore (*class in threading*), 966
BoundedSemaphore() (*multiprocessing.managers.SyncManager method*), 994
box() (*curses.window method*), 882
bpbynumber (*bdb.Breakpoint attribute*), 1928
bpformat() (*bdb.Breakpoint method*), 1928
bplist (*bdb.Breakpoint attribute*), 1928
bpprint() (*bdb.Breakpoint method*), 1928

BRANCH (*monitoring event*), 2019
Break (*class in ast*), 2185
break (*pdb command*), 1939
break_anywhere() (*bdb.Bdb method*), 1930
break_here() (*bdb.Bdb method*), 1930
break_long_words (*textwrap.TextWrapper attribute*), 172
break_on_hyphens (*textwrap.TextWrapper attribute*), 172
Breakpoint (*class in bdb*), 1927
breakpoint()
    built-in function, 7
breakpointhook() (*in module sys*), 1993
breakpoints, 1710
broadcast_address (*ipaddress.IPv4Network attribute*), 1580
broadcast_address (*ipaddress.IPv6Network attribute*), 1583
broken (*asyncio.Barrier attribute*), 1103
broken (*threading.Barrier attribute*), 969
BrokenBarrierError, 969, 1103
BrokenExecutor, 1030
BrokenPipeError, 115
BrokenProcessPool, 1030
BrokenThreadPool, 1030
BROWSER, 1443, 1444
BS (*in module curses.ascii*), 903
BsdDbShelf (*class in shelve*), 543
buf (*multiprocessing.shared_memory.SharedMemory attribute*), 1018
--buffer
    unittest command line option, 1806
Buffer (*class in collections.abc*), 285
buffer (*io.TextIOBase attribute*), 770
buffer (*unittest.TestResult attribute*), 1828
buffer protocol
    binary sequence types, 62
    str (*built-in class*), 51
buffer size, I/O, 23
buffer_info() (*array.array method*), 295
buffer_size (*xml.parsers.expat.xmlparser attribute*), 1435
buffer_text (*xml.parsers.expat.xmlparser attribute*), 1435
buffer_updated() (*asyncio.BufferedProtocol method*), 1149
buffer_used (*xml.parsers.expat.xmlparser attribute*), 1435
BufferedIOBase (*class in io*), 766
BufferedProtocol (*class in asyncio*), 1147
BufferedRandom (*class in io*), 770
BufferedReader (*class in io*), 769
BufferedRWPair (*class in io*), 770
BufferedWriter (*class in io*), 769
BufferError, 109
BufferFlags (*class in inspect*), 2110
BufferingFormatter (*class in logging*), 833
BufferingHandler (*class in logging.handlers*), 868
BufferTooShort, 980
BUILD_CONST_KEY_MAP (*opcode*), 2238
BUILD_LIST (*opcode*), 2238
BUILD_MAP (*opcode*), 2238
build_opener() (*in module urllib.request*), 1458
BUILD_SET (*opcode*), 2238
BUILD_SLICE (*opcode*), 2243
BUILD_STRING (*opcode*), 2238
BUILD_TUPLE (*opcode*), 2238
built-in
    types, 35
built-in function
    __import__(), 31
    abs(), 5
    aiter(), 6
    all(), 6
    anext(), 6
    any(), 6
    ascii(), 6
    bin(), 6
    breakpoint(), 7
    callable(), 8
    chr(), 8
    classmethod(), 8
    compile, 102, 307
    compile(), 8
    complex, 37
    delattr(), 10
    dir(), 11
    divmod(), 11
    enumerate(), 12
    eval, 102, 315, 316
    eval(), 12
    exec, 13, 102
    exec(), 13
    filter(), 14
    float, 37
    format(), 15
    getattr(), 15
    globals(), 15
    hasattr(), 15
    hash, 47
    hash(), 16
    help(), 16
    hex(), 16
    id(), 17
    input(), 17
    int, 37
    isinstance(), 18
    issubclass(), 18

iter(), 18
len, 45, 89
len(), 18
locals(), 19
map(), 19
max, 45
max(), 19
min, 45
min(), 20
multiprocessing.Manager(), 993
next(), 20
oct(), 20
open(), 21
ord(), 23
pow(), 24
print(), 24
property.deleter(), 25
property.getter(), 25
property.setter(), 25
repr(), 26
reversed(), 26
round(), 26
setattr(), 26
slice, 2243
sorted(), 27
staticmethod(), 27
sum(), 28
type, 102
vars(), 30
zip(), 30
builtin_module_names (*in module sys*), 1993
BuiltinFunctionType (*in module types*), 307
BuiltinImporter (*class in importlib.machinery*), 2139
BuiltinMethodType (*in module types*), 307
builtins
module, 31, 2030
busy_retry() (*in module test.support*), 1905
BUTTON_ALT (*in module curses*), 900
BUTTON_CTRL (*in module curses*), 900
BUTTON_SHIFT (*in module curses*), 900
buttonbox() (*tkinter.simpledialog.Dialog method*), 1679
BUTTONn_CLICKED (*in module curses*), 900
BUTTONn_DOUBLE_CLICKED (*in module curses*), 900
BUTTONn_PRESSED (*in module curses*), 900
BUTTONn_RELEASED (*in module curses*), 900
BUTTONn_TRIPLE_CLICKED (*in module curses*), 900
bye() (*in module turtle*), 1643
byref() (*in module ctypes*), 946
bytearray
formatting, 77
interpolation, 77
methods, 65
object, 47, 63, 64
bytearray (*built-in class*), 64
bytecode, **2323**
byte-code
file, 2219
Bytecode (*class in dis*), 2226
BYTECODE_SUFFIXES (*in module importlib.machinery*), 2138
Bytecode.codeobj (*in module dis*), 2227
Bytecode.first_line (*in module dis*), 2227
BytecodeTestCase (*class in test.support.bytecode_helper*), 1915
byteorder (*in module sys*), 1993
bytes
formatting, 77
interpolation, 77
methods, 65
object, 63
str (*built-in class*), 51
bytes (*built-in class*), 63
bytes (*uuid.UUID attribute*), 1525
bytes-like object, **2323**
bytes_le (*uuid.UUID attribute*), 1525
bytes_warning (*sys.flags attribute*), 1998
BytesFeedParser (*class in email.parser*), 1281
BytesGenerator (*class in email.generator*), 1284
BytesHeaderParser (*class in email.parser*), 1282
BytesIO (*class in io*), 768
BytesParser (*class in email.parser*), 1282
ByteString (*class in collections.abc*), 283
ByteString (*class in typing*), 1769
byteswap() (*array.array method*), 295
BytesWarning, 117
bz2
module, 595
BZ2Compressor (*class in bz2*), 597
BZ2Decompressor (*class in bz2*), 597
BZ2File (*class in bz2*), 595

C

C
language, 36
structures, 183
-C
dis command line option, 2226
trace command line option, 1960
-c
calendar command line option, 261
random command line option, 401
tarfile command line option, 632
trace command line option, 1959
unittest command line option, 1806
zipapp command line option, 1986
zipfile command line option, 617

C14NWriterTarget (*class in xml.etree.ElementTree*), 1399
c_bool (*class in ctypes*), 952
c_byte (*class in ctypes*), 950
c_char (*class in ctypes*), 950
c_char_p (*class in ctypes*), 950
C_CONTIGUOUS (*inspect.BufferFlags attribute*), 2110
c_contiguous (*memoryview attribute*), 86
c_double (*class in ctypes*), 950
c_float (*class in ctypes*), 950
c_int (*class in ctypes*), 950
c_int8 (*class in ctypes*), 950
c_int16 (*class in ctypes*), 950
c_int32 (*class in ctypes*), 950
c_int64 (*class in ctypes*), 950
c_long (*class in ctypes*), 950
c_longdouble (*class in ctypes*), 950
c_longlong (*class in ctypes*), 950
C_RAISE (*monitoring event*), 2019
C_RETURN (*monitoring event*), 2019
c_short (*class in ctypes*), 951
c_size_t (*class in ctypes*), 951
c_ssize_t (*class in ctypes*), 951
c_time_t (*class in ctypes*), 951
c_ubyte (*class in ctypes*), 951
c_uint (*class in ctypes*), 951
c_uint8 (*class in ctypes*), 951
c_uint16 (*class in ctypes*), 951
c_uint32 (*class in ctypes*), 951
c_uint64 (*class in ctypes*), 951
c_ulong (*class in ctypes*), 951
c_ulonglong (*class in ctypes*), 951
c_ushort (*class in ctypes*), 951
c_void_p (*class in ctypes*), 951
c_wchar (*class in ctypes*), 951
c_wchar_p (*class in ctypes*), 952
CACHE (*opcode*), 2232
cache() (*in module functools*), 436
cache_from_source() (*in module importlib.util*), 2144
cached (*importlib.machinery.ModuleSpec attribute*), 2143
cached_property() (*in module functools*), 436
CacheFTPHandler (*class in urllib.request*), 1462
calcobjsize() (*in module test.support*), 1908
calcsize() (*in module struct*), 184
calcvobjsize() (*in module test.support*), 1908
calendar
    module, 254
Calendar (*class in calendar*), 254
calendar command line option
    -c, 261
    --css, 261
    -e, 261
    --encoding, 261
    -f, 261
    --first-weekday, 261
    -h, 260
    --help, 260
    -L, 261
    -l, 261
    --lines, 261
    --locale, 261
    -m, 261
    month, 261
    --months, 261
    -s, 261
    --spacing, 261
    -t, 261
    --type, 261
    -w, 261
    --width, 261
    year, 261
calendar() (*in module calendar*), 258
Call (*class in ast*), 2176
CALL (*monitoring event*), 2019
CALL (*opcode*), 2242
call() (*in module operator*), 449
call() (*in module subprocess*), 1045
call() (*in module unittest.mock*), 1870
call_args (*unittest.mock.Mock attribute*), 1844
call_args_list (*unittest.mock.Mock attribute*), 1844
call_at() (*asyncio.loop method*), 1116
call_count (*unittest.mock.Mock attribute*), 1842
call_exception_handler() (*asyncio.loop method*), 1129
CALL_FUNCTION_EX (*opcode*), 2242
CALL_INTRINSIC_1 (*opcode*), 2245
CALL_INTRINSIC_2 (*opcode*), 2246
CALL_KW (*opcode*), 2242
call_later() (*asyncio.loop method*), 1116
call_list() (*unittest.mock.call method*), 1870
call_soon() (*asyncio.loop method*), 1115
call_soon_threadsafe() (*asyncio.loop method*), 1115
call_tracing() (*in module sys*), 1993
callable, **2323**
Callable (*class in collections.abc*), 283
Callable (*in module typing*), 1772
callable()
    built-in function, 8
CallableProxyType (*in module weakref*), 301
callback, **2323**
callback (*optparse.Option attribute*), 2304
callback() (*contextlib.ExitStack method*), 2063
callback_args (*optparse.Option attribute*), 2304
callback_kwargs (*optparse.Option attribute*), 2304
callbacks (*in module gc*), 2089
called (*unittest.mock.Mock attribute*), 1842

CalledProcessError, 1033
CAN (*in module curses.ascii*), 905
CAN_BCM (*in module socket*), 1181
can_change_color() (*in module curses*), 874
can_fetch() (*urllib.robotparser.RobotFileParser method*), 1487
CAN_ISOTP (*in module socket*), 1181
CAN_J1939 (*in module socket*), 1181
CAN_RAW_FD_FRAMES (*in module socket*), 1181
CAN_RAW_JOIN_FILTERS (*in module socket*), 1181
can_symlink() (*in module test.support.os_helper*), 1917
can_write_eof() (*asyncio.StreamWriter method*), 1092
can_write_eof() (*asyncio.WriteTransport method*), 1145
can_xattr() (*in module test.support.os_helper*), 1917
CANCEL (*in module tkinter.messagebox*), 1684
cancel() (*asyncio.Future method*), 1140
cancel() (*asyncio.Handle method*), 1132
cancel() (*asyncio.Task method*), 1086
cancel() (*concurrent.futures.Future method*), 1028
cancel() (*sched.scheduler method*), 1053
cancel() (*threading.Timer method*), 968
cancel() (*tkinter.dnd.DndHandler method*), 1685
cancel_command() (*tkinter.filedialog.FileDialog method*), 1681
cancel_dump_traceback_later() (*in module faulthandler*), 1934
cancel_join_thread() (*multiprocessing.Queue method*), 982
cancelled() (*asyncio.Future method*), 1139
cancelled() (*asyncio.Handle method*), 1132
cancelled() (*asyncio.Task method*), 1087
cancelled() (*concurrent.futures.Future method*), 1028
CancelledError, 1030, 1111
cancelling() (*asyncio.Task method*), 1088
CannotSendHeader, 1494
CannotSendRequest, 1494
canonic() (*bdb.Bdb method*), 1929
canonical() (*decimal.Context method*), 374
canonical() (*decimal.Decimal method*), 365
canonicalize() (*in module xml.etree.ElementTree*), 1389
capa() (*poplib.POP3 method*), 1508
capitalize() (*bytearray method*), 72
capitalize() (*bytes method*), 72
capitalize() (*str method*), 52
CapsuleType (*class in types*), 311
captured_stderr() (*in module test.support*), 1907
captured_stdin() (*in module test.support*), 1907
captured_stdout() (*in module test.support*), 1907
captureWarnings() (*in module logging*), 844
capwords() (*in module string*), 132
casefold() (*str method*), 52
cast() (*in module ctypes*), 946
cast() (*in module typing*), 1758
cast() (*memoryview method*), 83
--catch
    unittest command line option, 1806
catch_threading_exception() (*in module test.support.threading_helper*), 1915
catch_unraisable_exception() (*in module test.support*), 1910
catch_warnings (*class in warnings*), 2043
category() (*in module unicodedata*), 173
cbreak() (*in module curses*), 874
cbrt() (*in module math*), 350
ccc() (*ftplib.FTP_TLS method*), 1506
C-contiguous, 2324
cdf() (*statistics.NormalDist method*), 415
CDLL (*class in ctypes*), 940
ceil() (*in module math*), 37, 346
CellType (*in module types*), 307
center() (*bytearray method*), 69
center() (*bytes method*), 69
center() (*str method*), 52
CERT_NONE (*in module ssl*), 1210
CERT_OPTIONAL (*in module ssl*), 1210
CERT_REQUIRED (*in module ssl*), 1210
cert_store_stats() (*ssl.SSLContext method*), 1223
cert_time_to_seconds() (*in module ssl*), 1208
CertificateError, 1208
certificates, 1231
cfmakecbreak() (*in module tty*), 2272
cfmakeraw() (*in module tty*), 2271
CFUNCTYPE() (*in module ctypes*), 943
cget() (*tkinter.font.Font method*), 1678
cgi_directories (*http.server.CGIHTTPRequestHandler attribute*), 1544
CGIHandler (*class in wsgiref.handlers*), 1451
CGIHTTPRequestHandler (*class in http.server*), 1543
CGIXMLRPCRequestHandler (*class in xmlrpc.server*), 1567
chain() (*in module itertools*), 422
chaining
    comparisons, 36
    exception, 107
ChainMap (*class in collections*), 262
ChainMap (*class in typing*), 1768
change_cwd() (*in module test.support.os_helper*), 1917
CHANNEL_BINDING_TYPES (*in module ssl*), 1215
CHAR_MAX (*in module locale*), 1608
character, 173
CharacterDataHandler() (*xml.parsers.expat.xmlparser method*), 1436
characters() (*xml.sax.handler.ContentHandler method*), 1424

characters_written (*BlockingIOError attribute*), 115
Charset (*class in email.charset*), 1324
charset() (*gettext.NullTranslations method*), 1596
chdir() (*in module contextlib*), 2060
chdir() (*in module os*), 714
check (*lzma.LZMADecompressor attribute*), 603
check() (*imaplib.IMAP4 method*), 1512
check() (*in module tabnanny*), 2217
check__all__() (*in module test.support*), 1911
check_call() (*in module subprocess*), 1046
check_disallow_instantiation() (*in module test.support*), 1912
CHECK_EG_MATCH (*opcode*), 2235
CHECK_EXC_MATCH (*opcode*), 2235
check_free_after_iterating() (*in module test.support*), 1911
check_hostname (*ssl.SSLContext attribute*), 1228
check_impl_detail() (*in module test.support*), 1906
check_no_resource_warning() (*in module test.support.warnings_helper*), 1920
check_output() (*doctest.OutputChecker method*), 1799
check_output() (*in module subprocess*), 1046
check_returncode() (*subprocess.CompletedProcess method*), 1032
check_syntax_error() (*in module test.support*), 1910
check_syntax_warning() (*in module test.support.warnings_helper*), 1920
check_unused_args() (*string.Formatter method*), 123
check_warnings() (*in module test.support.warnings_helper*), 1920
checkcache() (*in module linecache*), 510
CHECKED_HASH (*py_compile.PycInvalidationMode attribute*), 2220
checkfuncname() (*in module bdb*), 1932
checksizeof() (*in module test.support*), 1908
checksum
    Cyclic Redundancy Check, 588
chflags() (*in module os*), 714
chgat() (*curses.window method*), 882
childNodes (*xml.dom.Node attribute*), 1404
ChildProcessError, 115
children (*pyclbr.Class attribute*), 2219
children (*pyclbr.Function attribute*), 2218
children (*tkinter.Tk attribute*), 1665
chksum (*tarfile.TarInfo attribute*), 627
chmod() (*in module os*), 715
chmod() (*pathlib.Path method*), 478
--choice
    random command line option, 401
choice() (*in module random*), 394
choice() (*in module secrets*), 685
choices (*optparse.Option attribute*), 2304
choices() (*in module random*), 394
Chooser (*class in tkinter.colorchooser*), 1677
chown() (*in module os*), 716
chown() (*in module shutil*), 515
chr()
    built-in function, 8
chroot() (*in module os*), 716
CHRTYPE (*in module tarfile*), 621
cipher() (*ssl.SSLSocket method*), 1220
circle() (*in module turtle*), 1622
CIRCUMFLEX (*in module token*), 2210
CIRCUMFLEXEQUAL (*in module token*), 2210
Clamped (*class in decimal*), 379
class, **2323**
Class (*class in pyclbr*), 2218
Class (*class in symtable*), 2206
CLASS (*symtable.SymbolTableType attribute*), 2204
class variable, **2323**
ClassDef (*class in ast*), 2197
classmethod()
    built-in function, 8
ClassMethodDescriptorType (*in module types*), 308
ClassVar (*in module typing*), 1736
CLD_CONTINUED (*in module os*), 753
CLD_DUMPED (*in module os*), 753
CLD_EXITED (*in module os*), 753
CLD_KILLED (*in module os*), 753
CLD_STOPPED (*in module os*), 753
CLD_TRAPPED (*in module os*), 753
clean() (*mailbox.Maildir method*), 1345
cleandoc() (*in module inspect*), 2096
CleanImport (*class in test.support.import_helper*), 1919
cleanup() (*tempfile.TemporaryDirectory method*), 502
CLEANUP_THROW (*opcode*), 2234
clear (*pdb command*), 1939
Clear Breakpoint, 1710
clear() (*array.array method*), 296
clear() (*asyncio.Event method*), 1099
clear() (*collections.deque method*), 268
clear() (*curses.window method*), 883
clear() (*dbm.gnu.gdbm method*), 550
clear() (*dbm.ndbm.ndbm method*), 551
clear() (*dict method*), 90
clear() (*email.message.EmailMessage method*), 1279
clear() (*frozenset method*), 88
clear() (*http.cookiejar.CookieJar method*), 1551
clear() (*in module turtle*), 1630
clear() (*mailbox.Mailbox method*), 1344
clear() (*sequence method*), 47
clear() (*threading.Event method*), 967
clear() (*xml.etree.ElementTree.Element method*), 1394

clear_all_breaks() (*bdb.Bdb method*), 1931
clear_all_file_breaks() (*bdb.Bdb method*), 1931
clear_bpbynumber() (*bdb.Bdb method*), 1931
clear_break() (*bdb.Bdb method*), 1931
clear_cache() (*in module filecmp*), 498
clear_cache() (*zoneinfo.ZoneInfo class method*), 251
clear_content() (*email.message.EmailMessage method*), 1280
clear_flags() (*decimal.Context method*), 373
clear_frames() (*in module traceback*), 2078
clear_history() (*in module readline*), 178
clear_overloads() (*in module typing*), 1762
clear_session_cookies() (*http.cookiejar.CookieJar method*), 1551
clear_traces() (*in module tracemalloc*), 1967
clear_traps() (*decimal.Context method*), 373
clearcache() (*in module linecache*), 510
clearok() (*curses.window method*), 883
clearscreen() (*in module turtle*), 1637
clearstamp() (*in module turtle*), 1623
clearstamps() (*in module turtle*), 1623
Client() (*in module multiprocessing.connection*), 1003
client_address (*http.server.BaseHTTPRequestHandler attribute*), 1539
client_address (*socketserver.BaseRequestHandler attribute*), 1534
CLOCK_BOOTTIME (*in module time*), 785
clock_getres() (*in module time*), 776
clock_gettime() (*in module time*), 776
clock_gettime_ns() (*in module time*), 776
CLOCK_HIGHRES (*in module time*), 785
CLOCK_MONOTONIC (*in module time*), 785
CLOCK_MONOTONIC_RAW (*in module time*), 785
CLOCK_MONOTONIC_RAW_APPROX (*in module time*), 785
CLOCK_PROCESS_CPUTIME_ID (*in module time*), 785
CLOCK_PROF (*in module time*), 785
CLOCK_REALTIME (*in module time*), 786
clock_seq (*uuid.UUID attribute*), 1526
clock_seq_hi_variant (*uuid.UUID attribute*), 1526
clock_seq_low (*uuid.UUID attribute*), 1526
clock_settime() (*in module time*), 776
clock_settime_ns() (*in module time*), 777
CLOCK_TAI (*in module time*), 785
CLOCK_THREAD_CPUTIME_ID (*in module time*), 786
CLOCK_UPTIME (*in module time*), 786
CLOCK_UPTIME_RAW (*in module time*), 786
CLOCK_UPTIME_RAW_APPROX (*in module time*), 786
clone() (*email.generator.BytesGenerator method*), 1285
clone() (*email.generator.Generator method*), 1286
clone() (*email.policy.Policy method*), 1289
clone() (*in module turtle*), 1635
CLONE_FILES (*in module os*), 699
CLONE_FS (*in module os*), 699
CLONE_NEWCGROUP (*in module os*), 699
CLONE_NEWIPC (*in module os*), 699
CLONE_NEWNET (*in module os*), 699
CLONE_NEWNS (*in module os*), 699
CLONE_NEWPID (*in module os*), 699
CLONE_NEWTIME (*in module os*), 699
CLONE_NEWUSER (*in module os*), 699
CLONE_NEWUTS (*in module os*), 699
CLONE_SIGHAND (*in module os*), 699
CLONE_SYSVSEM (*in module os*), 699
CLONE_THREAD (*in module os*), 699
CLONE_VM (*in module os*), 699
cloneNode() (*xml.dom.Node method*), 1405
close() (*asyncio.AbstractChildWatcher method*), 1159
close() (*asyncio.BaseTransport method*), 1143
close() (*asyncio.loop method*), 1114
close() (*asyncio.Runner method*), 1068
close() (*asyncio.Server method*), 1133
close() (*asyncio.StreamWriter method*), 1092
close() (*asyncio.SubprocessTransport method*), 1146
close() (*contextlib.ExitStack method*), 2063
close() (*dbm.dumb.dumbdbm method*), 552
close() (*dbm.gnu.gdbm method*), 550
close() (*dbm.ndbm.ndbm method*), 551
close() (*email.parser.BytesFeedParser method*), 1281
close() (*ftplib.FTP method*), 1504
close() (*html.parser.HTMLParser method*), 1375
close() (*http.client.HTTPConnection method*), 1496
close() (*imaplib.IMAP4 method*), 1512
close() (*in module fileinput*), 490
close() (*in module os*), 699
close() (*in module socket*), 1186
close() (*io.IOBase method*), 764
close() (*logging.FileHandler method*), 858
close() (*logging.Handler method*), 831
close() (*logging.handlers.MemoryHandler method*), 869
close() (*logging.handlers.NTEventLogHandler method*), 867
close() (*logging.handlers.SocketHandler method*), 863
close() (*logging.handlers.SysLogHandler method*), 865
close() (*mailbox.Mailbox method*), 1344
close() (*mailbox.Maildir method*), 1347
close() (*mailbox.MH method*), 1350
close() (*mmap.mmap method*), 1266
close() (*multiprocessing.connection.Connection method*), 986
close() (*multiprocessing.connection.Listener method*), 1003
close() (*multiprocessing.pool.Pool method*), 1001
close() (*multiprocessing.Process method*), 979
close() (*multiprocessing.Queue method*), 982

close() (*multiprocessing.shared_memory.SharedMemory method*), 1018
close() (*multiprocessing.SimpleQueue method*), 983
close() (*os.scandir method*), 723
close() (*select.devpoll method*), 1244
close() (*select.epoll method*), 1245
close() (*select.kqueue method*), 1247
close() (*selectors.BaseSelector method*), 1252
close() (*shelve.Shelf method*), 542
close() (*socket.socket method*), 1192
close() (*sqlite3.Blob method*), 574
close() (*sqlite3.Connection method*), 561
close() (*sqlite3.Cursor method*), 572
close() (*tarfile.TarFile method*), 625
close() (*urllib.request.BaseHandler method*), 1465
close() (*wave.Wave_read method*), 1590
close() (*wave.Wave_write method*), 1591
Close() (*winreg.PyHKEY method*), 2263
close() (*xml.etree.ElementTree.TreeBuilder method*), 1398
close() (*xml.etree.ElementTree.XMLParser method*), 1399
close() (*xml.etree.ElementTree.XMLPullParser method*), 1400
close() (*xml.sax.xmlreader.IncrementalParser method*), 1430
close() (*zipfile.ZipFile method*), 608
close_clients() (*asyncio.Server method*), 1133
close_connection (*http.server.BaseHTTPRequestHandler attribute*), 1539
closed (*http.client.HTTPResponse attribute*), 1498
closed (*io.IOBase attribute*), 764
closed (*mmap.mmap attribute*), 1266
closed (*select.devpoll attribute*), 1244
closed (*select.epoll attribute*), 1246
closed (*select.kqueue attribute*), 1247
CloseKey() (*in module winreg*), 2254
closelog() (*in module syslog*), 2282
closerange() (*in module os*), 700
closing() (*in module contextlib*), 2056
clrtobot() (*curses.window method*), 883
clrtoeol() (*curses.window method*), 883
cmath
    module, 355
cmd
    module, 1651, 1935
Cmd (*class in cmd*), 1651
cmd (*subprocess.CalledProcessError attribute*), 1033
cmd (*subprocess.TimeoutExpired attribute*), 1033
cmdloop() (*cmd.Cmd method*), 1651
cmdqueue (*cmd.Cmd attribute*), 1653
cmp() (*in module filecmp*), 498
cmp_op (*in module dis*), 2247
cmp_to_key() (*in module functools*), 437
cmpfiles() (*in module filecmp*), 498
CMSG_LEN() (*in module socket*), 1190
CMSG_SPACE() (*in module socket*), 1190
CO_ASYNC_GENERATOR (*in module inspect*), 2110
CO_COROUTINE (*in module inspect*), 2109
CO_GENERATOR (*in module inspect*), 2109
CO_ITERABLE_COROUTINE (*in module inspect*), 2109
CO_NESTED (*in module inspect*), 2109
CO_NEWLOCALS (*in module inspect*), 2109
CO_OPTIMIZED (*in module inspect*), 2109
CO_VARARGS (*in module inspect*), 2109
CO_VARKEYWORDS (*in module inspect*), 2109
code
    module, 2115
code (*SystemExit attribute*), 114
code (*urllib.error.HTTPError attribute*), 1486
code (*urllib.response.addinfourl attribute*), 1477
code (*xml.etree.ElementTree.ParseError attribute*), 1401
code (*xml.parsers.expat.ExpatError attribute*), 1438
code object, 102, 545
code_context (*inspect.FrameInfo attribute*), 2105
code_context (*inspect.Traceback attribute*), 2105
code_info() (*in module dis*), 2227
Codec (*class in codecs*), 196
CodecInfo (*class in codecs*), 191
Codecs, 191
    decode, 191
    encode, 191
codecs
    module, 191
coded_value (*http.cookies.Morsel attribute*), 1547
codeop
    module, 2117
codepoint2name (*in module html.entities*), 1379
codes (*in module xml.parsers.expat.errors*), 1440
CODESET (*in module locale*), 1604
CodeType (*class in types*), 307
col_offset (*ast.AST attribute*), 2169
collapse_addresses() (*in module ipaddress*), 1586
collapse_rfc2231_value() (*in module email.utils*), 1329
collect() (*in module gc*), 2086
collectedDurations (*unittest.TestResult attribute*), 1828
Collection (*class in collections.abc*), 283
Collection (*class in typing*), 1769
collections
    module, 261
collections.abc
    module, 280
colno (*json.JSONDecodeError attribute*), 1338
colno (*re.PatternError attribute*), 146
COLON (*in module token*), 2209

colon (*mailbox.Maildir attribute*), 1345
COLONEQUAL (*in module token*), 2211
color() (*in module turtle*), 1629
COLOR_BLACK (*in module curses*), 901
COLOR_BLUE (*in module curses*), 901
color_content() (*in module curses*), 874
COLOR_CYAN (*in module curses*), 901
COLOR_GREEN (*in module curses*), 901
COLOR_MAGENTA (*in module curses*), 901
color_pair() (*in module curses*), 874
COLOR_PAIRS (*in module curses*), 888
COLOR_RED (*in module curses*), 901
COLOR_WHITE (*in module curses*), 901
COLOR_YELLOW (*in module curses*), 901
colormode() (*in module turtle*), 1642
COLORS (*in module curses*), 888
colorsys
    module, 1592
COLS (*in module curses*), 888
column() (*tkinter.ttk.Treeview method*), 1698
columnize() (*cmd.Cmd method*), 1652
COLUMNS, 881
columns (*os.terminal_size attribute*), 712
comb() (*in module math*), 346
combinations() (*in module itertools*), 422
combinations_with_replacement() (*in module itertools*), 423
combine() (*datetime.datetime class method*), 222
combining() (*in module unicodedata*), 174
Combobox (*class in tkinter.ttk*), 1690
COMMA (*in module token*), 2209
command (*http.server.BaseHTTPRequestHandler attribute*), 1539
CommandCompiler (*class in codeop*), 2118
commands (*pdb command*), 1940
comment (*http.cookiejar.Cookie attribute*), 1557
comment (*http.cookies.Morsel attribute*), 1546
COMMENT (*in module token*), 2211
comment (*zipfile.ZipFile attribute*), 612
comment (*zipfile.ZipInfo attribute*), 615
Comment() (*in module xml.etree.ElementTree*), 1389
comment() (*xml.etree.ElementTree.TreeBuilder method*), 1398
comment() (*xml.sax.handler.LexicalHandler method*), 1426
comment_url (*http.cookiejar.Cookie attribute*), 1557
commenters (*shlex.shlex attribute*), 1659
CommentHandler() (*xml.parsers.expat.xmlparser method*), 1437
commit() (*sqlite3.Connection method*), 560
common (*filecmp.dircmp attribute*), 499
common_dirs (*filecmp.dircmp attribute*), 499
common_files (*filecmp.dircmp attribute*), 499
common_funny (*filecmp.dircmp attribute*), 499
common_types (*in module mimetypes*), 1363
commonpath() (*in module os.path*), 482
commonprefix() (*in module os.path*), 483
communicate() (*asyncio.subprocess.Process method*), 1105
communicate() (*subprocess.Popen method*), 1041
--compact
    json.tool command line option, 1341
Compare (*class in ast*), 2175
compare() (*decimal.Context method*), 374
compare() (*decimal.Decimal method*), 365
compare() (*difflib.Differ method*), 165
compare_digest() (*in module hmac*), 684
compare_digest() (*in module secrets*), 687
compare_networks() (*ipaddress.IPv4Network method*), 1582
compare_networks() (*ipaddress.IPv6Network method*), 1583
COMPARE_OP (*opcode*), 2239
compare_signal() (*decimal.Context method*), 374
compare_signal() (*decimal.Decimal method*), 365
compare_to() (*tracemalloc.Snapshot method*), 1969
compare_total() (*decimal.Context method*), 374
compare_total() (*decimal.Decimal method*), 365
compare_total_mag() (*decimal.Context method*), 374
compare_total_mag() (*decimal.Decimal method*), 365
comparing
    objects, 36
comparison
    operator, 36
COMPARISON_FLAGS (*in module doctest*), 1787
comparisons
    chaining, 36
Compat32 (*class in email.policy*), 1293
compat32 (*in module email.policy*), 1294
compile
    built-in function, 102, 307
Compile (*class in codeop*), 2118
compile()
    built-in function, 8
compile() (*in module py_compile*), 2219
compile() (*in module re*), 142
compile_command() (*in module code*), 2115
compile_command() (*in module codeop*), 2117
compile_dir() (*in module compileall*), 2223
compile_file() (*in module compileall*), 2224
compile_path() (*in module compileall*), 2224
compileall
    module, 2221
compileall command line option
    -b, 2222
    -d, 2221

directory, 2221
-e, 2222
-f, 2221
file, 2221
--hardlink-dupes, 2222
-i, 2222
--invalidation-mode, 2222
-j, 2222
-l, 2221
-o, 2222
-p, 2221
-q, 2221
-r, 2222
-s, 2221
-x, 2222
compiler_flag (*__future__._Feature attribute*), 2086
complete() (*rlcompleter.Completer method*), 182
complete_statement() (*in module sqlite3*), 556
completedefault() (*cmd.Cmd method*), 1652
CompletedProcess (*class in subprocess*), 1032
Completer (*class in rlcompleter*), 182
complex
built-in function, 37
complex (*built-in class*), 9
Complex (*class in numbers*), 343
complex number, **2323**
literals, 36
object, 36
comprehension (*class in ast*), 2178
--compress
zipapp command line option, 1986
compress() (*bz2.BZ2Compressor method*), 597
compress() (*in module bz2*), 598
compress() (*in module gzip*), 593
compress() (*in module itertools*), 423
compress() (*in module lzma*), 603
compress() (*in module zlib*), 587
compress() (*lzma.LZMACompressor method*), 602
compress() (*zlib.Compress method*), 589
compress_size (*zipfile.ZipInfo attribute*), 616
compress_type (*zipfile.ZipInfo attribute*), 615
compressed (*ipaddress.IPv4Address attribute*), 1574
compressed (*ipaddress.IPv4Network attribute*), 1580
compressed (*ipaddress.IPv6Address attribute*), 1576
compressed (*ipaddress.IPv6Network attribute*), 1583
compression() (*ssl.SSLSocket method*), 1220
CompressionError, 620
compressobj() (*in module zlib*), 588
COMSPEC, 750, 1036
concat() (*in module operator*), 448
Concatenate (*in module typing*), 1735
concatenation
operation, 45
concurrent.futures
module, 1023
cond (*bdb.Breakpoint attribute*), 1928
Condition (*class in asyncio*), 1099
Condition (*class in multiprocessing*), 988
Condition (*class in threading*), 964
condition (*pdb command*), 1940
Condition() (*multiprocessing.managers.SyncManager method*), 995
config() (*tkinter.font.Font method*), 1678
configparser
module, 645
ConfigParser (*class in configparser*), 658
configuration
file, 645
file, debugger, 1939
file, path, 2112
configuration information, 2023
configure() (*tkinter.ttk.Style method*), 1701
configure_mock() (*unittest.mock.Mock method*), 1841
CONFORM (*enum.FlagBoundary attribute*), 335
confstr() (*in module os*), 756
confstr_names (*in module os*), 756
conjugate() (*complex number method*), 37
conjugate() (*decimal.Decimal method*), 365
conjugate() (*numbers.Complex method*), 343
connect() (*ftplib.FTP method*), 1501
connect() (*http.client.HTTPConnection method*), 1496
connect() (*in module sqlite3*), 555
connect() (*multiprocessing.managers.BaseManager method*), 993
connect() (*smtplib.SMTP method*), 1520
connect() (*socket.socket method*), 1192
connect_accepted_socket() (*asyncio.loop method*), 1122
connect_ex() (*socket.socket method*), 1193
connect_read_pipe() (*asyncio.loop method*), 1127
connect_write_pipe() (*asyncio.loop method*), 1127
Connection (*class in multiprocessing.connection*), 986
Connection (*class in sqlite3*), 560
connection (*sqlite3.Cursor attribute*), 573
connection_lost() (*asyncio.BaseProtocol method*), 1147
connection_made() (*asyncio.BaseProtocol method*), 1147
ConnectionAbortedError, 115
ConnectionError, 115
ConnectionRefusedError, 115
ConnectionResetError, 115
ConnectRegistry() (*in module winreg*), 2254
const (*optparse.Option attribute*), 2304
Constant (*class in ast*), 2171
constructor() (*in module copyreg*), 541

consumed (*asyncio.LimitOverrunError attribute*), 1112
container
    iteration over, 44
Container (*class in collections.abc*), 283
Container (*class in typing*), 1769
contains() (*in module operator*), 448
CONTAINS_OP (*opcode*), 2239
content (*urllib.error.ContentTooShortError attribute*), 1487
content type
    MIME, 1362
content_disposition (*email.headerregistry.ContentDispositionHeader attribute*), 1298
content_manager (*email.policy.EmailPolicy attribute*), 1291
content_type (*email.headerregistry.ContentTypeHeader attribute*), 1298
ContentDispositionHeader (*class in email.headerregistry*), 1298
ContentHandler (*class in xml.sax.handler*), 1421
ContentManager (*class in email.contentmanager*), 1301
contents (*ctypes._Pointer attribute*), 954
contents() (*importlib.abc.ResourceReader method*), 2137
contents() (*importlib.resources.abc.ResourceReader method*), 2154
contents() (*in module importlib.resources*), 2153
ContentTooShortError, 1487
ContentTransferEncoding (*class in email.headerregistry*), 1298
ContentTypeHeader (*class in email.headerregistry*), 1298
Context (*class in contextvars*), 1059
Context (*class in decimal*), 372
context (*ssl.SSLSocket attribute*), 1221
context management protocol, 94
context manager, 94, **2324**
context variable, **2324**
context_diff() (*in module difflib*), 158
ContextDecorator (*class in contextlib*), 2060
contextlib
    module, 2054
ContextManager (*class in typing*), 1773
contextmanager() (*in module contextlib*), 2055
ContextVar (*class in contextvars*), 1058
contextvars
    module, 1058
CONTIG (*inspect.BufferFlags attribute*), 2110
CONTIG_RO (*inspect.BufferFlags attribute*), 2110
contiguous, **2324**
contiguous (*memoryview attribute*), 86
Continue (*class in ast*), 2185
continue (*pdb command*), 1940
CONTINUOUS (*enum.EnumCheck attribute*), 334
control() (*select.kqueue method*), 1247
controlnames (*in module curses.ascii*), 906
CONTTYPE (*in module tarfile*), 621
conversions
    numeric, 37
convert_arg_line_to_args() (*argparse.ArgumentParser method*), 821
convert_field() (*string.Formatter method*), 123
CONVERT_VALUE (*opcode*), 2243
Cookie (*class in http.cookiejar*), 1550
CookieError, 1545
CookieJar (*class in http.cookiejar*), 1549
cookiejar (*urllib.request.HTTPCookieProcessor attribute*), 1467
CookiePolicy (*class in http.cookiejar*), 1549
Coordinated Universal Time, 775
Copy, 1710
copy
    module, 312, 541
    protocol, 531
COPY (*opcode*), 2231
copy() (*collections.deque method*), 268
copy() (*contextvars.Context method*), 1060
copy() (*decimal.Context method*), 373
copy() (*dict method*), 90
copy() (*frozenset method*), 87
copy() (*hashlib.hash method*), 674
copy() (*hmac.HMAC method*), 684
copy() (*http.cookies.Morsel method*), 1547
copy() (*imaplib.IMAP4 method*), 1512
copy() (*in module copy*), 312
copy() (*in module multiprocessing.sharedctypes*), 991
copy() (*in module shutil*), 512
copy() (*sequence method*), 47
copy() (*tkinter.font.Font method*), 1678
copy() (*types.MappingProxyType method*), 310
copy() (*zlib.Compress method*), 589
copy() (*zlib.Decompress method*), 590
copy2() (*in module shutil*), 512
copy_abs() (*decimal.Context method*), 374
copy_abs() (*decimal.Decimal method*), 365
copy_context() (*in module contextvars*), 1059
copy_decimal() (*decimal.Context method*), 373
copy_file_range() (*in module os*), 700
COPY_FREE_VARS (*opcode*), 2242
copy_location() (*in module ast*), 2201
copy_negate() (*decimal.Context method*), 374
copy_negate() (*decimal.Decimal method*), 366
copy_sign() (*decimal.Context method*), 374
copy_sign() (*decimal.Decimal method*), 366
copyfile() (*in module shutil*), 511
copyfileobj() (*in module shutil*), 511

copying files, 510
copymode() (*in module shutil*), 511
copyreg
    module, 541
copyright (*built-in variable*), 34
copyright (*in module sys*), 1993
copysign() (*in module math*), 347
copystat() (*in module shutil*), 511
copytree() (*in module shutil*), 513
coroutine, **2324**
Coroutine (*class in collections.abc*), 284
Coroutine (*class in typing*), 1770
coroutine function, **2324**
coroutine() (*in module types*), 312
CoroutineType (*in module types*), 307
correlation() (*in module statistics*), 412
cos() (*in module cmath*), 356
cos() (*in module math*), 352
cosh() (*in module cmath*), 357
cosh() (*in module math*), 353
--count
    trace command line option, 1959
count (*tracemalloc.Statistic attribute*), 1970
count (*tracemalloc.StatisticDiff attribute*), 1971
count() (*array.array method*), 295
count() (*bytearray method*), 65
count() (*bytes method*), 65
count() (*collections.deque method*), 268
count() (*in module itertools*), 424
count() (*multiprocessing.shared_memory.ShareableList method*), 1021
count() (*sequence method*), 45
count() (*str method*), 52
count_diff (*tracemalloc.StatisticDiff attribute*), 1971
Counter (*class in collections*), 265
Counter (*class in typing*), 1768
countOf() (*in module operator*), 448
countTestCases() (*unittest.TestCase method*), 1821
countTestCases() (*unittest.TestSuite method*), 1825
covariance() (*in module statistics*), 412
CoverageResults (*class in trace*), 1961
--coverdir
    trace command line option, 1960
cProfile
    module, 1947
CPU time, 778, 783
cpu_count() (*in module multiprocessing*), 984
cpu_count() (*in module os*), 756
CPython, **2324**
cpython_only() (*in module test.support*), 1909
CR (*in module curses.ascii*), 904
crawl_delay() (*urllib.robotparser.RobotFileParser method*), 1487
CRC (*zipfile.ZipInfo attribute*), 616
crc32() (*in module binascii*), 1370
crc32() (*in module zlib*), 588
crc_hqx() (*in module binascii*), 1370
--create
    tarfile command line option, 632
    zipfile command line option, 617
create() (*imaplib.IMAP4 method*), 1512
create() (*in module venv*), 1981
create() (*venv.EnvBuilder method*), 1979
create_aggregate() (*sqlite3.Connection method*), 561
create_archive() (*in module zipapp*), 1986
create_autospec() (*in module unittest.mock*), 1871
CREATE_BREAKAWAY_FROM_JOB (*in module subprocess*), 1045
create_collation() (*sqlite3.Connection method*), 563
create_configuration() (*venv.EnvBuilder method*), 1980
create_connection() (*asyncio.loop method*), 1117
create_connection() (*in module socket*), 1185
create_datagram_endpoint() (*asyncio.loop method*), 1119
create_decimal() (*decimal.Context method*), 373
create_decimal_from_float() (*decimal.Context method*), 373
create_default_context() (*in module ssl*), 1205
CREATE_DEFAULT_ERROR_MODE (*in module subprocess*), 1045
create_eager_task_factory() (*in module asyncio*), 1077
create_empty_file() (*in module test.support.os_helper*), 1917
create_function() (*sqlite3.Connection method*), 561
create_future() (*asyncio.loop method*), 1117
create_git_ignore_file() (*venv.EnvBuilder method*), 1981
create_module() (*importlib.abc.Loader method*), 2132
create_module() (*importlib.machinery.ExtensionFileLoader method*), 2141
create_module() (*zipimport.zipimporter method*), 2120
CREATE_NEW_CONSOLE (*in module subprocess*), 1044
CREATE_NEW_PROCESS_GROUP (*in module subprocess*), 1044
CREATE_NO_WINDOW (*in module subprocess*), 1045
create_server() (*asyncio.loop method*), 1121
create_server() (*in module socket*), 1185
create_stats() (*profile.Profile method*), 1948
create_string_buffer() (*in module ctypes*), 946
create_subprocess_exec() (*in module asyncio*),

1104
create_subprocess_shell() (*in module asyncio*), 1104
create_system (*zipfile.ZipInfo attribute*), 615
create_task() (*asyncio.loop method*), 1117
create_task() (*asyncio.TaskGroup method*), 1074
create_task() (*in module asyncio*), 1072
create_unicode_buffer() (*in module ctypes*), 946
create_unix_connection() (*asyncio.loop method*), 1120
create_unix_server() (*asyncio.loop method*), 1122
create_version (*zipfile.ZipInfo attribute*), 615
create_window_function() (*sqlite3.Connection method*), 562
createAttribute() (*xml.dom.Document method*), 1407
createAttributeNS() (*xml.dom.Document method*), 1407
createComment() (*xml.dom.Document method*), 1407
createDocument() (*xml.dom.DOMImplementation method*), 1403
createDocumentType() (*xml.dom.DOMImplementation method*), 1404
createElement() (*xml.dom.Document method*), 1407
createElementNS() (*xml.dom.Document method*), 1407
createfilehandler() (*_tkinter.Widget.tk method*), 1677
CreateKey() (*in module winreg*), 2254
CreateKeyEx() (*in module winreg*), 2255
createLock() (*logging.Handler method*), 830
createLock() (*logging.NullHandler method*), 859
createProcessingInstruction() (*xml.dom.Document method*), 1407
createSocket() (*logging.handlers.SocketHandler method*), 864
createSocket() (*logging.handlers.SysLogHandler method*), 865
createTextNode() (*xml.dom.Document method*), 1407
credits (*built-in variable*), 34
CRITICAL (*in module logging*), 830
critical() (*in module logging*), 840
critical() (*logging.Logger method*), 828
CRNCYSTR (*in module locale*), 1605
CRT_ASSEMBLY_VERSION (*in module msvcrt*), 2253
CRT_ASSERT (*in module msvcrt*), 2253
CRT_ERROR (*in module msvcrt*), 2253
CRT_WARN (*in module msvcrt*), 2253
CRTDBG_MODE_DEBUG (*in module msvcrt*), 2253
CRTDBG_MODE_FILE (*in module msvcrt*), 2253
CRTDBG_MODE_WNDW (*in module msvcrt*), 2253
CRTDBG_REPORT_MODE (*in module msvcrt*), 2253
CrtSetReportFile() (*in module msvcrt*), 2253
CrtSetReportMode() (*in module msvcrt*), 2253
cryptography, 671
--css
    calendar command line option, 261
cssclass_month (*calendar.HTMLCalendar attribute*), 256
cssclass_month_head (*calendar.HTMLCalendar attribute*), 256
cssclass_noday (*calendar.HTMLCalendar attribute*), 256
cssclass_year (*calendar.HTMLCalendar attribute*), 256
cssclass_year_head (*calendar.HTMLCalendar attribute*), 257
cssclasses (*calendar.HTMLCalendar attribute*), 256
cssclasses_weekday_head (*calendar.HTMLCalendar attribute*), 256
csv, 637
    module, 637
cte (*email.headerregistry.ContentTransferEncoding attribute*), 1298
cte_type (*email.policy.Policy attribute*), 1289
ctermid() (*in module os*), 691
ctime() (*datetime.date method*), 219
ctime() (*datetime.datetime method*), 229
ctime() (*in module time*), 777
ctrl() (*in module curses.ascii*), 906
CTRL_BREAK_EVENT (*in module signal*), 1257
CTRL_C_EVENT (*in module signal*), 1256
ctypes
    module, 920
curdir (*in module os*), 757
currency() (*in module locale*), 1607
current() (*tkinter.ttk.Combobox method*), 1690
current_process() (*in module multiprocessing*), 984
current_task() (*in module asyncio*), 1084
current_thread() (*in module threading*), 956
CurrentByteIndex (*xml.parsers.expat.xmlparser attribute*), 1435
CurrentColumnNumber (*xml.parsers.expat.xmlparser attribute*), 1435
currentframe() (*in module inspect*), 2106
CurrentLineNumber (*xml.parsers.expat.xmlparser attribute*), 1436
curs_set() (*in module curses*), 874
curses
    module, 873
curses.ascii
    module, 903
curses.panel
    module, 907
curses.textpad
    module, 901

Cursor (*class in sqlite3*), 571
cursor() (*sqlite3.Connection method*), 560
cursyncup() (*curses.window method*), 883
Cut, 1710
cwd() (*ftplib.FTP method*), 1504
cwd() (*pathlib.Path class method*), 469
cycle() (*in module itertools*), 424
CycleError, 341
Cyclic Redundancy Check, 588

D

-d
    compileall command line option, 2221
    gzip command line option, 594
D_FMT (*in module locale*), 1604
D_T_FMT (*in module locale*), 1604
daemon (*multiprocessing.Process attribute*), 978
daemon (*threading.Thread attribute*), 961
daemon_threads (*socketserver.ThreadingMixIn attribute*), 1530
data
    packing binary, 183
    tabular, 637
data (*collections.UserDict attribute*), 279
data (*collections.UserList attribute*), 279
data (*collections.UserString attribute*), 280
data (*select.kevent attribute*), 1249
data (*selectors.SelectorKey attribute*), 1251
data (*urllib.request.Request attribute*), 1463
data (*xml.dom.Comment attribute*), 1409
data (*xml.dom.ProcessingInstruction attribute*), 1410
data (*xml.dom.Text attribute*), 1409
data (*xmlrpc.client.Binary attribute*), 1562
data() (*xml.etree.ElementTree.TreeBuilder method*), 1398
data_filter() (*in module tarfile*), 630
data_open() (*urllib.request.DataHandler method*), 1470
data_received() (*asyncio.Protocol method*), 1148
database
    Unicode, 173
DatabaseError, 575
databases, 551
dataclass() (*in module dataclasses*), 2044
dataclass_transform() (*in module typing*), 1759
dataclasses
    module, 2043
DataError, 575
datagram_received() (*asyncio.DatagramProtocol method*), 1149
DatagramHandler (*class in logging.handlers*), 864
DatagramProtocol (*class in asyncio*), 1147
DatagramRequestHandler (*class in socketserver*), 1534
DatagramTransport (*class in asyncio*), 1143
DataHandler (*class in urllib.request*), 1462
date (*class in datetime*), 215
date() (*datetime.datetime method*), 226
date_time (*zipfile.ZipInfo attribute*), 615
date_time_string() (*http.server.BaseHTTPRequestHandler method*), 1541
DateHeader (*class in email.headerregistry*), 1297
datetime
    module, 209
datetime (*class in datetime*), 220
DateTime (*class in xmlrpc.client*), 1561
datetime (*email.headerregistry.DateHeader attribute*), 1297
Day (*class in calendar*), 258
day (*datetime.date attribute*), 216
day (*datetime.datetime attribute*), 224
DAY_1 (*in module locale*), 1604
DAY_2 (*in module locale*), 1604
DAY_3 (*in module locale*), 1604
DAY_4 (*in module locale*), 1604
DAY_5 (*in module locale*), 1604
DAY_6 (*in module locale*), 1604
DAY_7 (*in module locale*), 1604
day_abbr (*in module calendar*), 258
day_name (*in module calendar*), 258
daylight (*in module time*), 786
Daylight Saving Time, 775
DbfilenameShelf (*class in shelve*), 543
dbm
    module, 546
dbm.dumb
    module, 551
dbm.gnu
    module, 543, 549
dbm.ndbm
    module, 543, 550
dbm.sqlite3
    module, 548
DC1 (*in module curses.ascii*), 904
DC2 (*in module curses.ascii*), 904
DC3 (*in module curses.ascii*), 904
DC4 (*in module curses.ascii*), 904
dcgettext() (*in module locale*), 1610
deactivate_stack_trampoline() (*in module sys*), 2013
debug (*imaplib.IMAP4 attribute*), 1516
DEBUG (*in module logging*), 830
DEBUG (*in module re*), 140
debug (*pdb command*), 1943
debug (*shlex.shlex attribute*), 1660
debug (*sys.flags attribute*), 1998
debug (*zipfile.ZipFile attribute*), 612

debug() (*in module doctest*), 1801
debug() (*in module logging*), 839
debug() (*logging.Logger method*), 827
debug() (*unittest.TestCase method*), 1814
debug() (*unittest.TestSuite method*), 1825
DEBUG_BYTECODE_SUFFIXES (*in module importlib.machinery*), 2138
DEBUG_COLLECTABLE (*in module gc*), 2090
DEBUG_LEAK (*in module gc*), 2090
DEBUG_SAVEALL (*in module gc*), 2090
debug_src() (*in module doctest*), 1801
DEBUG_STATS (*in module gc*), 2090
DEBUG_UNCOLLECTABLE (*in module gc*), 2090
debugger, 957, 1709, 2003, 2011
    configuration file, 1939
debugging, 1935
debuglevel (*http.client.HTTPResponse attribute*), 1498
DebugRunner (*class in doctest*), 1801
DECEMBER (*in module calendar*), 259
decimal
    module, 359
Decimal (*class in decimal*), 363
decimal() (*in module unicodedata*), 173
DecimalException (*class in decimal*), 379
decode
    Codecs, 191
decode (*codecs.CodecInfo attribute*), 191
decode() (*bytearray method*), 66
decode() (*bytes method*), 66
decode() (*codecs.Codec method*), 196
decode() (*codecs.IncrementalDecoder method*), 198
decode() (*in module base64*), 1368
decode() (*in module codecs*), 191
decode() (*in module quopri*), 1371
decode() (*json.JSONDecoder method*), 1336
decode() (*xmlrpc.client.Binary method*), 1562
decode() (*xmlrpc.client.DateTime method*), 1561
decode_header() (*in module email.header*), 1323
decode_params() (*in module email.utils*), 1329
decode_rfc2231() (*in module email.utils*), 1329
decode_source() (*in module importlib.util*), 2145
decodebytes() (*in module base64*), 1368
DecodedGenerator (*class in email.generator*), 1286
decodestring() (*in module quopri*), 1371
decomposition() (*in module unicodedata*), 174
--decompress
    gzip command line option, 594
decompress() (*bz2.BZ2Decompressor method*), 597
decompress() (*in module bz2*), 598
decompress() (*in module gzip*), 593
decompress() (*in module lzma*), 603
decompress() (*in module zlib*), 588
decompress() (*lzma.LZMADecompressor method*), 603
decompress() (*zlib.Decompress method*), 590
decompressobj() (*in module zlib*), 589
decorator, **2324**
DEDENT (*in module token*), 2208
dedent() (*in module textwrap*), 170
deepcopy() (*in module copy*), 312
def_prog_mode() (*in module curses*), 875
def_shell_mode() (*in module curses*), 875
default (*in module email.policy*), 1292
DEFAULT (*in module unittest.mock*), 1870
default (*inspect.Parameter attribute*), 2099
default (*optparse.Option attribute*), 2303
default() (*cmd.Cmd method*), 1652
default() (*json.JSONEncoder method*), 1337
DEFAULT_BUFFER_SIZE (*in module io*), 762
default_bufsize (*in module xml.dom.pulldom*), 1418
default_exception_handler() (*asyncio.loop method*), 1129
default_factory (*collections.defaultdict attribute*), 272
DEFAULT_FORMAT (*in module tarfile*), 621
DEFAULT_IGNORES (*in module filecmp*), 499
default_loader() (*in module xml.etree.ElementInclude*), 1393
default_max_str_digits (*sys.int_info attribute*), 2006
default_open() (*urllib.request.BaseHandler method*), 1465
DEFAULT_PROTOCOL (*in module pickle*), 525
DEFAULT_TIMEOUT (*unittest.mock.ThreadingMock attribute*), 1852
default_timer() (*in module timeit*), 1954
DefaultContext (*class in decimal*), 372
DefaultCookiePolicy (*class in http.cookiejar*), 1549
defaultdict (*class in collections*), 271
DefaultDict (*class in typing*), 1768
DefaultEventLoopPolicy (*class in asyncio*), 1157
DefaultHandler() (*xml.parsers.expat.xmlparser method*), 1437
DefaultHandlerExpand() (*xml.parsers.expat.xmlparser method*), 1437
defaults() (*configparser.ConfigParser method*), 659
DefaultSelector (*class in selectors*), 1252
defaultTestLoader (*in module unittest*), 1830
defaultTestResult() (*unittest.TestCase method*), 1822
defects (*email.headerregistry.BaseHeader attribute*), 1296
defects (*email.message.EmailMessage attribute*), 1280
defects (*email.message.Message attribute*), 1318
defpath (*in module os*), 758
DefragResult (*class in urllib.parse*), 1483
DefragResultBytes (*class in urllib.parse*), 1483
degrees() (*in module math*), 353

degrees() (*in module turtle*), 1626
del
    statement, 47, 89
Del (*class in ast*), 2173
DEL (*in module curses.ascii*), 905
del_param() (*email.message.EmailMessage method*), 1276
del_param() (*email.message.Message method*), 1316
delattr()
    built-in function, 10
delay() (*in module turtle*), 1639
delay_output() (*in module curses*), 875
delayload (*http.cookiejar.FileCookieJar attribute*), 1552
delch() (*curses.window method*), 883
dele() (*poplib.POP3 method*), 1509
Delete (*class in ast*), 2182
delete() (*ftplib.FTP method*), 1504
delete() (*imaplib.IMAP4 method*), 1512
delete() (*tkinter.ttk.Treeview method*), 1698
DELETE_ATTR (*opcode*), 2237
DELETE_DEREF (*opcode*), 2241
DELETE_FAST (*opcode*), 2241
DELETE_GLOBAL (*opcode*), 2237
DELETE_NAME (*opcode*), 2236
DELETE_SUBSCR (*opcode*), 2233
deleteacl() (*imaplib.IMAP4 method*), 1513
deletefilehandler() (*_tkinter.Widget.tk method*), 1677
DeleteKey() (*in module winreg*), 2255
DeleteKeyEx() (*in module winreg*), 2255
deleteln() (*curses.window method*), 883
deleteMe() (*bdb.Breakpoint method*), 1927
DeleteValue() (*in module winreg*), 2256
delimiter (*csv.Dialect attribute*), 642
delitem() (*in module operator*), 449
deliver_challenge() (*in module multiprocessing.connection*), 1003
delocalize() (*in module locale*), 1608
demo_app() (*in module wsgiref.simple_server*), 1449
denominator (*fractions.Fraction attribute*), 390
denominator (*numbers.Rational attribute*), 344
deprecated() (*in module warnings*), 2042
DeprecationWarning, 116
deque (*class in collections*), 268
Deque (*class in typing*), 1768
dequeue() (*logging.handlers.QueueListener method*), 872
DER_cert_to_PEM_cert() (*in module ssl*), 1209
derive() (*BaseExceptionGroup method*), 118
derwin() (*curses.window method*), 883
description (*inspect.Parameter.kind attribute*), 2100
description (*sqlite3.Cursor attribute*), 573
descriptor, **2324**
deserialize() (*sqlite3.Connection method*), 569
dest (*optparse.Option attribute*), 2303
detach() (*io.BufferedIOBase method*), 766
detach() (*io.TextIOBase method*), 770
detach() (*socket.socket method*), 1193
detach() (*tkinter.ttk.Treeview method*), 1698
detach() (*weakref.finalize method*), 300
Detach() (*winreg.PyHKEY method*), 2263
DETACHED_PROCESS (*in module subprocess*), 1045
--details
    inspect command line option, 2111
detect_api_mismatch() (*in module test.support*), 1911
detect_encoding() (*in module tokenize*), 2213
deterministic profiling, 1944
dev_mode (*sys.flags attribute*), 1998
device_encoding() (*in module os*), 700
devmajor (*tarfile.TarInfo attribute*), 627
devminor (*tarfile.TarInfo attribute*), 627
devnull (*in module os*), 758
DEVNULL (*in module subprocess*), 1032
devpoll() (*in module select*), 1242
DevpollSelector (*class in selectors*), 1252
dgettext() (*in module gettext*), 1594
dgettext() (*in module locale*), 1610
Dialect (*class in csv*), 640
dialect (*csv.csvreader attribute*), 643
dialect (*csv.csvwriter attribute*), 643
Dialog (*class in tkinter.commondialog*), 1682
Dialog (*class in tkinter.simpledialog*), 1679
dict (*built-in class*), 89
Dict (*class in ast*), 2172
Dict (*class in typing*), 1767
dict() (*multiprocessing.managers.SyncManager method*), 995
DICT_MERGE (*opcode*), 2239
DICT_UPDATE (*opcode*), 2238
DictComp (*class in ast*), 2177
dictConfig() (*in module logging.config*), 845
dictionary, **2324**
    object, 89
    type, operations on, 89
dictionary comprehension, **2325**
dictionary view, **2325**
DictReader (*class in csv*), 639
DictWriter (*class in csv*), 639
diff_bytes() (*in module difflib*), 160
diff_files (*filecmp.dircmp attribute*), 499
Differ (*class in difflib*), 157
difference() (*frozenset method*), 87
difference_update() (*frozenset method*), 88
difflib
    module, 156
dig (*sys.float_info attribute*), 2000
digest() (*hashlib.hash method*), 674

digest() (*hashlib.shake method*), 674
digest() (*hmac.HMAC method*), 684
digest() (*in module hmac*), 683
digest_size (*hmac.HMAC attribute*), 684
digit() (*in module unicodedata*), 173
digits (*in module string*), 121
dir()
    built-in function, 11
dir() (*ftplib.FTP method*), 1504
dircmp (*class in filecmp*), 498
directory
    changing, 714
    compileall command line option, 2221
    creating, 719
    deleting, 514, 721
    site-packages, 2111
    traversal, 732, 733
    walking, 732, 733
Directory (*class in tkinter.filedialog*), 1680
DirEntry (*class in os*), 723
dirname() (*in module os.path*), 483
dirs_double_event() (*tkinter.filedialog.FileDialog method*), 1681
dirs_select_event() (*tkinter.filedialog.FileDialog method*), 1681
DirsOnSysPath (*class in test.support.import_helper*), 1919
DIRTYPE (*in module tarfile*), 621
dis
    module, 2225
dis command line option
    -C, 2226
    -h, 2226
    --help, 2226
    -O, 2226
    --show-caches, 2226
    --show-offsets, 2226
dis() (*dis.Bytecode method*), 2227
dis() (*in module dis*), 2228
dis() (*in module pickletools*), 2249
DISABLE (*in module sys.monitoring*), 2021
disable (*pdb command*), 1939
disable() (*bdb.Breakpoint method*), 1928
disable() (*in module faulthandler*), 1934
disable() (*in module gc*), 2086
disable() (*in module logging*), 840
disable() (*profile.Profile method*), 1948
disable_faulthandler() (*in module test.support*), 1907
disable_gc() (*in module test.support*), 1907
disable_interspersed_args() (*optparse.OptionParser method*), 2308
disabled (*logging.Logger attribute*), 826
DisableReflectionKey() (*in module winreg*), 2259
disassemble() (*in module dis*), 2228
discard (*http.cookiejar.Cookie attribute*), 1556
discard() (*frozenset method*), 88
discard() (*mailbox.Mailbox method*), 1342
discard() (*mailbox.MH method*), 1349
discover() (*unittest.TestLoader method*), 1826
disk_usage() (*in module shutil*), 515
dispatch_call() (*bdb.Bdb method*), 1929
dispatch_exception() (*bdb.Bdb method*), 1930
dispatch_line() (*bdb.Bdb method*), 1929
dispatch_return() (*bdb.Bdb method*), 1929
dispatch_table (*pickle.Pickler attribute*), 527
DISPLAY, 1665
display (*pdb command*), 1941
display_name (*email.headerregistry.Address attribute*), 1300
display_name (*email.headerregistry.Group attribute*), 1301
displayhook() (*in module sys*), 1994
dist() (*in module math*), 352
distance() (*in module turtle*), 1625
Div (*class in ast*), 2174
divide() (*decimal.Context method*), 374
divide_int() (*decimal.Context method*), 374
DivisionByZero (*class in decimal*), 379
divmod()
    built-in function, 11
divmod() (*decimal.Context method*), 374
DLE (*in module curses.ascii*), 904
DllCanUnloadNow() (*in module ctypes*), 946
DllGetClassObject() (*in module ctypes*), 947
dllhandle (*in module sys*), 1994
dnd_start() (*in module tkinter.dnd*), 1685
DndHandler (*class in tkinter.dnd*), 1685
dngettext() (*in module gettext*), 1594
dnpgettext() (*in module gettext*), 1594
do_clear() (*bdb.Bdb method*), 1930
do_command() (*curses.textpad.Textbox method*), 902
do_GET() (*http.server.SimpleHTTPRequestHandler method*), 1542
do_handshake() (*ssl.SSLSocket method*), 1218
do_HEAD() (*http.server.SimpleHTTPRequestHandler method*), 1542
do_help() (*cmd.Cmd method*), 1652
do_POST() (*http.server.CGIHTTPRequestHandler method*), 1544
doc (*json.JSONDecodeError attribute*), 1338
doc_header (*cmd.Cmd attribute*), 1653
DocCGIXMLRPCRequestHandler (*class in xmlrpc.server*), 1571
DocFileSuite() (*in module doctest*), 1792
doClassCleanups() (*unittest.TestCase class method*), 1823
doCleanups() (*unittest.TestCase method*), 1822

docmd() (*smtplib.SMTP method*), 1520
docstring, **2325**
docstring (*doctest.DocTest attribute*), 1795
doctest
    module, 1779
DocTest (*class in doctest*), 1794
DocTestFailure, 1802
DocTestFinder (*class in doctest*), 1796
DocTestParser (*class in doctest*), 1797
DocTestRunner (*class in doctest*), 1797
DocTestSuite() (*in module doctest*), 1792
doctype() (*xml.etree.ElementTree.TreeBuilder method*), 1398
documentation
    generation, 1774
    online, 1774
documentElement (*xml.dom.Document attribute*), 1407
DocXMLRPCRequestHandler (*class in xmlrpc.server*), 1571
DocXMLRPCServer (*class in xmlrpc.server*), 1571
domain (*email.headerregistry.Address attribute*), 1300
domain (*http.cookiejar.Cookie attribute*), 1556
domain (*http.cookies.Morsel attribute*), 1546
domain (*tracemalloc.DomainFilter attribute*), 1968
domain (*tracemalloc.Filter attribute*), 1968
domain (*tracemalloc.Trace attribute*), 1971
domain_initial_dot (*http.cookiejar.Cookie attribute*), 1557
domain_return_ok() (*http.cookiejar.CookiePolicy method*), 1553
domain_specified (*http.cookiejar.Cookie attribute*), 1557
DomainFilter (*class in tracemalloc*), 1968
DomainLiberal (*http.cookiejar.DefaultCookiePolicy attribute*), 1556
DomainRFC2965Match (*http.cookiejar.DefaultCookiePolicy attribute*), 1556
DomainStrict (*http.cookiejar.DefaultCookiePolicy attribute*), 1556
DomainStrictNoDots (*http.cookiejar.DefaultCookiePolicy attribute*), 1555
DomainStrictNonDomain (*http.cookiejar.DefaultCookiePolicy attribute*), 1556
DOMEventStream (*class in xml.dom.pulldom*), 1418
DOMException, 1410
doModuleCleanups() (*in module unittest*), 1834
DomstringSizeErr, 1410
done() (*asyncio.Future method*), 1139
done() (*asyncio.Task method*), 1085
done() (*concurrent.futures.Future method*), 1028
done() (*graphlib.TopologicalSorter method*), 340
done() (*in module turtle*), 1641
DONT_ACCEPT_BLANKLINE (*in module doctest*), 1786
DONT_ACCEPT_TRUE_FOR_1 (*in module doctest*), 1786
dont_write_bytecode (*in module sys*), 1995
dont_write_bytecode (*sys.flags attribute*), 1998
doRollover() (*logging.handlers.RotatingFileHandler method*), 861
doRollover() (*logging.handlers.TimedRotatingFileHandler method*), 863
DOT (*in module token*), 2209
dot() (*in module turtle*), 1622
DOTALL (*in module re*), 141
doublequote (*csv.Dialect attribute*), 642
DOUBLESLASH (*in module token*), 2210
DOUBLESLASHEQUAL (*in module token*), 2210
DOUBLESTAR (*in module token*), 2210
DOUBLESTAREQUAL (*in module token*), 2210
doupdate() (*in module curses*), 875
down (*pdb command*), 1939
down() (*in module turtle*), 1626
dpgettext() (*in module gettext*), 1594
drain() (*asyncio.StreamWriter method*), 1093
drive (*pathlib.PurePath attribute*), 460
drop_whitespace (*textwrap.TextWrapper attribute*), 172
dropwhile() (*in module itertools*), 424
dst() (*datetime.datetime method*), 227
dst() (*datetime.time method*), 235
dst() (*datetime.timezone method*), 243
dst() (*datetime.tzinfo method*), 237
DTDHandler (*class in xml.sax.handler*), 1421
duck-typing, **2325**
dump() (*in module ast*), 2202
dump() (*in module json*), 1333
dump() (*in module marshal*), 545
dump() (*in module pickle*), 526
dump() (*in module plistlib*), 668
dump() (*in module xml.etree.ElementTree*), 1389
dump() (*pickle.Pickler method*), 527
dump() (*tracemalloc.Snapshot method*), 1969
dump_stats() (*profile.Profile method*), 1948
dump_stats() (*pstats.Stats method*), 1949
dump_traceback() (*in module faulthandler*), 1933
dump_traceback_later() (*in module faulthandler*), 1934
dumps() (*in module json*), 1334
dumps() (*in module marshal*), 545
dumps() (*in module pickle*), 526
dumps() (*in module plistlib*), 668
dumps() (*in module xmlrpc.client*), 1565
dup() (*in module os*), 700
dup() (*socket.socket method*), 1193

dup2() (*in module os*), 701
DuplicateOptionError, 663
DuplicateSectionError, 663
--durations
    unittest command line option, 1806
dwFlags (*subprocess.STARTUPINFO attribute*), 1043
DynamicClassAttribute() (*in module types*), 311

E

-e
    calendar command line option, 261
    compileall command line option, 2222
    tarfile command line option, 632
    tokenize command line option, 2214
    zipfile command line option, 617
e (*in module cmath*), 358
e (*in module math*), 354
E2BIG (*in module errno*), 913
EACCES (*in module errno*), 913
EADDRINUSE (*in module errno*), 918
EADDRNOTAVAIL (*in module errno*), 918
EADV (*in module errno*), 916
EAFNOSUPPORT (*in module errno*), 918
EAFP, **2325**
EAGAIN (*in module errno*), 913
eager_task_factory() (*in module asyncio*), 1077
EALREADY (*in module errno*), 919
east_asian_width() (*in module unicodedata*), 174
EBADE (*in module errno*), 915
EBADF (*in module errno*), 913
EBADFD (*in module errno*), 917
EBADMSG (*in module errno*), 917
EBADR (*in module errno*), 916
EBADRQC (*in module errno*), 916
EBADSLT (*in module errno*), 916
EBFONT (*in module errno*), 916
EBUSY (*in module errno*), 914
ECANCELED (*in module errno*), 919
ECHILD (*in module errno*), 913
echo() (*in module curses*), 875
echochar() (*curses.window method*), 883
ECHRNG (*in module errno*), 915
ECOMM (*in module errno*), 916
ECONNABORTED (*in module errno*), 918
ECONNREFUSED (*in module errno*), 919
ECONNRESET (*in module errno*), 918
EDEADLK (*in module errno*), 915
EDEADLOCK (*in module errno*), 916
EDESTADDRREQ (*in module errno*), 917
edit() (*curses.textpad.Textbox method*), 902
EDOM (*in module errno*), 915
EDOTDOT (*in module errno*), 917
EDQUOT (*in module errno*), 919
EEXIST (*in module errno*), 914
EFAULT (*in module errno*), 914
EFBIG (*in module errno*), 914
EFD_CLOEXEC (*in module os*), 736
EFD_NONBLOCK (*in module os*), 736
EFD_SEMAPHORE (*in module os*), 736
effective() (*in module bdb*), 1932
ehlo() (*smtplib.SMTP method*), 1520
ehlo_or_helo_if_needed() (*smtplib.SMTP method*), 1520
EHOSTDOWN (*in module errno*), 919
EHOSTUNREACH (*in module errno*), 919
EIDRM (*in module errno*), 915
EILSEQ (*in module errno*), 917
EINPROGRESS (*in module errno*), 919
EINTR (*in module errno*), 913
EINVAL (*in module errno*), 914
EIO (*in module errno*), 913
EISCONN (*in module errno*), 918
EISDIR (*in module errno*), 914
EISNAM (*in module errno*), 919
EJECT (*enum.FlagBoundary attribute*), 335
EL2HLT (*in module errno*), 915
EL2NSYNC (*in module errno*), 915
EL3HLT (*in module errno*), 915
EL3RST (*in module errno*), 915
Element (*class in xml.etree.ElementTree*), 1393
element_create() (*tkinter.ttk.Style method*), 1703
element_names() (*tkinter.ttk.Style method*), 1704
element_options() (*tkinter.ttk.Style method*), 1704
ElementDeclHandler() (*xml.parsers.expat.xmlparser method*), 1436
elements() (*collections.Counter method*), 265
ElementTree (*class in xml.etree.ElementTree*), 1396
ELIBACC (*in module errno*), 917
ELIBBAD (*in module errno*), 917
ELIBEXEC (*in module errno*), 917
ELIBMAX (*in module errno*), 917
ELIBSCN (*in module errno*), 917
Ellipsis (*built-in variable*), 33
ELLIPSIS (*in module doctest*), 1786
ELLIPSIS (*in module token*), 2211
EllipsisType (*in module types*), 309
ELNRNG (*in module errno*), 915
ELOOP (*in module errno*), 915
EM (*in module curses.ascii*), 905
email
    module, 1271
email.charset
    module, 1324
email.contentmanager
    module, 1301
email.encoders
    module, 1326
email.errors

module, 1294
email.generator
module, 1284
email.header
module, 1321
email.headerregistry
module, 1295
email.iterators
module, 1329
email.message
module, 1272
EmailMessage (*class in email.message*), 1272
email.mime
module, 1319
email.mime.application
module, 1320
email.mime.audio
module, 1320
email.mime.base
module, 1319
email.mime.image
module, 1320
email.mime.message
module, 1321
email.mime.multipart
module, 1319
email.mime.nonmultipart
module, 1319
email.mime.text
module, 1321
email.parser
module, 1280
email.policy
module, 1287
EmailPolicy (*class in email.policy*), 1291
email.utils
module, 1327
EMFILE (*in module errno*), 914
emit() (*logging.FileHandler method*), 858
emit() (*logging.Handler method*), 831
emit() (*logging.handlers.BufferingHandler method*), 869
emit() (*logging.handlers.DatagramHandler method*), 864
emit() (*logging.handlers.HTTPHandler method*), 870
emit() (*logging.handlers.NTEventLogHandler method*), 867
emit() (*logging.handlers.QueueHandler method*), 870
emit() (*logging.handlers.RotatingFileHandler method*), 861
emit() (*logging.handlers.SMTPHandler method*), 868
emit() (*logging.handlers.SocketHandler method*), 863
emit() (*logging.handlers.SysLogHandler method*), 865
emit() (*logging.handlers.TimedRotatingFileHandler method*), 863
emit() (*logging.handlers.WatchedFileHandler method*), 859
emit() (*logging.NullHandler method*), 859
emit() (*logging.StreamHandler method*), 858
EMLINK (*in module errno*), 914
Empty, 1055
empty (*inspect.Parameter attribute*), 2099
empty (*inspect.Signature attribute*), 2098
empty() (*asyncio.Queue method*), 1108
empty() (*multiprocessing.Queue method*), 982
empty() (*multiprocessing.SimpleQueue method*), 983
empty() (*queue.Queue method*), 1055
empty() (*queue.SimpleQueue method*), 1057
empty() (*sched.scheduler method*), 1053
EMPTY_NAMESPACE (*in module xml.dom*), 1402
emptyline() (*cmd.Cmd method*), 1652
emscripten_version (*sys._emscripten_info attribute*), 1995
EMSGSIZE (*in module errno*), 917
EMULTIHOP (*in module errno*), 916
enable (*pdb command*), 1939
enable() (*bdb.Breakpoint method*), 1927
enable() (*imaplib.IMAP4 method*), 1513
enable() (*in module faulthandler*), 1934
enable() (*in module gc*), 2086
enable() (*profile.Profile method*), 1948
enable_callback_tracebacks() (*in module sqlite3*), 557
enable_interspersed_args() (*optparse.OptionParser method*), 2308
enable_load_extension() (*sqlite3.Connection method*), 565
enable_traversal() (*tkinter.ttk.Notebook method*), 1693
ENABLE_USER_SITE (*in module site*), 2113
enabled (*bdb.Breakpoint attribute*), 1928
EnableReflectionKey() (*in module winreg*), 2259
ENAMETOOLONG (*in module errno*), 915
ENAVAIL (*in module errno*), 919
enclose() (*curses.window method*), 883
encode
Codecs, 191
encode (*codecs.CodecInfo attribute*), 191
encode() (*codecs.Codec method*), 196
encode() (*codecs.IncrementalEncoder method*), 197
encode() (*email.header.Header method*), 1323
encode() (*in module base64*), 1368
encode() (*in module codecs*), 191
encode() (*in module quopri*), 1371
encode() (*json.JSONEncoder method*), 1337
encode() (*str method*), 52
encode() (*xmlrpc.client.Binary method*), 1562
encode() (*xmlrpc.client.DateTime method*), 1561
encode_7or8bit() (*in module email.encoders*), 1327

encode_base64() (*in module email.encoders*), 1326
encode_noop() (*in module email.encoders*), 1327
encode_quopri() (*in module email.encoders*), 1326
encode_rfc2231() (*in module email.utils*), 1329
encodebytes() (*in module base64*), 1368
EncodedFile() (*in module codecs*), 193
encodePriority() (*logging.handlers.SysLogHandler method*), 866
encodestring() (*in module quopri*), 1371
encoding
    base64, 1365
    quoted-printable, 1371
--encoding
    calendar command line option, 261
encoding (*curses.window attribute*), 883
ENCODING (*in module tarfile*), 620
ENCODING (*in module token*), 2211
encoding (*io.TextIOBase attribute*), 770
encoding (*UnicodeError attribute*), 114
encodings_map (*in module mimetypes*), 1363
encodings_map (*mimetypes.MimeTypes attribute*), 1364
encodings.idna
    module, 207
encodings.mbcs
    module, 208
encodings.utf_8_sig
    module, 208
EncodingWarning, 117
end (*UnicodeError attribute*), 114
end() (*re.Match method*), 150
end() (*xml.etree.ElementTree.TreeBuilder method*), 1398
END_ASYNC_FOR (*opcode*), 2234
end_col_offset (*ast.AST attribute*), 2169
end_fill() (*in module turtle*), 1630
END_FOR (*opcode*), 2231
end_headers() (*http.server.BaseHTTPRequestHandler method*), 1541
end_lineno (*ast.AST attribute*), 2169
end_lineno (*SyntaxError attribute*), 113
end_lineno (*traceback.TracebackException attribute*), 2080
end_ns() (*xml.etree.ElementTree.TreeBuilder method*), 1398
end_offset (*SyntaxError attribute*), 113
end_offset (*traceback.TracebackException attribute*), 2080
end_poly() (*in module turtle*), 1635
END_SEND (*opcode*), 2231
endCDATA() (*xml.sax.handler.LexicalHandler method*), 1426
EndCdataSectionHandler() (*xml.parsers.expat.xmlparser method*), 1437
EndDoctypeDeclHandler() (*xml.parsers.expat.xmlparser method*), 1436
endDocument() (*xml.sax.handler.ContentHandler method*), 1423
endDTD() (*xml.sax.handler.LexicalHandler method*), 1426
endElement() (*xml.sax.handler.ContentHandler method*), 1424
EndElementHandler() (*xml.parsers.expat.xmlparser method*), 1436
endElementNS() (*xml.sax.handler.ContentHandler method*), 1424
endheaders() (*http.client.HTTPConnection method*), 1496
ENDMARKER (*in module token*), 2208
EndNamespaceDeclHandler() (*xml.parsers.expat.xmlparser method*), 1437
endpos (*re.Match attribute*), 150
endPrefixMapping() (*xml.sax.handler.ContentHandler method*), 1423
endswith() (*bytearray method*), 67
endswith() (*bytes method*), 67
endswith() (*str method*), 52
endwin() (*in module curses*), 875
ENETDOWN (*in module errno*), 918
ENETRESET (*in module errno*), 918
ENETUNREACH (*in module errno*), 918
ENFILE (*in module errno*), 914
ENOANO (*in module errno*), 916
ENOBUFS (*in module errno*), 918
ENOCSI (*in module errno*), 915
ENODATA (*in module errno*), 916
ENODEV (*in module errno*), 914
ENOENT (*in module errno*), 913
ENOEXEC (*in module errno*), 913
ENOLCK (*in module errno*), 915
ENOLINK (*in module errno*), 916
ENOMEM (*in module errno*), 913
ENOMSG (*in module errno*), 915
ENONET (*in module errno*), 916
ENOPKG (*in module errno*), 916
ENOPROTOOPT (*in module errno*), 918
ENOSPC (*in module errno*), 914
ENOSR (*in module errno*), 916
ENOSTR (*in module errno*), 916
ENOSYS (*in module errno*), 915
ENOTBLK (*in module errno*), 914
ENOTCAPABLE (*in module errno*), 919
ENOTCONN (*in module errno*), 918
ENOTDIR (*in module errno*), 914
ENOTEMPTY (*in module errno*), 915
ENOTNAM (*in module errno*), 919
ENOTRECOVERABLE (*in module errno*), 920

ENOTSOCK (*in module errno*), 917
ENOTSUP (*in module errno*), 918
ENOTTY (*in module errno*), 914
ENOTUNIQ (*in module errno*), 917
ENQ (*in module curses.ascii*), 903
enqueue() (*logging.handlers.QueueHandler method*), 871
enqueue_sentinel() (*logging.handlers.QueueListener method*), 872
ensure_directories() (*venv.EnvBuilder method*), 1979
ensure_future() (*in module asyncio*), 1138
ensurepip
    module, 1973
enter() (*sched.scheduler method*), 1053
enter_async_context() (*contextlib.AsyncExitStack method*), 2063
enter_context() (*contextlib.ExitStack method*), 2062
enterabs() (*sched.scheduler method*), 1053
enterAsyncContext() (*unittest.IsolatedAsyncioTestCase method*), 1823
enterClassContext() (*unittest.TestCase class method*), 1822
enterContext() (*unittest.TestCase method*), 1822
enterModuleContext() (*in module unittest*), 1834
entities (*xml.dom.DocumentType attribute*), 1406
EntityDeclHandler() (*xml.parsers.expat.xmlparser method*), 1437
entitydefs (*in module html.entities*), 1378
EntityResolver (*class in xml.sax.handler*), 1421
enum
    module, 323
Enum (*class in enum*), 327
enum_certificates() (*in module ssl*), 1209
enum_crls() (*in module ssl*), 1209
EnumCheck (*class in enum*), 334
enumerate()
    built-in function, 12
enumerate() (*in module threading*), 957
EnumKey() (*in module winreg*), 2256
EnumType (*class in enum*), 325
EnumValue() (*in module winreg*), 2256
EnvBuilder (*class in venv*), 1978
environ (*in module os*), 691
environ (*in module posix*), 2268
environb (*in module os*), 692
environment variable
    BROWSER, 1443, 1444
    COLUMNS, 881
    COMSPEC, 750, 1036
    DISPLAY, 1665
    HOME, 483, 484, 1665
    HOMEDRIVE, 483
    HOMEPATH, 483
    IDLESTARTUP, 1714
    LANG, 1593, 1594, 1602, 1606
    LANGUAGE, 1593, 1594
    LC_ALL, 1593, 1594
    LC_MESSAGES, 1593, 1594
    LINES, 875, 881
    LOGNAME, 694, 873
    no_proxy, 1461
    PAGER, 1775
    PATH, 480, 741, 747, 748, 758, 1035, 1443, 1977, 1978, 2112
    PYTHON_CPU_COUNT, 757, 984
    PYTHON_DOM, 1402
    PYTHON_GIL, 2327
    PYTHONASYNCIODEBUG, 1130, 1171, 1776
    PYTHONBREAKPOINT, 7, 1994
    PYTHONCASEOK, 32
    PYTHONCOERCECLOCALE, 691
    PYTHONDEVMODE, 1775
    PYTHONDONTWRITEBYTECODE, 1995
    PYTHONFAULTHANDLER, 1776, 1933
    PYTHONHOME, 1914, 2162, 2163
    PYTHONINTMAXSTRDIGITS, 105, 2006
    PYTHONIOENCODING, 690, 2014
    PYTHONLEGACYWINDOWSFSENCODING, 2013, 2014
    PYTHONLEGACYWINDOWSSTDIO, 2014
    PYTHONMALLOC, 1776
    PYTHONNOUSERSITE, 2113
    PYTHONPATH, 1914, 2008, 2162
    PYTHONPLATLIBDIR, 2162
    PYTHONPYCACHEPREFIX, 1995
    PYTHONSAFEPATH, 2008, 2319
    PYTHONSTARTUP, 180, 1066, 1714, 2006, 2113
    PYTHONTRACEMALLOC, 1962, 1968
    PYTHONTZPATH, 250, 254
    PYTHONUNBUFFERED, 2014
    PYTHONUSERBASE, 2113, 2114
    PYTHONUSERSITE, 1914
    PYTHONUTF8, 691, 2014
    PYTHONWARNDEFAULTENCODING, 761
    PYTHONWARNINGS, 1776, 2037, 2038
    SOURCE_DATE_EPOCH, 2220, 2222
    SSLKEYLOGFILE, 1206
    SystemRoot, 1038
    TEMP, 503
    TERM, 879
    TMP, 503
    TMPDIR, 503
    TZ, 783, 784
    USER, 873
    USERNAME, 483, 694, 873
    USERPROFILE, 483

environment variables
    deleting, 698
    setting, 695
EnvironmentError, 115
Environments
    virtual, 1975
EnvironmentVarGuard (*class in test.support.os_helper*), 1916
ENXIO (*in module errno*), 913
eof (*bz2.BZ2Decompressor attribute*), 598
eof (*lzma.LZMADecompressor attribute*), 603
eof (*shlex.shlex attribute*), 1660
eof (*ssl.MemoryBIO attribute*), 1239
eof (*zlib.Decompress attribute*), 590
eof_received() (*asyncio.BufferedProtocol method*), 1149
eof_received() (*asyncio.Protocol method*), 1148
EOFError, 109
EOPNOTSUPP (*in module errno*), 918
EOT (*in module curses.ascii*), 903
EOVERFLOW (*in module errno*), 917
EOWNERDEAD (*in module errno*), 919
EPERM (*in module errno*), 913
EPFNOSUPPORT (*in module errno*), 918
epilogue (*email.message.EmailMessage attribute*), 1280
epilogue (*email.message.Message attribute*), 1318
EPIPE (*in module errno*), 914
epoch, 775
epoll() (*in module select*), 1242
EpollSelector (*class in selectors*), 1252
EPROTO (*in module errno*), 916
EPROTONOSUPPORT (*in module errno*), 918
EPROTOTYPE (*in module errno*), 917
epsilon (*sys.float_info attribute*), 2000
Eq (*class in ast*), 2175
eq() (*in module operator*), 446
EQEQUAL (*in module token*), 2209
EQFULL (*in module errno*), 919
EQUAL (*in module token*), 2209
ERA (*in module locale*), 1605
ERA_D_FMT (*in module locale*), 1606
ERA_D_T_FMT (*in module locale*), 1606
ERA_T_FMT (*in module locale*), 1606
ERANGE (*in module errno*), 915
erase() (*curses.window method*), 883
erasechar() (*in module curses*), 875
EREMCHG (*in module errno*), 917
EREMOTE (*in module errno*), 916
EREMOTEIO (*in module errno*), 919
ERESTART (*in module errno*), 917
erf() (*in module math*), 353
erfc() (*in module math*), 353
EROFS (*in module errno*), 914
ERR (*in module curses*), 888
errcheck (*ctypes._FuncPtr attribute*), 943
errcode (*xmlrpc.client.ProtocolError attribute*), 1563
errmsg (*xmlrpc.client.ProtocolError attribute*), 1563
errno
    module, 111, 913
errno (*OSError attribute*), 111
Error, 312, 516, 575, 641, 663, 1360, 1371, 1444, 1589, 1601
error, 184, 546, 549551, 587, 689, 874, 1062, 1178, 1242, 1432, 2277, 2288
ERROR (*in module logging*), 830
ERROR (*in module tkinter.messagebox*), 1684
error handler's name
    backslashreplace, 194
    ignore, 194
    namereplace, 194
    replace, 194
    strict, 194
    surrogateescape, 194
    surrogatepass, 195
    xmlcharrefreplace, 194
error() (*argparse.ArgumentParser method*), 821
error() (*in module logging*), 839
error() (*logging.Logger method*), 828
error() (*urllib.request.OpenerDirector method*), 1464
error() (*xml.sax.handler.ErrorHandler method*), 1425
error_body (*wsgiref.handlers.BaseHandler attribute*), 1454
error_content_type (*http.server.BaseHTTPRequestHandler attribute*), 1539
error_headers (*wsgiref.handlers.BaseHandler attribute*), 1454
error_leader() (*shlex.shlex method*), 1659
error_message_format (*http.server.BaseHTTPRequestHandler attribute*), 1539
error_output() (*wsgiref.handlers.BaseHandler method*), 1454
error_perm, 1506
error_proto, 1506, 1508
error_received() (*asyncio.DatagramProtocol method*), 1149
error_reply, 1506
error_status (*wsgiref.handlers.BaseHandler attribute*), 1454
error_temp, 1506
ErrorByteIndex (*xml.parsers.expat.xmlparser attribute*), 1435
errorcode (*in module errno*), 913
ErrorCode (*xml.parsers.expat.xmlparser attribute*), 1435
ErrorColumnNumber (*xml.parsers.expat.xmlparser attribute*), 1435
ErrorHandler (*class in xml.sax.handler*), 1421

errorlevel (*tarfile.TarFile attribute*), 624
ErrorLineNumber (*xml.parsers.expat.xmlparser attribute*), 1435
Errors
    logging, 823
errors (*io.TextIOBase attribute*), 770
errors (*unittest.TestLoader attribute*), 1825
errors (*unittest.TestResult attribute*), 1828
ErrorStream (*class in wsgiref.types*), 1455
ErrorString() (*in module xml.parsers.expat*), 1432
ERRORTOKEN (*in module token*), 2211
ESC (*in module curses.ascii*), 905
escape (*shlex.shlex attribute*), 1659
escape() (*in module glob*), 507
escape() (*in module html*), 1373
escape() (*in module re*), 145
escape() (*in module xml.sax.saxutils*), 1426
escapechar (*csv.Dialect attribute*), 642
escapedquotes (*shlex.shlex attribute*), 1659
ESHUTDOWN (*in module errno*), 918
ESOCKTNOSUPPORT (*in module errno*), 918
ESPIPE (*in module errno*), 914
ESRCH (*in module errno*), 913
ESRMNT (*in module errno*), 916
ESTALE (*in module errno*), 919
ESTRPIPE (*in module errno*), 917
ETB (*in module curses.ascii*), 904
ETH_P_ALL (*in module socket*), 1182
ETHERTYPE_ARP (*in module socket*), 1184
ETHERTYPE_IP (*in module socket*), 1184
ETHERTYPE_IPV6 (*in module socket*), 1184
ETHERTYPE_VLAN (*in module socket*), 1184
ETIME (*in module errno*), 916
ETIMEDOUT (*in module errno*), 919
Etiny() (*decimal.Context method*), 374
ETOOMANYREFS (*in module errno*), 918
Etop() (*decimal.Context method*), 374
ETX (*in module curses.ascii*), 903
ETXTBSY (*in module errno*), 914
EUCLEAN (*in module errno*), 919
EUNATCH (*in module errno*), 915
EUSERS (*in module errno*), 917
eval
    built-in function, 102, 315, 316
eval()
    built-in function, 12
Event (*class in asyncio*), 1098
Event (*class in multiprocessing*), 988
Event (*class in threading*), 967
event scheduling, 1052
Event() (*multiprocessing.managers.SyncManager method*), 995
EVENT_READ (*in module selectors*), 1250
EVENT_WRITE (*in module selectors*), 1250
eventfd() (*in module os*), 735
eventfd_read() (*in module os*), 735
eventfd_write() (*in module os*), 736
EventLoop (*class in asyncio*), 1135
events (*selectors.SelectorKey attribute*), 1251
events (*widgets*), 1675
EWOULDBLOCK (*in module errno*), 915
EX_CANTCREAT (*in module os*), 743
EX_CONFIG (*in module os*), 743
EX_DATAERR (*in module os*), 742
EX_IOERR (*in module os*), 743
EX_NOHOST (*in module os*), 742
EX_NOINPUT (*in module os*), 742
EX_NOPERM (*in module os*), 743
EX_NOTFOUND (*in module os*), 743
EX_NOUSER (*in module os*), 742
EX_OK (*in module os*), 742
EX_OSERR (*in module os*), 742
EX_OSFILE (*in module os*), 743
EX_PROTOCOL (*in module os*), 743
EX_SOFTWARE (*in module os*), 742
EX_TEMPFAIL (*in module os*), 743
EX_UNAVAILABLE (*in module os*), 742
EX_USAGE (*in module os*), 742
--exact
    tokenize command line option, 2214
Example (*class in doctest*), 1795
example (*doctest.DocTestFailure attribute*), 1802
example (*doctest.UnexpectedException attribute*), 1802
examples (*doctest.DocTest attribute*), 1794
exc_info (*doctest.UnexpectedException attribute*), 1802
exc_info() (*in module sys*), 1996
exc_msg (*doctest.Example attribute*), 1795
exc_type (*traceback.TracebackException attribute*), 2080
exc_type_str (*traceback.TracebackException attribute*), 2080
excel (*class in csv*), 640
excel_tab (*class in csv*), 640
except
    statement, 107
ExceptHandler (*class in ast*), 2186
excepthook() (*in module sys*), 1995
excepthook() (*in module threading*), 956
Exception, 109
exception
    chaining, 107
EXCEPTION (*in module _tkinter*), 1677
exception() (*asyncio.Future method*), 1140
exception() (*asyncio.Task method*), 1085
exception() (*concurrent.futures.Future method*), 1028
exception() (*in module logging*), 840
exception() (*in module sys*), 1996
exception() (*logging.Logger method*), 828

EXCEPTION_HANDLED (*monitoring event*), 2019
ExceptionGroup, 117
exceptions (*BaseExceptionGroup attribute*), 118
exceptions (*pdb command*), 1943
exceptions (*traceback.TracebackException attribute*), 2079
EXCLAMATION (*in module token*), 2211
EXDEV (*in module errno*), 914
exec
    built-in function, 13, 102
exec()
    built-in function, 13
exec_module() (*importlib.abc.InspectLoader method*), 2134
exec_module() (*importlib.abc.Loader method*), 2132
exec_module() (*importlib.abc.SourceLoader method*), 2136
exec_module() (*importlib.machinery.ExtensionFileLoader method*), 2142
exec_module() (*zipimport.zipimporter method*), 2120
exec_prefix (*in module sys*), 1996
execl() (*in module os*), 741
execle() (*in module os*), 741
execlp() (*in module os*), 741
execlpe() (*in module os*), 741
executable (*in module sys*), 1997
Executable Zip Files, 1985
execute() (*sqlite3.Connection method*), 561
execute() (*sqlite3.Cursor method*), 571
executemany() (*sqlite3.Connection method*), 561
executemany() (*sqlite3.Cursor method*), 571
executescript() (*sqlite3.Connection method*), 561
executescript() (*sqlite3.Cursor method*), 572
ExecutionLoader (*class in importlib.abc*), 2134
Executor (*class in concurrent.futures*), 1023
execv() (*in module os*), 741
execve() (*in module os*), 741
execvp() (*in module os*), 741
execvpe() (*in module os*), 741
EXFULL (*in module errno*), 916
exists() (*in module os.path*), 483
exists() (*pathlib.Path method*), 470
exists() (*tkinter.ttk.Treeview method*), 1698
exists() (*zipfile.Path method*), 613
exit (*built-in variable*), 34
exit() (*argparse.ArgumentParser method*), 821
exit() (*in module _thread*), 1062
exit() (*in module sys*), 1997
exitcode (*multiprocessing.Process attribute*), 979
exitonclick() (*in module turtle*), 1643
ExitStack (*class in contextlib*), 2062
exp() (*decimal.Context method*), 375
exp() (*decimal.Decimal method*), 366
exp() (*in module cmath*), 356
exp() (*in module math*), 350
exp2() (*in module math*), 350
expand() (*re.Match method*), 148
expand_tabs (*textwrap.TextWrapper attribute*), 171
ExpandEnvironmentStrings() (*in module winreg*), 2256
expandNode() (*xml.dom.pulldom.DOMEventStream method*), 1418
expandtabs() (*bytearray method*), 72
expandtabs() (*bytes method*), 72
expandtabs() (*str method*), 53
expanduser() (*in module os.path*), 483
expanduser() (*pathlib.Path method*), 469
expandvars() (*in module os.path*), 484
Expat, 1432
ExpatError, 1432
expected (*asyncio.IncompleteReadError attribute*), 1112
expectedFailure() (*in module unittest*), 1812
expectedFailures (*unittest.TestResult attribute*), 1828
expired() (*asyncio.Timeout method*), 1079
expires (*http.cookiejar.Cookie attribute*), 1556
expires (*http.cookies.Morsel attribute*), 1546
exploded (*ipaddress.IPv4Address attribute*), 1574
exploded (*ipaddress.IPv4Network attribute*), 1580
exploded (*ipaddress.IPv6Address attribute*), 1576
exploded (*ipaddress.IPv6Network attribute*), 1583
expm1() (*in module math*), 351
expovariate() (*in module random*), 396
Expr (*class in ast*), 2174
expression, **2325**
Expression (*class in ast*), 2170
expunge() (*imaplib.IMAP4 method*), 1513
extend() (*array.array method*), 295
extend() (*collections.deque method*), 268
extend() (*sequence method*), 47
extend() (*xml.etree.ElementTree.Element method*), 1394
extend_path() (*in module pkgutil*), 2121
EXTENDED_ARG (*opcode*), 2243
ExtendedContext (*class in decimal*), 372
ExtendedInterpolation (*class in configparser*), 650
extendleft() (*collections.deque method*), 268
extension module, **2325**
EXTENSION_SUFFIXES (*in module importlib.machinery*), 2138
ExtensionFileLoader (*class in importlib.machinery*), 2141
extensions_map (*http.server.SimpleHTTPRequestHandler attribute*), 1542
External Data Representation, 525
external_attr (*zipfile.ZipInfo attribute*), 616

ExternalClashError, 1360
ExternalEntityParserCreate() (*xml.parsers.expat.xmlparser method*), 1434
ExternalEntityRefHandler() (*xml.parsers.expat.xmlparser method*), 1437
extra (*zipfile.ZipInfo attribute*), 615
--extract
  tarfile command line option, 632
  zipfile command line option, 617
extract() (*tarfile.TarFile method*), 624
extract() (*traceback.StackSummary class method*), 2081
extract() (*zipfile.ZipFile method*), 609
extract_cookies() (*http.cookiejar.CookieJar method*), 1551
extract_stack() (*in module traceback*), 2078
extract_tb() (*in module traceback*), 2077
extract_version (*zipfile.ZipInfo attribute*), 615
extractall() (*tarfile.TarFile method*), 623
extractall() (*zipfile.ZipFile method*), 610
ExtractError, 620
extractfile() (*tarfile.TarFile method*), 624
extraction_filter (*tarfile.TarFile attribute*), 625
extsep (*in module os*), 758

F

-f
  calendar command line option, 261
  compileall command line option, 2221
  random command line option, 401
  trace command line option, 1960
  unittest command line option, 1806
f-string, **2325**
F_CONTIGUOUS (*inspect.BufferFlags attribute*), 2110
f_contiguous (*memoryview attribute*), 86
F_LOCK (*in module os*), 703
F_OK (*in module os*), 714
F_TEST (*in module os*), 703
F_TLOCK (*in module os*), 703
F_ULOCK (*in module os*), 703
fabs() (*in module math*), 347
factorial() (*in module math*), 347
factory() (*importlib.util.LazyLoader class method*), 2147
fail() (*unittest.TestCase method*), 1821
FAIL_FAST (*in module doctest*), 1787
failed (*doctest.TestResults attribute*), 1797
--failfast
  unittest command line option, 1806
failfast (*unittest.TestResult attribute*), 1829
failureException, 1793
failureException (*unittest.TestCase attribute*), 1821
failures (*doctest.DocTestRunner attribute*), 1799
failures (*unittest.TestResult attribute*), 1828
FakePath (*class in test.support.os_helper*), 1917
False, 35, 43
false, 35
False (*Built-in object*), 35
False (*built-in variable*), 33
families() (*in module tkinter.font*), 1678
family (*socket.socket attribute*), 1199
FancyURLopener (*class in urllib.request*), 1475
--fast
  gzip command line option, 594
fast (*pickle.Pickler attribute*), 528
FastChildWatcher (*class in asyncio*), 1159
fatalError() (*xml.sax.handler.ErrorHandler method*), 1425
Fault (*class in xmlrpc.client*), 1563
faultCode (*xmlrpc.client.Fault attribute*), 1563
faulthandler
  module, 1933
faultString (*xmlrpc.client.Fault attribute*), 1563
fchdir() (*in module os*), 716
fchmod() (*in module os*), 701
fchown() (*in module os*), 701
fcntl
  module, 2274
fcntl() (*in module fcntl*), 2275
fd (*selectors.SelectorKey attribute*), 1251
fd() (*in module turtle*), 1619
fd_count() (*in module test.support.os_helper*), 1917
fdatasync() (*in module os*), 701
fdopen() (*in module os*), 699
feature_external_ges (*in module xml.sax.handler*), 1422
feature_external_pes (*in module xml.sax.handler*), 1422
feature_namespace_prefixes (*in module xml.sax.handler*), 1421
feature_namespaces (*in module xml.sax.handler*), 1421
feature_string_interning (*in module xml.sax.handler*), 1421
feature_validation (*in module xml.sax.handler*), 1422
FEBRUARY (*in module calendar*), 259
feed() (*email.parser.BytesFeedParser method*), 1281
feed() (*html.parser.HTMLParser method*), 1375
feed() (*xml.etree.ElementTree.XMLParser method*), 1399
feed() (*xml.etree.ElementTree.XMLPullParser method*), 1400
feed() (*xml.sax.xmlreader.IncrementalParser method*), 1430
feed_eof() (*asyncio.StreamReader method*), 1091
FeedParser (*class in email.parser*), 1281
fetch() (*imaplib.IMAP4 method*), 1513

fetchall() (*sqlite3.Cursor method*), 572
fetchmany() (*sqlite3.Cursor method*), 572
fetchone() (*sqlite3.Cursor method*), 572
FF (*in module curses.ascii*), 904
fflags (*select.kevent attribute*), 1248
Field (*class in dataclasses*), 2047
field() (*in module dataclasses*), 2046
field_size_limit() (*in module csv*), 638
fieldnames (*csv.DictReader attribute*), 643
fields (*uuid.UUID attribute*), 1525
fields() (*in module dataclasses*), 2047
FIFOTYPE (*in module tarfile*), 621
file
    byte-code, 2219
    compileall command line option, 2221
    configuration, 645
    copying, 510
    debugger configuration, 1939
    gzip command line option, 594
    .ini, 645
    large files, 2267
    mime.types, 1363
    modes, 21
    path configuration, 2112
    .pdbrc, 1939
    plist, 667
    temporary, 500
--file
    trace command line option, 1960
file (*bdb.Breakpoint attribute*), 1928
file (*pyclbr.Class attribute*), 2218
file (*pyclbr.Function attribute*), 2218
file control
    UNIX, 2274
file name
    temporary, 500
file object, **2325**
    io module, 760
    open() built-in function, 21
file-like object, **2325**
FILE_ATTRIBUTE_ARCHIVE (*in module stat*), 497
FILE_ATTRIBUTE_COMPRESSED (*in module stat*), 497
FILE_ATTRIBUTE_DEVICE (*in module stat*), 497
FILE_ATTRIBUTE_DIRECTORY (*in module stat*), 497
FILE_ATTRIBUTE_ENCRYPTED (*in module stat*), 497
FILE_ATTRIBUTE_HIDDEN (*in module stat*), 497
FILE_ATTRIBUTE_INTEGRITY_STREAM (*in module stat*), 497
FILE_ATTRIBUTE_NO_SCRUB_DATA (*in module stat*), 497
FILE_ATTRIBUTE_NORMAL (*in module stat*), 497
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED (*in module stat*), 497
FILE_ATTRIBUTE_OFFLINE (*in module stat*), 497
FILE_ATTRIBUTE_READONLY (*in module stat*), 497
FILE_ATTRIBUTE_REPARSE_POINT (*in module stat*), 497
FILE_ATTRIBUTE_SPARSE_FILE (*in module stat*), 497
FILE_ATTRIBUTE_SYSTEM (*in module stat*), 497
FILE_ATTRIBUTE_TEMPORARY (*in module stat*), 497
FILE_ATTRIBUTE_VIRTUAL (*in module stat*), 497
file_digest() (*in module hashlib*), 674
file_open() (*urllib.request.FileHandler method*), 1470
file_size (*zipfile.ZipInfo attribute*), 616
filecmp
    module, 498
fileConfig() (*in module logging.config*), 845
FileCookieJar (*class in http.cookiejar*), 1549
FileDialog (*class in tkinter.filedialog*), 1680
FileExistsError, 115
FileFinder (*class in importlib.machinery*), 2140
FileHandler (*class in logging*), 858
FileHandler (*class in urllib.request*), 1462
fileinput
    module, 489
FileInput (*class in fileinput*), 490
FileIO (*class in io*), 767
filelineno() (*in module fileinput*), 490
FileLoader (*class in importlib.abc*), 2135
filemode() (*in module stat*), 493
filename (*doctest.DocTest attribute*), 1794
filename (*http.cookiejar.FileCookieJar attribute*), 1552
filename (*inspect.FrameInfo attribute*), 2105
filename (*inspect.Traceback attribute*), 2105
filename (*netrc.NetrcParseError attribute*), 666
filename (*OSError attribute*), 111
filename (*SyntaxError attribute*), 113
filename (*traceback.FrameSummary attribute*), 2082
filename (*traceback.TracebackException attribute*), 2080
filename (*tracemalloc.Frame attribute*), 1969
filename (*zipfile.ZipFile attribute*), 612
filename (*zipfile.ZipInfo attribute*), 615
filename() (*in module fileinput*), 489
filename2 (*OSError attribute*), 111
filename_only (*in module tabnanny*), 2217
filename_pattern (*tracemalloc.Filter attribute*), 1969
filenames
    pathname expansion, 506
    wildcard expansion, 508
fileno() (*bz2.BZ2File method*), 596
fileno() (*http.client.HTTPResponse method*), 1497
fileno() (*in module fileinput*), 490
fileno() (*io.IOBase method*), 764

fileno() (*multiprocessing.connection.Connection method*), 986
fileno() (*select.devpoll method*), 1244
fileno() (*select.epoll method*), 1246
fileno() (*select.kqueue method*), 1247
fileno() (*selectors.DevpollSelector method*), 1252
fileno() (*selectors.EpollSelector method*), 1252
fileno() (*selectors.KqueueSelector method*), 1252
fileno() (*socketserver.BaseServer method*), 1531
fileno() (*socket.socket method*), 1193
FileNotFoundError, 115
fileobj (*selectors.SelectorKey attribute*), 1251
files() (*importlib.abc.TraversableResources method*), 2138
files() (*importlib.resources.abc.TraversableResources method*), 2155
files() (*in module importlib.resources*), 2150
files_double_event() (*tkinter.filedialog.FileDialog method*), 1681
files_select_event() (*tkinter.filedialog.FileDialog method*), 1681
filesystem encoding and error handler, **2325**
FileType (*class in argparse*), 817
FileWrapper (*class in wsgiref.types*), 1455
FileWrapper (*class in wsgiref.util*), 1448
fill() (*in module textwrap*), 169
fill() (*textwrap.TextWrapper method*), 173
fillcolor() (*in module turtle*), 1628
filling() (*in module turtle*), 1630
fillvalue (*reprlib.Repr attribute*), 321
--filter
    tarfile command line option, 633
Filter (*class in logging*), 834
Filter (*class in tracemalloc*), 1968
filter (*select.kevent attribute*), 1248
filter()
    built-in function, 14
filter() (*in module curses*), 875
filter() (*in module fnmatch*), 509
filter() (*logging.Filter method*), 834
filter() (*logging.Handler method*), 831
filter() (*logging.Logger method*), 829
filter_command() (*tkinter.filedialog.FileDialog method*), 1681
FILTER_DIR (*in module unittest.mock*), 1872
filter_traces() (*tracemalloc.Snapshot method*), 1970
FilterError, 620
filterfalse() (*in module itertools*), 425
filterwarnings() (*in module warnings*), 2042
Final (*in module typing*), 1737
final() (*in module typing*), 1762
finalize (*class in weakref*), 300
find() (*bytearray method*), 67
find() (*bytes method*), 67
find() (*doctest.DocTestFinder method*), 1796
find() (*in module gettext*), 1594
find() (*mmap.mmap method*), 1266
find() (*str method*), 53
find() (*xml.etree.ElementTree.Element method*), 1394
find() (*xml.etree.ElementTree.ElementTree method*), 1396
find_class() (*pickle protocol*), 538
find_class() (*pickle.Unpickler method*), 528
find_library() (*in module ctypes.util*), 947
find_loader() (*in module pkgutil*), 2122
find_longest_match() (*difflib.SequenceMatcher method*), 161
find_msvcrt() (*in module ctypes.util*), 947
find_spec() (*importlib.abc.MetaPathFinder method*), 2131
find_spec() (*importlib.abc.PathEntryFinder method*), 2132
find_spec() (*importlib.machinery.FileFinder method*), 2140
find_spec() (*importlib.machinery.PathFinder class method*), 2139
find_spec() (*in module importlib.util*), 2145
find_spec() (*zipimport.zipimporter method*), 2120
find_unused_port() (*in module test.support.socket_helper*), 1913
find_user_password() (*urllib.request.HTTPPasswordMgr method*), 1468
find_user_password() (*urllib.request.HTTPPasswordMgrWithPriorAuth method*), 1468
findall() (*in module re*), 143
findall() (*re.Pattern method*), 147
findall() (*xml.etree.ElementTree.Element method*), 1394
findall() (*xml.etree.ElementTree.ElementTree method*), 1396
findCaller() (*logging.Logger method*), 829
finder, **2326**
findfile() (*in module test.support*), 1906
finditer() (*in module re*), 144
finditer() (*re.Pattern method*), 147
findlabels() (*in module dis*), 2229
findlinestarts() (*in module dis*), 2229
findtext() (*xml.etree.ElementTree.Element method*), 1394
findtext() (*xml.etree.ElementTree.ElementTree method*), 1396
finish() (*socketserver.BaseRequestHandler method*), 1534
finish() (*tkinter.dnd.DndHandler method*), 1685
finish_request() (*socketserver.BaseServer method*),

1533
FIRST_COMPLETED (*in module asyncio*), 1081
FIRST_COMPLETED (*in module concurrent.futures*), 1029
FIRST_EXCEPTION (*in module asyncio*), 1081
FIRST_EXCEPTION (*in module concurrent.futures*), 1029
firstChild (*xml.dom.Node attribute*), 1404
firstkey() (*dbm.gnu.gdbm method*), 549
--first-weekday
calendar command line option, 261
firstweekday() (*in module calendar*), 257
fix_missing_locations() (*in module ast*), 2200
fix_sentence_endings (*textwrap.TextWrapper attribute*), 172
Flag (*class in enum*), 331
flag_bits (*zipfile.ZipInfo attribute*), 616
FlagBoundary (*class in enum*), 335
flags (*in module sys*), 1997
flags (*re.Pattern attribute*), 147
flags (*select.kevent attribute*), 1248
flash() (*in module curses*), 875
flatten() (*email.generator.BytesGenerator method*), 1285
flatten() (*email.generator.Generator method*), 1286
flattening
objects, 523
float
built-in function, 37
--float
random command line option, 401
float (*built-in class*), 14
float_info (*in module sys*), 1999
float_repr_style (*in module sys*), 2001
floating-point
literals, 36
object, 36
FloatingPointError, 109
FloatOperation (*class in decimal*), 380
flock() (*in module fcntl*), 2276
floor division, **2326**
floor() (*in module math*), 37, 347
FloorDiv (*class in ast*), 2174
floordiv() (*in module operator*), 447
flush() (*bz2.BZ2Compressor method*), 597
flush() (*io.BufferedWriter method*), 769
flush() (*io.IOBase method*), 764
flush() (*logging.Handler method*), 831
flush() (*logging.handlers.BufferingHandler method*), 869
flush() (*logging.handlers.MemoryHandler method*), 869
flush() (*logging.StreamHandler method*), 858
flush() (*lzma.LZMACompressor method*), 602
flush() (*mailbox.Mailbox method*), 1344
flush() (*mailbox.Maildir method*), 1347
flush() (*mailbox.MH method*), 1349
flush() (*mmap.mmap method*), 1266
flush() (*xml.etree.ElementTree.XMLParser method*), 1399
flush() (*xml.etree.ElementTree.XMLPullParser method*), 1400
flush() (*zlib.Compress method*), 589
flush() (*zlib.Decompress method*), 590
flush_headers() (*http.server.BaseHTTPRequestHandler method*), 1541
flush_std_streams() (*in module test.support*), 1908
flushinp() (*in module curses*), 875
FlushKey() (*in module winreg*), 2256
fma() (*decimal.Context method*), 375
fma() (*decimal.Decimal method*), 366
fma() (*in module math*), 347
fmean() (*in module statistics*), 405
fmod() (*in module math*), 347
FMT_BINARY (*in module plistlib*), 668
FMT_XML (*in module plistlib*), 668
fnmatch
module, 508
fnmatch() (*in module fnmatch*), 509
fnmatchcase() (*in module fnmatch*), 509
focus() (*tkinter.ttk.Treeview method*), 1698
fold (*datetime.datetime attribute*), 224
fold (*datetime.time attribute*), 233
fold() (*email.headerregistry.BaseHeader method*), 1296
fold() (*email.policy.Compat32 method*), 1293
fold() (*email.policy.EmailPolicy method*), 1292
fold() (*email.policy.Policy method*), 1290
fold_binary() (*email.policy.Compat32 method*), 1293
fold_binary() (*email.policy.EmailPolicy method*), 1292
fold_binary() (*email.policy.Policy method*), 1290
Font (*class in tkinter.font*), 1678
For (*class in ast*), 2184
FOR_ITER (*opcode*), 2241
forget() (*in module test.support.import_helper*), 1918
forget() (*tkinter.ttk.Notebook method*), 1693
fork() (*in module os*), 743
fork() (*in module pty*), 2272
ForkingMixIn (*class in socketserver*), 1530
ForkingTCPServer (*class in socketserver*), 1531
ForkingUDPServer (*class in socketserver*), 1531
ForkingUnixDatagramServer (*class in socketserver*), 1531
ForkingUnixStreamServer (*class in socketserver*), 1531
forkpty() (*in module os*), 744
FORMAT (*inspect.BufferFlags attribute*), 2110

format (*memoryview attribute*), 85
format (*multiprocessing.shared_memory.ShareableList attribute*), 1021
format (*struct.Struct attribute*), 190
format()
    built-in function, 15
format() (*inspect.Signature method*), 2098
format() (*logging.BufferingFormatter method*), 833
format() (*logging.Formatter method*), 832
format() (*logging.Handler method*), 831
format() (*pprint.PrettyPrinter method*), 316
format() (*str method*), 53
format() (*string.Formatter method*), 122
format() (*traceback.StackSummary method*), 2081
format() (*traceback.TracebackException method*), 2080
format() (*tracemalloc.Traceback method*), 1972
format_datetime() (*in module email.utils*), 1329
format_exc() (*in module traceback*), 2078
format_exception() (*in module traceback*), 2078
format_exception_only() (*in module traceback*), 2078
format_exception_only() (*traceback.TracebackException method*), 2080
format_field() (*string.Formatter method*), 123
format_frame_summary() (*traceback.StackSummary method*), 2081
format_help() (*argparse.ArgumentParser method*), 820
format_list() (*in module traceback*), 2078
format_map() (*str method*), 53
FORMAT_SIMPLE (*opcode*), 2244
FORMAT_SPEC (*opcode*), 2244
format_stack() (*in module traceback*), 2078
format_stack_entry() (*bdb.Bdb method*), 1932
format_string() (*in module locale*), 1607
format_tb() (*in module traceback*), 2078
format_usage() (*argparse.ArgumentParser method*), 820
formataddr() (*in module email.utils*), 1327
formatargvalues() (*in module inspect*), 2103
formatdate() (*in module email.utils*), 1328
FormatError, 1360
FormatError() (*in module ctypes*), 947
formatException() (*logging.Formatter method*), 833
formatFooter() (*logging.BufferingFormatter method*), 833
formatHeader() (*logging.BufferingFormatter method*), 833
formatmonth() (*calendar.HTMLCalendar method*), 256
formatmonth() (*calendar.TextCalendar method*), 255
formatmonthname() (*calendar.HTMLCalendar method*), 256
formatStack() (*logging.Formatter method*), 833
FormattedValue (*class in ast*), 2171
Formatter (*class in logging*), 832
Formatter (*class in string*), 122
formatTime() (*logging.Formatter method*), 832
formatting
    bytearray (%), 77
    bytes (%), 77
formatting, string (%), 60
formatwarning() (*in module warnings*), 2042
formatyear() (*calendar.HTMLCalendar method*), 256
formatyear() (*calendar.TextCalendar method*), 255
formatyearpage() (*calendar.HTMLCalendar method*), 256
Fortran contiguous, 2324
forward() (*in module turtle*), 1619
ForwardRef (*class in typing*), 1765
fp (*urllib.error.HTTPError attribute*), 1486
fpathconf() (*in module os*), 701
Fraction (*class in fractions*), 388
fractions
    module, 388
Frame (*class in tracemalloc*), 1969
frame (*inspect.FrameInfo attribute*), 2105
frame (*tkinter.scrolledtext.ScrolledText attribute*), 1684
FrameInfo (*class in inspect*), 2105
FrameSummary (*class in traceback*), 2082
FrameType (*in module types*), 310
free threading, **2326**
free_tool_id() (*in module sys.monitoring*), 2018
freedesktop_os_release() (*in module platform*), 912
freeze() (*in module gc*), 2089
freeze_support() (*in module multiprocessing*), 984
frexp() (*in module math*), 347
FRIDAY (*in module calendar*), 258
from_address() (*ctypes._CData method*), 949
from_buffer() (*ctypes._CData method*), 948
from_buffer_copy() (*ctypes._CData method*), 948
from_bytes() (*int class method*), 40
from_callable() (*inspect.Signature class method*), 2098
from_decimal() (*fractions.Fraction class method*), 390
from_exception() (*traceback.TracebackException class method*), 2080
from_file() (*zipfile.ZipInfo class method*), 614
from_file() (*zoneinfo.ZoneInfo class method*), 250
from_float() (*decimal.Decimal class method*), 366
from_float() (*fractions.Fraction class method*), 390
from_iterable() (*itertools.chain class method*), 422
from_list() (*traceback.StackSummary class method*), 2081
from_param() (*ctypes._CData method*), 949

from_samples() (*statistics.NormalDist class method*), 414
from_traceback() (*dis.Bytecode class method*), 2227
from_uri() (*pathlib.Path class method*), 468
frombuf() (*tarfile.TarInfo class method*), 626
frombytes() (*array.array method*), 295
fromfd() (*in module socket*), 1186
fromfd() (*select.epoll method*), 1246
fromfd() (*select.kqueue method*), 1247
fromfile() (*array.array method*), 295
fromhex() (*bytearray class method*), 64
fromhex() (*bytes class method*), 63
fromhex() (*float class method*), 41
fromisocalendar() (*datetime.date class method*), 216
fromisocalendar() (*datetime.datetime class method*), 223
fromisoformat() (*datetime.date class method*), 215
fromisoformat() (*datetime.datetime class method*), 222
fromisoformat() (*datetime.time class method*), 233
fromkeys() (*collections.Counter method*), 266
fromkeys() (*dict class method*), 90
fromlist() (*array.array method*), 295
fromordinal() (*datetime.date class method*), 215
fromordinal() (*datetime.datetime class method*), 222
fromshare() (*in module socket*), 1186
fromstring() (*in module xml.etree.ElementTree*), 1390
fromstringlist() (*in module xml.etree.ElementTree*), 1390
fromtarfile() (*tarfile.TarInfo class method*), 626
fromtimestamp() (*datetime.date class method*), 215
fromtimestamp() (*datetime.datetime class method*), 221
fromunicode() (*array.array method*), 295
fromutc() (*datetime.timezone method*), 243
fromutc() (*datetime.tzinfo method*), 238
FrozenImporter (*class in importlib.machinery*), 2139
FrozenInstanceError, 2050
frozenset (*built-in class*), 86
FrozenSet (*class in typing*), 1767
FS (*in module curses.ascii*), 905
fs_is_case_insensitive() (*in module test.support.os_helper*), 1917
FS_NONASCII (*in module test.support.os_helper*), 1916
fsdecode() (*in module os*), 692
fsencode() (*in module os*), 692
fspath() (*in module os*), 692
fstat() (*in module os*), 702
fstatvfs() (*in module os*), 702
FSTRING_END (*in module token*), 2211
FSTRING_MIDDLE (*in module token*), 2211
FSTRING_START (*in module token*), 2211
fsum() (*in module math*), 347
fsync() (*in module os*), 702
FTP, 1476
    ftplib (*standard module*), 1500
    protocol, 1476, 1500
FTP (*class in ftplib*), 1500
ftp_open() (*urllib.request.FTPHandler method*), 1470
FTP_TLS (*class in ftplib*), 1505
FTPHandler (*class in urllib.request*), 1462
ftplib
    module, 1500
ftruncate() (*in module os*), 702
Full, 1055
FULL (*inspect.BufferFlags attribute*), 2110
full() (*asyncio.Queue method*), 1108
full() (*multiprocessing.Queue method*), 982
full() (*queue.Queue method*), 1055
full_match() (*pathlib.PurePath method*), 464
FULL_RO (*inspect.BufferFlags attribute*), 2110
full_url (*urllib.request.Request attribute*), 1462
fullmatch() (*in module re*), 143
fullmatch() (*re.Pattern method*), 147
fully_trusted_filter() (*in module tarfile*), 629
func (*functools.partial attribute*), 446
funcname (*bdb.Breakpoint attribute*), 1928
function, **2326**
Function (*class in pyclbr*), 2218
Function (*class in symtable*), 2206
function (*inspect.FrameInfo attribute*), 2105
function (*inspect.Traceback attribute*), 2105
FUNCTION (*symtable.SymbolTableType attribute*), 2204
function annotation, **2326**
FunctionDef (*class in ast*), 2195
FunctionTestCase (*class in unittest*), 1824
FunctionType (*class in ast*), 2170
FunctionType (*in module types*), 307
functools
    module, 436
funny_files (*filecmp.dircmp attribute*), 499
Future (*class in asyncio*), 1139
Future (*class in concurrent.futures*), 1028
FutureWarning, 117
fwalk() (*in module os*), 733

G

-g
    trace command line option, 1960
gaierror, 1178
gamma() (*in module math*), 353
gammavariate() (*in module random*), 396
garbage (*in module gc*), 2089
garbage collection, **2326**
gather() (*curses.textpad.Textbox method*), 902
gather() (*in module asyncio*), 1075
gauss() (*in module random*), 396

gc
    module, 2086
gc_collect() (*in module test.support*), 1907
gcd() (*in module math*), 347
ge() (*in module operator*), 446
generate_tokens() (*in module tokenize*), 2213
generator, 2326, **2327**
Generator (*class in collections.abc*), 283
Generator (*class in email.generator*), 1285
Generator (*class in typing*), 1772
generator expression, **2327**
generator iterator, **2327**
GeneratorExit, 109
GeneratorExp (*class in ast*), 2177
GeneratorType (*in module types*), 307
Generic
    Alias, 95
Generic (*class in typing*), 1742
generic function, **2327**
generic type, **2327**
generic_visit() (*ast.NodeVisitor method*), 2201
GenericAlias
    object, 95
GenericAlias (*class in types*), 309
genops() (*in module pickletools*), 2249
geometric_mean() (*in module statistics*), 405
get() (*asyncio.Queue method*), 1108
get() (*configparser.ConfigParser method*), 661
get() (*contextvars.Context method*), 1060
get() (*contextvars.ContextVar method*), 1058
get() (*dict method*), 90
get() (*email.message.EmailMessage method*), 1275
get() (*email.message.Message method*), 1314
get() (*in module webbrowser*), 1444
get() (*mailbox.Mailbox method*), 1343
get() (*multiprocessing.pool.AsyncResult method*), 1002
get() (*multiprocessing.Queue method*), 982
get() (*multiprocessing.SimpleQueue method*), 983
get() (*queue.Queue method*), 1055
get() (*queue.SimpleQueue method*), 1057
get() (*tkinter.ttk.Combobox method*), 1690
get() (*tkinter.ttk.Spinbox method*), 1691
get() (*types.MappingProxyType method*), 310
get() (*xml.etree.ElementTree.Element method*), 1394
GET_AITER (*opcode*), 2233
get_all() (*email.message.EmailMessage method*), 1275
get_all() (*email.message.Message method*), 1314
get_all() (*wsgiref.headers.Headers method*), 1448
get_all_breaks() (*bdb.Bdb method*), 1931
get_all_start_methods() (*in module multiprocessing*), 984
GET_ANEXT (*opcode*), 2234
get_annotations() (*in module inspect*), 2104
get_app() (*wsgiref.simple_server.WSGIServer method*), 1450
get_archive_formats() (*in module shutil*), 518
get_args() (*in module typing*), 1764
get_asyncgen_hooks() (*in module sys*), 2004
get_attribute() (*in module test.support*), 1910
GET_AWAITABLE (*opcode*), 2233
get_begidx() (*in module readline*), 180
get_blocking() (*in module os*), 702
get_body() (*email.message.EmailMessage method*), 1278
get_body_encoding() (*email.charset.Charset method*), 1325
get_boundary() (*email.message.EmailMessage method*), 1276
get_boundary() (*email.message.Message method*), 1317
get_bpbynumber() (*bdb.Bdb method*), 1931
get_break() (*bdb.Bdb method*), 1931
get_breaks() (*bdb.Bdb method*), 1931
get_buffer() (*asyncio.BufferedProtocol method*), 1149
get_bytes() (*mailbox.Mailbox method*), 1343
get_ca_certs() (*ssl.SSLContext method*), 1224
get_cache_token() (*in module abc*), 2074
get_channel_binding() (*ssl.SSLSocket method*), 1220
get_charset() (*email.message.Message method*), 1313
get_charsets() (*email.message.EmailMessage method*), 1277
get_charsets() (*email.message.Message method*), 1317
get_child_watcher() (*asyncio.AbstractEventLoopPolicy method*), 1157
get_child_watcher() (*in module asyncio*), 1158
get_children() (*symtable.SymbolTable method*), 2206
get_children() (*tkinter.ttk.Treeview method*), 1697
get_ciphers() (*ssl.SSLContext method*), 1224
get_clock_info() (*in module time*), 777
get_close_matches() (*in module difflib*), 158
get_code() (*importlib.abc.InspectLoader method*), 2134
get_code() (*importlib.abc.SourceLoader method*), 2136
get_code() (*importlib.machinery.ExtensionFileLoader method*), 2142
get_code() (*importlib.machinery.SourcelessFileLoader method*), 2141
get_code() (*zipimport.zipimporter method*), 2120
get_completer() (*in module readline*), 179
get_completer_delims() (*in module readline*), 180

get_completion_type() (*in module readline*), 179
get_config_h_filename() (*in module sysconfig*), 2029
get_config_var() (*in module sysconfig*), 2023
get_config_vars() (*in module sysconfig*), 2023
get_content() (*email.contentmanager.ContentManager method*), 1301
get_content() (*email.message.EmailMessage method*), 1279
get_content() (*in module email.contentmanager*), 1302
get_content_charset() (*email.message.EmailMessage method*), 1277
get_content_charset() (*email.message.Message method*), 1317
get_content_disposition() (*email.message.EmailMessage method*), 1277
get_content_disposition() (*email.message.Message method*), 1317
get_content_maintype() (*email.message.EmailMessage method*), 1275
get_content_maintype() (*email.message.Message method*), 1315
get_content_subtype() (*email.message.EmailMessage method*), 1276
get_content_subtype() (*email.message.Message method*), 1315
get_content_type() (*email.message.EmailMessage method*), 1275
get_content_type() (*email.message.Message method*), 1315
get_context() (*asyncio.Handle method*), 1132
get_context() (*asyncio.Task method*), 1086
get_context() (*in module multiprocessing*), 985
get_coro() (*asyncio.Task method*), 1086
get_coroutine_origin_tracking_depth() (*in module sys*), 2004
get_count() (*in module gc*), 2087
get_current_history_length() (*in module readline*), 178
get_data() (*importlib.abc.FileLoader method*), 2135
get_data() (*importlib.abc.ResourceLoader method*), 2133
get_data() (*in module pkgutil*), 2124
get_data() (*zipimport.zipimporter method*), 2120
get_date() (*mailbox.MaildirMessage method*), 1353
get_debug() (*asyncio.loop method*), 1130
get_debug() (*in module gc*), 2087
get_default() (*argparse.ArgumentParser method*), 820
get_default_scheme() (*in module sysconfig*), 2027
get_default_type() (*email.message.EmailMessage method*), 1276
get_default_type() (*email.message.Message method*), 1315
get_default_verify_paths() (*in module ssl*), 1209
get_dialect() (*in module csv*), 638
get_disassembly_as_string() (*test.support.bytecode_helper.BytecodeTestCase method*), 1915
get_docstring() (*in module ast*), 2200
get_doctest() (*doctest.DocTestParser method*), 1797
get_endidx() (*in module readline*), 180
get_environ() (*wsgiref.simple_server.WSGIRequestHandler method*), 1450
get_errno() (*in module ctypes*), 947
get_escdelay() (*in module curses*), 879
get_event_loop() (*asyncio.AbstractEventLoopPolicy method*), 1157
get_event_loop() (*in module asyncio*), 1112
get_event_loop_policy() (*in module asyncio*), 1156
get_events() (*in module sys.monitoring*), 2021
get_examples() (*doctest.DocTestParser method*), 1797
get_exception_handler() (*asyncio.loop method*), 1129
get_exec_path() (*in module os*), 693
get_extra_info() (*asyncio.BaseTransport method*), 1143
get_extra_info() (*asyncio.StreamWriter method*), 1092
get_field() (*string.Formatter method*), 122
get_file() (*mailbox.Babyl method*), 1350
get_file() (*mailbox.Mailbox method*), 1343
get_file() (*mailbox.Maildir method*), 1347
get_file() (*mailbox.mbox method*), 1348
get_file() (*mailbox.MH method*), 1349
get_file() (*mailbox.MMDF method*), 1351
get_file_breaks() (*bdb.Bdb method*), 1931
get_filename() (*email.message.EmailMessage method*), 1276
get_filename() (*email.message.Message method*), 1316
get_filename() (*importlib.abc.ExecutionLoader method*), 2134
get_filename() (*importlib.abc.FileLoader method*), 2135
get_filename() (*importlib.machinery.ExtensionFileLoader method*), 2142
get_filename() (*zipimport.zipimporter method*), 2120
get_filter() (*tkinter.filedialog.FileDialog method*), 1681
get_flags() (*mailbox.Maildir method*), 1346
get_flags() (*mailbox.MaildirMessage method*), 1353

get_flags() (*mailbox.mboxMessage method*), 1355
get_flags() (*mailbox.MMDFMessage method*), 1359
get_folder() (*mailbox.Maildir method*), 1345
get_folder() (*mailbox.MH method*), 1349
get_frees() (*symtable.Function method*), 2206
get_freeze_count() (*in module gc*), 2089
get_from() (*mailbox.mboxMessage method*), 1354
get_from() (*mailbox.MMDFMessage method*), 1359
get_full_url() (*urllib.request.Request method*), 1463
get_globals() (*symtable.Function method*), 2206
get_grouped_opcodes() (*difflib.SequenceMatcher method*), 163
get_handle_inheritable() (*in module os*), 712
get_header() (*urllib.request.Request method*), 1464
get_history_item() (*in module readline*), 178
get_history_length() (*in module readline*), 178
get_id() (*symtable.SymbolTable method*), 2205
get_ident() (*in module _thread*), 1062
get_ident() (*in module threading*), 956
get_identifiers() (*string.Template method*), 131
get_identifiers() (*symtable.SymbolTable method*), 2205
get_importer() (*in module pkgutil*), 2122
get_info() (*mailbox.Maildir method*), 1346
get_info() (*mailbox.MaildirMessage method*), 1353
get_inheritable() (*in module os*), 712
get_inheritable() (*socket.socket method*), 1193
get_instructions() (*in module dis*), 2229
get_int_max_str_digits() (*in module sys*), 2002
get_interpreter() (*in module zipapp*), 1987
GET_ITER (*opcode*), 2232
get_key() (*selectors.BaseSelector method*), 1252
get_labels() (*mailbox.Babyl method*), 1350
get_labels() (*mailbox.BabylMessage method*), 1357
get_last_error() (*in module ctypes*), 947
GET_LEN (*opcode*), 2236
get_line_buffer() (*in module readline*), 178
get_lineno() (*symtable.SymbolTable method*), 2205
get_loader() (*in module pkgutil*), 2122
get_local_events() (*in module sys.monitoring*), 2021
get_locals() (*symtable.Function method*), 2206
get_logger() (*in module multiprocessing*), 1006
get_loop() (*asyncio.Future method*), 1140
get_loop() (*asyncio.Runner method*), 1068
get_loop() (*asyncio.Server method*), 1133
get_makefile_filename() (*in module sysconfig*), 2029
get_map() (*selectors.BaseSelector method*), 1252
get_matching_blocks() (*difflib.SequenceMatcher method*), 162
get_message() (*mailbox.Mailbox method*), 1343
get_method() (*urllib.request.Request method*), 1463
get_methods() (*symtable.Class method*), 2206
get_mixed_type_key() (*in module ipaddress*), 1587
get_name() (*asyncio.Task method*), 1086
get_name() (*symtable.Symbol method*), 2207
get_name() (*symtable.SymbolTable method*), 2205
get_namespace() (*symtable.Symbol method*), 2207
get_namespaces() (*symtable.Symbol method*), 2207
get_native_id() (*in module _thread*), 1063
get_native_id() (*in module threading*), 956
get_nonlocals() (*symtable.Function method*), 2206
get_nonstandard_attr() (*http.cookiejar.Cookie method*), 1557
get_nowait() (*asyncio.Queue method*), 1109
get_nowait() (*multiprocessing.Queue method*), 982
get_nowait() (*queue.Queue method*), 1056
get_nowait() (*queue.SimpleQueue method*), 1057
get_object_traceback() (*in module tracemalloc*), 1967
get_objects() (*in module gc*), 2087
get_opcodes() (*difflib.SequenceMatcher method*), 162
get_option() (*optparse.OptionParser method*), 2308
get_option_group() (*optparse.OptionParser method*), 2298
get_origin() (*in module typing*), 1764
get_original_bases() (*in module types*), 306
get_original_stdout() (*in module test.support*), 1907
get_osfhandle() (*in module msvcrt*), 2252
get_output_charset() (*email.charset.Charset method*), 1325
get_overloads() (*in module typing*), 1761
get_pagesize() (*in module test.support*), 1906
get_param() (*email.message.Message method*), 1316
get_parameters() (*symtable.Function method*), 2206
get_params() (*email.message.Message method*), 1315
get_path() (*in module sysconfig*), 2027
get_path_names() (*in module sysconfig*), 2027
get_paths() (*in module sysconfig*), 2028
get_payload() (*email.message.Message method*), 1312
get_pid() (*asyncio.SubprocessTransport method*), 1146
get_pipe_transport() (*asyncio.SubprocessTransport method*), 1146
get_platform() (*in module sysconfig*), 2028
get_poly() (*in module turtle*), 1635
get_preferred_scheme() (*in module sysconfig*), 2027
get_protocol() (*asyncio.BaseTransport method*), 1144
get_protocol_members() (*in module typing*), 1764
get_proxy_response_headers() (*http.client.HTTPConnection method*), 1496
get_python_version() (*in module sysconfig*), 2028
get_ready() (*graphlib.TopologicalSorter method*), 340
get_referents() (*in module gc*), 2088

get_referrers() (*in module gc*), 2087
get_request() (*socketserver.BaseServer method*), 1533
get_returncode() (*asyncio.SubprocessTransport method*), 1146
get_running_loop() (*in module asyncio*), 1112
get_scheme() (*wsgiref.handlers.BaseHandler method*), 1453
get_scheme_names() (*in module sysconfig*), 2027
get_selection() (*tkinter.filedialog.FileDialog method*), 1681
get_sequences() (*mailbox.MH method*), 1349
get_sequences() (*mailbox.MHMessage method*), 1356
get_server() (*multiprocessing.managers.BaseManager method*), 993
get_server_certificate() (*in module ssl*), 1208
get_shapepoly() (*in module turtle*), 1634
get_source() (*importlib.abc.InspectLoader method*), 2134
get_source() (*importlib.abc.SourceLoader method*), 2136
get_source() (*importlib.machinery.ExtensionFileLoader method*), 2142
get_source() (*importlib.machinery.SourcelessFileLoader method*), 2141
get_source() (*zipimport.zipimporter method*), 2120
get_source_segment() (*in module ast*), 2200
get_stack() (*asyncio.Task method*), 1085
get_stack() (*bdb.Bdb method*), 1931
get_start_method() (*in module multiprocessing*), 985
get_starttag_text() (*html.parser.HTMLParser method*), 1375
get_stats() (*in module gc*), 2087
get_stats_profile() (*pstats.Stats method*), 1951
get_stderr() (*wsgiref.handlers.BaseHandler method*), 1453
get_stderr() (*wsgiref.simple_server.WSGIRequestHandler method*), 1450
get_stdin() (*wsgiref.handlers.BaseHandler method*), 1452
get_string() (*mailbox.Mailbox method*), 1343
get_subdir() (*mailbox.MaildirMessage method*), 1352
get_symbols() (*symtable.SymbolTable method*), 2206
get_tabsize() (*in module curses*), 879
get_task_factory() (*asyncio.loop method*), 1117
get_terminal_size() (*in module os*), 712
get_terminal_size() (*in module shutil*), 521
get_threshold() (*in module gc*), 2087
get_token() (*shlex.shlex method*), 1658
get_tool() (*in module sys.monitoring*), 2018
get_traceback_limit() (*in module tracemalloc*), 1967
get_traced_memory() (*in module tracemalloc*), 1967
get_tracemalloc_memory() (*in module tracemalloc*), 1967
get_type() (*symtable.SymbolTable method*), 2205
get_type_hints() (*in module typing*), 1763
get_unixfrom() (*email.message.EmailMessage method*), 1274
get_unixfrom() (*email.message.Message method*), 1312
get_unpack_formats() (*in module shutil*), 519
get_unverified_chain() (*ssl.SSLSocket method*), 1220
get_usage() (*optparse.OptionParser method*), 2310
get_value() (*string.Formatter method*), 122
get_verified_chain() (*ssl.SSLSocket method*), 1220
get_version() (*optparse.OptionParser method*), 2299
get_visible() (*mailbox.BabylMessage method*), 1357
get_wch() (*curses.window method*), 884
get_write_buffer_limits() (*asyncio.WriteTransport method*), 1145
get_write_buffer_size() (*asyncio.WriteTransport method*), 1145
GET_YIELD_FROM_ITER (*opcode*), 2232
getacl() (*imaplib.IMAP4 method*), 1513
getaddresses() (*in module email.utils*), 1328
getaddrinfo() (*asyncio.loop method*), 1126
getaddrinfo() (*in module socket*), 1186
getallocatedblocks() (*in module sys*), 2001
getandroidapilevel() (*in module sys*), 2001
getannotation() (*imaplib.IMAP4 method*), 1513
getargvalues() (*in module inspect*), 2102
getasyncgenlocals() (*in module inspect*), 2109
getasyncgenstate() (*in module inspect*), 2108
getatime() (*in module os.path*), 484
getattr()
    built-in function, 15
getattr_static() (*in module inspect*), 2107
getAttribute() (*xml.dom.Element method*), 1408
getAttributeNode() (*xml.dom.Element method*), 1408
getAttributeNodeNS() (*xml.dom.Element method*), 1408
getAttributeNS() (*xml.dom.Element method*), 1408
GetBase() (*xml.parsers.expat.xmlparser method*), 1434
getbegyx() (*curses.window method*), 883
getbkgd() (*curses.window method*), 884
getblocking() (*socket.socket method*), 1194
getboolean() (*configparser.ConfigParser method*), 661
getbuffer() (*io.BytesIO method*), 768
getByteStream() (*xml.sax.xmlreader.InputSource method*), 1431
getcallargs() (*in module inspect*), 2103

getcanvas() (*in module turtle*), 1642
getch() (*curses.window method*), 884
getch() (*in module msvcrt*), 2252
getCharacterStream() (*xml.sax.xmlreader.InputSource method*), 1431
getche() (*in module msvcrt*), 2252
getChild() (*logging.Logger method*), 827
getChildren() (*logging.Logger method*), 827
getclasstree() (*in module inspect*), 2102
getclosurevars() (*in module inspect*), 2103
getcode() (*http.client.HTTPResponse method*), 1498
getcode() (*urllib.response.addinfourl method*), 1477
getColumnNumber() (*xml.sax.xmlreader.Locator method*), 1430
getcomments() (*in module inspect*), 2096
getcompname() (*wave.Wave_read method*), 1590
getcomptype() (*wave.Wave_read method*), 1590
getconfig() (*sqlite3.Connection method*), 568
getContentHandler() (*xml.sax.xmlreader.XMLReader method*), 1429
getcontext() (*in module decimal*), 371
getcoroutinelocals() (*in module inspect*), 2109
getcoroutinestate() (*in module inspect*), 2108
getctime() (*in module os.path*), 484
getcwd() (*in module os*), 716
getcwdb() (*in module os*), 716
getdecoder() (*in module codecs*), 192
getdefaultencoding() (*in module sys*), 2001
getdefaultlocale() (*in module locale*), 1606
getdefaulttimeout() (*in module socket*), 1190
getdlopenflags() (*in module sys*), 2001
getdoc() (*in module inspect*), 2096
getDOMImplementation() (*in module xml.dom*), 1402
getDTDHandler() (*xml.sax.xmlreader.XMLReader method*), 1429
getEffectiveLevel() (*logging.Logger method*), 826
getegid() (*in module os*), 693
getElementsByTagName() (*xml.dom.Document method*), 1407
getElementsByTagName() (*xml.dom.Element method*), 1407
getElementsByTagNameNS() (*xml.dom.Document method*), 1407
getElementsByTagNameNS() (*xml.dom.Element method*), 1407
getencoder() (*in module codecs*), 192
getencoding() (*in module locale*), 1607
getEncoding() (*xml.sax.xmlreader.InputSource method*), 1431
getEntityResolver() (*xml.sax.xmlreader.XMLReader method*), 1429
getenv() (*in module os*), 692
getenvb() (*in module os*), 693
getErrorHandler() (*xml.sax.xmlreader.XMLReader method*), 1429
geteuid() (*in module os*), 693
getEvent() (*xml.dom.pulldom.DOMEventStream method*), 1418
getEventCategory() (*logging.handlers.NTEventLogHandler method*), 867
getEventType() (*logging.handlers.NTEventLogHandler method*), 868
getException() (*xml.sax.SAXException method*), 1421
getFeature() (*xml.sax.xmlreader.XMLReader method*), 1429
getfile() (*in module inspect*), 2096
getFilesToDelete() (*logging.handlers.TimedRotatingFileHandler method*), 863
getfilesystemencodeerrors() (*in module sys*), 2002
getfilesystemencoding() (*in module sys*), 2001
getfloat() (*configparser.ConfigParser method*), 661
getfqdn() (*in module socket*), 1187
getframeinfo() (*in module inspect*), 2106
getframerate() (*wave.Wave_read method*), 1590
getfullargspec() (*in module inspect*), 2102
getgeneratorlocals() (*in module inspect*), 2109
getgeneratorstate() (*in module inspect*), 2108
getgid() (*in module os*), 693
getgrall() (*in module grp*), 2269
getgrgid() (*in module grp*), 2269
getgrnam() (*in module grp*), 2269
getgrouplist() (*in module os*), 693
getgroups() (*in module os*), 693
getHandlerByName() (*in module logging*), 841
getHandlerNames() (*in module logging*), 841
getheader() (*http.client.HTTPResponse method*), 1497
getheaders() (*http.client.HTTPResponse method*), 1497
gethostbyaddr() (*in module socket*), 698, 1188
gethostbyname() (*in module socket*), 1187
gethostbyname_ex() (*in module socket*), 1187
gethostname() (*in module socket*), 698, 1187
getincrementaldecoder() (*in module codecs*), 192
getincrementalencoder() (*in module codecs*), 192
getinfo() (*zipfile.ZipFile method*), 608
getinnerframes() (*in module inspect*), 2106

GetInputContext() (*xml.parsers.expat.xmlparser method*), 1434
getint() (*configparser.ConfigParser method*), 661
getitem() (*in module operator*), 449
getitimer() (*in module signal*), 1259
getkey() (*curses.window method*), 884
GetLastError() (*in module ctypes*), 947
getLength() (*xml.sax.xmlreader.Attributes method*), 1431
getLevelName() (*in module logging*), 840
getLevelNamesMapping() (*in module logging*), 840
getlimit() (*sqlite3.Connection method*), 567
getline() (*in module linecache*), 510
getLineNumber() (*xml.sax.xmlreader.Locator method*), 1430
getloadavg() (*in module os*), 757
getlocale() (*in module locale*), 1606
getLogger() (*in module logging*), 839
getLoggerClass() (*in module logging*), 839
getlogin() (*in module os*), 694
getLogRecordFactory() (*in module logging*), 839
getMandatoryRelease() (*__future__._Feature method*), 2086
getmark() (*wave.Wave_read method*), 1590
getmarkers() (*wave.Wave_read method*), 1590
getmaxyx() (*curses.window method*), 884
getmember() (*tarfile.TarFile method*), 623
getmembers() (*in module inspect*), 2092
getmembers() (*tarfile.TarFile method*), 623
getmembers_static() (*in module inspect*), 2092
getMessage() (*logging.LogRecord method*), 835
getMessage() (*xml.sax.SAXException method*), 1421
getMessageID() (*logging.handlers.NTEventLogHandler method*), 868
getmodule() (*in module inspect*), 2096
getmodulename() (*in module inspect*), 2093
getmouse() (*in module curses*), 875
getmro() (*in module inspect*), 2103
getmtime() (*in module os.path*), 484
getName() (*threading.Thread method*), 960
getNameByQName() (*xml.sax.xmlreader.AttributesNS method*), 1432
getnameinfo() (*asyncio.loop method*), 1126
getnameinfo() (*in module socket*), 1188
getnames() (*tarfile.TarFile method*), 623
getNames() (*xml.sax.xmlreader.Attributes method*), 1431
getnchannels() (*wave.Wave_read method*), 1590
getnframes() (*wave.Wave_read method*), 1590
getnode, 1527
getnode() (*in module uuid*), 1526
getopt
    module, 2287
getopt() (*in module getopt*), 2287
GetoptError, 2288
getOptionalRelease() (*__future__._Feature method*), 2086
getouterframes() (*in module inspect*), 2106
getoutput() (*in module subprocess*), 1051
getpagesize() (*in module resource*), 2281
getparams() (*wave.Wave_read method*), 1590
getparyx() (*curses.window method*), 884
getpass
    module, 872
getpass() (*in module getpass*), 872
GetPassWarning, 873
getpeercert() (*ssl.SSLSocket method*), 1219
getpeername() (*socket.socket method*), 1193
getpen() (*in module turtle*), 1635
getpgid() (*in module os*), 694
getpgrp() (*in module os*), 694
getpid() (*in module os*), 694
getpos() (*html.parser.HTMLParser method*), 1375
getppid() (*in module os*), 694
getpreferredencoding() (*in module locale*), 1606
getpriority() (*in module os*), 694
getprofile() (*in module sys*), 2003
getprofile() (*in module threading*), 957
getProperty() (*xml.sax.xmlreader.XMLReader method*), 1429
getprotobyname() (*in module socket*), 1188
getproxies() (*in module urllib.request*), 1459
getPublicId() (*xml.sax.xmlreader.InputSource method*), 1430
getPublicId() (*xml.sax.xmlreader.Locator method*), 1430
getpwall() (*in module pwd*), 2269
getpwnam() (*in module pwd*), 2269
getpwuid() (*in module pwd*), 2268
getQNameByName() (*xml.sax.xmlreader.AttributesNS method*), 1432
getQNames() (*xml.sax.xmlreader.AttributesNS method*), 1432
getquota() (*imaplib.IMAP4 method*), 1513
getquotaroot() (*imaplib.IMAP4 method*), 1513
getrandbits() (*in module random*), 394
getrandbits() (*random.Random method*), 397
getrandom() (*in module os*), 758
getreader() (*in module codecs*), 192
getrecursionlimit() (*in module sys*), 2002
getrefcount() (*in module sys*), 2002
GetReparseDeferralEnabled() (*xml.parsers.expat.xmlparser method*), 1435
getresgid() (*in module os*), 695
getresponse() (*http.client.HTTPConnection method*), 1495
getresuid() (*in module os*), 695

getrlimit() (*in module resource*), 2277
getroot() (*xml.etree.ElementTree.ElementTree method*), 1396
getrusage() (*in module resource*), 2280
getsampwidth() (*wave.Wave_read method*), 1590
getscreen() (*in module turtle*), 1636
getservbyname() (*in module socket*), 1188
getservbyport() (*in module socket*), 1188
GetSetDescriptorType (*in module types*), 310
getshapes() (*in module turtle*), 1642
getsid() (*in module os*), 697
getsignal() (*in module signal*), 1257
getsitepackages() (*in module site*), 2113
getsize() (*in module os.path*), 484
getsizeof() (*in module sys*), 2002
getsockname() (*socket.socket method*), 1193
getsockopt() (*socket.socket method*), 1193
getsource() (*in module inspect*), 2096
getsourcefile() (*in module inspect*), 2096
getsourcelines() (*in module inspect*), 2096
getstate() (*codecs.IncrementalDecoder method*), 198
getstate() (*codecs.IncrementalEncoder method*), 197
getstate() (*in module random*), 393
getstate() (*random.Random method*), 397
getstatusoutput() (*in module subprocess*), 1050
getstr() (*curses.window method*), 884
getSubject() (*logging.handlers.SMTPHandler method*), 868
getswitchinterval() (*in module sys*), 2003
getSystemId() (*xml.sax.xmlreader.InputSource method*), 1430
getSystemId() (*xml.sax.xmlreader.Locator method*), 1430
getsyx() (*in module curses*), 876
gettarinfo() (*tarfile.TarFile method*), 625
gettempdir() (*in module tempfile*), 503
gettempdirb() (*in module tempfile*), 504
gettempprefix() (*in module tempfile*), 504
gettempprefixb() (*in module tempfile*), 504
getTestCaseNames() (*unittest.TestLoader method*), 1826
gettext
    module, 1593
gettext() (*gettext.GNUTranslations method*), 1597
gettext() (*gettext.NullTranslations method*), 1596
gettext() (*in module gettext*), 1593
gettext() (*in module locale*), 1610
gettimeout() (*socket.socket method*), 1194
gettrace() (*in module sys*), 2003
gettrace() (*in module threading*), 957
getturtle() (*in module turtle*), 1635
getType() (*xml.sax.xmlreader.Attributes method*), 1431
getuid() (*in module os*), 695
getunicodeinternedsize() (*in module sys*), 2001
geturl() (*http.client.HTTPResponse method*), 1498
geturl() (*urllib.parse.urllib.parse.SplitResult method*), 1483
geturl() (*urllib.response.addinfourl method*), 1476
getuser() (*in module getpass*), 873
getuserbase() (*in module site*), 2114
getusersitepackages() (*in module site*), 2114
getvalue() (*io.BytesIO method*), 768
getvalue() (*io.StringIO method*), 773
getValue() (*xml.sax.xmlreader.Attributes method*), 1431
getValueByQName() (*xml.sax.xmlreader.AttributesNS method*), 1432
getwch() (*in module msvcrt*), 2252
getwche() (*in module msvcrt*), 2252
getweakrefcount() (*in module weakref*), 298
getweakrefs() (*in module weakref*), 298
getwelcome() (*ftplib.FTP method*), 1502
getwelcome() (*poplib.POP3 method*), 1508
getwin() (*in module curses*), 876
getwindowsversion() (*in module sys*), 2003
getwriter() (*in module codecs*), 192
getxattr() (*in module os*), 739
getyx() (*curses.window method*), 884
gid (*tarfile.TarInfo attribute*), 627
GIL, **2327**
glob
    module, 506, 508
glob() (*in module glob*), 506
glob() (*pathlib.Path method*), 473
Global (*class in ast*), 2197
global interpreter lock, **2327**
global_enum() (*in module enum*), 338
globals()
    built-in function, 15
globs (*doctest.DocTest attribute*), 1794
gmtime() (*in module time*), 777
gname (*tarfile.TarInfo attribute*), 627
GNOME, 1598
GNU_FORMAT (*in module tarfile*), 621
gnu_getopt() (*in module getopt*), 2288
GNUTranslations (*class in gettext*), 1597
GNUTYPE_LONGLINK (*in module tarfile*), 621
GNUTYPE_LONGNAME (*in module tarfile*), 621
GNUTYPE_SPARSE (*in module tarfile*), 621
go() (*tkinter.filedialog.FileDialog method*), 1681
got (*doctest.DocTestFailure attribute*), 1802
goto() (*in module turtle*), 1620
grantpt() (*in module os*), 702
Graphical User Interface, 1663
graphlib
    module, 339
GREATER (*in module token*), 2209
GREATEREQUAL (*in module token*), 2210

Greenwich Mean Time, 775
GRND_NONBLOCK (*in module os*), 759
GRND_RANDOM (*in module os*), 759
Group (*class in email.headerregistry*), 1301
group() (*pathlib.Path method*), 478
group() (*re.Match method*), 148
groupby() (*in module itertools*), 425
groupdict() (*re.Match method*), 149
groupindex (*re.Pattern attribute*), 147
groups (*email.headerregistry.AddressHeader attribute*), 1297
groups (*re.Pattern attribute*), 147
groups() (*re.Match method*), 149
grp
    module, 2269
GS (*in module curses.ascii*), 905
Gt (*class in ast*), 2175
gt() (*in module operator*), 446
GtE (*class in ast*), 2175
guess_all_extensions() (*in module mimetypes*), 1362
guess_all_extensions() (*mimetypes.MimeTypes method*), 1364
guess_extension() (*in module mimetypes*), 1362
guess_extension() (*mimetypes.MimeTypes method*), 1364
guess_file_type() (*in module mimetypes*), 1362
guess_file_type() (*mimetypes.MimeTypes method*), 1364
guess_scheme() (*in module wsgiref.util*), 1446
guess_type() (*in module mimetypes*), 1362
guess_type() (*mimetypes.MimeTypes method*), 1364
GUI, 1663
gzip
    module, 591
gzip command line option
    --best, 594
    -d, 594
    --decompress, 594
    --fast, 594
    file, 594
    -h, 594
    --help, 594
GzipFile (*class in gzip*), 591

H

-h
    ast command line option, 2203
    calendar command line option, 260
    dis command line option, 2226
    gzip command line option, 594
    json.tool command line option, 1341
    python--m-sqlite3-[-h]-[-v]-[filename]-[sql] command line option, 577
    random command line option, 401
    timeit command line option, 1956
    tokenize command line option, 2214
    uuid command line option, 1528
    zipapp command line option, 1986
halfdelay() (*in module curses*), 876
Handle (*class in asyncio*), 1132
handle() (*http.server.BaseHTTPRequestHandler method*), 1540
handle() (*logging.Handler method*), 831
handle() (*logging.handlers.QueueListener method*), 872
handle() (*logging.Logger method*), 829
handle() (*logging.NullHandler method*), 859
handle() (*socketserver.BaseRequestHandler method*), 1533
handle() (*wsgiref.simple_server.WSGIRequestHandler method*), 1450
handle_charref() (*html.parser.HTMLParser method*), 1375
handle_comment() (*html.parser.HTMLParser method*), 1376
handle_data() (*html.parser.HTMLParser method*), 1375
handle_decl() (*html.parser.HTMLParser method*), 1376
handle_defect() (*email.policy.Policy method*), 1289
handle_endtag() (*html.parser.HTMLParser method*), 1375
handle_entityref() (*html.parser.HTMLParser method*), 1375
handle_error() (*socketserver.BaseServer method*), 1533
handle_expect_100() (*http.server.BaseHTTPRequestHandler method*), 1540
handle_one_request() (*http.server.BaseHTTPRequestHandler method*), 1540
handle_pi() (*html.parser.HTMLParser method*), 1376
handle_request() (*socketserver.BaseServer method*), 1531
handle_request() (*xmlrpc.server.CGIXMLRPCRequestHandler method*), 1571
handle_startendtag() (*html.parser.HTMLParser method*), 1375
handle_starttag() (*html.parser.HTMLParser method*), 1375
handle_timeout() (*socketserver.BaseServer method*), 1533
handleError() (*logging.Handler method*), 831
handleError() (*logging.handlers.SocketHandler method*), 863

Handler (*class in logging*), 830
Handlers (*class in signal*), 1254
handlers (*logging.Logger attribute*), 826
hardlink_to() (*pathlib.Path method*), 477
--hardlink-dupes
    compileall command line option, 2222
harmonic_mean() (*in module statistics*), 405
HAS_ALPN (*in module ssl*), 1214
has_children() (*symtable.SymbolTable method*), 2205
has_colors() (*in module curses*), 876
has_default() (*typing.ParamSpec method*), 1748
has_default() (*typing.TypeVar method*), 1745
has_default() (*typing.TypeVarTuple method*), 1747
has_dualstack_ipv6() (*in module socket*), 1186
HAS_ECDH (*in module ssl*), 1215
has_extended_color_support() (*in module curses*), 876
has_extn() (*smtplib.SMTP method*), 1520
has_header() (*csv.Sniffer method*), 640
has_header() (*urllib.request.Request method*), 1463
has_ic() (*in module curses*), 876
has_il() (*in module curses*), 876
has_ipv6 (*in module socket*), 1183
has_key() (*in module curses*), 876
has_location (*importlib.machinery.ModuleSpec attribute*), 2143
HAS_NEVER_CHECK_COMMON_NAME (*in module ssl*), 1214
has_nonstandard_attr() (*http.cookiejar.Cookie method*), 1557
HAS_NPN (*in module ssl*), 1215
has_option() (*configparser.ConfigParser method*), 660
has_option() (*optparse.OptionParser method*), 2308
HAS_PSK (*in module ssl*), 1215
has_section() (*configparser.ConfigParser method*), 660
HAS_SNI (*in module ssl*), 1215
HAS_SSLv2 (*in module ssl*), 1215
HAS_SSLv3 (*in module ssl*), 1215
has_ticket (*ssl.SSLSession attribute*), 1239
HAS_TLSv1 (*in module ssl*), 1215
HAS_TLSv1_1 (*in module ssl*), 1215
HAS_TLSv1_2 (*in module ssl*), 1215
HAS_TLSv1_3 (*in module ssl*), 1215
hasarg (*in module dis*), 2247
hasattr()
    built-in function, 15
hasAttribute() (*xml.dom.Element method*), 1408
hasAttributeNS() (*xml.dom.Element method*), 1408
hasAttributes() (*xml.dom.Node method*), 1405
hasChildNodes() (*xml.dom.Node method*), 1405
hascompare (*in module dis*), 2248
hasconst (*in module dis*), 2247
hasexc (*in module dis*), 2248
hasFeature() (*xml.dom.DOMImplementation method*), 1403
hasfree (*in module dis*), 2248
hash
    built-in function, 47
hash()
    built-in function, 16
hash-based pyc, **2327**
hash_bits (*sys.hash_info attribute*), 2005
hash_info (*in module sys*), 2004
hash_randomization (*sys.flags attribute*), 1998
hashable, **2327**
Hashable (*class in collections.abc*), 283
Hashable (*class in typing*), 1773
hasHandlers() (*logging.Logger method*), 829
hash.block_size (*in module hashlib*), 673
hash.digest_size (*in module hashlib*), 673
hashlib
    module, 671
hasjabs (*in module dis*), 2248
hasjrel (*in module dis*), 2248
hasjump (*in module dis*), 2248
haslocal (*in module dis*), 2248
hasname (*in module dis*), 2248
HAVE_ARGUMENT (*opcode*), 2245
HAVE_CONTEXTVAR (*in module decimal*), 378
HAVE_DOCSTRINGS (*in module test.support*), 1905
HAVE_THREADS (*in module decimal*), 378
HCI_DATA_DIR (*in module socket*), 1183
HCI_FILTER (*in module socket*), 1183
HCI_TIME_STAMP (*in module socket*), 1183
Header (*class in email.header*), 1322
header_encode() (*email.charset.Charset method*), 1325
header_encode_lines() (*email.charset.Charset method*), 1325
header_encoding (*email.charset.Charset attribute*), 1324
header_factory (*email.policy.EmailPolicy attribute*), 1291
header_fetch_parse() (*email.policy.Compat32 method*), 1293
header_fetch_parse() (*email.policy.EmailPolicy method*), 1292
header_fetch_parse() (*email.policy.Policy method*), 1290
header_items() (*urllib.request.Request method*), 1464
header_max_count() (*email.policy.EmailPolicy method*), 1291
header_max_count() (*email.policy.Policy method*), 1290
header_offset (*zipfile.ZipInfo attribute*), 616

header_source_parse() (*email.policy.Compat32 method*), 1293
header_source_parse() (*email.policy.EmailPolicy method*), 1292
header_source_parse() (*email.policy.Policy method*), 1290
header_store_parse() (*email.policy.Compat32 method*), 1293
header_store_parse() (*email.policy.EmailPolicy method*), 1292
header_store_parse() (*email.policy.Policy method*), 1290
HeaderDefect, 1294
HeaderError, 620
HeaderParseError, 1294
HeaderParser (*class in email.parser*), 1283
HeaderRegistry (*class in email.headerregistry*), 1298
headers
    MIME, 1362
Headers (*class in wsgiref.headers*), 1448
headers (*http.client.HTTPResponse attribute*), 1498
headers (*http.server.BaseHTTPRequestHandler attribute*), 1539
headers (*urllib.error.HTTPError attribute*), 1486
headers (*urllib.response.addinfourl attribute*), 1476
headers (*xmlrpc.client.ProtocolError attribute*), 1563
heading() (*in module turtle*), 1625
heading() (*tkinter.ttk.Treeview method*), 1698
heapify() (*in module heapq*), 286
heapmin() (*in module msvcrt*), 2253
heappop() (*in module heapq*), 286
heappush() (*in module heapq*), 286
heappushpop() (*in module heapq*), 286
heapq
    module, 286
heapreplace() (*in module heapq*), 286
helo() (*smtplib.SMTP method*), 1520
help
    online, 1774
--help
    ast command line option, 2203
    calendar command line option, 260
    dis command line option, 2226
    gzip command line option, 594
    json.tool command line option, 1341
    python--m-sqlite3-[-h]-[-v]-[filename]-[sql] command line option, 577
    random command line option, 401
    timeit command line option, 1956
    tokenize command line option, 2214
    trace command line option, 1959
    uuid command line option, 1528
    zipapp command line option, 1986
help (*optparse.Option attribute*), 2304
help (*pdb command*), 1939
help()
    built-in function, 16
herror, 1178
hex (*uuid.UUID attribute*), 1526
hex()
    built-in function, 16
hex() (*bytearray method*), 65
hex() (*bytes method*), 63
hex() (*float method*), 41
hex() (*memoryview method*), 82
hexadecimal
    literals, 36
hexdigest() (*hashlib.hash method*), 674
hexdigest() (*hashlib.shake method*), 674
hexdigest() (*hmac.HMAC method*), 684
hexdigits (*in module string*), 121
hexlify() (*in module binascii*), 1370
hexversion (*in module sys*), 2005
hidden() (*curses.panel.Panel method*), 907
hide() (*curses.panel.Panel method*), 907
hide() (*tkinter.ttk.Notebook method*), 1693
hide_cookie2 (*http.cookiejar.CookiePolicy attribute*), 1554
hideturtle() (*in module turtle*), 1631
HierarchyRequestErr, 1410
HIGH_PRIORITY_CLASS (*in module subprocess*), 1044
HIGHEST_PROTOCOL (*in module pickle*), 525
hits (*bdb.Breakpoint attribute*), 1928
HKEY_CLASSES_ROOT (*in module winreg*), 2260
HKEY_CURRENT_CONFIG (*in module winreg*), 2260
HKEY_CURRENT_USER (*in module winreg*), 2260
HKEY_DYN_DATA (*in module winreg*), 2260
HKEY_LOCAL_MACHINE (*in module winreg*), 2260
HKEY_PERFORMANCE_DATA (*in module winreg*), 2260
HKEY_USERS (*in module winreg*), 2260
hline() (*curses.window method*), 884
hls_to_rgb() (*in module colorsys*), 1592
hmac
    module, 683
HOME, 483, 484, 1665
home() (*in module turtle*), 1622
home() (*pathlib.Path class method*), 469
HOMEDRIVE, 483
HOMEPATH, 483
hook_compressed() (*in module fileinput*), 491
hook_encoded() (*in module fileinput*), 491
host (*urllib.request.Request attribute*), 1462
hostmask (*ipaddress.IPv4Network attribute*), 1580
hostmask (*ipaddress.IPv6Network attribute*), 1583
hostname_checks_common_name (*ssl.SSLContext attribute*), 1229
hosts (*netrc.netrc attribute*), 666

hosts() (*ipaddress.IPv4Network method*), 1580
hosts() (*ipaddress.IPv6Network method*), 1583
hour (*datetime.datetime attribute*), 224
hour (*datetime.time attribute*), 232
HRESULT (*class in ctypes*), 952
hStdError (*subprocess.STARTUPINFO attribute*), 1043
hStdInput (*subprocess.STARTUPINFO attribute*), 1043
hStdOutput (*subprocess.STARTUPINFO attribute*), 1043
hsv_to_rgb() (*in module colorsys*), 1592
HT (*in module curses.ascii*), 903
ht() (*in module turtle*), 1631
HTML, 1373, 1476
html
    module, 1373
html5 (*in module html.entities*), 1378
HTMLCalendar (*class in calendar*), 256
HtmlDiff (*class in difflib*), 157
html.entities
    module, 1378
html.parser
    module, 1373
HTMLParser (*class in html.parser*), 1373
htonl() (*in module socket*), 1189
htons() (*in module socket*), 1189
HTTP
    http (*standard module*), 1488
    http.client (*standard module*), 1492
    protocol, 1476, 1488, 1492, 1538
http
    module, 1488
HTTP (*in module email.policy*), 1292
http_error_301() (*urllib.request.HTTPRedirectHandler method*), 1467
http_error_302() (*urllib.request.HTTPRedirectHandler method*), 1467
http_error_303() (*urllib.request.HTTPRedirectHandler method*), 1467
http_error_307() (*urllib.request.HTTPRedirectHandler method*), 1467
http_error_308() (*urllib.request.HTTPRedirectHandler method*), 1467
http_error_401() (*urllib.request.HTTPBasicAuthHandler method*), 1469
http_error_401() (*urllib.request.HTTPDigestAuthHandler method*), 1469
http_error_407() (*urllib.request.ProxyBasicAuthHandler method*), 1469
http_error_407() (*urllib.request.ProxyDigestAuthHandler method*), 1469
http_error_auth_reqed() (*urllib.request.AbstractBasicAuthHandler method*), 1468
http_error_auth_reqed() (*urllib.request.AbstractDigestAuthHandler method*), 1469
http_error_default() (*urllib.request.BaseHandler method*), 1466
http_open() (*urllib.request.HTTPHandler method*), 1469
HTTP_PORT (*in module http.client*), 1494
http_response() (*urllib.request.HTTPErrorProcessor method*), 1470
http_version (*wsgiref.handlers.BaseHandler attribute*), 1454
HTTPBasicAuthHandler (*class in urllib.request*), 1461
http.client
    module, 1492
HTTPConnection (*class in http.client*), 1492
http.cookiejar
    module, 1549
HTTPCookieProcessor (*class in urllib.request*), 1460
http.cookies
    module, 1545
httpd, 1538
HTTPDefaultErrorHandler (*class in urllib.request*), 1460
HTTPDigestAuthHandler (*class in urllib.request*), 1461
HTTPError, 1486
HTTPErrorProcessor (*class in urllib.request*), 1462
HTTPException, 1493
HTTPHandler (*class in logging.handlers*), 869
HTTPHandler (*class in urllib.request*), 1462
HTTPMessage (*class in http.client*), 1500
HTTPMethod (*class in http*), 1491
httponly (*http.cookies.Morsel attribute*), 1546
HTTPPasswordMgr (*class in urllib.request*), 1461
HTTPPasswordMgrWithDefaultRealm (*class in urllib.request*), 1461
HTTPPasswordMgrWithPriorAuth (*class in urllib.request*), 1461
HTTPRedirectHandler (*class in urllib.request*), 1460
HTTPResponse (*class in http.client*), 1493
https_open() (*urllib.request.HTTPSHandler method*), 1469
HTTPS_PORT (*in module http.client*), 1494
https_response() (*url-*

lib.request.HTTPErrorProcessor method), 1470
HTTPSConnection (*class in http.client*), 1492
http.server
    module, 1538
    security, 1544
HTTPServer (*class in http.server*), 1538
HTTPSHandler (*class in urllib.request*), 1462
HTTPStatus (*class in http*), 1488
HV_GUID_BROADCAST (*in module socket*), 1183
HV_GUID_CHILDREN (*in module socket*), 1183
HV_GUID_LOOPBACK (*in module socket*), 1183
HV_GUID_PARENT (*in module socket*), 1183
HV_GUID_WILDCARD (*in module socket*), 1183
HV_GUID_ZERO (*in module socket*), 1183
HV_PROTOCOL_RAW (*in module socket*), 1183
HVSOCKET_ADDRESS_FLAG_PASSTHRU (*in module socket*), 1183
HVSOCKET_CONNECT_TIMEOUT (*in module socket*), 1183
HVSOCKET_CONNECT_TIMEOUT_MAX (*in module socket*), 1183
HVSOCKET_CONNECTED_SUSPEND (*in module socket*), 1183
hypot() (*in module math*), 352

I

-i
    ast command line option, 2204
    compileall command line option, 2222
    random command line option, 401
I (*in module re*), 140
I/O control
    buffering, 23, 1194
    POSIX, 2270
    tty, 2270
    UNIX, 2274
iadd() (*in module operator*), 452
iand() (*in module operator*), 452
iconcat() (*in module operator*), 452
id (*ssl.SSLSession attribute*), 1239
id()
    built-in function, 17
id() (*unittest.TestCase method*), 1822
idcok() (*curses.window method*), 884
ident (*select.kevent attribute*), 1248
ident (*threading.Thread attribute*), 960
identchars (*cmd.Cmd attribute*), 1653
identify() (*tkinter.ttk.Notebook method*), 1693
identify() (*tkinter.ttk.Treeview method*), 1699
identify() (*tkinter.ttk.Widget method*), 1689
identify_column() (*tkinter.ttk.Treeview method*), 1699
identify_element() (*tkinter.ttk.Treeview method*), 1699
identify_region() (*tkinter.ttk.Treeview method*), 1699
identify_row() (*tkinter.ttk.Treeview method*), 1699
IDLE, 1706, **2328**
IDLE_PRIORITY_CLASS (*in module subprocess*), 1044
idlelib
    module, 1718
IDLESTARTUP, 1714
idlok() (*curses.window method*), 884
if
    statement, 35
If (*class in ast*), 2183
if_indextoname() (*in module socket*), 1191
if_nameindex() (*in module socket*), 1190
if_nametoindex() (*in module socket*), 1191
IfExp (*class in ast*), 2176
ifloordiv() (*in module operator*), 452
iglob() (*in module glob*), 507
ignorableWhitespace()
    (*xml.sax.handler.ContentHandler method*), 1424
ignore
    error handler's name, 194
ignore (*bdb.Breakpoint attribute*), 1928
IGNORE (*in module tkinter.messagebox*), 1683
ignore (*pdb command*), 1940
ignore_environment (*sys.flags attribute*), 1998
ignore_errors() (*in module codecs*), 195
IGNORE_EXCEPTION_DETAIL (*in module doctest*), 1786
ignore_patterns() (*in module shutil*), 513
ignore_warnings() (*in module test.support.warnings_helper*), 1919
IGNORECASE (*in module re*), 140
--ignore-dir
    trace command line option, 1960
--ignore-module
    trace command line option, 1960
IISCGIHandler (*class in wsgiref.handlers*), 1451
IllegalMonthError, 259
IllegalWeekdayError, 259
ilshift() (*in module operator*), 452
imag (*numbers.Complex attribute*), 343
imag (*sys.hash_info attribute*), 2005
imap() (*multiprocessing.pool.Pool method*), 1001
IMAP4
    protocol, 1510
IMAP4 (*class in imaplib*), 1510
IMAP4_SSL
    protocol, 1510
IMAP4_SSL (*class in imaplib*), 1511
IMAP4_stream

protocol, 1510
IMAP4_stream (*class in imaplib*), 1511
IMAP4.abort, 1510
IMAP4.error, 1510
IMAP4.readonly, 1511
imap_unordered() (*multiprocessing.pool.Pool method*), 1001
imaplib
module, 1510
imatmul() (*in module operator*), 453
immedok() (*curses.window method*), 884
immortal, **2328**
immutable, **2328**
sequence types, 47
imod() (*in module operator*), 452
impl_detail() (*in module test.support*), 1909
implementation (*in module sys*), 2005
import
statement, 31, 2111
Import (*class in ast*), 2182
import path, **2328**
import_fresh_module() (*in module test.support.import_helper*), 1918
IMPORT_FROM (*opcode*), 2240
import_module() (*in module importlib*), 2130
import_module() (*in module test.support.import_helper*), 1919
IMPORT_NAME (*opcode*), 2240
importer, **2328**
ImportError, 109
ImportFrom (*class in ast*), 2182
importing, **2328**
importlib
module, 2128
importlib.abc
module, 2131
importlib.machinery
module, 2138
importlib.metadata
module, 2155
importlib.resources
module, 2150
importlib.resources.abc
module, 2153
importlib.util
module, 2144
ImportWarning, 117
ImproperConnectionState, 1493
imul() (*in module operator*), 452
in
operator, 36, 45
In (*class in ast*), 2175
in_dll() (*ctypes._CData method*), 949
in_table_a1() (*in module stringprep*), 175
in_table_b1() (*in module stringprep*), 175
in_table_c3() (*in module stringprep*), 176
in_table_c4() (*in module stringprep*), 176
in_table_c5() (*in module stringprep*), 176
in_table_c6() (*in module stringprep*), 176
in_table_c7() (*in module stringprep*), 176
in_table_c8() (*in module stringprep*), 176
in_table_c9() (*in module stringprep*), 176
in_table_c11() (*in module stringprep*), 176
in_table_c11_c12() (*in module stringprep*), 176
in_table_c12() (*in module stringprep*), 176
in_table_c21() (*in module stringprep*), 176
in_table_c21_c22() (*in module stringprep*), 176
in_table_c22() (*in module stringprep*), 176
in_table_d1() (*in module stringprep*), 176
in_table_d2() (*in module stringprep*), 176
in_transaction (*sqlite3.Connection attribute*), 570
inch() (*curses.window method*), 884
include() (*in module xml.etree.ElementInclude*), 1393
--include-attributes
ast command line option, 2203
inclusive (*tracemalloc.DomainFilter attribute*), 1968
inclusive (*tracemalloc.Filter attribute*), 1969
Incomplete, 1371
IncompleteRead, 1493
IncompleteReadError, 1111
increment_lineno() (*in module ast*), 2201
IncrementalDecoder (*class in codecs*), 198
incrementaldecoder (*codecs.CodecInfo attribute*), 191
IncrementalEncoder (*class in codecs*), 197
incrementalencoder (*codecs.CodecInfo attribute*), 191
IncrementalNewlineDecoder (*class in io*), 773
IncrementalParser (*class in xml.sax.xmlreader*), 1428
--indent
ast command line option, 2204
json.tool command line option, 1341
indent (*doctest.Example attribute*), 1795
INDENT (*in module token*), 2208
indent (*reprlib.Repr attribute*), 321
indent() (*in module textwrap*), 170
indent() (*in module xml.etree.ElementTree*), 1390
IndentationError, 113
--indentlevel
pickletools command line option, 2249
index (*inspect.FrameInfo attribute*), 2105
index (*inspect.Traceback attribute*), 2105
index() (*array.array method*), 295
index() (*bytearray method*), 67
index() (*bytes method*), 67
index() (*collections.deque method*), 268
index() (*in module operator*), 447

index() (*multiprocessing.shared_memory.ShareableList method*), 1021
index() (*sequence method*), 45
index() (*str method*), 54
index() (*tkinter.ttk.Notebook method*), 1693
index() (*tkinter.ttk.Treeview method*), 1699
IndexError, 110
indexOf() (*in module operator*), 449
IndexSizeErr, 1410
INDIRECT (*inspect.BufferFlags attribute*), 2110
inet_aton() (*in module socket*), 1189
inet_ntoa() (*in module socket*), 1189
inet_ntop() (*in module socket*), 1189
inet_pton() (*in module socket*), 1189
Inexact (*class in decimal*), 379
inf (*in module cmath*), 358
inf (*in module math*), 354
inf (*sys.hash_info attribute*), 2004
infile
    json.tool command line option, 1340
infile (*shlex.shlex attribute*), 1659
Infinity, 14
infj (*in module cmath*), 358
--info
    zipapp command line option, 1986
INFO (*in module logging*), 830
INFO (*in module tkinter.messagebox*), 1684
info() (*dis.Bytecode method*), 2227
info() (*gettext.NullTranslations method*), 1596
info() (*http.client.HTTPResponse method*), 1498
info() (*in module logging*), 839
info() (*logging.Logger method*), 828
info() (*urllib.response.addinfourl method*), 1476
infolist() (*zipfile.ZipFile method*), 608
.ini
    file, 645
ini file, 645
init() (*in module mimetypes*), 1363
init_color() (*in module curses*), 876
init_pair() (*in module curses*), 876
inited (*in module mimetypes*), 1363
initgroups() (*in module os*), 695
initial_indent (*textwrap.TextWrapper attribute*), 172
initscr() (*in module curses*), 876
inode() (*os.DirEntry method*), 724
input()
    built-in function, 17
input() (*in module fileinput*), 489
input_charset (*email.charset.Charset attribute*), 1324
input_codec (*email.charset.Charset attribute*), 1324
InputSource (*class in xml.sax.xmlreader*), 1428
InputStream (*class in wsgiref.types*), 1455
insch() (*curses.window method*), 885
insdelln() (*curses.window method*), 885
insert() (*array.array method*), 296
insert() (*collections.deque method*), 269
insert() (*sequence method*), 47
insert() (*tkinter.ttk.Notebook method*), 1693
insert() (*tkinter.ttk.Treeview method*), 1699
insert() (*xml.etree.ElementTree.Element method*), 1395
insert_text() (*in module readline*), 178
insertBefore() (*xml.dom.Node method*), 1405
insertln() (*curses.window method*), 885
insnstr() (*curses.window method*), 885
insort() (*in module bisect*), 291
insort_left() (*in module bisect*), 291
insort_right() (*in module bisect*), 291
inspect
    module, 2090
inspect (*sys.flags attribute*), 1998
inspect command line option
    --details, 2111
InspectLoader (*class in importlib.abc*), 2134
insstr() (*curses.window method*), 885
install() (*gettext.NullTranslations method*), 1596
install() (*in module gettext*), 1595
install_opener() (*in module urllib.request*), 1458
install_scripts() (*venv.EnvBuilder method*), 1981
installHandler() (*in module unittest*), 1835
instate() (*tkinter.ttk.Widget method*), 1689
instr() (*curses.window method*), 885
instream (*shlex.shlex attribute*), 1659
Instruction (*class in dis*), 2230
INSTRUCTION (*monitoring event*), 2019
Instruction.arg (*in module dis*), 2230
Instruction.argrepr (*in module dis*), 2230
Instruction.argval (*in module dis*), 2230
Instruction.baseopcode (*in module dis*), 2230
Instruction.baseopname (*in module dis*), 2230
Instruction.cache_offset (*in module dis*), 2230
Instruction.end_offset (*in module dis*), 2230
Instruction.is_jump_target (*in module dis*), 2230
Instruction.jump_target (*in module dis*), 2230
Instruction.line_number (*in module dis*), 2230
Instruction.offset (*in module dis*), 2230
Instruction.oparg (*in module dis*), 2230
Instruction.opcode (*in module dis*), 2230
Instruction.opname (*in module dis*), 2230
Instruction.positions (*in module dis*), 2230
Instruction.start_offset (*in module dis*), 2230
Instruction.starts_line (*in module dis*), 2230
int
    built-in function, 37
int (*built-in class*), 17
int (*uuid.UUID attribute*), 1526

Int2AP() (*in module imaplib*), 1511
int_info (*in module sys*), 2006
int_max_str_digits (*sys.flags attribute*), 1998
integer
    literals, 36
    object, 36
    types, operations on, 38
--integer
    random command line option, 401
Integral (*class in numbers*), 344
Integrated Development Environment, 1706
IntegrityError, 576
IntEnum (*class in enum*), 330
interact (*pdb command*), 1942
interact() (*code.InteractiveConsole method*), 2117
interact() (*in module code*), 2115
interactive, **2328**
Interactive (*class in ast*), 2170
interactive (*sys.flags attribute*), 1998
InteractiveConsole (*class in code*), 2115
InteractiveInterpreter (*class in code*), 2115
InterfaceError, 575
intern() (*in module sys*), 2006
internal_attr (*zipfile.ZipInfo attribute*), 616
Internaldate2tuple() (*in module imaplib*), 1511
InternalError, 576
internalSubset (*xml.dom.DocumentType attribute*), 1406
Internet, 1443
INTERNET_TIMEOUT (*in module test.support*), 1904
interpolation
    bytearray (%), 77
    bytes (%), 77
interpolation, string (%), 60
InterpolationDepthError, 663
InterpolationError, 663
InterpolationMissingOptionError, 663
InterpolationSyntaxError, 663
interpreted, **2328**
interpreter prompts, 2010
interpreter shutdown, **2328**
interpreter_requires_environment() (*in module test.support.script_helper*), 1913
interrupt() (*sqlite3.Connection method*), 564
interrupt_main() (*in module _thread*), 1062
InterruptedError, 116
intersection() (*frozenset method*), 87
intersection_update() (*frozenset method*), 88
IntFlag (*class in enum*), 333
intro (*cmd.Cmd attribute*), 1653
InuseAttributeErr, 1410
inv() (*in module operator*), 447
inv_cdf() (*statistics.NormalDist method*), 415
InvalidAccessErr, 1410
invalidate_caches() (*importlib.abc.MetaPathFinder method*), 2132
invalidate_caches() (*importlib.abc.PathEntryFinder method*), 2132
invalidate_caches() (*importlib.machinery.FileFinder method*), 2140
invalidate_caches() (*importlib.machinery.PathFinder class method*), 2139
invalidate_caches() (*in module importlib*), 2130
invalidate_caches() (*zipimport.zipimporter method*), 2121
--invalidation-mode
    compileall command line option, 2222
InvalidCharacterErr, 1410
InvalidModificationErr, 1410
InvalidOperation (*class in decimal*), 379
InvalidStateErr, 1410
InvalidStateError, 1030, 1111
InvalidTZPathWarning, 254
InvalidURL, 1493
Invert (*class in ast*), 2174
invert() (*in module operator*), 447
io
    module, 760
IO (*class in typing*), 1758
IO_REPARSE_TAG_APPEXECLINK (*in module stat*), 497
IO_REPARSE_TAG_MOUNT_POINT (*in module stat*), 497
IO_REPARSE_TAG_SYMLINK (*in module stat*), 497
IOBase (*class in io*), 764
ioctl() (*in module fcntl*), 2275
ioctl() (*socket.socket method*), 1194
IOCTL_VM_SOCKETS_GET_LOCAL_CID (*in module socket*), 1182
IOError, 115
ior() (*in module operator*), 453
ios_ver() (*in module platform*), 911
io.StringIO
    object, 51
ip (*ipaddress.IPv4Interface attribute*), 1585
ip (*ipaddress.IPv6Interface attribute*), 1585
ip_address() (*in module ipaddress*), 1573
ip_interface() (*in module ipaddress*), 1573
ip_network() (*in module ipaddress*), 1573
ipaddress
    module, 1572
ipow() (*in module operator*), 453
ipv4_mapped (*ipaddress.IPv6Address attribute*), 1577
IPv4Address (*class in ipaddress*), 1573
IPv4Interface (*class in ipaddress*), 1585
IPv4Network (*class in ipaddress*), 1579

IPV6_ENABLED (*in module test.support.socket_helper*), 1913
ipv6_mapped (*ipaddress.IPv4Address attribute*), 1575
IPv6Address (*class in ipaddress*), 1576
IPv6Interface (*class in ipaddress*), 1585
IPv6Network (*class in ipaddress*), 1582
irshift() (*in module operator*), 453
is
    operator, 36
Is (*class in ast*), 2175
is not
    operator, 36
is_() (*in module operator*), 447
is_absolute() (*pathlib.PurePath method*), 463
is_active() (*asyncio.AbstractChildWatcher method*), 1158
is_active() (*graphlib.TopologicalSorter method*), 340
is_alive() (*multiprocessing.Process method*), 978
is_alive() (*threading.Thread method*), 961
is_android (*in module test.support*), 1903
is_annotated() (*symtable.Symbol method*), 2207
is_assigned() (*symtable.Symbol method*), 2207
is_async (*pyclbr.Function attribute*), 2218
is_attachment() (*email.message.EmailMessage method*), 1277
is_authenticated() (*urllib.request.HTTPPasswordMgrWithPriorAuth method*), 1468
is_block_device() (*pathlib.Path method*), 471
is_blocked() (*http.cookiejar.DefaultCookiePolicy method*), 1555
is_canonical() (*decimal.Context method*), 375
is_canonical() (*decimal.Decimal method*), 366
is_char_device() (*pathlib.Path method*), 472
IS_CHARACTER_JUNK() (*in module difflib*), 161
is_check_supported() (*in module lzma*), 604
is_closed() (*asyncio.loop method*), 1114
is_closing() (*asyncio.BaseTransport method*), 1143
is_closing() (*asyncio.StreamWriter method*), 1093
is_dataclass() (*in module dataclasses*), 2049
is_declared_global() (*symtable.Symbol method*), 2207
is_dir() (*importlib.abc.Traversable method*), 2137
is_dir() (*importlib.resources.abc.Traversable method*), 2154
is_dir() (*os.DirEntry method*), 724
is_dir() (*pathlib.Path method*), 471
is_dir() (*zipfile.Path method*), 612
is_dir() (*zipfile.ZipInfo method*), 615
is_enabled() (*in module faulthandler*), 1934
is_expired() (*http.cookiejar.Cookie method*), 1557
is_fifo() (*pathlib.Path method*), 471
is_file() (*importlib.abc.Traversable method*), 2137
is_file() (*importlib.resources.abc.Traversable method*), 2154
is_file() (*os.DirEntry method*), 724
is_file() (*pathlib.Path method*), 471
is_file() (*zipfile.Path method*), 612
is_finalized() (*in module gc*), 2088
is_finalizing() (*in module sys*), 2006
is_finite() (*decimal.Context method*), 375
is_finite() (*decimal.Decimal method*), 367
is_free() (*symtable.Symbol method*), 2207
is_global (*ipaddress.IPv4Address attribute*), 1575
is_global (*ipaddress.IPv6Address attribute*), 1576
is_global() (*symtable.Symbol method*), 2207
is_hop_by_hop() (*in module wsgiref.util*), 1448
is_imported() (*symtable.Symbol method*), 2207
is_infinite() (*decimal.Context method*), 375
is_infinite() (*decimal.Decimal method*), 367
is_integer() (*float method*), 41
is_integer() (*fractions.Fraction method*), 390
is_integer() (*int method*), 41
is_junction() (*os.DirEntry method*), 725
is_junction() (*pathlib.Path method*), 471
is_jython (*in module test.support*), 1903
IS_LINE_JUNK() (*in module difflib*), 160
is_linetouched() (*curses.window method*), 885
is_link_local (*ipaddress.IPv4Address attribute*), 1575
is_link_local (*ipaddress.IPv4Network attribute*), 1580
is_link_local (*ipaddress.IPv6Address attribute*), 1577
is_link_local (*ipaddress.IPv6Network attribute*), 1583
is_local() (*symtable.Symbol method*), 2207
is_loopback (*ipaddress.IPv4Address attribute*), 1575
is_loopback (*ipaddress.IPv4Network attribute*), 1580
is_loopback (*ipaddress.IPv6Address attribute*), 1577
is_loopback (*ipaddress.IPv6Network attribute*), 1583
is_mount() (*pathlib.Path method*), 471
is_multicast (*ipaddress.IPv4Address attribute*), 1574
is_multicast (*ipaddress.IPv4Network attribute*), 1579
is_multicast (*ipaddress.IPv6Address attribute*), 1576
is_multicast (*ipaddress.IPv6Network attribute*), 1582
is_multipart() (*email.message.EmailMessage method*), 1273
is_multipart() (*email.message.Message method*), 1312
is_namespace() (*symtable.Symbol method*), 2207
is_nan() (*decimal.Context method*), 375
is_nan() (*decimal.Decimal method*), 367
is_nested() (*symtable.SymbolTable method*), 2205
is_nonlocal() (*symtable.Symbol method*), 2207
is_normal() (*decimal.Context method*), 375
is_normal() (*decimal.Decimal method*), 367

is_normalized() (*in module unicodedata*), 174
is_not() (*in module operator*), 447
is_not_allowed() (*http.cookiejar.DefaultCookiePolicy method*), 1555
IS_OP (*opcode*), 2239
is_optimized() (*symtable.SymbolTable method*), 2205
is_package() (*importlib.abc.InspectLoader method*), 2134
is_package() (*importlib.abc.SourceLoader method*), 2136
is_package() (*importlib.machinery.ExtensionFileLoader method*), 2142
is_package() (*importlib.machinery.SourceFileLoader method*), 2140
is_package() (*importlib.machinery.SourcelessFileLoader method*), 2141
is_package() (*zipimport.zipimporter method*), 2120
is_parameter() (*symtable.Symbol method*), 2207
is_private (*ipaddress.IPv4Address attribute*), 1574
is_private (*ipaddress.IPv4Network attribute*), 1579
is_private (*ipaddress.IPv6Address attribute*), 1576
is_private (*ipaddress.IPv6Network attribute*), 1582
is_protocol() (*in module typing*), 1765
is_python_build() (*in module sysconfig*), 2029
is_qnan() (*decimal.Context method*), 375
is_qnan() (*decimal.Decimal method*), 367
is_reading() (*asyncio.ReadTransport method*), 1144
is_referenced() (*symtable.Symbol method*), 2207
is_relative_to() (*pathlib.PurePath method*), 463
is_reserved (*ipaddress.IPv4Address attribute*), 1575
is_reserved (*ipaddress.IPv4Network attribute*), 1579
is_reserved (*ipaddress.IPv6Address attribute*), 1577
is_reserved (*ipaddress.IPv6Network attribute*), 1582
is_reserved() (*pathlib.PurePath method*), 463
is_resource() (*importlib.abc.ResourceReader method*), 2137
is_resource() (*importlib.resources.abc.ResourceReader method*), 2154
is_resource() (*in module importlib.resources*), 2152
is_resource_enabled() (*in module test.support*), 1906
is_running() (*asyncio.loop method*), 1114
is_safe (*uuid.UUID attribute*), 1526
is_serving() (*asyncio.Server method*), 1134
is_set() (*asyncio.Event method*), 1099
is_set() (*threading.Event method*), 967
is_signed() (*decimal.Context method*), 375
is_signed() (*decimal.Decimal method*), 367
is_site_local (*ipaddress.IPv6Address attribute*), 1577
is_site_local (*ipaddress.IPv6Network attribute*), 1583
is_skipped_line() (*bdb.Bdb method*), 1930
is_snan() (*decimal.Context method*), 375
is_snan() (*decimal.Decimal method*), 367
is_socket() (*pathlib.Path method*), 471
is_stack_trampoline_active() (*in module sys*), 2013
is_subnormal() (*decimal.Context method*), 375
is_subnormal() (*decimal.Decimal method*), 367
is_symlink() (*os.DirEntry method*), 724
is_symlink() (*pathlib.Path method*), 471
is_symlink() (*zipfile.Path method*), 612
is_tarfile() (*in module tarfile*), 620
is_term_resized() (*in module curses*), 877
is_tracing() (*in module tracemalloc*), 1967
is_tracked() (*in module gc*), 2088
is_typeddict() (*in module typing*), 1765
is_unspecified (*ipaddress.IPv4Address attribute*), 1575
is_unspecified (*ipaddress.IPv4Network attribute*), 1579
is_unspecified (*ipaddress.IPv6Address attribute*), 1576
is_unspecified (*ipaddress.IPv6Network attribute*), 1582
is_valid() (*string.Template method*), 131
is_wintouched() (*curses.window method*), 885
is_zero() (*decimal.Context method*), 375
is_zero() (*decimal.Decimal method*), 367
is_zipfile() (*in module zipfile*), 606
isabs() (*in module os.path*), 484
isabstract() (*in module inspect*), 2095
IsADirectoryError, 116
isalnum() (*bytearray method*), 73
isalnum() (*bytes method*), 73
isalnum() (*in module curses.ascii*), 905
isalnum() (*str method*), 54
isalpha() (*bytearray method*), 73
isalpha() (*bytes method*), 73
isalpha() (*in module curses.ascii*), 905
isalpha() (*str method*), 54
isascii() (*bytearray method*), 73
isascii() (*bytes method*), 73
isascii() (*in module curses.ascii*), 905
isascii() (*str method*), 54
isasyncgen() (*in module inspect*), 2094
isasyncgenfunction() (*in module inspect*), 2094
isatty() (*in module os*), 702
isatty() (*io.IOBase method*), 764
isawaitable() (*in module inspect*), 2094
isblank() (*in module curses.ascii*), 905
isblk() (*tarfile.TarInfo method*), 628
isbuiltin() (*in module inspect*), 2094
ischr() (*tarfile.TarInfo method*), 628
isclass() (*in module inspect*), 2093

isclose() (*in module cmath*), 357
isclose() (*in module math*), 348
iscntrl() (*in module curses.ascii*), 905
iscode() (*in module inspect*), 2094
iscoroutine() (*in module asyncio*), 1084
iscoroutine() (*in module inspect*), 2094
iscoroutinefunction() (*in module inspect*), 2093
isctrl() (*in module curses.ascii*), 906
isDaemon() (*threading.Thread method*), 961
isdatadescriptor() (*in module inspect*), 2095
isdecimal() (*str method*), 54
isdev() (*tarfile.TarInfo method*), 628
isdevdrive() (*in module os.path*), 485
isdigit() (*bytearray method*), 73
isdigit() (*bytes method*), 73
isdigit() (*in module curses.ascii*), 905
isdigit() (*str method*), 54
isdir() (*in module os.path*), 484
isdir() (*tarfile.TarInfo method*), 628
isdisjoint() (*frozenset method*), 87
isdown() (*in module turtle*), 1627
iselement() (*in module xml.etree.ElementTree*), 1390
isenabled() (*in module gc*), 2086
isEnabledFor() (*logging.Logger method*), 826
isendwin() (*in module curses*), 877
ISEOF() (*in module token*), 2208
isfifo() (*tarfile.TarInfo method*), 628
isfile() (*in module os.path*), 484
isfile() (*tarfile.TarInfo method*), 628
isfinite() (*in module cmath*), 357
isfinite() (*in module math*), 348
isfirstline() (*in module fileinput*), 490
isframe() (*in module inspect*), 2094
isfunction() (*in module inspect*), 2093
isfuture() (*in module asyncio*), 1138
isgenerator() (*in module inspect*), 2093
isgeneratorfunction() (*in module inspect*), 2093
isgetsetdescriptor() (*in module inspect*), 2095
isgraph() (*in module curses.ascii*), 906
isidentifier() (*str method*), 54
isinf() (*in module cmath*), 357
isinf() (*in module math*), 348
isinstance()
    built-in function, 18
isjunction() (*in module os.path*), 484
iskeyword() (*in module keyword*), 2212
isleap() (*in module calendar*), 257
islice() (*in module itertools*), 426
islink() (*in module os.path*), 484
islnk() (*tarfile.TarInfo method*), 628
islower() (*bytearray method*), 73
islower() (*bytes method*), 73
islower() (*in module curses.ascii*), 906
islower() (*str method*), 54
ismemberdescriptor() (*in module inspect*), 2095
ismeta() (*in module curses.ascii*), 906
ismethod() (*in module inspect*), 2093
ismethoddescriptor() (*in module inspect*), 2095
ismethodwrapper() (*in module inspect*), 2095
ismodule() (*in module inspect*), 2093
ismount() (*in module os.path*), 485
isnan() (*in module cmath*), 357
isnan() (*in module math*), 348
ISNONTERMINAL() (*in module token*), 2208
IsNot (*class in ast*), 2175
isnumeric() (*str method*), 55
isocalendar() (*datetime.date method*), 218
isocalendar() (*datetime.datetime method*), 228
isoformat() (*datetime.date method*), 218
isoformat() (*datetime.datetime method*), 228
isoformat() (*datetime.time method*), 234
isolated (*sys.flags attribute*), 1998
IsolatedAsyncioTestCase (*class in unittest*), 1823
isolation_level (*sqlite3.Connection attribute*), 570
isoweekday() (*datetime.date method*), 218
isoweekday() (*datetime.datetime method*), 228
isprint() (*in module curses.ascii*), 906
isprintable() (*str method*), 55
ispunct() (*in module curses.ascii*), 906
isqrt() (*in module math*), 348
isreadable() (*in module pprint*), 315
isreadable() (*pprint.PrettyPrinter method*), 316
isrecursive() (*in module pprint*), 315
isrecursive() (*pprint.PrettyPrinter method*), 316
isreg() (*tarfile.TarInfo method*), 628
isreserved() (*in module os.path*), 485
isReservedKey() (*http.cookies.Morsel method*), 1547
isroutine() (*in module inspect*), 2095
isSameNode() (*xml.dom.Node method*), 1405
issoftkeyword() (*in module keyword*), 2212
isspace() (*bytearray method*), 74
isspace() (*bytes method*), 74
isspace() (*in module curses.ascii*), 906
isspace() (*str method*), 55
isstdin() (*in module fileinput*), 490
issubclass()
    built-in function, 18
issubset() (*frozenset method*), 87
issuperset() (*frozenset method*), 87
issym() (*tarfile.TarInfo method*), 628
ISTERMINAL() (*in module token*), 2208
istitle() (*bytearray method*), 74
istitle() (*bytes method*), 74
istitle() (*str method*), 55
istraceback() (*in module inspect*), 2094
isub() (*in module operator*), 453
isupper() (*bytearray method*), 74
isupper() (*bytes method*), 74

isupper() (*in module curses.ascii*), 906
isupper() (*str method*), 55
isvisible() (*in module turtle*), 1631
isxdigit() (*in module curses.ascii*), 906
ITALIC (*in module tkinter.font*), 1677
item() (*tkinter.ttk.Treeview method*), 1699
item() (*xml.dom.NamedNodeMap method*), 1409
item() (*xml.dom.NodeList method*), 1406
itemgetter() (*in module operator*), 450
items() (*configparser.ConfigParser method*), 661
items() (*contextvars.Context method*), 1060
items() (*dict method*), 91
items() (*email.message.EmailMessage method*), 1275
items() (*email.message.Message method*), 1314
items() (*mailbox.Mailbox method*), 1343
items() (*types.MappingProxyType method*), 310
items() (*xml.etree.ElementTree.Element method*), 1394
itemsize (*array.array attribute*), 294
itemsize (*memoryview attribute*), 85
ItemsView (*class in collections.abc*), 284
ItemsView (*class in typing*), 1769
iter()
    built-in function, 18
iter() (*xml.etree.ElementTree.Element method*), 1395
iter() (*xml.etree.ElementTree.ElementTree method*), 1396
iter_attachments() (*email.message.EmailMessage method*), 1278
iter_child_nodes() (*in module ast*), 2201
iter_fields() (*in module ast*), 2201
iter_importers() (*in module pkgutil*), 2122
iter_modules() (*in module pkgutil*), 2123
iter_parts() (*email.message.EmailMessage method*), 1278
iter_unpack() (*in module struct*), 184
iter_unpack() (*struct.Struct method*), 190
iterable, **2328**
Iterable (*class in collections.abc*), 283
Iterable (*class in typing*), 1772
iterator, **2329**
Iterator (*class in collections.abc*), 283
Iterator (*class in typing*), 1772
iterator protocol, 44
iterdecode() (*in module codecs*), 193
iterdir() (*importlib.abc.Traversable method*), 2137
iterdir() (*importlib.resources.abc.Traversable method*), 2154
iterdir() (*pathlib.Path method*), 473
iterdir() (*zipfile.Path method*), 612
iterdump() (*sqlite3.Connection method*), 566
iterencode() (*in module codecs*), 193
iterencode() (*json.JSONEncoder method*), 1337
iterfind() (*xml.etree.ElementTree.Element method*), 1395
iterfind() (*xml.etree.ElementTree.ElementTree method*), 1396
iteritems() (*mailbox.Mailbox method*), 1343
iterkeys() (*mailbox.Mailbox method*), 1342
itermonthdates() (*calendar.Calendar method*), 254
itermonthdays() (*calendar.Calendar method*), 254
itermonthdays2() (*calendar.Calendar method*), 254
itermonthdays3() (*calendar.Calendar method*), 254
itermonthdays4() (*calendar.Calendar method*), 255
iterparse() (*in module xml.etree.ElementTree*), 1390
itertext() (*xml.etree.ElementTree.Element method*), 1395
itertools
    module, 419
itervalues() (*mailbox.Mailbox method*), 1342
iterweekdays() (*calendar.Calendar method*), 254
ITIMER_PROF (*in module signal*), 1257
ITIMER_REAL (*in module signal*), 1257
ITIMER_VIRTUAL (*in module signal*), 1257
ItimerError, 1257
itruediv() (*in module operator*), 453
ixor() (*in module operator*), 453

J

-j
    compileall command line option, 2222
JANUARY (*in module calendar*), 259
java_ver() (*in module platform*), 910
join() (*asyncio.Queue method*), 1109
join() (*bytearray method*), 67
join() (*bytes method*), 67
join() (*in module os.path*), 485
join() (*in module shlex*), 1656
join() (*multiprocessing.JoinableQueue method*), 983
join() (*multiprocessing.pool.Pool method*), 1001
join() (*multiprocessing.Process method*), 978
join() (*queue.Queue method*), 1056
join() (*str method*), 55
join() (*threading.Thread method*), 960
join_thread() (*in module test.support.threading_helper*), 1915
join_thread() (*multiprocessing.Queue method*), 982
JoinableQueue (*class in multiprocessing*), 983
JoinedStr (*class in ast*), 2171
joinpath() (*importlib.abc.Traversable method*), 2137
joinpath() (*importlib.resources.abc.Traversable method*), 2154
joinpath() (*pathlib.PurePath method*), 463
joinpath() (*zipfile.Path method*), 613
js_output() (*http.cookies.BaseCookie method*), 1546
js_output() (*http.cookies.Morsel method*), 1547
json
    module, 1331
JSONDecodeError, 1338

JSONDecoder (*class in json*), 1335
JSONEncoder (*class in json*), 1336
--json-lines
    json.tool command line option, 1340
json.tool
    module, 1340
json.tool command line option
    --compact, 1341
    -h, 1341
    --help, 1341
    --indent, 1341
    infile, 1340
    --json-lines, 1340
    --no-ensure-ascii, 1340
    --no-indent, 1341
    outfile, 1340
    --sort-keys, 1340
    --tab, 1341
JULY (*in module calendar*), 259
JUMP (*monitoring event*), 2019
JUMP (*opcode*), 2247
jump (*pdb command*), 1941
JUMP_BACKWARD (*opcode*), 2240
JUMP_BACKWARD_NO_INTERRUPT (*opcode*), 2240
JUMP_FORWARD (*opcode*), 2240
JUMP_NO_INTERRUPT (*opcode*), 2247
JUNE (*in module calendar*), 259

K

-k
    unittest command line option, 1806
kbhit() (*in module msvcrt*), 2252
kde() (*in module statistics*), 406
kde_random() (*in module statistics*), 407
KEEP (*enum.FlagBoundary attribute*), 336
kevent() (*in module select*), 1243
key (*http.cookies.Morsel attribute*), 1547
key (*zoneinfo.ZoneInfo attribute*), 251
key function, **2329**
KEY_A1 (*in module curses*), 892
KEY_A3 (*in module curses*), 892
KEY_ALL_ACCESS (*in module winreg*), 2260
KEY_B2 (*in module curses*), 892
KEY_BACKSPACE (*in module curses*), 890
KEY_BEG (*in module curses*), 892
KEY_BREAK (*in module curses*), 890
KEY_BTAB (*in module curses*), 892
KEY_C1 (*in module curses*), 892
KEY_C3 (*in module curses*), 892
KEY_CANCEL (*in module curses*), 892
KEY_CATAB (*in module curses*), 891
KEY_CLEAR (*in module curses*), 891
KEY_CLOSE (*in module curses*), 892
KEY_COMMAND (*in module curses*), 892
KEY_COPY (*in module curses*), 893
KEY_CREATE (*in module curses*), 893
KEY_CREATE_LINK (*in module winreg*), 2261
KEY_CREATE_SUB_KEY (*in module winreg*), 2261
KEY_CTAB (*in module curses*), 891
KEY_DC (*in module curses*), 891
KEY_DL (*in module curses*), 891
KEY_DOWN (*in module curses*), 890
KEY_EIC (*in module curses*), 891
KEY_END (*in module curses*), 893
KEY_ENTER (*in module curses*), 892
KEY_ENUMERATE_SUB_KEYS (*in module winreg*), 2261
KEY_EOL (*in module curses*), 891
KEY_EOS (*in module curses*), 891
KEY_EXECUTE (*in module winreg*), 2261
KEY_EXIT (*in module curses*), 893
KEY_F0 (*in module curses*), 890
KEY_FIND (*in module curses*), 893
KEY_Fn (*in module curses*), 890
KEY_HELP (*in module curses*), 893
KEY_HOME (*in module curses*), 890
KEY_IC (*in module curses*), 891
KEY_IL (*in module curses*), 891
KEY_LEFT (*in module curses*), 890
KEY_LL (*in module curses*), 892
KEY_MARK (*in module curses*), 893
KEY_MAX (*in module curses*), 896
KEY_MESSAGE (*in module curses*), 893
KEY_MIN (*in module curses*), 890
KEY_MOUSE (*in module curses*), 896
KEY_MOVE (*in module curses*), 893
KEY_NEXT (*in module curses*), 893
KEY_NOTIFY (*in module winreg*), 2261
KEY_NPAGE (*in module curses*), 891
KEY_OPEN (*in module curses*), 893
KEY_OPTIONS (*in module curses*), 893
KEY_PPAGE (*in module curses*), 891
KEY_PREVIOUS (*in module curses*), 893
KEY_PRINT (*in module curses*), 892
KEY_QUERY_VALUE (*in module winreg*), 2261
KEY_READ (*in module winreg*), 2261
KEY_REDO (*in module curses*), 893
KEY_REFERENCE (*in module curses*), 893
KEY_REFRESH (*in module curses*), 894
KEY_REPLACE (*in module curses*), 894
KEY_RESET (*in module curses*), 892
KEY_RESIZE (*in module curses*), 896
KEY_RESTART (*in module curses*), 894
KEY_RESUME (*in module curses*), 894
KEY_RIGHT (*in module curses*), 890
KEY_SAVE (*in module curses*), 894
KEY_SBEG (*in module curses*), 894
KEY_SCANCEL (*in module curses*), 894
KEY_SCOMMAND (*in module curses*), 894

KEY_SCOPY (*in module curses*), 894
KEY_SCREATE (*in module curses*), 894
KEY_SDC (*in module curses*), 894
KEY_SDL (*in module curses*), 894
KEY_SELECT (*in module curses*), 894
KEY_SEND (*in module curses*), 894
KEY_SEOL (*in module curses*), 894
KEY_SET_VALUE (*in module winreg*), 2261
KEY_SEXIT (*in module curses*), 895
KEY_SF (*in module curses*), 891
KEY_SFIND (*in module curses*), 895
KEY_SHELP (*in module curses*), 895
KEY_SHOME (*in module curses*), 895
KEY_SIC (*in module curses*), 895
KEY_SLEFT (*in module curses*), 895
KEY_SMESSAGE (*in module curses*), 895
KEY_SMOVE (*in module curses*), 895
KEY_SNEXT (*in module curses*), 895
KEY_SOPTIONS (*in module curses*), 895
KEY_SPREVIOUS (*in module curses*), 895
KEY_SPRINT (*in module curses*), 895
KEY_SR (*in module curses*), 891
KEY_SREDO (*in module curses*), 895
KEY_SREPLACE (*in module curses*), 895
KEY_SRESET (*in module curses*), 892
KEY_SRIGHT (*in module curses*), 895
KEY_SRSUME (*in module curses*), 896
KEY_SSAVE (*in module curses*), 896
KEY_SSUSPEND (*in module curses*), 896
KEY_STAB (*in module curses*), 891
KEY_SUNDO (*in module curses*), 896
KEY_SUSPEND (*in module curses*), 896
KEY_UNDO (*in module curses*), 896
KEY_UP (*in module curses*), 890
KEY_WOW64_32KEY (*in module winreg*), 2261
KEY_WOW64_64KEY (*in module winreg*), 2261
KEY_WRITE (*in module winreg*), 2260
KeyboardInterrupt, 110
KeyError, 110
keylog_filename (*ssl.SSLContext attribute*), 1228
keyname() (*in module curses*), 877
keypad() (*curses.window method*), 885
keyrefs() (*weakref.WeakKeyDictionary method*), 299
keys() (*contextvars.Context method*), 1060
keys() (*dict method*), 91
keys() (*email.message.EmailMessage method*), 1274
keys() (*email.message.Message method*), 1314
keys() (*mailbox.Mailbox method*), 1342
keys() (*sqlite3.Row method*), 574
keys() (*types.MappingProxyType method*), 310
keys() (*xml.etree.ElementTree.Element method*), 1394
KeysView (*class in collections.abc*), 284
KeysView (*class in typing*), 1769
keyword
    module, 2212
keyword (*class in ast*), 2176
keyword argument, **2329**
keywords (*functools.partial attribute*), 446
kill() (*asyncio.subprocess.Process method*), 1106
kill() (*asyncio.SubprocessTransport method*), 1146
kill() (*in module os*), 744
kill() (*multiprocessing.Process method*), 979
kill() (*subprocess.Popen method*), 1041
kill_python() (*in module test.support.script_helper*), 1914
killchar() (*in module curses*), 877
killpg() (*in module os*), 745
kind (*inspect.Parameter attribute*), 2099
knownfiles (*in module mimetypes*), 1363
kqueue() (*in module select*), 1243
KqueueSelector (*class in selectors*), 1252
KW_ONLY (*in module dataclasses*), 2049
kwargs (*inspect.BoundArguments attribute*), 2101
kwargs (*typing.ParamSpec attribute*), 1748
kwlist (*in module keyword*), 2212

L

-L
    calendar command line option, 261
-l
    calendar command line option, 261
    compileall command line option, 2221
    pickletools command line option, 2249
    tarfile command line option, 632
    trace command line option, 1959
    zipfile command line option, 617
L (*in module re*), 141
lambda, **2329**
Lambda (*class in ast*), 2195
LambdaType (*in module types*), 307
LANG, 1593, 1594, 1602, 1606
LANGUAGE, 1593, 1594
language
    C, 36
large files, 2267
LARGEST (*in module test.support*), 1905
LargeZipFile, 606
last_accepted (*multiprocessing.connection.Listener attribute*), 1004
last_exc (*in module sys*), 2007
last_traceback (*in module sys*), 2007
last_type (*in module sys*), 2007
last_value (*in module sys*), 2007
lastChild (*xml.dom.Node attribute*), 1404
lastcmd (*cmd.Cmd attribute*), 1653
lastgroup (*re.Match attribute*), 150
lastindex (*re.Match attribute*), 150
lastResort (*in module logging*), 843

lastrowid (*sqlite3.Cursor attribute*), 573
layout() (*tkinter.ttk.Style method*), 1702
lazycache() (*in module linecache*), 510
LazyLoader (*class in importlib.util*), 2146
LBRACE (*in module token*), 2209
LBYL, **2329**
LC_ALL, 1593, 1594
LC_ALL (*in module locale*), 1608
LC_COLLATE (*in module locale*), 1608
LC_CTYPE (*in module locale*), 1608
LC_MESSAGES, 1593, 1594
LC_MESSAGES (*in module locale*), 1608
LC_MONETARY (*in module locale*), 1608
LC_NUMERIC (*in module locale*), 1608
LC_TIME (*in module locale*), 1608
lchflags() (*in module os*), 716
lchmod() (*in module os*), 716
lchmod() (*pathlib.Path method*), 478
lchown() (*in module os*), 717
lcm() (*in module math*), 348
ldexp() (*in module math*), 348
le() (*in module operator*), 446
leapdays() (*in module calendar*), 257
leaveok() (*curses.window method*), 885
left (*filecmp.dircmp attribute*), 499
left() (*in module turtle*), 1620
left_list (*filecmp.dircmp attribute*), 499
left_only (*filecmp.dircmp attribute*), 499
LEFTSHIFT (*in module token*), 2210
LEFTSHIFTEQUAL (*in module token*), 2210
LEGACY_TRANSACTION_CONTROL (*in module sqlite3*), 557
len
    built-in function, 45, 89
len()
    built-in function, 18
length (*xml.dom.NamedNodeMap attribute*), 1409
length (*xml.dom.NodeList attribute*), 1406
length_hint() (*in module operator*), 449
LESS (*in module token*), 2209
LESSEQUAL (*in module token*), 2209
level (*logging.Logger attribute*), 825
LexicalHandler (*class in xml.sax.handler*), 1421
lexists() (*in module os.path*), 483
LF (*in module curses.ascii*), 904
lgamma() (*in module math*), 353
libc_ver() (*in module platform*), 912
LIBRARIES_ASSEMBLY_NAME_PREFIX (*in module msvcrt*), 2254
library (*in module dbm.ndbm*), 551
library (*ssl.SSLError attribute*), 1207
LibraryLoader (*class in ctypes*), 941
license (*built-in variable*), 34
LifoQueue (*class in asyncio*), 1110
LifoQueue (*class in queue*), 1054
light-weight processes, 1062
limit_denominator() (*fractions.Fraction method*), 390
LimitOverrunError, 1112
line (*bdb.Breakpoint attribute*), 1928
LINE (*monitoring event*), 2019
line (*traceback.FrameSummary attribute*), 2082
line_buffering (*io.TextIOWrapper attribute*), 772
line_num (*csv.csvreader attribute*), 643
linear_regression() (*in module statistics*), 413
line-buffered I/O, 23
linecache
    module, 510
lineno (*ast.AST attribute*), 2169
lineno (*doctest.DocTest attribute*), 1795
lineno (*doctest.Example attribute*), 1795
lineno (*inspect.FrameInfo attribute*), 2105
lineno (*inspect.Traceback attribute*), 2105
lineno (*json.JSONDecodeError attribute*), 1338
lineno (*netrc.NetrcParseError attribute*), 666
lineno (*pyclbr.Class attribute*), 2218
lineno (*pyclbr.Function attribute*), 2218
lineno (*re.PatternError attribute*), 146
lineno (*shlex.shlex attribute*), 1660
lineno (*SyntaxError attribute*), 113
lineno (*traceback.FrameSummary attribute*), 2082
lineno (*traceback.TracebackException attribute*), 2080
lineno (*tracemalloc.Filter attribute*), 1969
lineno (*tracemalloc.Frame attribute*), 1969
lineno (*xml.parsers.expat.ExpatError attribute*), 1438
lineno() (*in module fileinput*), 490
LINES, 875, 881
--lines
    calendar command line option, 261
LINES (*in module curses*), 888
lines (*os.terminal_size attribute*), 712
linesep (*email.policy.Policy attribute*), 1289
linesep (*in module os*), 758
lineterminator (*csv.Dialect attribute*), 642
LineTooLong, 1494
link() (*in module os*), 717
linkname (*tarfile.TarInfo attribute*), 627
LinkOutsideDestinationError, 620
list, **2329**
    object, 47, 48
    type, operations on, 47
--list
    tarfile command line option, 632
    zipfile command line option, 617
list (*built-in class*), 48
List (*class in ast*), 2172
List (*class in typing*), 1767
list (*pdb command*), 1941

list comprehension, **2329**
list() (*imaplib.IMAP4 method*), 1513
list() (*multiprocessing.managers.SyncManager method*), 995
list() (*poplib.POP3 method*), 1509
list() (*tarfile.TarFile method*), 623
LIST_APPEND (*opcode*), 2234
list_dialects() (*in module csv*), 638
LIST_EXTEND (*opcode*), 2238
list_folders() (*mailbox.Maildir method*), 1345
list_folders() (*mailbox.MH method*), 1349
ListComp (*class in ast*), 2177
listdir() (*in module os*), 717
listdrives() (*in module os*), 718
listen() (*in module logging.config*), 846
listen() (*in module turtle*), 1639
listen() (*socket.socket method*), 1194
Listener (*class in multiprocessing.connection*), 1003
listener (*logging.handlers.QueueHandler attribute*), 871
--listfuncs
    trace command line option, 1959
listMethods() (*xmlrpc.client.ServerProxy.system method*), 1560
listmounts() (*in module os*), 718
listvolumes() (*in module os*), 718
listxattr() (*in module os*), 739
Literal (*in module typing*), 1736
literal_eval() (*in module ast*), 2200
literals
    binary, 36
    complex number, 36
    floating-point, 36
    hexadecimal, 36
    integer, 36
    numeric, 36
    octal, 36
LiteralString (*in module typing*), 1731
LittleEndianStructure (*class in ctypes*), 952
LittleEndianUnion (*class in ctypes*), 952
ljust() (*bytearray method*), 69
ljust() (*bytes method*), 69
ljust() (*str method*), 55
LK_LOCK (*in module msvcrt*), 2251
LK_NBLCK (*in module msvcrt*), 2251
LK_NBRLCK (*in module msvcrt*), 2251
LK_RLCK (*in module msvcrt*), 2251
LK_UNLCK (*in module msvcrt*), 2251
ll (*pdb command*), 1941
LMTP (*class in smtplib*), 1518
ln() (*decimal.Context method*), 375
ln() (*decimal.Decimal method*), 367
LNKTYPE (*in module tarfile*), 621
Load (*class in ast*), 2173
load() (*http.cookiejar.FileCookieJar method*), 1552
load() (*http.cookies.BaseCookie method*), 1546
load() (*in module json*), 1334
load() (*in module marshal*), 545
load() (*in module pickle*), 526
load() (*in module plistlib*), 667
load() (*in module tomllib*), 664
load() (*pickle.Unpickler method*), 528
load() (*tracemalloc.Snapshot class method*), 1970
LOAD_ASSERTION_ERROR (*opcode*), 2236
LOAD_ATTR (*opcode*), 2239
LOAD_BUILD_CLASS (*opcode*), 2236
load_cert_chain() (*ssl.SSLContext method*), 1223
LOAD_CLOSURE (*opcode*), 2247
LOAD_CONST (*opcode*), 2237
load_default_certs() (*ssl.SSLContext method*), 1223
LOAD_DEREF (*opcode*), 2241
load_dh_params() (*ssl.SSLContext method*), 1226
load_extension() (*sqlite3.Connection method*), 566
LOAD_FAST (*opcode*), 2241
LOAD_FAST_AND_CLEAR (*opcode*), 2241
LOAD_FAST_CHECK (*opcode*), 2241
LOAD_FROM_DICT_OR_DEREF (*opcode*), 2241
LOAD_FROM_DICT_OR_GLOBALS (*opcode*), 2237
LOAD_GLOBAL (*opcode*), 2241
LOAD_LOCALS (*opcode*), 2237
LOAD_METHOD (*opcode*), 2247
load_module() (*importlib.abc.FileLoader method*), 2135
load_module() (*importlib.abc.InspectLoader method*), 2134
load_module() (*importlib.abc.Loader method*), 2133
load_module() (*importlib.abc.SourceLoader method*), 2136
load_module() (*importlib.machinery.SourceFileLoader method*), 2141
load_module() (*importlib.machinery.SourcelessFileLoader method*), 2141
load_module() (*zipimport.zipimporter method*), 2120
LOAD_NAME (*opcode*), 2237
load_package_tests() (*in module test.support*), 1910
LOAD_SUPER_ATTR (*opcode*), 2239
load_verify_locations() (*ssl.SSLContext method*), 1223
loader, **2330**
Loader (*class in importlib.abc*), 2132
loader (*importlib.machinery.ModuleSpec attribute*), 2142
loader_state (*importlib.machinery.ModuleSpec attribute*), 2143
LoadError, 1549

LoadFileDialog (*class in tkinter.filedialog*), 1681
LoadKey() (*in module winreg*), 2257
LoadLibrary() (*ctypes.LibraryLoader method*), 942
loads() (*in module json*), 1335
loads() (*in module marshal*), 546
loads() (*in module pickle*), 526
loads() (*in module plistlib*), 667
loads() (*in module tomllib*), 664
loads() (*in module xmlrpc.client*), 1565
loadTestsFromModule() (*unittest.TestLoader method*), 1826
loadTestsFromName() (*unittest.TestLoader method*), 1826
loadTestsFromNames() (*unittest.TestLoader method*), 1826
loadTestsFromTestCase() (*unittest.TestLoader method*), 1825
local (*class in threading*), 958
LOCAL_CREDS (*in module socket*), 1183
LOCAL_CREDS_PERSISTENT (*in module socket*), 1183
localcontext() (*in module decimal*), 371
locale
    module, 1601
--locale
    calendar command line option, 261
LOCALE (*in module re*), 141
locale encoding, **2330**
localeconv() (*in module locale*), 1602
LocaleHTMLCalendar (*class in calendar*), 257
LocaleTextCalendar (*class in calendar*), 257
localize() (*in module locale*), 1608
localName (*xml.dom.Attr attribute*), 1409
localName (*xml.dom.Node attribute*), 1404
--locals
    unittest command line option, 1806
locals()
    built-in function, 19
localtime() (*in module email.utils*), 1327
localtime() (*in module time*), 777
Locator (*class in xml.sax.xmlreader*), 1428
Lock (*class in asyncio*), 1097
Lock (*class in multiprocessing*), 988
Lock (*class in threading*), 961
lock (*sys.thread_info attribute*), 2015
lock() (*mailbox.Babyl method*), 1350
lock() (*mailbox.Mailbox method*), 1344
lock() (*mailbox.Maildir method*), 1347
lock() (*mailbox.mbox method*), 1348
lock() (*mailbox.MH method*), 1349
lock() (*mailbox.MMDF method*), 1351
Lock() (*multiprocessing.managers.SyncManager method*), 995
LOCK_EX (*in module fcntl*), 2276
LOCK_NB (*in module fcntl*), 2276
LOCK_SH (*in module fcntl*), 2276
LOCK_UN (*in module fcntl*), 2276
locked() (*_thread.lock method*), 1063
locked() (*asyncio.Condition method*), 1099
locked() (*asyncio.Lock method*), 1098
locked() (*asyncio.Semaphore method*), 1101
locked() (*threading.Lock method*), 962
lockf() (*in module fcntl*), 2276
lockf() (*in module os*), 703
locking() (*in module msvcrt*), 2251
LockType (*in module _thread*), 1062
log() (*in module cmath*), 356
log() (*in module logging*), 840
log() (*in module math*), 351
log() (*logging.Logger method*), 828
log1p() (*in module math*), 351
log2() (*in module math*), 351
log10() (*decimal.Context method*), 375
log10() (*decimal.Decimal method*), 367
log10() (*in module cmath*), 356
log10() (*in module math*), 351
LOG_ALERT (*in module syslog*), 2282
LOG_AUTH (*in module syslog*), 2283
LOG_AUTHPRIV (*in module syslog*), 2283
LOG_CONS (*in module syslog*), 2283
LOG_CRIT (*in module syslog*), 2282
LOG_CRON (*in module syslog*), 2283
LOG_DAEMON (*in module syslog*), 2283
log_date_time_string() (*http.server.BaseHTTPRequestHandler method*), 1541
LOG_DEBUG (*in module syslog*), 2282
LOG_EMERG (*in module syslog*), 2282
LOG_ERR (*in module syslog*), 2282
log_error() (*http.server.BaseHTTPRequestHandler method*), 1541
log_exception() (*wsgiref.handlers.BaseHandler method*), 1453
LOG_FTP (*in module syslog*), 2283
LOG_INFO (*in module syslog*), 2282
LOG_INSTALL (*in module syslog*), 2283
LOG_KERN (*in module syslog*), 2283
LOG_LAUNCHD (*in module syslog*), 2283
LOG_LOCAL0 (*in module syslog*), 2283
LOG_LOCAL1 (*in module syslog*), 2283
LOG_LOCAL2 (*in module syslog*), 2283
LOG_LOCAL3 (*in module syslog*), 2283
LOG_LOCAL4 (*in module syslog*), 2283
LOG_LOCAL5 (*in module syslog*), 2283
LOG_LOCAL6 (*in module syslog*), 2283
LOG_LOCAL7 (*in module syslog*), 2283
LOG_LPR (*in module syslog*), 2283
LOG_MAIL (*in module syslog*), 2283

log_message() (*http.server.BaseHTTPRequestHandler method*), 1541
LOG_NDELAY (*in module syslog*), 2283
LOG_NETINFO (*in module syslog*), 2283
LOG_NEWS (*in module syslog*), 2283
LOG_NOTICE (*in module syslog*), 2282
LOG_NOWAIT (*in module syslog*), 2283
LOG_ODELAY (*in module syslog*), 2283
LOG_PERROR (*in module syslog*), 2283
LOG_PID (*in module syslog*), 2283
LOG_RAS (*in module syslog*), 2283
LOG_REMOTEAUTH (*in module syslog*), 2283
log_request() (*http.server.BaseHTTPRequestHandler method*), 1541
LOG_SYSLOG (*in module syslog*), 2283
log_to_stderr() (*in module multiprocessing*), 1006
LOG_USER (*in module syslog*), 2283
LOG_UUCP (*in module syslog*), 2283
LOG_WARNING (*in module syslog*), 2282
logb() (*decimal.Context method*), 375
logb() (*decimal.Decimal method*), 367
Logger (*class in logging*), 825
LoggerAdapter (*class in logging*), 838
logging
    Errors, 823
    module, 823
logging.config
    module, 844
logging.handlers
    module, 857
logical_and() (*decimal.Context method*), 375
logical_and() (*decimal.Decimal method*), 367
logical_invert() (*decimal.Context method*), 375
logical_invert() (*decimal.Decimal method*), 367
logical_or() (*decimal.Context method*), 375
logical_or() (*decimal.Decimal method*), 367
logical_xor() (*decimal.Context method*), 375
logical_xor() (*decimal.Decimal method*), 367
login() (*ftplib.FTP method*), 1502
login() (*imaplib.IMAP4 method*), 1513
login() (*smtplib.SMTP method*), 1521
login_cram_md5() (*imaplib.IMAP4 method*), 1513
login_tty() (*in module os*), 703
LOGNAME, 694, 873
lognormvariate() (*in module random*), 396
logout() (*imaplib.IMAP4 method*), 1513
LogRecord (*class in logging*), 834
LONG_TIMEOUT (*in module test.support*), 1904
longMessage (*unittest.TestCase attribute*), 1821
longname() (*in module curses*), 877
lookup() (*in module codecs*), 191
lookup() (*in module unicodedata*), 173
lookup() (*symtable.SymbolTable method*), 2205
lookup() (*tkinter.ttk.Style method*), 1702
lookup_error() (*in module codecs*), 195
LookupError, 109
loop
    over mutable sequence, 45
loop_factory (*unittest.IsolatedAsyncioTestCase attribute*), 1823
LOOPBACK_TIMEOUT (*in module test.support*), 1903
lower() (*bytearray method*), 74
lower() (*bytes method*), 74
lower() (*str method*), 55
LPAR (*in module token*), 2208
lpAttributeList (*subprocess.STARTUPINFO attribute*), 1043
lru_cache() (*in module functools*), 437
lseek() (*in module os*), 703
LShift (*class in ast*), 2174
lshift() (*in module operator*), 447
LSQB (*in module token*), 2208
lstat() (*in module os*), 718
lstat() (*pathlib.Path method*), 470
lstrip() (*bytearray method*), 69
lstrip() (*bytes method*), 69
lstrip() (*str method*), 55
lsub() (*imaplib.IMAP4 method*), 1513
Lt (*class in ast*), 2175
lt() (*in module operator*), 446
lt() (*in module turtle*), 1620
LtE (*class in ast*), 2175
LWPCookieJar (*class in http.cookiejar*), 1553
lzma
    module, 600
LZMACompressor (*class in lzma*), 601
LZMADecompressor (*class in lzma*), 602
LZMAError, 600
LZMAFile (*class in lzma*), 600

M

-m
    ast command line option, 2203
    calendar command line option, 261
    pickletools command line option, 2249
    trace command line option, 1960
    zipapp command line option, 1986
M (*in module re*), 141
mac_ver() (*in module platform*), 911
machine() (*in module platform*), 909
macros (*netrc.netrc attribute*), 666
MADV_AUTOSYNC (*in module mmap*), 1267
MADV_CORE (*in module mmap*), 1267
MADV_DODUMP (*in module mmap*), 1267
MADV_DOFORK (*in module mmap*), 1267
MADV_DONTDUMP (*in module mmap*), 1267
MADV_DONTFORK (*in module mmap*), 1267
MADV_DONTNEED (*in module mmap*), 1267

MADV_FREE (*in module mmap*), 1267
MADV_FREE_REUSABLE (*in module mmap*), 1267
MADV_FREE_REUSE (*in module mmap*), 1267
MADV_HUGEPAGE (*in module mmap*), 1267
MADV_HWPOISON (*in module mmap*), 1267
MADV_MERGEABLE (*in module mmap*), 1267
MADV_NOCORE (*in module mmap*), 1267
MADV_NOHUGEPAGE (*in module mmap*), 1267
MADV_NORMAL (*in module mmap*), 1267
MADV_NOSYNC (*in module mmap*), 1267
MADV_PROTECT (*in module mmap*), 1267
MADV_RANDOM (*in module mmap*), 1267
MADV_REMOVE (*in module mmap*), 1267
MADV_SEQUENTIAL (*in module mmap*), 1267
MADV_SOFT_OFFLINE (*in module mmap*), 1267
MADV_UNMERGEABLE (*in module mmap*), 1267
MADV_WILLNEED (*in module mmap*), 1267
madvise() (*mmap.mmap method*), 1266
magic
    method, 2330
magic method, **2330**
MAGIC_NUMBER (*in module importlib.util*), 2144
MagicMock (*class in unittest.mock*), 1867
mailbox
    module, 1341
Mailbox (*class in mailbox*), 1341
Maildir (*class in mailbox*), 1345
MaildirMessage (*class in mailbox*), 1352
--main
    zipapp command line option, 1986
main() (*in module site*), 2113
main() (*in module unittest*), 1831
main_thread() (*in module threading*), 957
mainloop() (*in module turtle*), 1641
maintype (*email.headerregistry.ContentTypeHeader attribute*), 1298
major (*email.headerregistry.MIMEVersionHeader attribute*), 1298
major() (*in module os*), 720
make_alternative() (*email.message.EmailMessage method*), 1279
make_archive() (*in module shutil*), 517
make_bad_fd() (*in module test.support.os_helper*), 1917
MAKE_CELL (*opcode*), 2241
make_cookies() (*http.cookiejar.CookieJar method*), 1551
make_dataclass() (*in module dataclasses*), 2048
make_file() (*difflib.HtmlDiff method*), 157
MAKE_FUNCTION (*opcode*), 2243
make_header() (*in module email.header*), 1323
make_legacy_pyc() (*in module test.support.import_helper*), 1919
make_mixed() (*email.message.EmailMessage method*), 1279
make_msgid() (*in module email.utils*), 1327
make_parser() (*in module xml.sax*), 1419
make_pkg() (*in module test.support.script_helper*), 1914
make_related() (*email.message.EmailMessage method*), 1279
make_script() (*in module test.support.script_helper*), 1914
make_server() (*in module wsgiref.simple_server*), 1449
make_table() (*difflib.HtmlDiff method*), 158
make_zip_pkg() (*in module test.support.script_helper*), 1914
make_zip_script() (*in module test.support.script_helper*), 1914
makedev() (*in module os*), 720
makedirs() (*in module os*), 719
makeelement() (*xml.etree.ElementTree.Element method*), 1395
makefile() (*socket.socket method*), 1194
makeLogRecord() (*in module logging*), 841
makePickle() (*logging.handlers.SocketHandler method*), 863
makeRecord() (*logging.Logger method*), 829
makeSocket() (*logging.handlers.DatagramHandler method*), 864
makeSocket() (*logging.handlers.SocketHandler method*), 863
maketrans() (*bytearray static method*), 68
maketrans() (*bytes static method*), 68
maketrans() (*str static method*), 56
manager (*logging.LoggerAdapter attribute*), 838
mangle_from_ (*email.policy.Compat32 attribute*), 1293
mangle_from_ (*email.policy.Policy attribute*), 1289
mant_dig (*sys.float_info attribute*), 2000
map()
    built-in function, 19
map() (*concurrent.futures.Executor method*), 1023
map() (*multiprocessing.pool.Pool method*), 1001
map() (*tkinter.ttk.Style method*), 1702
MAP_32BIT (*in module mmap*), 1268
MAP_ADD (*opcode*), 2234
MAP_ALIGNED_SUPER (*in module mmap*), 1268
MAP_ANON (*in module mmap*), 1268
MAP_ANONYMOUS (*in module mmap*), 1268
map_async() (*multiprocessing.pool.Pool method*), 1001
MAP_CONCEAL (*in module mmap*), 1268
MAP_DENYWRITE (*in module mmap*), 1268
MAP_EXECUTABLE (*in module mmap*), 1268
MAP_HASSEMAPHORE (*in module mmap*), 1268
MAP_JIT (*in module mmap*), 1268
MAP_NOCACHE (*in module mmap*), 1268
MAP_NOEXTEND (*in module mmap*), 1268

MAP_NORESERVE (*in module mmap*), 1268
MAP_POPULATE (*in module mmap*), 1268
MAP_PRIVATE (*in module mmap*), 1268
MAP_RESILIENT_CODESIGN (*in module mmap*), 1268
MAP_RESILIENT_MEDIA (*in module mmap*), 1268
MAP_SHARED (*in module mmap*), 1268
MAP_STACK (*in module mmap*), 1268
map_table_b2() (*in module stringprep*), 175
map_table_b3() (*in module stringprep*), 175
map_to_type() (*email.headerregistry.HeaderRegistry method*), 1300
MAP_TPRO (*in module mmap*), 1268
MAP_TRANSLATED_ALLOW_EXECUTE (*in module mmap*), 1268
MAP_UNIX03 (*in module mmap*), 1268
mapLogRecord() (*logging.handlers.HTTPHandler method*), 869
mapping, **2330**
    object, 89
    types, operations on, 89
Mapping (*class in collections.abc*), 284
Mapping (*class in typing*), 1769
MappingProxyType (*class in types*), 310
MappingView (*class in collections.abc*), 284
MappingView (*class in typing*), 1770
mapPriority() (*logging.handlers.SysLogHandler method*), 867
maps (*collections.ChainMap attribute*), 262
MARCH (*in module calendar*), 259
markcoroutinefunction() (*in module inspect*), 2093
marshal
    module, 544
marshalling
    objects, 523
masking
    operations, 38
master (*tkinter.Tk attribute*), 1665
Match (*class in ast*), 2188
Match (*class in re*), 148
Match (*class in typing*), 1768
match() (*in module re*), 142
match() (*pathlib.PurePath method*), 464
match() (*re.Pattern method*), 146
match_case (*class in ast*), 2188
MATCH_CLASS (*opcode*), 2244
MATCH_KEYS (*opcode*), 2236
MATCH_MAPPING (*opcode*), 2236
MATCH_SEQUENCE (*opcode*), 2236
match_value() (*test.support.Matcher method*), 1912
MatchAs (*class in ast*), 2192
MatchClass (*class in ast*), 2191
Matcher (*class in test.support*), 1912
matches() (*test.support.Matcher method*), 1912
MatchMapping (*class in ast*), 2190
MatchOr (*class in ast*), 2193
MatchSequence (*class in ast*), 2189
MatchSingleton (*class in ast*), 2189
MatchStar (*class in ast*), 2190
MatchValue (*class in ast*), 2188
math
    module, 37, 346, 358
matmul() (*in module operator*), 448
MatMult (*class in ast*), 2174
max
    built-in function, 45
max (*datetime.date attribute*), 216
max (*datetime.datetime attribute*), 224
max (*datetime.time attribute*), 232
max (*datetime.timedelta attribute*), 212
max (*sys.float_info attribute*), 2000
max()
    built-in function, 19
max() (*decimal.Context method*), 376
max() (*decimal.Decimal method*), 368
max_10_exp (*sys.float_info attribute*), 2000
max_count (*email.headerregistry.BaseHeader attribute*), 1296
MAX_EMAX (*in module decimal*), 378
max_exp (*sys.float_info attribute*), 2000
MAX_INTERPOLATION_DEPTH (*in module configparser*), 662
max_line_length (*email.policy.Policy attribute*), 1288
max_lines (*textwrap.TextWrapper attribute*), 173
max_mag() (*decimal.Context method*), 376
max_mag() (*decimal.Decimal method*), 368
max_memuse (*in module test.support*), 1904
MAX_PREC (*in module decimal*), 378
max_prefixlen (*ipaddress.IPv4Address attribute*), 1574
max_prefixlen (*ipaddress.IPv4Network attribute*), 1579
max_prefixlen (*ipaddress.IPv6Address attribute*), 1576
max_prefixlen (*ipaddress.IPv6Network attribute*), 1582
MAX_Py_ssize_t (*in module test.support*), 1904
maxarray (*reprlib.Repr attribute*), 321
maxdeque (*reprlib.Repr attribute*), 321
maxdict (*reprlib.Repr attribute*), 321
maxDiff (*unittest.TestCase attribute*), 1821
maxfrozenset (*reprlib.Repr attribute*), 321
MAXIMUM_SUPPORTED (*ssl.TLSVersion attribute*), 1217
maximum_version (*ssl.SSLContext attribute*), 1228
maxlen (*collections.deque attribute*), 269
maxlevel (*reprlib.Repr attribute*), 321
maxlist (*reprlib.Repr attribute*), 321
maxlong (*reprlib.Repr attribute*), 321

maxother (*reprlib.Repr attribute*), 321
maxset (*reprlib.Repr attribute*), 321
maxsize (*asyncio.Queue attribute*), 1108
maxsize (*in module sys*), 2007
maxstring (*reprlib.Repr attribute*), 321
maxtuple (*reprlib.Repr attribute*), 321
maxunicode (*in module sys*), 2007
MAXYEAR (*in module datetime*), 210
MAY (*in module calendar*), 259
MB_ICONASTERISK (*in module winsound*), 2265
MB_ICONEXCLAMATION (*in module winsound*), 2265
MB_ICONHAND (*in module winsound*), 2265
MB_ICONQUESTION (*in module winsound*), 2265
MB_OK (*in module winsound*), 2265
mbox (*class in mailbox*), 1348
mboxMessage (*class in mailbox*), 1354
md5() (*in module hashlib*), 672
mean (*statistics.NormalDist attribute*), 414
mean() (*in module statistics*), 404
measure() (*tkinter.font.Font method*), 1678
median (*statistics.NormalDist attribute*), 414
median() (*in module statistics*), 407
median_grouped() (*in module statistics*), 408
median_high() (*in module statistics*), 408
median_low() (*in module statistics*), 408
member() (*in module enum*), 338
MemberDescriptorType (*in module types*), 310
memfd_create() (*in module os*), 734
memmove() (*in module ctypes*), 947
--memo
    pickletools command line option, 2249
MemoryBIO (*class in ssl*), 1239
MemoryError, 110
MemoryHandler (*class in logging.handlers*), 869
memoryview
    object, 63
memoryview (*built-in class*), 79
memset() (*in module ctypes*), 947
merge() (*in module heapq*), 287
message (*BaseExceptionGroup attribute*), 118
Message (*class in email.message*), 1311
Message (*class in mailbox*), 1352
Message (*class in tkinter.messagebox*), 1682
message digest, MD5, 671
message_factory (*email.policy.Policy attribute*), 1289
message_from_binary_file() (*in module email*), 1283
message_from_bytes() (*in module email*), 1283
message_from_file() (*in module email*), 1283
message_from_string() (*in module email*), 1283
MessageBeep() (*in module winsound*), 2263
MessageClass (*http.server.BaseHTTPRequestHandler attribute*), 1540
MessageDefect, 1294
MessageError, 1294
MessageParseError, 1294
messages (*in module xml.parsers.expat.errors*), 1440
meta path finder, **2330**
meta() (*in module curses*), 877
meta_path (*in module sys*), 2007
metaclass, **2330**
--metadata-encoding
    zipfile command line option, 617
MetaPathFinder (*class in importlib.abc*), 2131
metavar (*optparse.Option attribute*), 2304
MetavarTypeHelpFormatter (*class in argparse*), 794
method, **2330**
    magic, 2330
    object, 102
    special, 2335
method (*urllib.request.Request attribute*), 1463
method resolution order, **2330**
method_calls (*unittest.mock.Mock attribute*), 1845
methodcaller() (*in module operator*), 450
MethodDescriptorType (*in module types*), 308
methodHelp() (*xmlrpc.client.ServerProxy.system method*), 1560
methods
    bytearray, 65
    bytes, 65
    string, 52
methods (*pyclbr.Class attribute*), 2219
methodSignature() (*xmlrpc.client.ServerProxy.system method*), 1560
MethodType (*in module types*), 307
MethodWrapperType (*in module types*), 308
metrics() (*tkinter.font.Font method*), 1678
MFD_ALLOW_SEALING (*in module os*), 734
MFD_CLOEXEC (*in module os*), 734
MFD_HUGE_1GB (*in module os*), 734
MFD_HUGE_1MB (*in module os*), 734
MFD_HUGE_2GB (*in module os*), 734
MFD_HUGE_2MB (*in module os*), 734
MFD_HUGE_8MB (*in module os*), 734
MFD_HUGE_16GB (*in module os*), 734
MFD_HUGE_16MB (*in module os*), 734
MFD_HUGE_32MB (*in module os*), 734
MFD_HUGE_64KB (*in module os*), 734
MFD_HUGE_256MB (*in module os*), 734
MFD_HUGE_512KB (*in module os*), 734
MFD_HUGE_512MB (*in module os*), 734
MFD_HUGE_MASK (*in module os*), 734
MFD_HUGE_SHIFT (*in module os*), 734
MFD_HUGETLB (*in module os*), 734
MH (*class in mailbox*), 1348
MHMessage (*class in mailbox*), 1356
microsecond (*datetime.datetime attribute*), 224

microsecond (*datetime.time attribute*), 233
MIME
    base64 encoding, 1365
    content type, 1362
    headers, 1362
    quoted-printable encoding, 1371
MIMEApplication (*class in email.mime.application*), 1320
MIMEAudio (*class in email.mime.audio*), 1320
MIMEBase (*class in email.mime.base*), 1319
MIMEImage (*class in email.mime.image*), 1320
MIMEMessage (*class in email.mime.message*), 1321
MIMEMultipart (*class in email.mime.multipart*), 1319
MIMENonMultipart (*class in email.mime.nonmultipart*), 1319
MIMEPart (*class in email.message*), 1280
MIMEText (*class in email.mime.text*), 1321
mimetypes
    module, 1362
MimeTypes (*class in mimetypes*), 1364
MIMEVersionHeader (*class in email.headerregistry*), 1298
min
    built-in function, 45
min (*datetime.date attribute*), 216
min (*datetime.datetime attribute*), 224
min (*datetime.time attribute*), 232
min (*datetime.timedelta attribute*), 212
min (*sys.float_info attribute*), 2000
min()
    built-in function, 20
min() (*decimal.Context method*), 376
min() (*decimal.Decimal method*), 368
min_10_exp (*sys.float_info attribute*), 2000
MIN_EMIN (*in module decimal*), 378
MIN_ETINY (*in module decimal*), 378
min_exp (*sys.float_info attribute*), 2000
min_mag() (*decimal.Context method*), 376
min_mag() (*decimal.Decimal method*), 368
MINEQUAL (*in module token*), 2210
MINIMUM_SUPPORTED (*ssl.TLSVersion attribute*), 1217
minimum_version (*ssl.SSLContext attribute*), 1229
minor (*email.headerregistry.MIMEVersionHeader attribute*), 1298
minor() (*in module os*), 720
MINUS (*in module token*), 2209
minus() (*decimal.Context method*), 376
minute (*datetime.datetime attribute*), 224
minute (*datetime.time attribute*), 233
MINYEAR (*in module datetime*), 210
mirrored() (*in module unicodedata*), 174
misc_header (*cmd.Cmd attribute*), 1653
--missing
    trace command line option, 1960
MISSING (*contextvars.Token attribute*), 1059
MISSING (*in module dataclasses*), 2049
MISSING (*in module sys.monitoring*), 2022
MISSING_C_DOCSTRINGS (*in module test.support*), 1905
missing_compiler_executable() (*in module test.support*), 1911
MissingSectionHeaderError, 663
mkd() (*ftplib.FTP method*), 1504
mkdir() (*in module os*), 719
mkdir() (*pathlib.Path method*), 476
mkdir() (*zipfile.ZipFile method*), 611
mkdtemp() (*in module tempfile*), 503
mkfifo() (*in module os*), 720
mknod() (*in module os*), 720
mkstemp() (*in module tempfile*), 502
mktemp() (*in module tempfile*), 505
mktime() (*in module time*), 777
mktime_tz() (*in module email.utils*), 1328
mlsd() (*ftplib.FTP method*), 1503
mmap
    module, 1263
mmap (*class in mmap*), 1264
MMDF (*class in mailbox*), 1351
MMDFMessage (*class in mailbox*), 1358
Mock (*class in unittest.mock*), 1838
mock_add_spec() (*unittest.mock.Mock method*), 1841
mock_calls (*unittest.mock.Mock attribute*), 1845
mock_open() (*in module unittest.mock*), 1873
Mod (*class in ast*), 2174
mod() (*in module operator*), 447
--mode
    ast command line option, 2203
mode (*bz2.BZ2File attribute*), 596
mode (*gzip.GzipFile attribute*), 592
mode (*io.FileIO attribute*), 768
mode (*lzma.LZMAFile attribute*), 601
mode (*statistics.NormalDist attribute*), 414
mode (*tarfile.TarInfo attribute*), 626
mode() (*in module statistics*), 409
mode() (*in module turtle*), 1642
modes
    file, 21
modf() (*in module math*), 349
modified() (*urllib.robotparser.RobotFileParser method*), 1487
modify() (*select.devpoll method*), 1244
modify() (*select.epoll method*), 1246
modify() (*selectors.BaseSelector method*), 1251
modify() (*select.poll method*), 1246
module, **2330**
    __future__, 2085
    __main__, 2030, 2126, 2127
    _locale, 1601

_thread, 1062
_tkinter, 1666
abc, 2070
argparse, 787
array, 63, 293
ast, 2165
asyncio, 1065
atexit, 2075
base64, 1365, 1369
bdb, 1927, 1935
binascii, 1369
bisect, 290
builtins, 31, 2030
bz2, 595
calendar, 254
cmath, 355
cmd, 1651, 1935
code, 2115
codecs, 191
codeop, 2117
collections, 261
collections.abc, 280
colorsys, 1592
compileall, 2221
concurrent.futures, 1023
configparser, 645
contextlib, 2054
contextvars, 1058
copy, 312, 541
copyreg, 541
cProfile, 1947
csv, 637
ctypes, 920
curses, 873
curses.ascii, 903
curses.panel, 907
curses.textpad, 901
dataclasses, 2043
datetime, 209
dbm, 546
dbm.dumb, 551
dbm.gnu, 543, 549
dbm.ndbm, 543, 550
dbm.sqlite3, 548
decimal, 359
difflib, 156
dis, 2225
doctest, 1779
email, 1271
email.charset, 1324
email.contentmanager, 1301
email.encoders, 1326
email.errors, 1294
email.generator, 1284
email.header, 1321
email.headerregistry, 1295
email.iterators, 1329
email.message, 1272
email.mime, 1319
email.mime.application, 1320
email.mime.audio, 1320
email.mime.base, 1319
email.mime.image, 1320
email.mime.message, 1321
email.mime.multipart, 1319
email.mime.nonmultipart, 1319
email.mime.text, 1321
email.parser, 1280
email.policy, 1287
email.utils, 1327
encodings.idna, 207
encodings.mbcs, 208
encodings.utf_8_sig, 208
ensurepip, 1973
enum, 323
errno, 111, 913
faulthandler, 1933
fcntl, 2274
filecmp, 498
fileinput, 489
fnmatch, 508
fractions, 388
ftplib, 1500
functools, 436
gc, 2086
getopt, 2287
getpass, 872
gettext, 1593
glob, 506, 508
graphlib, 339
grp, 2269
gzip, 591
hashlib, 671
heapq, 286
hmac, 683
html, 1373
html.entities, 1378
html.parser, 1373
http, 1488
http.client, 1492
http.cookiejar, 1549
http.cookies, 1545
http.server, 1538
idlelib, 1718
imaplib, 1510
importlib, 2128
importlib.abc, 2131
importlib.machinery, 2138

importlib.metadata, 2155
importlib.resources, 2150
importlib.resources.abc, 2153
importlib.util, 2144
inspect, 2090
io, 760
ipaddress, 1572
itertools, 419
json, 1331
json.tool, 1340
keyword, 2212
linecache, 510
locale, 1601
logging, 823
logging.config, 844
logging.handlers, 857
lzma, 600
mailbox, 1341
marshal, 544
math, 37, 346, 358
mimetypes, 1362
mmap, 1263
modulefinder, 2125
msvcrt, 2251
multiprocessing, 970
multiprocessing.connection, 1003
multiprocessing.dummy, 1007
multiprocessing.managers, 993
multiprocessing.pool, 1000
multiprocessing.shared_memory, 1017
multiprocessing.sharedctypes, 991
netrc, 665
numbers, 343
operator, 446
optparse, 2289
os, 689, 2267
os.path, 482
pathlib, 455
pdb, 1935
pickle, 313, 523, 541, 544
pickletools, 2248
pkgutil, 2121
platform, 908
plistlib, 667
poplib, 1507
posix, 2267
pprint, 314
profile, 1947
pstats, 1949
pty, 706, 2272
pwd, 483, 2268
py_compile, 2219
pyclbr, 2217
pydoc, 1774
pyexpat, 1432
queue, 1054
quopri, 1371
random, 392
re, 52, 132, 508
readline, 177
reprlib, 320
resource, 2277
rlcompleter, 181
runpy, 2126
sched, 1052
search path, 510, 2008, 2111
secrets, 685
select, 1242
selectors, 1249
shelve, 541, 544
shlex, 1656
shutil, 510
signal, 1064, 1253
site, 2111
sitecustomize, 2112
smtplib, 1517
socket, 1175, 1443
socketserver, 1529
sqlite3, 553
ssl, 1204
stat, 491, 726
statistics, 402
string, 121
stringprep, 175
struct, 183, 1199
subprocess, 1030
symtable, 2204
sys, 23, 1991
sysconfig, 2023
syslog, 2281
sys.monitoring, 2017
tabnanny, 2216
tarfile, 618
tempfile, 500
termios, 2270
test, 1900
test.regrtest, 1902
test.support, 1903
test.support.bytecode_helper, 1915
test.support.import_helper, 1918
test.support.os_helper, 1916
test.support.script_helper, 1913
test.support.socket_helper, 1913
test.support.threading_helper, 1915
test.support.warnings_helper, 1919
textwrap, 169
threading, 955
time, 774

timeit, 1953
tkinter, 1663
tkinter.colorchooser, 1677
tkinter.commondialog, 1682
tkinter.dnd, 1685
tkinter.filedialog, 1679
tkinter.font, 1677
tkinter.messagebox, 1682
tkinter.scrolledtext, 1684
tkinter.simpledialog, 1679
tkinter.ttk, 1686
token, 2208
tokenize, 2212
tomllib, 664
trace, 1959
traceback, 2076
tracemalloc, 1961
tty, 2271
turtle, 1611
turtledemo, 1649
types, 102, 305
typing, 1719
unicodedata, 173
unittest, 1803
unittest.mock, 1835
urllib, 1457
urllib.error, 1486
urllib.parse, 1477
urllib.request, 1457, 1492
urllib.response, 1476
urllib.robotparser, 1487
usercustomize, 2113
uuid, 1524
venv, 1975
warnings, 2036
wave, 1589
weakref, 297
webbrowser, 1443
winreg, 2254
winsound, 2263
wsgiref, 1446
wsgiref.handlers, 1451
wsgiref.headers, 1448
wsgiref.simple_server, 1449
wsgiref.types, 1455
wsgiref.util, 1446
wsgiref.validate, 1450
xml, 1379
xml.dom, 1401
xml.dom.minidom, 1412
xml.dom.pulldom, 1417
xml.etree.ElementInclude, 1393
xml.etree.ElementTree, 1381
xml.parsers.expat, 1432
xml.parsers.expat.errors, 1440
xml.parsers.expat.model, 1439
xmlrpc.client, 1558
xmlrpc.server, 1566
xml.sax, 1419
xml.sax.handler, 1421
xml.sax.saxutils, 1426
xml.sax.xmlreader, 1428
zipapp, 1985
zipfile, 606
zipimport, 2119
zlib, 587
zoneinfo, 248
Module (*class in ast*), 2170
module (*pyclbr.Class attribute*), 2218
module (*pyclbr.Function attribute*), 2218
MODULE (*symtable.SymbolTableType attribute*), 2204
Module browser, 1706
module spec, **2330**
module_from_spec() (*in module importlib.util*), 2145
modulefinder
    module, 2125
ModuleFinder (*class in modulefinder*), 2125
ModuleInfo (*class in pkgutil*), 2121
ModuleNotFoundError, 110
modules (*in module sys*), 2008
modules (*modulefinder.ModuleFinder attribute*), 2125
modules_cleanup() (*in module test.support.import_helper*), 1919
modules_setup() (*in module test.support.import_helper*), 1919
ModuleSpec (*class in importlib.machinery*), 2142
ModuleType (*class in types*), 308
modulus (*sys.hash_info attribute*), 2004
MON_1 (*in module locale*), 1604
MON_2 (*in module locale*), 1604
MON_3 (*in module locale*), 1604
MON_4 (*in module locale*), 1604
MON_5 (*in module locale*), 1604
MON_6 (*in module locale*), 1604
MON_7 (*in module locale*), 1604
MON_8 (*in module locale*), 1604
MON_9 (*in module locale*), 1604
MON_10 (*in module locale*), 1604
MON_11 (*in module locale*), 1604
MON_12 (*in module locale*), 1604
MONDAY (*in module calendar*), 258
monotonic() (*in module time*), 778
monotonic_ns() (*in module time*), 778
month
    calendar command line option, 261
month (*calendar.IllegalMonthError attribute*), 259
Month (*class in calendar*), 259
month (*datetime.date attribute*), 216

month (*datetime.datetime attribute*), 224
month() (*in module calendar*), 258
month_abbr (*in module calendar*), 259
month_name (*in module calendar*), 259
monthcalendar() (*in module calendar*), 258
monthdatescalendar() (*calendar.Calendar method*), 255
monthdays2calendar() (*calendar.Calendar method*), 255
monthdayscalendar() (*calendar.Calendar method*), 255
monthrange() (*in module calendar*), 258
--months
    calendar command line option, 261
Morsel (*class in http.cookies*), 1546
most_common() (*collections.Counter method*), 265
mouseinterval() (*in module curses*), 877
mousemask() (*in module curses*), 877
move() (*curses.panel.Panel method*), 907
move() (*curses.window method*), 885
move() (*in module shutil*), 514
move() (*mmap.mmap method*), 1266
move() (*tkinter.ttk.Treeview method*), 1700
move_to_end() (*collections.OrderedDict method*), 277
MozillaCookieJar (*class in http.cookiejar*), 1553
MRO, **2331**
mro() (*class method*), 104
msg (*http.client.HTTPResponse attribute*), 1497
msg (*json.JSONDecodeError attribute*), 1338
msg (*netrc.NetrcParseError attribute*), 666
msg (*re.PatternError attribute*), 146
msg (*traceback.TracebackException attribute*), 2080
msvcrt
    module, 2251
mtime (*gzip.GzipFile attribute*), 592
mtime (*tarfile.TarInfo attribute*), 626
mtime() (*urllib.robotparser.RobotFileParser method*), 1487
mul() (*in module operator*), 448
Mult (*class in ast*), 2174
MultiCall (*class in xmlrpc.client*), 1564
MULTILINE (*in module re*), 141
MultilineContinuationError, 664
MultiLoopChildWatcher (*class in asyncio*), 1159
multimode() (*in module statistics*), 409
MultipartConversionError, 1294
multiply() (*decimal.Context method*), 376
multiprocessing
    module, 970
multiprocessing.connection
    module, 1003
multiprocessing.dummy
    module, 1007
multiprocessing.Manager()
    built-in function, 993
multiprocessing.managers
    module, 993
multiprocessing.pool
    module, 1000
multiprocessing.shared_memory
    module, 1017
multiprocessing.sharedctypes
    module, 991
mutable, **2331**
    sequence types, 47
mutable sequence
    loop over, 45
MutableMapping (*class in collections.abc*), 284
MutableMapping (*class in typing*), 1770
MutableSequence (*class in collections.abc*), 283
MutableSequence (*class in typing*), 1770
MutableSet (*class in collections.abc*), 284
MutableSet (*class in typing*), 1770
mvderwin() (*curses.window method*), 885
mvwin() (*curses.window method*), 886
myrights() (*imaplib.IMAP4 method*), 1513

N

-N
    uuid command line option, 1528
-n
    timeit command line option, 1956
    uuid command line option, 1528
N_TOKENS (*in module token*), 2211
n_waiting (*asyncio.Barrier attribute*), 1102
n_waiting (*threading.Barrier attribute*), 969
NAK (*in module curses.ascii*), 904
--name
    uuid command line option, 1528
name (*bz2.BZ2File attribute*), 596
Name (*class in ast*), 2173
name (*codecs.CodecInfo attribute*), 191
name (*contextvars.ContextVar attribute*), 1058
name (*doctest.DocTest attribute*), 1794
name (*email.headerregistry.BaseHeader attribute*), 1296
name (*enum.Enum attribute*), 327
name (*gzip.GzipFile attribute*), 592
name (*hashlib.hash attribute*), 673
name (*hmac.HMAC attribute*), 684
name (*http.cookiejar.Cookie attribute*), 1556
name (*ImportError attribute*), 109
name (*importlib.abc.FileLoader attribute*), 2135
name (*importlib.abc.Traversable attribute*), 2137
name (*importlib.machinery.AppleFrameworkLoader attribute*), 2144
name (*importlib.machinery.ExtensionFileLoader attribute*), 2141
name (*importlib.machinery.ModuleSpec attribute*), 2142

name (*importlib.machinery.SourceFileLoader attribute*), 2140
name (*importlib.machinery.SourcelessFileLoader attribute*), 2141
name (*importlib.resources.abc.Traversable attribute*), 2154
name (*in module os*), 689
NAME (*in module token*), 2208
name (*inspect.Parameter attribute*), 2099
name (*io.FileIO attribute*), 768
name (*logging.Logger attribute*), 825
name (*lzma.LZMAFile attribute*), 601
name (*multiprocessing.Process attribute*), 978
name (*multiprocessing.shared_memory.SharedMemory attribute*), 1018
name (*os.DirEntry attribute*), 724
name (*pathlib.PurePath attribute*), 462
name (*pyclbr.Class attribute*), 2218
name (*pyclbr.Function attribute*), 2218
name (*sys.thread_info attribute*), 2015
name (*tarfile.TarInfo attribute*), 626
name (*tempfile.TemporaryDirectory attribute*), 502
name (*threading.Thread attribute*), 960
name (*traceback.FrameSummary attribute*), 2082
name (*webbrowser.controller attribute*), 1446
name (*xml.dom.Attr attribute*), 1409
name (*xml.dom.DocumentType attribute*), 1406
name (*zipfile.Path attribute*), 612
name() (*in module unicodedata*), 173
name2codepoint (*in module html.entities*), 1378
Named Shared Memory, 1017
named tuple, **2331**
NAMED_FLAGS (*enum.EnumCheck attribute*), 334
NamedExpr (*class in ast*), 2176
NamedTemporaryFile() (*in module tempfile*), 501
NamedTuple (*class in typing*), 1750
namedtuple() (*in module collections*), 273
NameError, 110
namelist() (*zipfile.ZipFile method*), 609
nameprep() (*in module encodings.idna*), 208
namer (*logging.handlers.BaseRotatingHandler attribute*), 860
namereplace
    error handler's name, 194
namereplace_errors() (*in module codecs*), 196
names() (*in module tkinter.font*), 1678
namespace, **2331**
--namespace
    uuid command line option, 1528
Namespace (*class in argparse*), 813
Namespace (*class in multiprocessing.managers*), 995
namespace package, **2331**
namespace() (*imaplib.IMAP4 method*), 1513
Namespace() (*multiprocessing.managers.SyncManager method*), 995
NAMESPACE_DNS (*in module uuid*), 1527
NAMESPACE_OID (*in module uuid*), 1527
NAMESPACE_URL (*in module uuid*), 1527
NAMESPACE_X500 (*in module uuid*), 1527
NamespaceErr, 1410
NamespaceLoader (*class in importlib.machinery*), 2142
namespaceURI (*xml.dom.Node attribute*), 1404
nametofont() (*in module tkinter.font*), 1679
NaN, 14
nan (*in module cmath*), 358
nan (*in module math*), 354
nan (*sys.hash_info attribute*), 2004
nanj (*in module cmath*), 358
NannyNag, 2217
napms() (*in module curses*), 877
nargs (*optparse.Option attribute*), 2304
native_id (*threading.Thread attribute*), 960
nbytes (*memoryview attribute*), 84
ncurses_version (*in module curses*), 888
ND (*inspect.BufferFlags attribute*), 2110
ndiff() (*in module difflib*), 159
ndim (*memoryview attribute*), 85
ne() (*in module operator*), 446
needs_input (*bz2.BZ2Decompressor attribute*), 598
needs_input (*lzma.LZMADecompressor attribute*), 603
neg() (*in module operator*), 448
nested scope, **2331**
netmask (*ipaddress.IPv4Network attribute*), 1580
netmask (*ipaddress.IPv6Network attribute*), 1583
NetmaskValueError, 1587
netrc
    module, 665
netrc (*class in netrc*), 665
NetrcParseError, 666
netscape (*http.cookiejar.CookiePolicy attribute*), 1554
network (*ipaddress.IPv4Interface attribute*), 1585
network (*ipaddress.IPv6Interface attribute*), 1585
network_address (*ipaddress.IPv4Network attribute*), 1580
network_address (*ipaddress.IPv6Network attribute*), 1583
Never (*in module typing*), 1732
NEVER_EQ (*in module test.support*), 1905
new() (*in module hashlib*), 672
new() (*in module hmac*), 683
new-style class, **2331**
new_child() (*collections.ChainMap method*), 262
new_class() (*in module types*), 305
new_event_loop() (*asyncio.AbstractEventLoopPolicy method*), 1157
new_event_loop() (*in module asyncio*), 1113
new_panel() (*in module curses.panel*), 907
NEWLINE (*in module token*), 2208

newlines (*io.TextIOBase attribute*), 770
newpad() (*in module curses*), 877
NewType (*class in typing*), 1751
newwin() (*in module curses*), 877
next (*pdb command*), 1940
next()
    built-in function, 20
next() (*tarfile.TarFile method*), 623
next() (*tkinter.ttk.Treeview method*), 1700
next_minus() (*decimal.Context method*), 376
next_minus() (*decimal.Decimal method*), 368
next_plus() (*decimal.Context method*), 376
next_plus() (*decimal.Decimal method*), 368
next_toward() (*decimal.Context method*), 376
next_toward() (*decimal.Decimal method*), 368
nextafter() (*in module math*), 349
nextfile() (*in module fileinput*), 490
nextkey() (*dbm.gnu.gdbm method*), 550
nextSibling (*xml.dom.Node attribute*), 1404
ngettext() (*gettext.GNUTranslations method*), 1597
ngettext() (*gettext.NullTranslations method*), 1596
ngettext() (*in module gettext*), 1594
nice() (*in module os*), 745
NL (*in module curses.ascii*), 904
NL (*in module token*), 2211
nl() (*in module curses*), 878
nl_langinfo() (*in module locale*), 1604
nlargest() (*in module heapq*), 287
nlst() (*ftplib.FTP method*), 1504
NO (*in module tkinter.messagebox*), 1684
no_cache() (*zoneinfo.ZoneInfo class method*), 251
NO_EVENTS (*monitoring event*), 2019
no_proxy, 1461
no_site (*sys.flags attribute*), 1998
no_tracing() (*in module test.support*), 1909
no_type_check() (*in module typing*), 1762
no_type_check_decorator() (*in module typing*), 1762
no_user_site (*sys.flags attribute*), 1998
nocbreak() (*in module curses*), 878
NoDataAllowedErr, 1411
node (*uuid.UUID attribute*), 1526
node() (*in module platform*), 909
NoDefault (*in module typing*), 1765
nodelay() (*curses.window method*), 886
nodeName (*xml.dom.Node attribute*), 1404
NodeTransformer (*class in ast*), 2201
nodeType (*xml.dom.Node attribute*), 1404
nodeValue (*xml.dom.Node attribute*), 1405
NodeVisitor (*class in ast*), 2201
noecho() (*in module curses*), 878
--no-ensure-ascii
    json.tool command line option, 1340
NOEXPR (*in module locale*), 1605
NOFLAG (*in module re*), 141
--no-indent
    json.tool command line option, 1341
NoModificationAllowedErr, 1411
NonCallableMagicMock (*class in unittest.mock*), 1867
NonCallableMock (*class in unittest.mock*), 1846
None (*Built-in object*), 35
None (*built-in variable*), 33
NoneType (*in module types*), 307
nonl() (*in module curses*), 878
Nonlocal (*class in ast*), 2197
nonmember() (*in module enum*), 338
noop() (*imaplib.IMAP4 method*), 1514
noop() (*poplib.POP3 method*), 1509
NoOptionError, 663
NOP (*opcode*), 2231
noqiflush() (*in module curses*), 878
noraw() (*in module curses*), 878
--no-report
    trace command line option, 1960
NoReturn (*in module typing*), 1732
NORMAL (*in module tkinter.font*), 1677
NORMAL_PRIORITY_CLASS (*in module subprocess*), 1045
NormalDist (*class in statistics*), 414
normalize() (*decimal.Context method*), 376
normalize() (*decimal.Decimal method*), 368
normalize() (*in module locale*), 1607
normalize() (*in module unicodedata*), 174
normalize() (*xml.dom.Node method*), 1405
NORMALIZE_WHITESPACE (*in module doctest*), 1786
normalvariate() (*in module random*), 396
normcase() (*in module os.path*), 486
normpath() (*in module os.path*), 486
NoSectionError, 663
NoSuchMailboxError, 1360
not
    operator, 35
Not (*class in ast*), 2174
not in
    operator, 36, 45
not_() (*in module operator*), 447
NotADirectoryError, 116
notationDecl() (*xml.sax.handler.DTDHandler method*), 1425
NotationDeclHandler() (*xml.parsers.expat.xmlparser method*), 1437
notations (*xml.dom.DocumentType attribute*), 1406
NotConnected, 1493
Notebook (*class in tkinter.ttk*), 1693
NotEmptyError, 1360
NotEq (*class in ast*), 2175
NOTEQUAL (*in module token*), 2209

NotFoundErr, 1411
notify() (*asyncio.Condition method*), 1099
notify() (*threading.Condition method*), 965
notify_all() (*asyncio.Condition method*), 1100
notify_all() (*threading.Condition method*), 965
notimeout() (*curses.window method*), 886
NotImplemented (*built-in variable*), 33
NotImplementedError, 110
NotImplementedType (*in module types*), 308
NotIn (*class in ast*), 2175
NotRequired (*in module typing*), 1737
NOTSET (*in module logging*), 830
NotStandaloneHandler() (*xml.parsers.expat.xmlparser method*), 1437
NotSupportedErr, 1411
NotSupportedError, 576
--no-type-comments
    ast command line option, 2203
noutrefresh() (*curses.window method*), 886
NOVEMBER (*in module calendar*), 259
now() (*datetime.datetime class method*), 221
npgettext() (*gettext.GNUTranslations method*), 1597
npgettext() (*gettext.NullTranslations method*), 1596
npgettext() (*in module gettext*), 1594
NSIG (*in module signal*), 1257
nsmallest() (*in module heapq*), 287
NT_OFFSET (*in module token*), 2211
NTEventLogHandler (*class in logging.handlers*), 867
ntohl() (*in module socket*), 1188
ntohs() (*in module socket*), 1188
ntransfercmd() (*ftplib.FTP method*), 1503
NUL (*in module curses.ascii*), 903
nullcontext() (*in module contextlib*), 2057
NullHandler (*class in logging*), 859
NullTranslations (*class in gettext*), 1595
num_addresses (*ipaddress.IPv4Network attribute*), 1580
num_addresses (*ipaddress.IPv6Network attribute*), 1583
num_tickets (*ssl.SSLContext attribute*), 1229
--number
    timeit command line option, 1956
Number (*class in numbers*), 343
NUMBER (*in module token*), 2208
number_class() (*decimal.Context method*), 376
number_class() (*decimal.Decimal method*), 368
numbers
    module, 343
numerator (*fractions.Fraction attribute*), 390
numerator (*numbers.Rational attribute*), 344
numeric
    conversions, 37
    literals, 36
    object, 36
    types, operations on, 37
numeric() (*in module unicodedata*), 173
numinput() (*in module turtle*), 1641

O

-O
    dis command line option, 2226
-o
    compileall command line option, 2222
    pickletools command line option, 2249
    zipapp command line option, 1986
O_APPEND (*in module os*), 704
O_ASYNC (*in module os*), 705
O_BINARY (*in module os*), 705
O_CLOEXEC (*in module os*), 705
O_CREAT (*in module os*), 704
O_DIRECT (*in module os*), 705
O_DIRECTORY (*in module os*), 705
O_DSYNC (*in module os*), 705
O_EVTONLY (*in module os*), 705
O_EXCL (*in module os*), 704
O_EXLOCK (*in module os*), 705
O_FSYNC (*in module os*), 705
O_NDELAY (*in module os*), 705
O_NOATIME (*in module os*), 705
O_NOCTTY (*in module os*), 705
O_NOFOLLOW (*in module os*), 705
O_NOFOLLOW_ANY (*in module os*), 705
O_NOINHERIT (*in module os*), 705
O_NONBLOCK (*in module os*), 705
O_PATH (*in module os*), 705
O_RANDOM (*in module os*), 705
O_RDONLY (*in module os*), 704
O_RDWR (*in module os*), 704
O_RSYNC (*in module os*), 705
O_SEQUENTIAL (*in module os*), 705
O_SHLOCK (*in module os*), 705
O_SHORT_LIVED (*in module os*), 705
O_SYMLINK (*in module os*), 705
O_SYNC (*in module os*), 705
O_TEMPORARY (*in module os*), 705
O_TEXT (*in module os*), 705
O_TMPFILE (*in module os*), 705
O_TRUNC (*in module os*), 704
O_WRONLY (*in module os*), 704
obj (*memoryview attribute*), 84
object, **2331**
    Boolean, 36
    bytearray, 47, 63, 64
    bytes, 63
    code, 102, 545
    complex number, 36
    dictionary, 89
    floating-point, 36

GenericAlias, 95
integer, 36
io.StringIO, 51
list, 47, 48
mapping, 89
memoryview, 63
method, 102
numeric, 36
range, 49
sequence, 45
set, 86
socket, 1175
string, 51
traceback, 1996, 2076
tuple, 47, 49
type, 29
Union, 99
object (*built-in class*), 20
object (*UnicodeError attribute*), 114
objects
comparing, 36
flattening, 523
marshalling, 523
persistent, 523
pickling, 523
serializing, 523
oct()
built-in function, 20
octal
literals, 36
octdigits (*in module string*), 121
OCTOBER (*in module calendar*), 259
offset (*SyntaxError attribute*), 113
offset (*tarfile.TarInfo attribute*), 627
offset (*traceback.TracebackException attribute*), 2080
offset (*xml.parsers.expat.ExpatError attribute*), 1438
offset_data (*tarfile.TarInfo attribute*), 627
OK (*in module curses*), 888
OK (*in module tkinter.messagebox*), 1684
ok_command() (*tkinter.filedialog.LoadFileDialog method*), 1681
ok_command() (*tkinter.filedialog.SaveFileDialog method*), 1681
ok_event() (*tkinter.filedialog.FileDialog method*), 1681
OKCANCEL (*in module tkinter.messagebox*), 1684
old_value (*contextvars.Token attribute*), 1059
OleDLL (*class in ctypes*), 940
on_motion() (*tkinter.dnd.DndHandler method*), 1685
on_release() (*tkinter.dnd.DndHandler method*), 1685
onclick() (*in module turtle*), 1640
ondrag() (*in module turtle*), 1635
onecmd() (*cmd.Cmd method*), 1652
onkey() (*in module turtle*), 1639
onkeypress() (*in module turtle*), 1640
onkeyrelease() (*in module turtle*), 1639
onrelease() (*in module turtle*), 1634
onscreenclick() (*in module turtle*), 1640
ontimer() (*in module turtle*), 1640
OP (*in module token*), 2211
OP_ALL (*in module ssl*), 1212
OP_CIPHER_SERVER_PREFERENCE (*in module ssl*), 1213
OP_ENABLE_KTLS (*in module ssl*), 1214
OP_ENABLE_MIDDLEBOX_COMPAT (*in module ssl*), 1214
OP_IGNORE_UNEXPECTED_EOF (*in module ssl*), 1214
OP_LEGACY_SERVER_CONNECT (*in module ssl*), 1214
OP_NO_COMPRESSION (*in module ssl*), 1214
OP_NO_RENEGOTIATION (*in module ssl*), 1213
OP_NO_SSLv2 (*in module ssl*), 1212
OP_NO_SSLv3 (*in module ssl*), 1212
OP_NO_TICKET (*in module ssl*), 1214
OP_NO_TLSv1 (*in module ssl*), 1213
OP_NO_TLSv1_1 (*in module ssl*), 1213
OP_NO_TLSv1_2 (*in module ssl*), 1213
OP_NO_TLSv1_3 (*in module ssl*), 1213
OP_SINGLE_DH_USE (*in module ssl*), 1213
OP_SINGLE_ECDH_USE (*in module ssl*), 1213
Open (*class in tkinter.filedialog*), 1680
open()
built-in function, 21
open() (*imaplib.IMAP4 method*), 1514
open() (*importlib.abc.Traversable method*), 2137
open() (*importlib.resources.abc.Traversable method*), 2154
open() (*in module bz2*), 595
open() (*in module codecs*), 192
open() (*in module dbm*), 547
open() (*in module dbm.dumb*), 552
open() (*in module dbm.gnu*), 549
open() (*in module dbm.ndbm*), 551
open() (*in module dbm.sqlite3*), 548
open() (*in module gzip*), 591
open() (*in module io*), 762
open() (*in module lzma*), 600
open() (*in module os*), 704
open() (*in module shelve*), 541
open() (*in module tarfile*), 618
open() (*in module tokenize*), 2214
open() (*in module wave*), 1589
open() (*in module webbrowser*), 1444
open() (*pathlib.Path method*), 472
open() (*tarfile.TarFile class method*), 623
open() (*urllib.request.OpenerDirector method*), 1464
open() (*urllib.request.URLopener method*), 1474
open() (*webbrowser.controller method*), 1446
open() (*zipfile.Path method*), 612
open() (*zipfile.ZipFile method*), 609

open_binary() (*in module importlib.resources*), 2151
open_code() (*in module io*), 762
open_connection() (*in module asyncio*), 1089
open_flags (*in module dbm.gnu*), 549
open_new() (*in module webbrowser*), 1444
open_new() (*webbrowser.controller method*), 1446
open_new_tab() (*in module webbrowser*), 1444
open_new_tab() (*webbrowser.controller method*), 1446
open_osfhandle() (*in module msvcrt*), 2252
open_resource() (*importlib.abc.ResourceReader method*), 2137
open_resource() (*importlib.resources.abc.ResourceReader method*), 2153
open_text() (*in module importlib.resources*), 2151
open_unix_connection() (*in module asyncio*), 1090
open_unknown() (*urllib.request.URLopener method*), 1474
open_urlresource() (*in module test.support*), 1910
OpenerDirector (*class in urllib.request*), 1460
OpenKey() (*in module winreg*), 2257
OpenKeyEx() (*in module winreg*), 2257
openlog() (*in module syslog*), 2282
openpty() (*in module os*), 705
openpty() (*in module pty*), 2273
OpenSSL
    (use in module hashlib), 671
    (use in module ssl), 1204
OPENSSL_VERSION (*in module ssl*), 1216
OPENSSL_VERSION_INFO (*in module ssl*), 1216
OPENSSL_VERSION_NUMBER (*in module ssl*), 1216
operation
    concatenation, 45
    repetition, 45
    slice, 45
    subscript, 45
OperationalError, 575
operations
    bitwise, 38
    Boolean, 35
    masking, 38
    shifting, 38
operations on
    dictionary type, 89
    integer types, 38
    list type, 47
    mapping types, 89
    numeric types, 37
    sequence types, 45, 47
operator
    - (*minus*), 37
    % (*percent*), 37
    & (*ampersand*), 38
    * (*asterisk*), 37
    **, 37
    + (*plus*), 37
    / (*slash*), 37
    //, 37
    < (*less*), 36
    <<, 38
    <=, 36
    !=, 36
    ==, 36
    > (*greater*), 36
    >=, 36
    >>, 38
    ^ (*caret*), 38
    | (*vertical bar*), 38
    ~ (*tilde*), 38
    and, 35
    comparison, 36
    in, 36, 45
    is, 36
    is not, 36
    module, 446
    not, 35
    not in, 36, 45
    or, 35
opmap (*in module dis*), 2247
opname (*in module dis*), 2247
optim_args_from_interpreter_flags() (*in module test.support*), 1907
optimize (*sys.flags attribute*), 1998
optimize() (*in module pickletools*), 2250
optimized scope, **2331**
OPTIMIZED_BYTECODE_SUFFIXES (*in module importlib.machinery*), 2138
Option (*class in optparse*), 2303
Optional (*in module typing*), 1735
OptionConflictError, 2318
OptionError, 2318
OptionGroup (*class in optparse*), 2297
OptionParser (*class in optparse*), 2300
Options (*class in ssl*), 1214
options (*doctest.Example attribute*), 1795
options (*ssl.SSLContext attribute*), 1229
options() (*configparser.ConfigParser method*), 660
OptionValueError, 2318
optionxform() (*configparser.ConfigParser method*), 662
optparse
    module, 2289
or
    operator, 35
Or (*class in ast*), 2175
or_() (*in module operator*), 448

ord()
    built-in function, 23
ordered_attributes (*xml.parsers.expat.xmlparser attribute*), 1435
OrderedDict (*class in collections*), 277
OrderedDict (*class in typing*), 1768
orig_argv (*in module sys*), 2008
origin (*importlib.machinery.ModuleSpec attribute*), 2142
origin_req_host (*urllib.request.Request attribute*), 1463
origin_server (*wsgiref.handlers.BaseHandler attribute*), 1454
os
    module, 689, 2267
os_environ (*wsgiref.handlers.BaseHandler attribute*), 1453
OSError, 111
os.path
    module, 482
OUT_TO_DEFAULT (*in module msvcrt*), 2253
OUT_TO_MSGBOX (*in module msvcrt*), 2253
OUT_TO_STDERR (*in module msvcrt*), 2253
outfile
    json.tool command line option, 1340
--output
    pickletools command line option, 2249
    zipapp command line option, 1986
output (*subprocess.CalledProcessError attribute*), 1033
output (*subprocess.TimeoutExpired attribute*), 1033
output (*unittest.TestCase attribute*), 1818
output() (*http.cookies.BaseCookie method*), 1546
output() (*http.cookies.Morsel method*), 1547
output_charset (*email.charset.Charset attribute*), 1324
output_codec (*email.charset.Charset attribute*), 1324
output_difference() (*doctest.OutputChecker method*), 1799
OutputChecker (*class in doctest*), 1799
OutputString() (*http.cookies.Morsel method*), 1547
OutsideDestinationError, 620
Overflow (*class in decimal*), 379
OverflowError, 111
overlap() (*statistics.NormalDist method*), 415
overlaps() (*ipaddress.IPv4Network method*), 1580
overlaps() (*ipaddress.IPv6Network method*), 1583
overlay() (*curses.window method*), 886
overload() (*in module typing*), 1761
override() (*in module typing*), 1763
overwrite() (*curses.window method*), 886
owner() (*pathlib.Path method*), 478

P

-p
    compileall command line option, 2221
    pickletools command line option, 2249
    timeit command line option, 1956
    unittest-discover command line option, 1807
    zipapp command line option, 1986
p (*pdb command*), 1941
P_ALL (*in module os*), 752
P_DETACH (*in module os*), 749
P_NOWAIT (*in module os*), 748
P_NOWAITO (*in module os*), 748
P_OVERLAY (*in module os*), 749
P_PGID (*in module os*), 752
P_PID (*in module os*), 752
P_PIDFD (*in module os*), 752
P_WAIT (*in module os*), 749
pack() (*in module struct*), 184
pack() (*mailbox.MH method*), 1349
pack() (*struct.Struct method*), 190
pack_into() (*in module struct*), 184
pack_into() (*struct.Struct method*), 190
package, 2112, **2332**
packed (*ipaddress.IPv4Address attribute*), 1574
packed (*ipaddress.IPv6Address attribute*), 1576
packing
    binary data, 183
packing (*widgets*), 1671
PAGER, 1775
pair_content() (*in module curses*), 878
pair_number() (*in module curses*), 878
pairwise() (*in module itertools*), 426
parameter, **2332**
Parameter (*class in inspect*), 2099
ParameterizedMIMEHeader (*class in email.headerregistry*), 1298
parameters (*inspect.Signature attribute*), 2098
params (*email.headerregistry.ParameterizedMIMEHeader attribute*), 1298
ParamSpec (*class in ast*), 2194
ParamSpec (*class in typing*), 1747
ParamSpecArgs (*in module typing*), 1749
ParamSpecKwargs (*in module typing*), 1749
paramstyle (*in module sqlite3*), 558
pardir (*in module os*), 757
parent (*importlib.machinery.ModuleSpec attribute*), 2143
parent (*logging.Logger attribute*), 825
parent (*pathlib.PurePath attribute*), 461
parent (*pyclbr.Class attribute*), 2219
parent (*pyclbr.Function attribute*), 2218
parent (*urllib.request.BaseHandler attribute*), 1465
parent() (*tkinter.ttk.Treeview method*), 1700
parent_process() (*in module multiprocessing*), 984
parentNode (*xml.dom.Node attribute*), 1404
parents (*collections.ChainMap attribute*), 262
parents (*pathlib.PurePath attribute*), 461

paretovariate() (*in module random*), 396
parse() (*doctest.DocTestParser method*), 1797
parse() (*email.parser.BytesParser method*), 1282
parse() (*email.parser.Parser method*), 1282
parse() (*in module ast*), 2199
parse() (*in module xml.dom.minidom*), 1413
parse() (*in module xml.dom.pulldom*), 1418
parse() (*in module xml.etree.ElementTree*), 1391
parse() (*in module xml.sax*), 1419
parse() (*string.Formatter method*), 122
parse() (*urllib.robotparser.RobotFileParser method*), 1487
parse() (*xml.etree.ElementTree.ElementTree method*), 1396
Parse() (*xml.parsers.expat.xmlparser method*), 1433
parse() (*xml.sax.xmlreader.XMLReader method*), 1429
parse_and_bind() (*in module readline*), 177
parse_args() (*argparse.ArgumentParser method*), 810
parse_args() (*optparse.OptionParser method*), 2308
PARSE_COLNAMES (*in module sqlite3*), 557
parse_config_h() (*in module sysconfig*), 2029
PARSE_DECLTYPES (*in module sqlite3*), 557
parse_headers() (*in module http.client*), 1493
parse_intermixed_args() (*argparse.ArgumentParser method*), 822
parse_known_args() (*argparse.ArgumentParser method*), 821
parse_known_intermixed_args() (*argparse.ArgumentParser method*), 822
parse_qs() (*in module urllib.parse*), 1479
parse_qsl() (*in module urllib.parse*), 1480
parseaddr() (*in module email.utils*), 1327
parsebytes() (*email.parser.BytesParser method*), 1282
parsedate() (*in module email.utils*), 1328
parsedate_to_datetime() (*in module email.utils*), 1328
parsedate_tz() (*in module email.utils*), 1328
ParseError (*class in xml.etree.ElementTree*), 1401
ParseFile() (*xml.parsers.expat.xmlparser method*), 1433
ParseFlags() (*in module imaplib*), 1511
Parser (*class in email.parser*), 1282
parser (*pathlib.PurePath attribute*), 460
ParserCreate() (*in module xml.parsers.expat*), 1432
ParseResult (*class in urllib.parse*), 1483
ParseResultBytes (*class in urllib.parse*), 1483
parsestr() (*email.parser.Parser method*), 1282
parseString() (*in module xml.dom.minidom*), 1413
parseString() (*in module xml.dom.pulldom*), 1418
parseString() (*in module xml.sax*), 1419
parsing
    URL, 1477
ParsingError, 663
partial (*asyncio.IncompleteReadError attribute*), 1112
partial() (*imaplib.IMAP4 method*), 1514
partial() (*in module functools*), 440
partialmethod (*class in functools*), 440
parties (*asyncio.Barrier attribute*), 1102
parties (*threading.Barrier attribute*), 969
partition() (*bytearray method*), 68
partition() (*bytes method*), 68
partition() (*str method*), 56
parts (*pathlib.PurePath attribute*), 460
Pass (*class in ast*), 2182
pass_() (*poplib.POP3 method*), 1508
Paste, 1710
patch() (*in module test.support*), 1911
patch() (*in module unittest.mock*), 1856
patch.dict() (*in module unittest.mock*), 1860
patch.multiple() (*in module unittest.mock*), 1861
patch.object() (*in module unittest.mock*), 1859
patch.stopall() (*in module unittest.mock*), 1863
PATH, 480, 741, 747, 748, 758, 1035, 1443, 1977, 1978, 2112
path
    configuration file, 2112
    module search, 510, 2008, 2111
    operations, 455, 482
Path (*class in pathlib*), 467
Path (*class in zipfile*), 612
path (*http.cookiejar.Cookie attribute*), 1556
path (*http.cookies.Morsel attribute*), 1546
path (*http.server.BaseHTTPRequestHandler attribute*), 1539
path (*ImportError attribute*), 110
path (*importlib.abc.FileLoader attribute*), 2135
path (*importlib.machinery.AppleFrameworkLoader attribute*), 2144
path (*importlib.machinery.ExtensionFileLoader attribute*), 2141
path (*importlib.machinery.FileFinder attribute*), 2140
path (*importlib.machinery.SourceFileLoader attribute*), 2140
path (*importlib.machinery.SourcelessFileLoader attribute*), 2141
path (*in module sys*), 2008
path (*os.DirEntry attribute*), 724
path based finder, **2332**
Path browser, 1706
path entry, **2332**
path entry finder, **2332**
path entry hook, **2332**
path() (*in module importlib.resources*), 2152
path-like object, **2333**
path_hook() (*importlib.machinery.FileFinder class method*), 2140
path_hooks (*in module sys*), 2008

path_importer_cache (*in module sys*), 2008
path_mtime() (*importlib.abc.SourceLoader method*), 2136
path_return_ok() (*http.cookiejar.CookiePolicy method*), 1554
path_stats() (*importlib.abc.SourceLoader method*), 2135
path_stats() (*importlib.machinery.SourceFileLoader method*), 2140
pathconf() (*in module os*), 720
pathconf_names (*in module os*), 720
PathEntryFinder (*class in importlib.abc*), 2132
PathFinder (*class in importlib.machinery*), 2139
pathlib
    module, 455
PathLike (*class in os*), 692
pathname2url() (*in module urllib.request*), 1459
pathsep (*in module os*), 758
Path.stem (*in module zipfile*), 613
Path.suffix (*in module zipfile*), 613
Path.suffixes (*in module zipfile*), 613
--pattern
    unittest-discover command line option, 1807
Pattern (*class in re*), 146
Pattern (*class in typing*), 1768
pattern (*re.Pattern attribute*), 147
pattern (*re.PatternError attribute*), 146
PatternError, 146
pause() (*in module signal*), 1258
pause_reading() (*asyncio.ReadTransport method*), 1144
pause_writing() (*asyncio.BaseProtocol method*), 1148
PAX_FORMAT (*in module tarfile*), 621
pax_headers (*tarfile.TarFile attribute*), 626
pax_headers (*tarfile.TarInfo attribute*), 627
pbkdf2_hmac() (*in module hashlib*), 675
pd() (*in module turtle*), 1626
pdb
    module, 1935
Pdb (*class in pdb*), 1935, 1937
.pdbrc
    file, 1939
pdf() (*statistics.NormalDist method*), 415
peek() (*bz2.BZ2File method*), 596
peek() (*gzip.GzipFile method*), 592
peek() (*io.BufferedReader method*), 769
peek() (*lzma.LZMAFile method*), 601
peek() (*weakref.finalize method*), 300
PEM_cert_to_DER_cert() (*in module ssl*), 1209
pen() (*in module turtle*), 1627
pencolor() (*in module turtle*), 1628
pending (*ssl.MemoryBIO attribute*), 1239
pending() (*ssl.SSLSocket method*), 1221
PendingDeprecationWarning, 116
pendown() (*in module turtle*), 1626
pensize() (*in module turtle*), 1626
penup() (*in module turtle*), 1626
PEP, **2333**
PERCENT (*in module token*), 2209
PERCENTEQUAL (*in module token*), 2210
perf_counter() (*in module time*), 778
perf_counter_ns() (*in module time*), 778
Performance, 1953
perm() (*in module math*), 349
PermissionError, 116
permutations() (*in module itertools*), 427
persistence, 523
persistent
    objects, 523
persistent_id (*pickle protocol*), 532
persistent_id() (*pickle.Pickler method*), 527
persistent_load (*pickle protocol*), 532
persistent_load() (*pickle.Unpickler method*), 528
PF_CAN (*in module socket*), 1180
PF_DIVERT (*in module socket*), 1181
PF_PACKET (*in module socket*), 1182
PF_RDS (*in module socket*), 1182
pformat() (*in module pprint*), 315
pformat() (*pprint.PrettyPrinter method*), 316
pgettext() (*gettext.GNUTranslations method*), 1597
pgettext() (*gettext.NullTranslations method*), 1596
pgettext() (*in module gettext*), 1594
PGO (*in module test.support*), 1904
phase() (*in module cmath*), 355
pi (*in module cmath*), 358
pi (*in module math*), 354
pi() (*xml.etree.ElementTree.TreeBuilder method*), 1398
pickle
    module, 313, 523, 541, 544
pickle() (*in module copyreg*), 541
PickleBuffer (*class in pickle*), 528
PickleError, 526
Pickler (*class in pickle*), 526
pickletools
    module, 2248
pickletools command line option
    -a, 2249
    --annotate, 2249
    --indentlevel, 2249
    -l, 2249
    -m, 2249
    --memo, 2249
    -o, 2249
    --output, 2249
    -p, 2249
    --preamble, 2249

pickling
    objects, 523
PicklingError, 526
pid (*asyncio.subprocess.Process attribute*), 1107
pid (*multiprocessing.Process attribute*), 978
pid (*subprocess.Popen attribute*), 1042
PIDFD_NONBLOCK (*in module os*), 745
pidfd_open() (*in module os*), 745
pidfd_send_signal() (*in module signal*), 1258
PidfdChildWatcher (*class in asyncio*), 1159
PIPE (*in module subprocess*), 1032
Pipe() (*in module multiprocessing*), 981
pipe() (*in module os*), 706
pipe2() (*in module os*), 706
PIPE_BUF (*in module select*), 1244
pipe_connection_lost() (*asyncio.SubprocessProtocol method*), 1150
pipe_data_received() (*asyncio.SubprocessProtocol method*), 1150
PIPE_MAX_SIZE (*in module test.support*), 1904
pkgutil
    module, 2121
placeholder (*textwrap.TextWrapper attribute*), 173
platform
    module, 908
platform (*in module sys*), 2008
platform() (*in module platform*), 909
platlibdir (*in module sys*), 2009
PlaySound() (*in module winsound*), 2263
plist
    file, 667
plistlib
    module, 667
plock() (*in module os*), 745
PLUS (*in module token*), 2209
plus() (*decimal.Context method*), 376
PLUSEQUAL (*in module token*), 2210
pm() (*in module pdb*), 1937
POINTER() (*in module ctypes*), 947
pointer() (*in module ctypes*), 947
polar() (*in module cmath*), 356
Policy (*class in email.policy*), 1288
poll() (*in module select*), 1243
poll() (*multiprocessing.connection.Connection method*), 986
poll() (*select.devpoll method*), 1245
poll() (*select.epoll method*), 1246
poll() (*select.poll method*), 1247
poll() (*subprocess.Popen method*), 1040
PollSelector (*class in selectors*), 1252
Pool (*class in multiprocessing.pool*), 1000
pop() (*array.array method*), 296
pop() (*collections.deque method*), 269
pop() (*dict method*), 91
pop() (*frozenset method*), 88
pop() (*mailbox.Mailbox method*), 1344
pop() (*sequence method*), 47
POP3
    protocol, 1507
POP3 (*class in poplib*), 1507
POP3_SSL (*class in poplib*), 1507
pop_all() (*contextlib.ExitStack method*), 2063
POP_BLOCK (*opcode*), 2247
POP_EXCEPT (*opcode*), 2235
POP_JUMP_IF_FALSE (*opcode*), 2240
POP_JUMP_IF_NONE (*opcode*), 2240
POP_JUMP_IF_NOT_NONE (*opcode*), 2240
POP_JUMP_IF_TRUE (*opcode*), 2240
pop_source() (*shlex.shlex method*), 1659
POP_TOP (*opcode*), 2231
Popen (*class in subprocess*), 1035
popen() (*in module os*), 745, 1243
popitem() (*collections.OrderedDict method*), 277
popitem() (*dict method*), 91
popitem() (*mailbox.Mailbox method*), 1344
popleft() (*collections.deque method*), 269
poplib
    module, 1507
port (*http.cookiejar.Cookie attribute*), 1556
port_specified (*http.cookiejar.Cookie attribute*), 1557
portion, **2333**
pos (*json.JSONDecodeError attribute*), 1338
pos (*re.Match attribute*), 150
pos (*re.PatternError attribute*), 146
pos() (*in module operator*), 448
pos() (*in module turtle*), 1624
position (*xml.etree.ElementTree.ParseError attribute*), 1401
position() (*in module turtle*), 1624
positional argument, **2333**
Positions (*class in dis*), 2231
positions (*inspect.FrameInfo attribute*), 2105
positions (*inspect.Traceback attribute*), 2105
Positions.col_offset (*in module dis*), 2231
Positions.end_col_offset (*in module dis*), 2231
Positions.end_lineno (*in module dis*), 2231
Positions.lineno (*in module dis*), 2231
POSIX
    I/O control, 2270
    threads, 1062
posix
    module, 2267
POSIX Shared Memory, 1017
POSIX_FADV_DONTNEED (*in module os*), 706
POSIX_FADV_NOREUSE (*in module os*), 706
POSIX_FADV_NORMAL (*in module os*), 706
POSIX_FADV_RANDOM (*in module os*), 706

POSIX_FADV_SEQUENTIAL (*in module os*), 706
POSIX_FADV_WILLNEED (*in module os*), 706
posix_fadvise() (*in module os*), 706
posix_fallocate() (*in module os*), 706
posix_openpt() (*in module os*), 707
posix_spawn() (*in module os*), 746
POSIX_SPAWN_CLOSE (*in module os*), 746
POSIX_SPAWN_CLOSEFROM (*in module os*), 746
POSIX_SPAWN_DUP2 (*in module os*), 746
POSIX_SPAWN_OPEN (*in module os*), 746
posix_spawnp() (*in module os*), 747
PosixPath (*class in pathlib*), 467
post_handshake_auth (*ssl.SSLContext attribute*), 1229
post_mortem() (*in module pdb*), 1937
post_setup() (*venv.EnvBuilder method*), 1980
postcmd() (*cmd.Cmd method*), 1652
postloop() (*cmd.Cmd method*), 1653
Pow (*class in ast*), 2174
pow()
    built-in function, 24
pow() (*in module math*), 351
pow() (*in module operator*), 448
power() (*decimal.Context method*), 376
pp (*pdb command*), 1941
pp() (*in module pprint*), 314
pprint
    module, 314
pprint() (*in module pprint*), 315
pprint() (*pprint.PrettyPrinter method*), 316
prcal() (*in module calendar*), 258
pread() (*in module os*), 706
preadv() (*in module os*), 707
--preamble
    pickletools command line option, 2249
preamble (*email.message.EmailMessage attribute*), 1280
preamble (*email.message.Message attribute*), 1318
precmd() (*cmd.Cmd method*), 1652
prefix (*in module sys*), 2009
prefix (*xml.dom.Attr attribute*), 1409
prefix (*xml.dom.Node attribute*), 1404
prefix (*zipimport.zipimporter attribute*), 2121
PREFIXES (*in module site*), 2113
prefixlen (*ipaddress.IPv4Network attribute*), 1580
prefixlen (*ipaddress.IPv6Network attribute*), 1583
preloop() (*cmd.Cmd method*), 1653
prepare() (*graphlib.TopologicalSorter method*), 340
prepare() (*logging.handlers.QueueHandler method*), 870
prepare() (*logging.handlers.QueueListener method*), 872
prepare_class() (*in module types*), 305
prepare_input_source() (*in module xml.sax.saxutils*), 1427
PrepareProtocol (*class in sqlite3*), 575
PrettyPrinter (*class in pprint*), 315
prev() (*tkinter.ttk.Treeview method*), 1700
previousSibling (*xml.dom.Node attribute*), 1404
print()
    built-in function, 24
print() (*traceback.TracebackException method*), 2080
print_callees() (*pstats.Stats method*), 1951
print_callers() (*pstats.Stats method*), 1951
print_exc() (*in module traceback*), 2077
print_exc() (*timeit.Timer method*), 1956
print_exception() (*in module traceback*), 2077
print_help() (*argparse.ArgumentParser method*), 820
print_last() (*in module traceback*), 2077
print_stack() (*asyncio.Task method*), 1086
print_stack() (*in module traceback*), 2077
print_stats() (*profile.Profile method*), 1948
print_stats() (*pstats.Stats method*), 1950
print_tb() (*in module traceback*), 2077
print_usage() (*argparse.ArgumentParser method*), 820
print_usage() (*optparse.OptionParser method*), 2310
print_version() (*optparse.OptionParser method*), 2299
print_warning() (*in module test.support*), 1908
printable (*in module string*), 122
printdir() (*zipfile.ZipFile method*), 610
printf-style formatting, 60, 77
PRIO_DARWIN_BG (*in module os*), 695
PRIO_DARWIN_NONUI (*in module os*), 695
PRIO_DARWIN_PROCESS (*in module os*), 695
PRIO_DARWIN_THREAD (*in module os*), 695
PRIO_PGRP (*in module os*), 694
PRIO_PROCESS (*in module os*), 694
PRIO_USER (*in module os*), 694
PriorityQueue (*class in asyncio*), 1110
PriorityQueue (*class in queue*), 1054
prlimit() (*in module resource*), 2278
prmonth() (*calendar.TextCalendar method*), 255
prmonth() (*in module calendar*), 258
ProactorEventLoop (*class in asyncio*), 1135
process
    group, 693, 694
    id, 694
    id of parent, 694
    killing, 744, 745
    scheduling priority, 694, 697
    signalling, 744, 745
--process
    timeit command line option, 1956
Process (*class in multiprocessing*), 977
process() (*logging.LoggerAdapter method*), 838
process_cpu_count() (*in module os*), 757

process_exited() (*asyncio.SubprocessProtocol method*), 1150
process_request() (*socketserver.BaseServer method*), 1533
process_time() (*in module time*), 778
process_time_ns() (*in module time*), 779
process_tokens() (*in module tabnanny*), 2217
ProcessError, 980
processes, light-weight, 1062
ProcessingInstruction() (*in module xml.etree.ElementTree*), 1391
processingInstruction() (*xml.sax.handler.ContentHandler method*), 1425
ProcessingInstructionHandler() (*xml.parsers.expat.xmlparser method*), 1436
ProcessLookupError, 116
processor time, 778, 783
processor() (*in module platform*), 909
ProcessPoolExecutor (*class in concurrent.futures*), 1026
prod() (*in module math*), 349
product() (*in module itertools*), 427
profile
    module, 1947
Profile (*class in profile*), 1947
profile function, 957, 2003, 2010
profiler, 2003, 2010
profiling, deterministic, 1944
ProgrammingError, 576
Progressbar (*class in tkinter.ttk*), 1694
prompt (*cmd.Cmd attribute*), 1653
prompt_user_passwd() (*urllib.request.FancyURLopener method*), 1475
prompts, interpreter, 2010
propagate (*logging.Logger attribute*), 825
property (*built-in class*), 24
property list, 667
property() (*in module enum*), 337
property_declaration_handler (*in module xml.sax.handler*), 1422
property_dom_node (*in module xml.sax.handler*), 1422
property_lexical_handler (*in module xml.sax.handler*), 1422
property_xml_string (*in module xml.sax.handler*), 1422
property.deleter()
    built-in function, 25
property.getter()
    built-in function, 25
PropertyMock (*class in unittest.mock*), 1847
property.setter()
    built-in function, 25
prot_c() (*ftplib.FTP_TLS method*), 1506
prot_p() (*ftplib.FTP_TLS method*), 1506
proto (*socket.socket attribute*), 1199
protocol
    context management, 94
    copy, 531
    FTP, 1476, 1500
    HTTP, 1476, 1488, 1492, 1538
    IMAP4, 1510
    IMAP4_SSL, 1510
    IMAP4_stream, 1510
    iterator, 44
    POP3, 1507
    SMTP, 1517
Protocol (*class in asyncio*), 1147
Protocol (*class in typing*), 1752
protocol (*ssl.SSLContext attribute*), 1229
PROTOCOL_SSLv3 (*in module ssl*), 1212
PROTOCOL_SSLv23 (*in module ssl*), 1212
PROTOCOL_TLS (*in module ssl*), 1211
PROTOCOL_TLS_CLIENT (*in module ssl*), 1211
PROTOCOL_TLS_SERVER (*in module ssl*), 1211
PROTOCOL_TLSv1 (*in module ssl*), 1212
PROTOCOL_TLSv1_1 (*in module ssl*), 1212
PROTOCOL_TLSv1_2 (*in module ssl*), 1212
protocol_version (*http.server.BaseHTTPRequestHandler attribute*), 1540
PROTOCOL_VERSION (*imaplib.IMAP4 attribute*), 1516
ProtocolError (*class in xmlrpc.client*), 1563
provisional API, **2333**
provisional package, **2333**
proxy() (*in module weakref*), 298
proxyauth() (*imaplib.IMAP4 method*), 1514
ProxyBasicAuthHandler (*class in urllib.request*), 1461
ProxyDigestAuthHandler (*class in urllib.request*), 1462
ProxyHandler (*class in urllib.request*), 1460
ProxyType (*in module weakref*), 301
ProxyTypes (*in module weakref*), 301
pryear() (*calendar.TextCalendar method*), 255
ps1 (*in module sys*), 2010
ps2 (*in module sys*), 2010
pstats
    module, 1949
pstdev() (*in module statistics*), 409
pthread_getcpuclockid() (*in module time*), 776
pthread_kill() (*in module signal*), 1258
pthread_sigmask() (*in module signal*), 1259
pthreads, 1062
pthreads (*sys._emscripten_info attribute*), 1995
ptsname() (*in module os*), 707
pty
    module, 706, 2272

pu() (*in module turtle*), 1626
publicId (*xml.dom.DocumentType attribute*), 1406
PullDom (*class in xml.dom.pulldom*), 1418
punctuation (*in module string*), 122
punctuation_chars (*shlex.shlex attribute*), 1660
PurePath (*class in pathlib*), 457
PurePosixPath (*class in pathlib*), 458
PureWindowsPath (*class in pathlib*), 458
purge() (*in module re*), 145
Purpose.CLIENT_AUTH (*in module ssl*), 1216
Purpose.SERVER_AUTH (*in module ssl*), 1216
push() (*code.InteractiveConsole method*), 2117
push() (*contextlib.ExitStack method*), 2062
push_async_callback() (*contextlib.AsyncExitStack method*), 2063
push_async_exit() (*contextlib.AsyncExitStack method*), 2063
PUSH_EXC_INFO (*opcode*), 2235
PUSH_NULL (*opcode*), 2243
push_source() (*shlex.shlex method*), 1658
push_token() (*shlex.shlex method*), 1658
put() (*asyncio.Queue method*), 1109
put() (*multiprocessing.Queue method*), 982
put() (*multiprocessing.SimpleQueue method*), 983
put() (*queue.Queue method*), 1055
put() (*queue.SimpleQueue method*), 1057
put_nowait() (*asyncio.Queue method*), 1109
put_nowait() (*multiprocessing.Queue method*), 982
put_nowait() (*queue.Queue method*), 1055
put_nowait() (*queue.SimpleQueue method*), 1057
putch() (*in module msvcrt*), 2252
putenv() (*in module os*), 695
putheader() (*http.client.HTTPConnection method*), 1496
putp() (*in module curses*), 878
putrequest() (*http.client.HTTPConnection method*), 1496
putwch() (*in module msvcrt*), 2252
putwin() (*curses.window method*), 886
pvariance() (*in module statistics*), 410
pwd
    module, 483, 2268
pwd() (*ftplib.FTP method*), 1504
pwrite() (*in module os*), 708
pwritev() (*in module os*), 708
py_compile
    module, 2219
Py_DEBUG (*in module test.support*), 1904
py_object (*class in ctypes*), 952
PY_RESUME (*monitoring event*), 2019
PY_RETURN (*monitoring event*), 2019
PY_START (*monitoring event*), 2019
PY_THROW (*monitoring event*), 2019
PY_UNWIND (*monitoring event*), 2019
PY_YIELD (*monitoring event*), 2019
pycache_prefix (*in module sys*), 1995
PyCF_ALLOW_TOP_LEVEL_AWAIT (*in module ast*), 2203
PyCF_ONLY_AST (*in module ast*), 2203
PyCF_OPTIMIZED_AST (*in module ast*), 2203
PyCF_TYPE_COMMENTS (*in module ast*), 2203
PycInvalidationMode (*class in py_compile*), 2220
pyclbr
    module, 2217
PyCompileError, 2219
PyDLL (*class in ctypes*), 940
pydoc
    module, 1774
pyexpat
    module, 1432
PYFUNCTYPE() (*in module ctypes*), 944
--python
    zipapp command line option, 1986
Python 3000, **2333**
Python Editor, 1706
Python Enhancement Proposals
    PEP 1, 2333
    PEP 8, 27
    PEP 205, 301
    PEP 227, 2085
    PEP 235, 2129
    PEP 236, 2086
    PEP 237, 62, 79
    PEP 238, 2085, 2326
    PEP 246, 575
    PEP 249, 553, 556, 569, 575, 578, 585, 586
    PEP 255, 2085
    PEP 263, 2129, 2213
    PEP 273, 2119
    PEP 278, 2336
    PEP 282, 517, 844
    PEP 292, 130
    PEP 302, 32, 510, 2008, 2119, 21222124, 2126, 2129, 21322134, 2330
    PEP 305, 637
    PEP 307, 525
    PEP 324, 1031
    PEP 328, 32, 2085, 2129
    PEP 338, 2128
    PEP 342, 283
    PEP 343, 2067, 2085, 2324
    PEP 362, 2102, 2322, 2332
    PEP 366, 2128, 2129
    PEP 370, 2114
    PEP 378, 126
    PEP 380#use-of-stopiteration-to-return-values, 2021
    PEP 383, 194, 1175

PEP 387, 116, 117
PEP 393, 201, 2007
PEP 405, 1975
PEP 411, 2004, 2012, 2013, 2333
PEP 412, 437
PEP 420, 2129, 2331, 2333
PEP 421, 2006
PEP 428, 456
PEP 434, 1718
PEP 442, 2089
PEP 443, 2327
PEP 451, 2007, 2122, 21272129
PEP 453, 1973
PEP 461, 79
PEP 468, 277
PEP 475, 23, 116, 704, 709, 711, 751, 779, 1192, 1193, 11951198, 12441247, 1252, 1261
PEP 479, 112, 2085
PEP 483, 2327
PEP 484, 99, 1722, 1730, 1744, 1761, 2170, 2199, 2203, 2321, 2326, 2327, 2336
PEP 485, 348, 358
PEP 488, 1919, 2129, 2144, 2145, 2219
PEP 489, 2129, 2139, 2141, 2142, 2146
PEP 492, 285, 2109, 2110, 2322, 2324
PEP 495, 249
PEP 498, 2325
PEP 506, 685
PEP 515, 126, 389
PEP 519, 2333
PEP 524, 759
PEP 525, 285, 2004, 2012, 2110, 2322
PEP 526, 1736, 1751, 2043, 2051, 2199, 2203, 2321, 2336
PEP 529, 716, 2002, 2014
PEP 538, 1608
PEP 540, 690, 1608
PEP 544, 1730, 1752
PEP 552, 2129, 2220
PEP 557, 2043
PEP 560, 306
PEP 563, 1766, 2085
PEP 565, 116
PEP 566, 2158
PEP 567, 1058, 1115, 1116, 1140
PEP 574, 525, 538
PEP 578, 1923, 1991
PEP 584, 263, 272, 277, 299, 310, 692
PEP 585, 99, 281, 309, 17651774, 2327
PEP 586, 1736
PEP 589, 1757
PEP 591, 1737, 1762
PEP 593, 1740, 1764
PEP 597, 761
PEP 604, 101
PEP 610, 2160
PEP 612, 1723, 1728, 1736, 1749, 1772
PEP 613, 1734
PEP 615, 248
PEP 626, 2229
PEP 644, 1204
PEP 646, 1747
PEP 647, 1741
PEP 649, 2085
PEP 655, 1737, 1757
PEP 667, 19, 1937
PEP 673, 1733
PEP 675, 1732
PEP 681, 1761
PEP 682, 125
PEP 686, 691, 761
PEP 688, 285
PEP 692, 1742
PEP 695, 1731, 1744, 1745, 1747, 1749, 1774
PEP 698, 1763
PEP 702, 2043
PEP 703, 2326, 2327
PEP 705, 1737
PEP 706, 629
PEP 709, 19
PEP 742, 1741
PEP 3101, 122
PEP 3105, 2085
PEP 3112, 2085
PEP 3115, 306
PEP 3116, 2336
PEP 3118, 81
PEP 3119, 286, 2070
PEP 3120, 2130
PEP 3134, 108
PEP 3141, 343, 2070
PEP 3147, 1919, 2127, 2130, 2144, 2145, 2219, 2220, 22222224
PEP 3148, 1030
PEP 3149, 1991
PEP 3151, 116, 1178, 1242, 2277
PEP 3154, 525
PEP 3155, 2334
PEP 3333, 14461451, 1454, 1455
python_branch() (*in module platform*), 909
python_build() (*in module platform*), 909
python_compiler() (*in module platform*), 909
PYTHON_CPU_COUNT, 757, 984
PYTHON_DOM, 1402
PYTHON_GIL, 2327
python_implementation() (*in module platform*), 909

python_is_optimized() (*in module test.support*), 1906
python_revision() (*in module platform*), 909
python_version() (*in module platform*), 909
python_version_tuple() (*in module platform*), 909
PYTHONASYNCIODEBUG, 1130, 1171, 1776
PYTHONBREAKPOINT, 7, 1994
PYTHONCASEOK, 32
PYTHONCOERCECLOCALE, 691
PYTHONDEVMODE, 1775
PYTHONDONTWRITEBYTECODE, 1995
PYTHONFAULTHANDLER, 1776, 1933
PythonFinalizationError, 112
PYTHONHOME, 1914, 2162, 2163
Pythonic, **2333**
PYTHONINTMAXSTRDIGITS, 105, 2006
PYTHONIOENCODING, 690, 2014
PYTHONLEGACYWINDOWSFSENCODING, 2013, 2014
PYTHONLEGACYWINDOWSSTDIO, 2014
PYTHONMALLOC, 1776
python--m-py_compile command line option
    -, 2220
    <file>, 2220
    -q, 2220
    --quiet, 2220
python--m-sqlite3-[-h]-[-v]-[filename]-[sql] command line option
    -h, 577
    --help, 577
    -v, 577
    --version, 577
PYTHONNOUSERSITE, 2113
PYTHONPATH, 1914, 2008, 2162
PYTHONPLATLIBDIR, 2162
PYTHONPYCACHEPREFIX, 1995
PYTHONSAFEPATH, 2008, 2319
PYTHONSTARTUP, 180, 1066, 1714, 2006, 2113
PYTHONTRACEMALLOC, 1962, 1968
PYTHONTZPATH, 254
PYTHONUNBUFFERED, 2014
PYTHONUSERBASE, 2113, 2114
PYTHONUSERSITE, 1914
PYTHONUTF8, 691, 2014
PYTHONWARNDEFAULTENCODING, 761
PYTHONWARNINGS, 1776, 2037, 2038
PyZipFile (*class in zipfile*), 613

Q

-q
    compileall command line option, 2221
    python--m-py_compile command line option, 2220
qiflush() (*in module curses*), 878
QName (*class in xml.etree.ElementTree*), 1397
qsize() (*asyncio.Queue method*), 1109
qsize() (*multiprocessing.Queue method*), 982
qsize() (*queue.Queue method*), 1055
qsize() (*queue.SimpleQueue method*), 1057
qualified name, **2334**
quantiles() (*in module statistics*), 411
quantiles() (*statistics.NormalDist method*), 415
quantize() (*decimal.Context method*), 377
quantize() (*decimal.Decimal method*), 369
QueryInfoKey() (*in module winreg*), 2257
QueryReflectionKey() (*in module winreg*), 2259
QueryValue() (*in module winreg*), 2258
QueryValueEx() (*in module winreg*), 2258
QUESTION (*in module tkinter.messagebox*), 1684
queue
    module, 1054
Queue (*class in asyncio*), 1108
Queue (*class in multiprocessing*), 981
Queue (*class in queue*), 1054
queue (*sched.scheduler attribute*), 1054
Queue() (*multiprocessing.managers.SyncManager method*), 995
QueueEmpty, 1110
QueueFull, 1110
QueueHandler (*class in logging.handlers*), 870
QueueListener (*class in logging.handlers*), 871
QueueShutDown, 1110
quick_ratio() (*difflib.SequenceMatcher method*), 163
--quiet
    python--m-py_compile command line option, 2220
quiet (*sys.flags attribute*), 1998
quit (*built-in variable*), 34
quit (*pdb command*), 1943
quit() (*ftplib.FTP method*), 1504
quit() (*poplib.POP3 method*), 1509
quit() (*smtplib.SMTP method*), 1523
quit() (*tkinter.filedialog.FileDialog method*), 1681
quitting (*bdb.Bdb attribute*), 1931
quopri
    module, 1371
quote() (*in module email.utils*), 1327
quote() (*in module shlex*), 1657
quote() (*in module urllib.parse*), 1484
QUOTE_ALL (*in module csv*), 641
quote_from_bytes() (*in module urllib.parse*), 1484
QUOTE_MINIMAL (*in module csv*), 641
QUOTE_NONE (*in module csv*), 641
QUOTE_NONNUMERIC (*in module csv*), 641
QUOTE_NOTNULL (*in module csv*), 641

quote_plus() (*in module urllib.parse*), 1484
QUOTE_STRINGS (*in module csv*), 641
quoteattr() (*in module xml.sax.saxutils*), 1427
quotechar (*csv.Dialect attribute*), 642
quoted-printable
    encoding, 1371
quotes (*shlex.shlex attribute*), 1659
quoting (*csv.Dialect attribute*), 642

R

-R
    trace command line option, 1960
-r
    compileall command line option, 2222
    timeit command line option, 1956
    trace command line option, 1959
R_OK (*in module os*), 714
radians() (*in module math*), 353
radians() (*in module turtle*), 1626
radix (*sys.float_info attribute*), 2000
radix() (*decimal.Context method*), 377
radix() (*decimal.Decimal method*), 369
RADIXCHAR (*in module locale*), 1605
raise
    statement, 107
Raise (*class in ast*), 2181
RAISE (*monitoring event*), 2019
raise_on_defect (*email.policy.Policy attribute*), 1289
raise_signal() (*in module signal*), 1258
RAISE_VARARGS (*opcode*), 2242
raiseExceptions (*in module logging*), 843
RAND_add() (*in module ssl*), 1208
RAND_bytes() (*in module ssl*), 1208
RAND_status() (*in module ssl*), 1208
randbelow() (*in module secrets*), 685
randbits() (*in module secrets*), 686
randbytes() (*in module random*), 393
randint() (*in module random*), 393
random
    module, 392
Random (*class in random*), 397
random command line option
    -c, 401
    --choice, 401
    -f, 401
    --float, 401
    -h, 401
    --help, 401
    -i, 401
    --integer, 401
random() (*in module random*), 395
random() (*random.Random method*), 397
randrange() (*in module random*), 393
range
    object, 49
range (*built-in class*), 49
RARROW (*in module token*), 2211
ratio() (*difflib.SequenceMatcher method*), 163
Rational (*class in numbers*), 344
raw (*io.BufferedIOBase attribute*), 766
raw() (*in module curses*), 878
raw() (*pickle.PickleBuffer method*), 529
raw_data_manager (*in module email.contentmanager*), 1302
raw_decode() (*json.JSONDecoder method*), 1336
raw_input() (*code.InteractiveConsole method*), 2117
RawArray() (*in module multiprocessing.sharedctypes*), 991
RawConfigParser (*class in configparser*), 662
RawDescriptionHelpFormatter (*class in argparse*), 794
RawIOBase (*class in io*), 765
RawPen (*class in turtle*), 1644
RawTextHelpFormatter (*class in argparse*), 794
RawTurtle (*class in turtle*), 1644
RawValue() (*in module multiprocessing.sharedctypes*), 991
RBRACE (*in module token*), 2209
re
    module, 52, 132, 508
re (*re.Match attribute*), 150
READ (*inspect.BufferFlags attribute*), 2111
read() (*asyncio.StreamReader method*), 1091
read() (*codecs.StreamReader method*), 199
read() (*configparser.ConfigParser method*), 660
read() (*http.client.HTTPResponse method*), 1497
read() (*imaplib.IMAP4 method*), 1514
read() (*in module os*), 709
read() (*io.BufferedIOBase method*), 766
read() (*io.BufferedReader method*), 769
read() (*io.RawIOBase method*), 765
read() (*io.TextIOBase method*), 771
read() (*mimetypes.MimeTypes method*), 1365
read() (*mmap.mmap method*), 1266
read() (*sqlite3.Blob method*), 574
read() (*ssl.MemoryBIO method*), 1239
read() (*ssl.SSLSocket method*), 1218
read() (*urllib.robotparser.RobotFileParser method*), 1487
read() (*zipfile.ZipFile method*), 610
read1() (*bz2.BZ2File method*), 596
read1() (*io.BufferedIOBase method*), 767
read1() (*io.BufferedReader method*), 769
read1() (*io.BytesIO method*), 768
read_binary() (*in module importlib.resources*), 2152
read_byte() (*mmap.mmap method*), 1266
read_bytes() (*importlib.abc.Traversable method*), 2138

read_bytes() (*importlib.resources.abc.Traversable method*), 2155
read_bytes() (*pathlib.Path method*), 472
read_bytes() (*zipfile.Path method*), 613
read_dict() (*configparser.ConfigParser method*), 660
read_environ() (*in module wsgiref.handlers*), 1455
read_events() (*xml.etree.ElementTree.XMLPullParser method*), 1400
read_file() (*configparser.ConfigParser method*), 660
read_history_file() (*in module readline*), 178
read_init_file() (*in module readline*), 177
read_mime_types() (*in module mimetypes*), 1363
read_string() (*configparser.ConfigParser method*), 660
read_text() (*importlib.abc.Traversable method*), 2138
read_text() (*importlib.resources.abc.Traversable method*), 2155
read_text() (*in module importlib.resources*), 2152
read_text() (*pathlib.Path method*), 472
read_text() (*zipfile.Path method*), 613
read_token() (*shlex.shlex method*), 1658
read_windows_registry() (*mimetypes.MimeTypes method*), 1365
READABLE (*in module _tkinter*), 1677
readable() (*bz2.BZ2File method*), 596
readable() (*io.IOBase method*), 764
readall() (*io.RawIOBase method*), 766
reader() (*in module csv*), 637
ReadError, 620
readexactly() (*asyncio.StreamReader method*), 1091
readfp() (*mimetypes.MimeTypes method*), 1365
readframes() (*wave.Wave_read method*), 1590
readinto() (*bz2.BZ2File method*), 596
readinto() (*http.client.HTTPResponse method*), 1497
readinto() (*io.BufferedIOBase method*), 767
readinto() (*io.RawIOBase method*), 766
readinto1() (*io.BufferedIOBase method*), 767
readinto1() (*io.BytesIO method*), 769
readline
    module, 177
readline() (*asyncio.StreamReader method*), 1091
readline() (*codecs.StreamReader method*), 200
readline() (*imaplib.IMAP4 method*), 1514
readline() (*io.IOBase method*), 764
readline() (*io.TextIOBase method*), 771
readline() (*mmap.mmap method*), 1266
readlines() (*codecs.StreamReader method*), 200
readlines() (*io.IOBase method*), 764
readlink() (*in module os*), 721
readlink() (*pathlib.Path method*), 470
readmodule() (*in module pyclbr*), 2217
readmodule_ex() (*in module pyclbr*), 2217
ReadOnly (*in module typing*), 1737
readonly (*memoryview attribute*), 85
ReadTransport (*class in asyncio*), 1142
readuntil() (*asyncio.StreamReader method*), 1091
readv() (*in module os*), 710
ready() (*multiprocessing.pool.AsyncResult method*), 1002
Real (*class in numbers*), 343
real (*numbers.Complex attribute*), 343
real_max_memuse (*in module test.support*), 1905
real_quick_ratio() (*difflib.SequenceMatcher method*), 163
realpath() (*in module os.path*), 486
REALTIME_PRIORITY_CLASS (*in module subprocess*), 1045
reap_children() (*in module test.support*), 1910
reap_threads() (*in module test.support.threading_helper*), 1915
reason (*http.client.HTTPResponse attribute*), 1498
reason (*ssl.SSLError attribute*), 1207
reason (*UnicodeError attribute*), 114
reason (*urllib.error.HTTPError attribute*), 1486
reason (*urllib.error.URLError attribute*), 1486
reattach() (*tkinter.ttk.Treeview method*), 1700
recent() (*imaplib.IMAP4 method*), 1514
reconfigure() (*io.TextIOWrapper method*), 772
record_original_stdout() (*in module test.support*), 1907
RECORDS (*inspect.BufferFlags attribute*), 2110
records (*unittest.TestCase attribute*), 1818
RECORDS_RO (*inspect.BufferFlags attribute*), 2110
rect() (*in module cmath*), 356
rectangle() (*in module curses.textpad*), 901
RecursionError, 112
recursive_repr() (*in module reprlib*), 320
recv() (*multiprocessing.connection.Connection method*), 986
recv() (*socket.socket method*), 1194
recv_bytes() (*multiprocessing.connection.Connection method*), 986
recv_bytes_into() (*multiprocessing.connection.Connection method*), 987
recv_fds() (*in module socket*), 1191
recv_into() (*socket.socket method*), 1197
recvfrom() (*socket.socket method*), 1195
recvfrom_into() (*socket.socket method*), 1196
recvmsg() (*socket.socket method*), 1195
recvmsg_into() (*socket.socket method*), 1196
redirect_request() (*urllib.request.HTTPRedirectHandler method*), 1466
redirect_stderr() (*in module contextlib*), 2059
redirect_stdout() (*in module contextlib*), 2059
redisplay() (*in module readline*), 178
redrawln() (*curses.window method*), 886
redrawwin() (*curses.window method*), 886

reduce() (*in module functools*), 441
reducer_override() (*pickle.Pickler method*), 527
ref (*class in weakref*), 297
refcount_test() (*in module test.support*), 1909
reference count, **2334**
ReferenceError, 112
ReferenceType (*in module weakref*), 300
refold_source (*email.policy.EmailPolicy attribute*), 1291
refresh() (*curses.window method*), 886
REG_BINARY (*in module winreg*), 2261
REG_DWORD (*in module winreg*), 2261
REG_DWORD_BIG_ENDIAN (*in module winreg*), 2261
REG_DWORD_LITTLE_ENDIAN (*in module winreg*), 2261
REG_EXPAND_SZ (*in module winreg*), 2262
REG_FULL_RESOURCE_DESCRIPTOR (*in module winreg*), 2262
REG_LINK (*in module winreg*), 2262
REG_MULTI_SZ (*in module winreg*), 2262
REG_NONE (*in module winreg*), 2262
REG_QWORD (*in module winreg*), 2262
REG_QWORD_LITTLE_ENDIAN (*in module winreg*), 2262
REG_RESOURCE_LIST (*in module winreg*), 2262
REG_RESOURCE_REQUIREMENTS_LIST (*in module winreg*), 2262
REG_SZ (*in module winreg*), 2262
RegexFlag (*class in re*), 140
register() (*abc.ABCMeta method*), 2070
register() (*in module atexit*), 2075
register() (*in module codecs*), 192
register() (*in module faulthandler*), 1934
register() (*in module webbrowser*), 1444
register() (*multiprocessing.managers.BaseManager method*), 994
register() (*select.devpoll method*), 1244
register() (*select.epoll method*), 1246
register() (*selectors.BaseSelector method*), 1251
register() (*select.poll method*), 1246
register_adapter() (*in module sqlite3*), 557
register_archive_format() (*in module shutil*), 518
register_at_fork() (*in module os*), 747
register_callback() (*in module sys.monitoring*), 2022
register_converter() (*in module sqlite3*), 557
register_defect() (*email.policy.Policy method*), 1289
register_dialect() (*in module csv*), 638
register_error() (*in module codecs*), 195
register_function() (*xmlrpc.server.CGIXMLRPCRequestHandler method*), 1570
register_function() (*xmlrpc.server.SimpleXMLRPCServer method*), 1567
register_instance() (*xmlrpc.server.CGIXMLRPCRequestHandler method*), 1570
register_instance() (*xmlrpc.server.SimpleXMLRPCServer method*), 1567
register_introspection_functions() (*xmlrpc.server.CGIXMLRPCRequestHandler method*), 1571
register_introspection_functions() (*xmlrpc.server.SimpleXMLRPCServer method*), 1567
register_multicall_functions() (*xmlrpc.server.CGIXMLRPCRequestHandler method*), 1571
register_multicall_functions() (*xmlrpc.server.SimpleXMLRPCServer method*), 1568
register_namespace() (*in module xml.etree.ElementTree*), 1391
register_optionflag() (*in module doctest*), 1788
register_shape() (*in module turtle*), 1642
register_unpack_format() (*in module shutil*), 519
registerDOMImplementation() (*in module xml.dom*), 1402
registerResult() (*in module unittest*), 1835
REGTYPE (*in module tarfile*), 621
regular package, **2334**
relative
    URL, 1477
relative_to() (*pathlib.PurePath method*), 465
release() (*_thread.lock method*), 1063
release() (*asyncio.Condition method*), 1100
release() (*asyncio.Lock method*), 1098
release() (*asyncio.Semaphore method*), 1101
release() (*in module platform*), 910
release() (*logging.Handler method*), 830
release() (*memoryview method*), 82
release() (*multiprocessing.Lock method*), 989
release() (*multiprocessing.RLock method*), 989
release() (*pickle.PickleBuffer method*), 529
release() (*threading.Condition method*), 964
release() (*threading.Lock method*), 962
release() (*threading.RLock method*), 963
release() (*threading.Semaphore method*), 966
reload() (*in module importlib*), 2130
relpath() (*in module os.path*), 486
remainder() (*decimal.Context method*), 377
remainder() (*in module math*), 349
remainder_near() (*decimal.Context method*), 377
remainder_near() (*decimal.Decimal method*), 369

RemoteDisconnected, 1494
remove() (*array.array method*), 296
remove() (*collections.deque method*), 269
remove() (*frozenset method*), 88
remove() (*in module os*), 721
remove() (*mailbox.Mailbox method*), 1342
remove() (*mailbox.MH method*), 1349
remove() (*sequence method*), 47
remove() (*xml.etree.ElementTree.Element method*), 1395
remove_child_handler() (*asyncio.AbstractChildWatcher method*), 1158
remove_done_callback() (*asyncio.Future method*), 1140
remove_done_callback() (*asyncio.Task method*), 1085
remove_flag() (*mailbox.Maildir method*), 1346
remove_flag() (*mailbox.MaildirMessage method*), 1353
remove_flag() (*mailbox.mboxMessage method*), 1355
remove_flag() (*mailbox.MMDFMessage method*), 1359
remove_folder() (*mailbox.Maildir method*), 1345
remove_folder() (*mailbox.MH method*), 1349
remove_header() (*urllib.request.Request method*), 1463
remove_history_item() (*in module readline*), 178
remove_label() (*mailbox.BabylMessage method*), 1357
remove_option() (*configparser.ConfigParser method*), 662
remove_option() (*optparse.OptionParser method*), 2309
remove_reader() (*asyncio.loop method*), 1124
remove_section() (*configparser.ConfigParser method*), 662
remove_sequence() (*mailbox.MHMessage method*), 1356
remove_signal_handler() (*asyncio.loop method*), 1127
remove_writer() (*asyncio.loop method*), 1124
removeAttribute() (*xml.dom.Element method*), 1408
removeAttributeNode() (*xml.dom.Element method*), 1408
removeAttributeNS() (*xml.dom.Element method*), 1408
removeChild() (*xml.dom.Node method*), 1405
removedirs() (*in module os*), 721
removeFilter() (*logging.Handler method*), 831
removeFilter() (*logging.Logger method*), 828
removeHandler() (*in module unittest*), 1835
removeHandler() (*logging.Logger method*), 829
removeprefix() (*bytearray method*), 66
removeprefix() (*bytes method*), 66
removeprefix() (*str method*), 56
removeResult() (*in module unittest*), 1835
removesuffix() (*bytearray method*), 66
removesuffix() (*bytes method*), 66
removesuffix() (*str method*), 56
removexattr() (*in module os*), 739
rename() (*ftplib.FTP method*), 1504
rename() (*imaplib.IMAP4 method*), 1514
rename() (*in module os*), 721
rename() (*pathlib.Path method*), 477
renames() (*in module os*), 722
reopenIfNeeded() (*logging.handlers.WatchedFileHandler method*), 859
reorganize() (*dbm.gnu.gdbm method*), 550
--repeat
    timeit command line option, 1956
repeat() (*in module itertools*), 428
repeat() (*in module timeit*), 1954
repeat() (*timeit.Timer method*), 1955
repetition
    operation, 45
REPL, **2334**
replace
    error handler's name, 194
replace() (*bytearray method*), 68
replace() (*bytes method*), 68
replace() (*curses.panel.Panel method*), 908
replace() (*datetime.date method*), 217
replace() (*datetime.datetime method*), 226
replace() (*datetime.time method*), 234
replace() (*in module copy*), 312
replace() (*in module dataclasses*), 2049
replace() (*in module os*), 722
replace() (*inspect.Parameter method*), 2100
replace() (*inspect.Signature method*), 2098
replace() (*pathlib.Path method*), 477
replace() (*str method*), 56
replace() (*tarfile.TarInfo method*), 628
replace_errors() (*in module codecs*), 195
replace_header() (*email.message.EmailMessage method*), 1275
replace_header() (*email.message.Message method*), 1315
replace_history_item() (*in module readline*), 179
replace_whitespace (*textwrap.TextWrapper attribute*), 171
replaceChild() (*xml.dom.Node method*), 1405
ReplacePackage() (*in module modulefinder*), 2125
--report
    trace command line option, 1959
report() (*filecmp.dircmp method*), 498

report() (*modulefinder.ModuleFinder method*), 2125
REPORT_CDIFF (*in module doctest*), 1787
REPORT_ERRMODE (*in module msvcrt*), 2253
report_failure() (*doctest.DocTestRunner method*), 1798
report_full_closure() (*filecmp.dircmp method*), 499
REPORT_NDIFF (*in module doctest*), 1787
REPORT_ONLY_FIRST_FAILURE (*in module doctest*), 1787
report_partial_closure() (*filecmp.dircmp method*), 498
report_start() (*doctest.DocTestRunner method*), 1798
report_success() (*doctest.DocTestRunner method*), 1798
REPORT_UDIFF (*in module doctest*), 1787
report_unexpected_exception() (*doctest.DocTestRunner method*), 1798
REPORTING_FLAGS (*in module doctest*), 1788
Repr (*class in reprlib*), 320
repr()
    built-in function, 26
repr() (*in module reprlib*), 320
repr() (*reprlib.Repr method*), 322
repr1() (*reprlib.Repr method*), 322
ReprEnum (*class in enum*), 334
reprlib
    module, 320
Request (*class in urllib.request*), 1459
request (*socketserver.BaseRequestHandler attribute*), 1534
request() (*http.client.HTTPConnection method*), 1494
request_queue_size (*socketserver.BaseServer attribute*), 1532
request_rate() (*urllib.robotparser.RobotFileParser method*), 1487
request_uri() (*in module wsgiref.util*), 1446
request_version (*http.server.BaseHTTPRequestHandler attribute*), 1539
RequestHandlerClass (*socketserver.BaseServer attribute*), 1532
requestline (*http.server.BaseHTTPRequestHandler attribute*), 1539
Required (*in module typing*), 1737
requires() (*in module test.support*), 1906
requires_bz2() (*in module test.support*), 1909
requires_docstrings() (*in module test.support*), 1909
requires_freebsd_version() (*in module test.support*), 1909
requires_gil_enabled() (*in module test.support*), 1909
requires_gzip() (*in module test.support*), 1909
requires_IEEE_754() (*in module test.support*), 1909
requires_limited_api() (*in module test.support*), 1909
requires_linux_version() (*in module test.support*), 1909
requires_lzma() (*in module test.support*), 1909
requires_mac_version() (*in module test.support*), 1909
requires_resource() (*in module test.support*), 1909
requires_zlib() (*in module test.support*), 1909
RERAISE (*monitoring event*), 2019
RERAISE (*opcode*), 2235
reschedule() (*asyncio.Timeout method*), 1079
reserved (*zipfile.ZipInfo attribute*), 615
RESERVED_FUTURE (*in module uuid*), 1527
RESERVED_MICROSOFT (*in module uuid*), 1527
RESERVED_NCS (*in module uuid*), 1527
reset() (*asyncio.Barrier method*), 1102
reset() (*bdb.Bdb method*), 1929
reset() (*codecs.IncrementalDecoder method*), 198
reset() (*codecs.IncrementalEncoder method*), 197
reset() (*codecs.StreamReader method*), 200
reset() (*codecs.StreamWriter method*), 199
reset() (*contextvars.ContextVar method*), 1058
reset() (*html.parser.HTMLParser method*), 1375
reset() (*in module turtle*), 1630
reset() (*threading.Barrier method*), 969
reset() (*xml.dom.pulldom.DOMEventStream method*), 1419
reset() (*xml.sax.xmlreader.IncrementalParser method*), 1430
reset_mock() (*unittest.mock.AsyncMock method*), 1851
reset_mock() (*unittest.mock.Mock method*), 1841
reset_peak() (*in module tracemalloc*), 1967
reset_prog_mode() (*in module curses*), 878
reset_shell_mode() (*in module curses*), 878
reset_tzpath() (*in module zoneinfo*), 253
resetbuffer() (*code.InteractiveConsole method*), 2117
resetscreen() (*in module turtle*), 1638
resetty() (*in module curses*), 878
resetwarnings() (*in module warnings*), 2042
resize() (*curses.window method*), 887
resize() (*in module ctypes*), 947
resize() (*mmap.mmap method*), 1266
resize_term() (*in module curses*), 878
resizemode() (*in module turtle*), 1631
resizeterm() (*in module curses*), 879
resolution (*datetime.date attribute*), 216
resolution (*datetime.datetime attribute*), 224
resolution (*datetime.time attribute*), 232
resolution (*datetime.timedelta attribute*), 213
resolve() (*pathlib.Path method*), 469

resolve_bases() (*in module types*), 306
resolve_name() (*in module importlib.util*), 2145
resolve_name() (*in module pkgutil*), 2124
resolveEntity() (*xml.sax.handler.EntityResolver method*), 1425
resource
    module, 2277
resource_path() (*importlib.abc.ResourceReader method*), 2137
resource_path() (*importlib.resources.abc.ResourceReader method*), 2153
ResourceDenied, 1903
ResourceLoader (*class in importlib.abc*), 2133
ResourceReader (*class in importlib.abc*), 2136
ResourceReader (*class in importlib.resources.abc*), 2153
ResourceWarning, 117
response() (*imaplib.IMAP4 method*), 1514
ResponseNotReady, 1494
responses (*http.server.BaseHTTPRequestHandler attribute*), 1540
responses (*in module http.client*), 1494
restart (*pdb command*), 1943
restart_events() (*in module sys.monitoring*), 2021
restore() (*in module difflib*), 159
restore() (*test.support.SaveSignals method*), 1912
restype (*ctypes._FuncPtr attribute*), 942
result() (*asyncio.Future method*), 1139
result() (*asyncio.Task method*), 1085
result() (*concurrent.futures.Future method*), 1028
results() (*trace.Trace method*), 1961
RESUME (*opcode*), 2244
resume_reading() (*asyncio.ReadTransport method*), 1144
resume_writing() (*asyncio.BaseProtocol method*), 1148
retr() (*poplib.POP3 method*), 1509
retrbinary() (*ftplib.FTP method*), 1502
retrieve() (*urllib.request.URLopener method*), 1474
retrlines() (*ftplib.FTP method*), 1503
RETRY (*in module tkinter.messagebox*), 1683
RETRYCANCEL (*in module tkinter.messagebox*), 1684
Return (*class in ast*), 2196
return (*pdb command*), 1940
return_annotation (*inspect.Signature attribute*), 2098
RETURN_CONST (*opcode*), 2235
RETURN_GENERATOR (*opcode*), 2245
return_ok() (*http.cookiejar.CookiePolicy method*), 1553
RETURN_VALUE (*opcode*), 2235
return_value (*unittest.mock.Mock attribute*), 1842
returncode (*asyncio.subprocess.Process attribute*), 1107
returncode (*subprocess.CalledProcessError attribute*), 1033
returncode (*subprocess.CompletedProcess attribute*), 1032
returncode (*subprocess.Popen attribute*), 1042
retval (*pdb command*), 1943
reveal_type() (*in module typing*), 1759
reverse() (*array.array method*), 296
reverse() (*collections.deque method*), 269
reverse() (*sequence method*), 47
reverse_order() (*pstats.Stats method*), 1950
reverse_pointer (*ipaddress.IPv4Address attribute*), 1574
reverse_pointer (*ipaddress.IPv6Address attribute*), 1576
reversed()
    built-in function, 26
Reversible (*class in collections.abc*), 283
Reversible (*class in typing*), 1773
revert() (*http.cookiejar.FileCookieJar method*), 1552
rewind() (*wave.Wave_read method*), 1590
RFC
    RFC 821, 1517, 1519
    RFC 822, 781, 1304, 1321, 1496, 1521, 1522, 1524, 1597
    RFC 959, 1500
    RFC 1123, 781
    RFC 1321, 671
    RFC 1422, 1232, 1241
    RFC 1521, 1369, 1371
    RFC 1522, 1370, 1371
    RFC 1730, 1510
    RFC 1738, 1486
    RFC 1750, 1208
    RFC 1766, 1606
    RFC 1808, 1477, 1478, 1486
    RFC 1869, 1517, 1519
    RFC 1939, 1507
    RFC 2045, 1271, 1275, 1298, 1315, 1316, 1322, 1365, 1368
    RFC 2045#section-6.8, 1562
    RFC 2046, 1271, 1303, 1322
    RFC 2047, 1271, 1291, 1296, 1297, 1322, 1323, 1328
    RFC 2060, 1510, 1515
    RFC 2068, 1545
    RFC 2104, 683
    RFC 2109, 15451547, 1549, 1550, 15551557
    RFC 2183, 1271, 1277, 1317
    RFC 2231, 1271, 1275, 1276, 1314, 1316, 1322, 1329
    RFC 2295, 1490
    RFC 2324, 1490
    RFC 2342, 1514

RFC 2368, 1486
RFC 2373, 1574, 1575
RFC 2396, 1480, 1484, 1486
RFC 2397, 1470
RFC 2449, 1508
RFC 2518, 1489
RFC 2595, 1507, 1509
RFC 2616, 1448, 1451, 1466, 1467, 1475, 1486
RFC 2616#section-5.1.2, 1494, 1495
RFC 2616#section-14.23, 1495
RFC 2640, 1500, 1501, 1505
RFC 2732, 1486
RFC 2774, 1490
RFC 2821, 1271
RFC 2822, 781, 1313, 13211323, 13271329, 1352, 1493, 1539
RFC 2964, 1550
RFC 2965, 1460, 1463, 1549, 1550, 15531556, 1558
RFC 3056, 1577
RFC 3171, 1574
RFC 3229, 1489
RFC 3280, 1219
RFC 3330, 1575
RFC 3454, 175
RFC 3490, 206208
RFC 3490#section-3.1, 208
RFC 3492, 206, 207
RFC 3493, 1204
RFC 3501, 1516
RFC 3542, 1190
RFC 3548, 1369
RFC 3659, 1503
RFC 3879, 1577
RFC 3927, 1575
RFC 3986, 1477, 1479, 1481, 1482, 1484, 1485, 1539
RFC 4007, 1576, 1577
RFC 4086, 1241
RFC 4122, 15241527
RFC 4180, 637
RFC 4193, 1577
RFC 4217, 1505
RFC 4291, 1575, 1576
RFC 4380, 1577
RFC 4627, 1331, 1339
RFC 4648, 13651368, 2319
RFC 4918, 1489, 1490
RFC 4954, 1521
RFC 5161, 1513
RFC 5246, 1216, 1241
RFC 5280, 1206, 1208, 1241
RFC 5321, 1301
RFC 5322, 1271, 1272, 1282, 1285, 1286, 1288, 1290, 1291, 12941297, 1300, 1301, 1310, 1523
RFC 5424, 866
RFC 5735, 1575
RFC 5789, 1491
RFC 5842, 1489, 1490
RFC 5891, 207
RFC 5895, 207
RFC 5929, 1220
RFC 6066, 1215, 1225, 1242
RFC 6531, 1273, 1291, 1518
RFC 6532, 1271, 1272, 1282, 1291
RFC 6585, 1490
RFC 6855, 1513
RFC 6856, 1509
RFC 7159, 1331, 1338, 1339
RFC 7230, 1459, 1497
RFC 7301, 1214, 1225
RFC 7525, 1242
RFC 7693, 676
RFC 7725, 1490
RFC 7914, 675
RFC 8089, 468
RFC 8297, 1489
RFC 8305, 1118
RFC 8470, 1490
RFC 9110, 14891491
rfc2109 (*http.cookiejar.Cookie attribute*), 1557
rfc2109_as_netscape (*http.cookiejar.DefaultCookiePolicy attribute*), 1555
rfc2965 (*http.cookiejar.CookiePolicy attribute*), 1554
RFC_4122 (*in module uuid*), 1527
rfile (*http.server.BaseHTTPRequestHandler attribute*), 1539
rfile (*socketserver.DatagramRequestHandler attribute*), 1534
rfind() (*bytearray method*), 68
rfind() (*bytes method*), 68
rfind() (*mmap.mmap method*), 1267
rfind() (*str method*), 57
rgb_to_hls() (*in module colorsys*), 1592
rgb_to_hsv() (*in module colorsys*), 1592
rgb_to_yiq() (*in module colorsys*), 1592
rglob() (*pathlib.Path method*), 474
right (*filecmp.dircmp attribute*), 499
right() (*in module turtle*), 1619
right_list (*filecmp.dircmp attribute*), 499
right_only (*filecmp.dircmp attribute*), 499
RIGHTSHIFT (*in module token*), 2210
RIGHTSHIFTEQUAL (*in module token*), 2210
rindex() (*bytearray method*), 68
rindex() (*bytes method*), 68
rindex() (*str method*), 57

rjust() (*bytearray method*), 70
rjust() (*bytes method*), 70
rjust() (*str method*), 57
rlcompleter
    module, 181
RLIM_INFINITY (*in module resource*), 2277
RLIMIT_AS (*in module resource*), 2279
RLIMIT_CORE (*in module resource*), 2278
RLIMIT_CPU (*in module resource*), 2278
RLIMIT_DATA (*in module resource*), 2278
RLIMIT_FSIZE (*in module resource*), 2278
RLIMIT_KQUEUES (*in module resource*), 2280
RLIMIT_MEMLOCK (*in module resource*), 2279
RLIMIT_MSGQUEUE (*in module resource*), 2279
RLIMIT_NICE (*in module resource*), 2279
RLIMIT_NOFILE (*in module resource*), 2278
RLIMIT_NPROC (*in module resource*), 2278
RLIMIT_NPTS (*in module resource*), 2280
RLIMIT_OFILE (*in module resource*), 2278
RLIMIT_RSS (*in module resource*), 2278
RLIMIT_RTPRIO (*in module resource*), 2279
RLIMIT_RTTIME (*in module resource*), 2279
RLIMIT_SBSIZE (*in module resource*), 2279
RLIMIT_SIGPENDING (*in module resource*), 2279
RLIMIT_STACK (*in module resource*), 2278
RLIMIT_SWAP (*in module resource*), 2279
RLIMIT_VMEM (*in module resource*), 2279
RLock (*class in multiprocessing*), 989
RLock (*class in threading*), 962
RLock() (*multiprocessing.managers.SyncManager method*), 995
rmd() (*ftplib.FTP method*), 1504
rmdir() (*in module os*), 722
rmdir() (*in module test.support.os_helper*), 1917
rmdir() (*pathlib.Path method*), 478
rmtree() (*in module shutil*), 513
rmtree() (*in module test.support.os_helper*), 1917
RobotFileParser (*class in urllib.robotparser*), 1487
robots.txt, 1487
rollback() (*sqlite3.Connection method*), 560
rollover() (*tempfile.SpooledTemporaryFile method*), 502
ROMAN (*in module tkinter.font*), 1677
root (*pathlib.PurePath attribute*), 460
rotate() (*collections.deque method*), 269
rotate() (*decimal.Context method*), 377
rotate() (*decimal.Decimal method*), 369
rotate() (*logging.handlers.BaseRotatingHandler method*), 860
RotatingFileHandler (*class in logging.handlers*), 861
rotation_filename() (*logging.handlers.BaseRotatingHandler method*), 860
rotator (*logging.handlers.BaseRotatingHandler attribute*), 860
round()
    built-in function, 26
ROUND_05UP (*in module decimal*), 378
ROUND_CEILING (*in module decimal*), 378
ROUND_DOWN (*in module decimal*), 378
ROUND_FLOOR (*in module decimal*), 378
ROUND_HALF_DOWN (*in module decimal*), 378
ROUND_HALF_EVEN (*in module decimal*), 378
ROUND_HALF_UP (*in module decimal*), 378
ROUND_UP (*in module decimal*), 378
Rounded (*class in decimal*), 380
rounds (*sys.float_info attribute*), 2000
Row (*class in sqlite3*), 574
row_factory (*sqlite3.Connection attribute*), 570
row_factory (*sqlite3.Cursor attribute*), 573
rowcount (*sqlite3.Cursor attribute*), 573
RPAR (*in module token*), 2208
rpartition() (*bytearray method*), 68
rpartition() (*bytes method*), 68
rpartition() (*str method*), 57
rpc_paths (*xmlrpc.server.SimpleXMLRPCRequestHandler attribute*), 1568
rpop() (*poplib.POP3 method*), 1508
RS (*in module curses.ascii*), 905
rset() (*poplib.POP3 method*), 1509
RShift (*class in ast*), 2174
rshift() (*in module operator*), 448
rsplit() (*bytearray method*), 70
rsplit() (*bytes method*), 70
rsplit() (*str method*), 57
RSQB (*in module token*), 2209
rstrip() (*bytearray method*), 70
rstrip() (*bytes method*), 70
rstrip() (*str method*), 57
rt() (*in module turtle*), 1619
RTLD_DEEPBIND (*in module os*), 758
RTLD_GLOBAL (*in module os*), 758
RTLD_LAZY (*in module os*), 758
RTLD_LOCAL (*in module os*), 758
RTLD_NODELETE (*in module os*), 758
RTLD_NOLOAD (*in module os*), 758
RTLD_NOW (*in module os*), 758
ruler (*cmd.Cmd attribute*), 1653
run (*pdb command*), 1943
Run script, 1708
run() (*asyncio.Runner method*), 1068
run() (*bdb.Bdb method*), 1932
run() (*contextvars.Context method*), 1059
run() (*doctest.DocTestRunner method*), 1798
run() (*in module asyncio*), 1067
run() (*in module pdb*), 1937
run() (*in module profile*), 1947

run() (*in module subprocess*), 1031
run() (*multiprocessing.Process method*), 977
run() (*pdb.Pdb method*), 1938
run() (*profile.Profile method*), 1948
run() (*sched.scheduler method*), 1053
run() (*threading.Thread method*), 959
run() (*trace.Trace method*), 1960
run() (*unittest.IsolatedAsyncioTestCase method*), 1823
run() (*unittest.TestCase method*), 1814
run() (*unittest.TestSuite method*), 1825
run() (*unittest.TextTestRunner method*), 1831
run() (*wsgiref.handlers.BaseHandler method*), 1452
run_coroutine_threadsafe() (*in module asyncio*), 1083
run_docstring_examples() (*in module doctest*), 1791
run_forever() (*asyncio.loop method*), 1114
run_in_executor() (*asyncio.loop method*), 1128
run_in_subinterp() (*in module test.support*), 1911
run_module() (*in module runpy*), 2126
run_path() (*in module runpy*), 2127
run_python_until_end() (*in module test.support.script_helper*), 1914
run_script() (*modulefinder.ModuleFinder method*), 2125
run_until_complete() (*asyncio.loop method*), 1114
run_with_locale() (*in module test.support*), 1908
run_with_tz() (*in module test.support*), 1909
runcall() (*bdb.Bdb method*), 1932
runcall() (*in module pdb*), 1937
runcall() (*pdb.Pdb method*), 1938
runcall() (*profile.Profile method*), 1948
runcode() (*code.InteractiveInterpreter method*), 2116
runctx() (*bdb.Bdb method*), 1932
runctx() (*in module profile*), 1947
runctx() (*profile.Profile method*), 1948
runctx() (*trace.Trace method*), 1960
runeval() (*bdb.Bdb method*), 1932
runeval() (*in module pdb*), 1937
runeval() (*pdb.Pdb method*), 1938
runfunc() (*trace.Trace method*), 1960
Runner (*class in asyncio*), 1067
running() (*concurrent.futures.Future method*), 1028
runpy
    module, 2126
runsource() (*code.InteractiveInterpreter method*), 2116
runtime (*sys._emscripten_info attribute*), 1995
runtime_checkable() (*in module typing*), 1752
RuntimeError, 112
RuntimeWarning, 117
RUSAGE_BOTH (*in module resource*), 2281
RUSAGE_CHILDREN (*in module resource*), 2281
RUSAGE_SELF (*in module resource*), 2281
RUSAGE_THREAD (*in module resource*), 2281
RWF_APPEND (*in module os*), 708
RWF_DSYNC (*in module os*), 708
RWF_HIPRI (*in module os*), 707
RWF_NOWAIT (*in module os*), 707
RWF_SYNC (*in module os*), 708

S

-s
    calendar command line option, 261
    compileall command line option, 2221
    timeit command line option, 1956
    trace command line option, 1960
    unittest-discover command line option, 1807
S (*in module re*), 141
S_ENFMT (*in module stat*), 495
S_IEXEC (*in module stat*), 495
S_IFBLK (*in module stat*), 494
S_IFCHR (*in module stat*), 494
S_IFDIR (*in module stat*), 494
S_IFDOOR (*in module stat*), 494
S_IFIFO (*in module stat*), 494
S_IFLNK (*in module stat*), 494
S_IFMT() (*in module stat*), 492
S_IFPORT (*in module stat*), 494
S_IFREG (*in module stat*), 494
S_IFSOCK (*in module stat*), 493
S_IFWHT (*in module stat*), 494
S_IMODE() (*in module stat*), 492
S_IREAD (*in module stat*), 495
S_IRGRP (*in module stat*), 495
S_IROTH (*in module stat*), 495
S_IRUSR (*in module stat*), 495
S_IRWXG (*in module stat*), 495
S_IRWXO (*in module stat*), 495
S_IRWXU (*in module stat*), 494
S_ISBLK() (*in module stat*), 491
S_ISCHR() (*in module stat*), 491
S_ISDIR() (*in module stat*), 491
S_ISDOOR() (*in module stat*), 492
S_ISFIFO() (*in module stat*), 491
S_ISGID (*in module stat*), 494
S_ISLNK() (*in module stat*), 492
S_ISPORT() (*in module stat*), 492
S_ISREG() (*in module stat*), 491
S_ISSOCK() (*in module stat*), 492
S_ISUID (*in module stat*), 494
S_ISVTX (*in module stat*), 494
S_ISWHT() (*in module stat*), 492
S_IWGRP (*in module stat*), 495
S_IWOTH (*in module stat*), 495
S_IWRITE (*in module stat*), 495

S_IWUSR (*in module stat*), 495
S_IXGRP (*in module stat*), 495
S_IXOTH (*in module stat*), 495
S_IXUSR (*in module stat*), 495
safe (*uuid.SafeUUID attribute*), 1525
safe_path (*sys.flags attribute*), 1998
safe_substitute() (*string.Template method*), 130
SafeChildWatcher (*class in asyncio*), 1159
saferepr() (*in module pprint*), 315
SafeUUID (*class in uuid*), 1525
same_files (*filecmp.dircmp attribute*), 499
same_quantum() (*decimal.Context method*), 377
same_quantum() (*decimal.Decimal method*), 369
samefile() (*in module os.path*), 486
samefile() (*pathlib.Path method*), 472
SameFileError, 511
sameopenfile() (*in module os.path*), 487
samesite (*http.cookies.Morsel attribute*), 1546
samestat() (*in module os.path*), 487
sample() (*in module random*), 394
samples() (*statistics.NormalDist method*), 415
SATURDAY (*in module calendar*), 258
save() (*http.cookiejar.FileCookieJar method*), 1552
save() (*test.support.SaveSignals method*), 1912
SaveAs (*class in tkinter.filedialog*), 1680
SAVEDCWD (*in module test.support.os_helper*), 1916
SaveFileDialog (*class in tkinter.filedialog*), 1681
SaveKey() (*in module winreg*), 2258
SaveSignals (*class in test.support*), 1912
savetty() (*in module curses*), 879
SAX2DOM (*class in xml.dom.pulldom*), 1418
SAXException, 1420
SAXNotRecognizedException, 1420
SAXNotSupportedException, 1420
SAXParseException, 1420
scaleb() (*decimal.Context method*), 377
scaleb() (*decimal.Decimal method*), 369
scandir() (*in module os*), 722
scanf (*C function*), 152
sched
    module, 1052
SCHED_BATCH (*in module os*), 755
SCHED_FIFO (*in module os*), 755
sched_get_priority_max() (*in module os*), 755
sched_get_priority_min() (*in module os*), 755
sched_getaffinity() (*in module os*), 756
sched_getparam() (*in module os*), 756
sched_getscheduler() (*in module os*), 756
SCHED_IDLE (*in module os*), 755
SCHED_OTHER (*in module os*), 755
sched_param (*class in os*), 755
sched_priority (*os.sched_param attribute*), 755
SCHED_RESET_ON_FORK (*in module os*), 755
SCHED_RR (*in module os*), 755
sched_rr_get_interval() (*in module os*), 756
sched_setaffinity() (*in module os*), 756
sched_setparam() (*in module os*), 756
sched_setscheduler() (*in module os*), 755
SCHED_SPORADIC (*in module os*), 755
sched_yield() (*in module os*), 756
scheduler (*class in sched*), 1052
SCM_CREDS2 (*in module socket*), 1183
scope_id (*ipaddress.IPv6Address attribute*), 1577
Screen (*class in turtle*), 1644
screensize() (*in module turtle*), 1638
script_from_examples() (*in module doctest*), 1800
scroll() (*curses.window method*), 887
ScrolledCanvas (*class in turtle*), 1644
ScrolledText (*class in tkinter.scrolledtext*), 1684
scrollok() (*curses.window method*), 887
scrypt() (*in module hashlib*), 675
seal() (*in module unittest.mock*), 1877
search
    path, module, 510, 2008, 2111
search() (*imaplib.IMAP4 method*), 1514
search() (*in module re*), 142
search() (*re.Pattern method*), 146
second (*datetime.datetime attribute*), 224
second (*datetime.time attribute*), 233
seconds since the epoch, 775
secrets
    module, 685
SECTCRE (*configparser.ConfigParser attribute*), 656
sections() (*configparser.ConfigParser method*), 659
secure (*http.cookiejar.Cookie attribute*), 1556
secure (*http.cookies.Morsel attribute*), 1546
secure hash algorithm, SHA1, SHA2, SHA224, SHA256, SHA384, SHA512, SHA3, Shake, Blake2, 671
Secure Sockets Layer, 1204
security
    http.server, 1544
security considerations, 2318
security_level (*ssl.SSLContext attribute*), 1229
see() (*tkinter.ttk.Treeview method*), 1700
seed() (*in module random*), 393
seed() (*random.Random method*), 397
seed_bits (*sys.hash_info attribute*), 2005
seek() (*io.IOBase method*), 765
seek() (*io.TextIOBase method*), 771
seek() (*io.TextIOWrapper method*), 772
seek() (*mmap.mmap method*), 1267
seek() (*sqlite3.Blob method*), 575
SEEK_CUR (*in module os*), 703
SEEK_DATA (*in module os*), 704
SEEK_END (*in module os*), 703
SEEK_HOLE (*in module os*), 704

SEEK_SET (*in module os*), 703
seekable() (*bz2.BZ2File method*), 596
seekable() (*io.IOBase method*), 765
seekable() (*mmap.mmap method*), 1267
select
    module, 1242
select() (*imaplib.IMAP4 method*), 1514
select() (*in module select*), 1243
select() (*selectors.BaseSelector method*), 1251
select() (*tkinter.ttk.Notebook method*), 1693
selected_alpn_protocol() (*ssl.SSLSocket method*), 1220
selected_npn_protocol() (*ssl.SSLSocket method*), 1220
selection() (*tkinter.ttk.Treeview method*), 1700
selection_add() (*tkinter.ttk.Treeview method*), 1700
selection_remove() (*tkinter.ttk.Treeview method*), 1700
selection_set() (*tkinter.ttk.Treeview method*), 1700
selection_toggle() (*tkinter.ttk.Treeview method*), 1700
selector (*urllib.request.Request attribute*), 1463
SelectorEventLoop (*class in asyncio*), 1134
SelectorKey (*class in selectors*), 1250
selectors
    module, 1249
SelectSelector (*class in selectors*), 1252
Self (*in module typing*), 1732
Semaphore (*class in asyncio*), 1100
Semaphore (*class in multiprocessing*), 989
Semaphore (*class in threading*), 965
Semaphore() (*multiprocessing.managers.SyncManager method*), 995
semaphores, binary, 1062
SEMI (*in module token*), 2209
SEND (*opcode*), 2245
send() (*http.client.HTTPConnection method*), 1497
send() (*imaplib.IMAP4 method*), 1514
send() (*logging.handlers.DatagramHandler method*), 865
send() (*logging.handlers.SocketHandler method*), 864
send() (*multiprocessing.connection.Connection method*), 986
send() (*socket.socket method*), 1197
send_bytes() (*multiprocessing.connection.Connection method*), 986
send_error() (*http.server.BaseHTTPRequestHandler method*), 1540
send_fds() (*in module socket*), 1191
send_header() (*http.server.BaseHTTPRequestHandler method*), 1541
send_message() (*smtplib.SMTP method*), 1523
send_response() (*http.server.BaseHTTPRequestHandler method*), 1540
send_response_only() (*http.server.BaseHTTPRequestHandler method*), 1541
send_signal() (*asyncio.subprocess.Process method*), 1106
send_signal() (*asyncio.SubprocessTransport method*), 1146
send_signal() (*subprocess.Popen method*), 1041
sendall() (*socket.socket method*), 1197
sendcmd() (*ftplib.FTP method*), 1502
sendfile() (*asyncio.loop method*), 1123
sendfile() (*in module os*), 709
sendfile() (*socket.socket method*), 1198
sendfile() (*wsgiref.handlers.BaseHandler method*), 1454
SendfileNotAvailableError, 1111
sendmail() (*smtplib.SMTP method*), 1522
sendmsg() (*socket.socket method*), 1197
sendmsg_afalg() (*socket.socket method*), 1198
sendto() (*asyncio.DatagramTransport method*), 1146
sendto() (*socket.socket method*), 1197
sentinel (*in module unittest.mock*), 1869
sentinel (*multiprocessing.Process attribute*), 979
sep (*in module os*), 757
SEPTEMBER (*in module calendar*), 259
sequence, **2334**
    iteration, 44
    object, 45
    types, immutable, 47
    types, mutable, 47
    types, operations on, 45, 47
Sequence (*class in collections.abc*), 283
Sequence (*class in typing*), 1770
SequenceMatcher (*class in difflib*), 161
serialize() (*sqlite3.Connection method*), 568
serializing
    objects, 523
serve_forever() (*asyncio.Server method*), 1134
serve_forever() (*socketserver.BaseServer method*), 1532
server
    WWW, 1538
Server (*class in asyncio*), 1133
server (*http.server.BaseHTTPRequestHandler attribute*), 1539
server (*socketserver.BaseRequestHandler attribute*), 1534
server_activate() (*socketserver.BaseServer method*), 1533
server_address (*socketserver.BaseServer attribute*), 1532
server_bind() (*socketserver.BaseServer method*), 1533

server_close() (*socketserver.BaseServer method*), 1532
server_hostname (*ssl.SSLSocket attribute*), 1221
server_side (*ssl.SSLSocket attribute*), 1221
server_software (*wsgiref.handlers.BaseHandler attribute*), 1453
server_version (*http.server.BaseHTTPRequestHandler attribute*), 1539
server_version (*http.server.SimpleHTTPRequestHandler attribute*), 1542
ServerProxy (*class in xmlrpc.client*), 1558
service_actions() (*socketserver.BaseServer method*), 1532
session (*ssl.SSLSocket attribute*), 1221
session_reused (*ssl.SSLSocket attribute*), 1222
session_stats() (*ssl.SSLContext method*), 1228
set
    object, 86
set (*built-in class*), 86
Set (*class in ast*), 2172
Set (*class in collections.abc*), 284
Set (*class in typing*), 1767
Set Breakpoint, 1710
set comprehension, **2334**
set() (*asyncio.Event method*), 1098
set() (*configparser.ConfigParser method*), 661
set() (*configparser.RawConfigParser method*), 663
set() (*contextvars.ContextVar method*), 1058
set() (*http.cookies.Morsel method*), 1547
set() (*test.support.os_helper.EnvironmentVarGuard method*), 1917
set() (*threading.Event method*), 967
set() (*tkinter.ttk.Combobox method*), 1690
set() (*tkinter.ttk.Spinbox method*), 1691
set() (*tkinter.ttk.Treeview method*), 1700
set() (*xml.etree.ElementTree.Element method*), 1394
SET_ADD (*opcode*), 2234
set_allowed_domains() (*http.cookiejar.DefaultCookiePolicy method*), 1555
set_alpn_protocols() (*ssl.SSLContext method*), 1225
set_app() (*wsgiref.simple_server.WSGIServer method*), 1450
set_asyncgen_hooks() (*in module sys*), 2012
set_authorizer() (*sqlite3.Connection method*), 564
set_auto_history() (*in module readline*), 179
set_blocked_domains() (*http.cookiejar.DefaultCookiePolicy method*), 1555
set_blocking() (*in module os*), 710
set_boundary() (*email.message.EmailMessage method*), 1276
set_boundary() (*email.message.Message method*), 1317
set_break() (*bdb.Bdb method*), 1931
set_charset() (*email.message.Message method*), 1313
set_child_watcher() (*asyncio.AbstractEventLoopPolicy method*), 1157
set_child_watcher() (*in module asyncio*), 1158
set_children() (*tkinter.ttk.Treeview method*), 1698
set_ciphers() (*ssl.SSLContext method*), 1225
set_completer() (*in module readline*), 179
set_completer_delims() (*in module readline*), 180
set_completion_display_matches_hook() (*in module readline*), 180
set_content() (*email.contentmanager.ContentManager method*), 1301
set_content() (*email.message.EmailMessage method*), 1279
set_content() (*in module email.contentmanager*), 1302
set_continue() (*bdb.Bdb method*), 1931
set_cookie() (*http.cookiejar.CookieJar method*), 1551
set_cookie_if_ok() (*http.cookiejar.CookieJar method*), 1551
set_coroutine_origin_tracking_depth() (*in module sys*), 2012
set_data() (*importlib.abc.SourceLoader method*), 2136
set_data() (*importlib.machinery.SourceFileLoader method*), 2141
set_date() (*mailbox.MaildirMessage method*), 1353
set_debug() (*asyncio.loop method*), 1130
set_debug() (*in module gc*), 2087
set_debuglevel() (*ftplib.FTP method*), 1501
set_debuglevel() (*http.client.HTTPConnection method*), 1495
set_debuglevel() (*poplib.POP3 method*), 1508
set_debuglevel() (*smtplib.SMTP method*), 1520
set_default_executor() (*asyncio.loop method*), 1129
set_default_type() (*email.message.EmailMessage method*), 1276
set_default_type() (*email.message.Message method*), 1315
set_default_verify_paths() (*ssl.SSLContext method*), 1225
set_defaults() (*argparse.ArgumentParser method*), 820
set_defaults() (*optparse.OptionParser method*), 2310
set_ecdh_curve() (*ssl.SSLContext method*), 1226
set_errno() (*in module ctypes*), 948
set_error_mode() (*in module msvcrt*), 2253

set_escdelay() (*in module curses*), 879
set_event_loop() (*asyncio.AbstractEventLoopPolicy method*), 1157
set_event_loop() (*in module asyncio*), 1113
set_event_loop_policy() (*in module asyncio*), 1156
set_events() (*in module sys.monitoring*), 2021
set_exception() (*asyncio.Future method*), 1139
set_exception() (*concurrent.futures.Future method*), 1029
set_exception_handler() (*asyncio.loop method*), 1129
set_executable() (*in module multiprocessing*), 985
set_filter() (*tkinter.filedialog.FileDialog method*), 1681
set_flags() (*mailbox.Maildir method*), 1346
set_flags() (*mailbox.MaildirMessage method*), 1353
set_flags() (*mailbox.mboxMessage method*), 1355
set_flags() (*mailbox.MMDFMessage method*), 1359
set_forkserver_preload() (*in module multiprocessing*), 985
set_from() (*mailbox.mboxMessage method*), 1354
set_from() (*mailbox.MMDFMessage method*), 1359
SET_FUNCTION_ATTRIBUTE (*opcode*), 2243
set_handle_inheritable() (*in module os*), 712
set_history_length() (*in module readline*), 178
set_info() (*mailbox.Maildir method*), 1346
set_info() (*mailbox.MaildirMessage method*), 1353
set_inheritable() (*in module os*), 712
set_inheritable() (*socket.socket method*), 1198
set_int_max_str_digits() (*in module sys*), 2010
set_labels() (*mailbox.BabylMessage method*), 1357
set_last_error() (*in module ctypes*), 948
set_local_events() (*in module sys.monitoring*), 2021
set_memlimit() (*in module test.support*), 1906
set_name() (*asyncio.Task method*), 1086
set_next() (*bdb.Bdb method*), 1930
set_nonstandard_attr() (*http.cookiejar.Cookie method*), 1557
set_npn_protocols() (*ssl.SSLContext method*), 1225
set_ok() (*http.cookiejar.CookiePolicy method*), 1553
set_param() (*email.message.EmailMessage method*), 1276
set_param() (*email.message.Message method*), 1316
set_pasv() (*ftplib.FTP method*), 1503
set_payload() (*email.message.Message method*), 1313
set_policy() (*http.cookiejar.CookieJar method*), 1551
set_pre_input_hook() (*in module readline*), 179
set_progress_handler() (*sqlite3.Connection method*), 564
set_protocol() (*asyncio.BaseTransport method*), 1144
set_proxy() (*urllib.request.Request method*), 1464
set_psk_client_callback() (*ssl.SSLContext method*), 1230
set_psk_server_callback() (*ssl.SSLContext method*), 1231
set_quit() (*bdb.Bdb method*), 1931
set_result() (*asyncio.Future method*), 1139
set_result() (*concurrent.futures.Future method*), 1029
set_return() (*bdb.Bdb method*), 1930
set_running_or_notify_cancel() (*concurrent.futures.Future method*), 1028
set_selection() (*tkinter.filedialog.FileDialog method*), 1681
set_seq1() (*difflib.SequenceMatcher method*), 161
set_seq2() (*difflib.SequenceMatcher method*), 161
set_seqs() (*difflib.SequenceMatcher method*), 161
set_sequences() (*mailbox.MH method*), 1349
set_sequences() (*mailbox.MHMessage method*), 1356
set_server_documentation() (*xmlrpc.server.DocCGIXMLRPCRequestHandler method*), 1572
set_server_documentation() (*xmlrpc.server.DocXMLRPCServer method*), 1572
set_server_name() (*xmlrpc.server.DocCGIXMLRPCRequestHandler method*), 1572
set_server_name() (*xmlrpc.server.DocXMLRPCServer method*), 1572
set_server_title() (*xmlrpc.server.DocCGIXMLRPCRequestHandler method*), 1572
set_server_title() (*xmlrpc.server.DocXMLRPCServer method*), 1572
set_servername_callback (*ssl.SSLContext attribute*), 1226
set_start_method() (*in module multiprocessing*), 985
set_startup_hook() (*in module readline*), 179
set_step() (*bdb.Bdb method*), 1930
set_subdir() (*mailbox.MaildirMessage method*), 1353
set_tabsize() (*in module curses*), 879
set_task_factory() (*asyncio.loop method*), 1117
set_threshold() (*in module gc*), 2087
set_trace() (*bdb.Bdb method*), 1931
set_trace() (*in module bdb*), 1932
set_trace() (*in module pdb*), 1937
set_trace() (*pdb.Pdb method*), 1938
set_trace_callback() (*sqlite3.Connection method*), 565
set_tunnel() (*http.client.HTTPConnection method*),

1495
set_type() (*email.message.Message method*), 1316
set_unittest_reportflags() (*in module doctest*), 1793
set_unixfrom() (*email.message.EmailMessage method*), 1273
set_unixfrom() (*email.message.Message method*), 1312
set_until() (*bdb.Bdb method*), 1931
SET_UPDATE (*opcode*), 2238
set_url() (*urllib.robotparser.RobotFileParser method*), 1487
set_usage() (*optparse.OptionParser method*), 2310
set_userptr() (*curses.panel.Panel method*), 908
set_visible() (*mailbox.BabylMessage method*), 1357
set_wakeup_fd() (*in module signal*), 1259
set_write_buffer_limits() (*asyncio.WriteTransport method*), 1145
setacl() (*imaplib.IMAP4 method*), 1515
setannotation() (*imaplib.IMAP4 method*), 1515
setattr()
    built-in function, 26
setAttribute() (*xml.dom.Element method*), 1408
setAttributeNode() (*xml.dom.Element method*), 1408
setAttributeNodeNS() (*xml.dom.Element method*), 1408
setAttributeNS() (*xml.dom.Element method*), 1408
SetBase() (*xml.parsers.expat.xmlparser method*), 1433
setblocking() (*socket.socket method*), 1198
setByteStream() (*xml.sax.xmlreader.InputSource method*), 1431
setcbreak() (*in module tty*), 2272
setCharacterStream() (*xml.sax.xmlreader.InputSource method*), 1431
SetComp (*class in ast*), 2177
setcomptype() (*wave.Wave_write method*), 1591
setconfig() (*sqlite3.Connection method*), 568
setContentHandler() (*xml.sax.xmlreader.XMLReader method*), 1429
setcontext() (*in module decimal*), 371
setDaemon() (*threading.Thread method*), 961
setdefault() (*dict method*), 91
setdefault() (*http.cookies.Morsel method*), 1547
setdefaulttimeout() (*in module socket*), 1190
setdlopenflags() (*in module sys*), 2010
setDocumentLocator() (*xml.sax.handler.ContentHandler method*), 1423
setDTDHandler() (*xml.sax.xmlreader.XMLReader method*), 1429
setegid() (*in module os*), 696
setEncoding() (*xml.sax.xmlreader.InputSource method*), 1430
setEntityResolver() (*xml.sax.xmlreader.XMLReader method*), 1429
setErrorHandler() (*xml.sax.xmlreader.XMLReader method*), 1429
seteuid() (*in module os*), 696
setFeature() (*xml.sax.xmlreader.XMLReader method*), 1429
setfirstweekday() (*in module calendar*), 257
setFormatter() (*logging.Handler method*), 831
setframerate() (*wave.Wave_write method*), 1591
setgid() (*in module os*), 696
setgroups() (*in module os*), 696
seth() (*in module turtle*), 1621
setheading() (*in module turtle*), 1621
sethostname() (*in module socket*), 1190
setinputsizes() (*sqlite3.Cursor method*), 572
setitem() (*in module operator*), 449
setitimer() (*in module signal*), 1259
setLevel() (*logging.Handler method*), 830
setLevel() (*logging.Logger method*), 826
setlimit() (*sqlite3.Connection method*), 568
setlocale() (*in module locale*), 1602
setLocale() (*xml.sax.xmlreader.XMLReader method*), 1429
setLoggerClass() (*in module logging*), 842
setlogmask() (*in module syslog*), 2282
setLogRecordFactory() (*in module logging*), 842
setMaxConns() (*urllib.request.CacheFTPHandler method*), 1470
setmode() (*in module msvcrt*), 2251
setName() (*threading.Thread method*), 960
setnchannels() (*wave.Wave_write method*), 1591
setnframes() (*wave.Wave_write method*), 1591
setns() (*in module os*), 696
setoutputsize() (*sqlite3.Cursor method*), 573
SetParamEntityParsing() (*xml.parsers.expat.xmlparser method*), 1434
setparams() (*wave.Wave_write method*), 1591
setpassword() (*zipfile.ZipFile method*), 610
setpgid() (*in module os*), 697
setpgrp() (*in module os*), 697
setpos() (*in module turtle*), 1620
setpos() (*wave.Wave_read method*), 1590
setposition() (*in module turtle*), 1620
setpriority() (*in module os*), 697
setprofile() (*in module sys*), 2010
setprofile() (*in module threading*), 957
setprofile_all_threads() (*in module threading*), 957
setProperty() (*xml.sax.xmlreader.XMLReader method*), 1429

setPublicId() (*xml.sax.xmlreader.InputSource method*), 1430
setquota() (*imaplib.IMAP4 method*), 1515
setraw() (*in module tty*), 2272
setrecursionlimit() (*in module sys*), 2011
setregid() (*in module os*), 697
SetReparseDeferralEnabled() (*xml.parsers.expat.xmlparser method*), 1434
setresgid() (*in module os*), 697
setresuid() (*in module os*), 697
setreuid() (*in module os*), 697
setrlimit() (*in module resource*), 2277
setsampwidth() (*wave.Wave_write method*), 1591
setscrreg() (*curses.window method*), 887
setsid() (*in module os*), 697
setsockopt() (*socket.socket method*), 1198
setstate() (*codecs.IncrementalDecoder method*), 198
setstate() (*codecs.IncrementalEncoder method*), 197
setstate() (*in module random*), 393
setstate() (*random.Random method*), 397
setStream() (*logging.StreamHandler method*), 858
setswitchinterval() (*in module sys*), 2011
setswitchinterval() (*in module test.support*), 1906
setSystemId() (*xml.sax.xmlreader.InputSource method*), 1430
setsyx() (*in module curses*), 879
setTarget() (*logging.handlers.MemoryHandler method*), 869
settimeout() (*socket.socket method*), 1198
setTimeout() (*urllib.request.CacheFTPHandler method*), 1470
settrace() (*in module sys*), 2011
settrace() (*in module threading*), 957
settrace_all_threads() (*in module threading*), 957
setuid() (*in module os*), 697
setundobuffer() (*in module turtle*), 1636
--setup
    timeit command line option, 1956
setup() (*in module turtle*), 1643
setup() (*socketserver.BaseRequestHandler method*), 1533
setUp() (*unittest.TestCase method*), 1813
SETUP_ANNOTATIONS (*opcode*), 2235
SETUP_CLEANUP (*opcode*), 2247
setup_environ() (*wsgiref.handlers.BaseHandler method*), 1453
SETUP_FINALLY (*opcode*), 2246
setup_python() (*venv.EnvBuilder method*), 1980
setup_scripts() (*venv.EnvBuilder method*), 1980
setup_testing_defaults() (*in module wsgiref.util*), 1447
SETUP_WITH (*opcode*), 2247
setUpClass() (*unittest.TestCase method*), 1814
setupterm() (*in module curses*), 879
SetValue() (*in module winreg*), 2258
SetValueEx() (*in module winreg*), 2259
setworldcoordinates() (*in module turtle*), 1638
setx() (*in module turtle*), 1621
setxattr() (*in module os*), 740
sety() (*in module turtle*), 1621
SF_APPEND (*in module stat*), 496
SF_ARCHIVED (*in module stat*), 496
SF_DATALESS (*in module stat*), 497
SF_FIRMLINK (*in module stat*), 497
SF_IMMUTABLE (*in module stat*), 496
SF_MNOWAIT (*in module os*), 709
SF_NOCACHE (*in module os*), 710
SF_NODISKIO (*in module os*), 709
SF_NOUNLINK (*in module stat*), 496
SF_RESTRICTED (*in module stat*), 496
SF_SETTABLE (*in module stat*), 496
SF_SNAPSHOT (*in module stat*), 497
SF_SUPPORTED (*in module stat*), 496
SF_SYNC (*in module os*), 709
SF_SYNTHETIC (*in module stat*), 496
sha1() (*in module hashlib*), 672
sha3_224() (*in module hashlib*), 673
sha3_256() (*in module hashlib*), 673
sha3_384() (*in module hashlib*), 673
sha3_512() (*in module hashlib*), 673
sha224() (*in module hashlib*), 672
sha256() (*in module hashlib*), 672
sha384() (*in module hashlib*), 673
sha512() (*in module hashlib*), 673
shake_128() (*in module hashlib*), 674
shake_256() (*in module hashlib*), 674
Shape (*class in turtle*), 1645
shape (*memoryview attribute*), 86
shape() (*in module turtle*), 1631
shapesize() (*in module turtle*), 1632
shapetransform() (*in module turtle*), 1633
share() (*socket.socket method*), 1199
ShareableList (*class in multiprocessing.shared_memory*), 1020
ShareableList() (*multiprocessing.managers.SharedMemoryManager method*), 1020
Shared Memory, 1017
shared_ciphers() (*ssl.SSLSocket method*), 1220
shared_memory (*sys._emscripten_info attribute*), 1995
SharedMemory (*class in multiprocessing.shared_memory*), 1017
SharedMemory() (*multiprocessing.managers.SharedMemoryManager method*), 1019
SharedMemoryManager (*class in multiprocessing.managers*), 1019

shearfactor() (*in module turtle*), 1632
Shelf (*class in shelve*), 543
shelve
    module, 541, 544
shield() (*in module asyncio*), 1077
shift() (*decimal.Context method*), 377
shift() (*decimal.Decimal method*), 370
shift_path_info() (*in module wsgiref.util*), 1447
shifting
    operations, 38
shlex
    module, 1656
shlex (*class in shlex*), 1657
shm (*multiprocessing.shared_memory.ShareableList attribute*), 1021
SHORT_TIMEOUT (*in module test.support*), 1904
shortDescription() (*unittest.TestCase method*), 1822
shorten() (*in module textwrap*), 170
shouldFlush() (*logging.handlers.BufferingHandler method*), 869
shouldFlush() (*logging.handlers.MemoryHandler method*), 869
shouldStop (*unittest.TestResult attribute*), 1828
show() (*curses.panel.Panel method*), 908
show() (*tkinter.commondialog.Dialog method*), 1682
show() (*tkinter.messagebox.Message method*), 1683
show_code() (*in module dis*), 2227
show_flag_values() (*in module enum*), 338
--show-caches
    dis command line option, 2226
showerror() (*in module tkinter.messagebox*), 1683
showinfo() (*in module tkinter.messagebox*), 1683
--show-offsets
    dis command line option, 2226
showsyntaxerror() (*code.InteractiveInterpreter method*), 2116
showtraceback() (*code.InteractiveInterpreter method*), 2116
showturtle() (*in module turtle*), 1631
showwarning() (*in module tkinter.messagebox*), 1683
showwarning() (*in module warnings*), 2041
shuffle() (*in module random*), 394
ShutDown, 1055
shutdown() (*asyncio.Queue method*), 1109
shutdown() (*concurrent.futures.Executor method*), 1024
shutdown() (*imaplib.IMAP4 method*), 1515
shutdown() (*in module logging*), 842
shutdown() (*multiprocessing.managers.BaseManager method*), 994
shutdown() (*queue.Queue method*), 1057
shutdown() (*socketserver.BaseServer method*), 1532
shutdown() (*socket.socket method*), 1199
shutdown_asyncgens() (*asyncio.loop method*), 1114
shutdown_default_executor() (*asyncio.loop method*), 1115
shutil
    module, 510
SI (*in module curses.ascii*), 904
side_effect (*unittest.mock.Mock attribute*), 1843
SIG_BLOCK (*in module signal*), 1257
SIG_DFL (*in module signal*), 1254
SIG_IGN (*in module signal*), 1255
SIG_SETMASK (*in module signal*), 1257
SIG_UNBLOCK (*in module signal*), 1257
SIGABRT (*in module signal*), 1255
SIGALRM (*in module signal*), 1255
SIGBREAK (*in module signal*), 1255
SIGBUS (*in module signal*), 1255
SIGCHLD (*in module signal*), 1255
SIGCLD (*in module signal*), 1255
SIGCONT (*in module signal*), 1255
SIGFPE (*in module signal*), 1255
SIGHUP (*in module signal*), 1255
SIGILL (*in module signal*), 1255
SIGINT (*in module signal*), 1255
siginterrupt() (*in module signal*), 1260
SIGKILL (*in module signal*), 1256
Sigmasks (*class in signal*), 1254
signal
    module, 1064, 1253
signal() (*in module signal*), 1260
Signals (*class in signal*), 1254
Signature (*class in inspect*), 2097
signature (*inspect.BoundArguments attribute*), 2101
signature() (*in module inspect*), 2097
sigpending() (*in module signal*), 1260
SIGPIPE (*in module signal*), 1256
SIGSEGV (*in module signal*), 1256
SIGSTKFLT (*in module signal*), 1256
SIGTERM (*in module signal*), 1256
sigtimedwait() (*in module signal*), 1261
SIGUSR1 (*in module signal*), 1256
SIGUSR2 (*in module signal*), 1256
sigwait() (*in module signal*), 1260
sigwaitinfo() (*in module signal*), 1261
SIGWINCH (*in module signal*), 1256
SIMPLE (*inspect.BufferFlags attribute*), 2110
Simple Mail Transfer Protocol, 1517
SimpleCookie (*class in http.cookies*), 1545
simplefilter() (*in module warnings*), 2042
SimpleHandler (*class in wsgiref.handlers*), 1452
SimpleHTTPRequestHandler (*class in http.server*), 1542
SimpleNamespace (*class in types*), 311
SimpleQueue (*class in multiprocessing*), 983

SimpleQueue (*class in queue*), 1055
SimpleXMLRPCRequestHandler (*class in xmlrpc.server*), 1567
SimpleXMLRPCServer (*class in xmlrpc.server*), 1566
sin() (*in module cmath*), 356
sin() (*in module math*), 352
single dispatch, **2335**
SingleAddressHeader (*class in email.headerregistry*), 1298
singledispatch() (*in module functools*), 441
singledispatchmethod (*class in functools*), 444
sinh() (*in module cmath*), 357
sinh() (*in module math*), 353
SIO_KEEPALIVE_VALS (*in module socket*), 1182
SIO_LOOPBACK_FAST_PATH (*in module socket*), 1182
SIO_RCVALL (*in module socket*), 1182
site
    module, 2111
site command line option
    --user-base, 2114
    --user-site, 2114
site_maps() (*urllib.robotparser.RobotFileParser method*), 1487
sitecustomize
    module, 2112
site-packages
    directory, 2111
sixtofour (*ipaddress.IPv6Address attribute*), 1577
size (*multiprocessing.shared_memory.SharedMemory attribute*), 1018
size (*struct.Struct attribute*), 190
size (*tarfile.TarInfo attribute*), 626
size (*tracemalloc.Statistic attribute*), 1970
size (*tracemalloc.StatisticDiff attribute*), 1971
size (*tracemalloc.Trace attribute*), 1971
size() (*ftplib.FTP method*), 1504
size() (*mmap.mmap method*), 1267
size_diff (*tracemalloc.StatisticDiff attribute*), 1971
Sized (*class in collections.abc*), 283
Sized (*class in typing*), 1773
sizeof() (*in module ctypes*), 948
sizeof_digit (*sys.int_info attribute*), 2006
SKIP (*in module doctest*), 1787
skip() (*in module unittest*), 1812
skip_if_broken_multiprocessing_synchronize() (*in module test.support*), 1912
skip_unless_bind_unix_socket() (*in module test.support.socket_helper*), 1913
skip_unless_symlink() (*in module test.support.os_helper*), 1917
skip_unless_xattr() (*in module test.support.os_helper*), 1917
skipIf() (*in module unittest*), 1812
skipinitialspace (*csv.Dialect attribute*), 642
skipped (*doctest.TestResults attribute*), 1797
skipped (*unittest.TestResult attribute*), 1828
skippedEntity() (*xml.sax.handler.ContentHandler method*), 1425
skips (*doctest.DocTestRunner attribute*), 1799
SkipTest, 1812
skipTest() (*unittest.TestCase method*), 1814
skipUnless() (*in module unittest*), 1812
SLASH (*in module token*), 2209
SLASHEQUAL (*in module token*), 2210
sleep() (*in module asyncio*), 1075
sleep() (*in module time*), 779
sleeping_retry() (*in module test.support*), 1905
slice, **2335**
    assignment, 47
    built-in function, 2243
    operation, 45
slice (*built-in class*), 27
Slice (*class in ast*), 2177
slow_callback_duration (*asyncio.loop attribute*), 1130
SMALLEST (*in module test.support*), 1905
SMTP
    protocol, 1517
SMTP (*class in smtplib*), 1517
SMTP (*in module email.policy*), 1292
SMTP_SSL (*class in smtplib*), 1518
SMTPAuthenticationError, 1519
SMTPConnectError, 1519
SMTPDataError, 1519
SMTPException, 1518
SMTPHandler (*class in logging.handlers*), 868
SMTPHeloError, 1519
smtplib
    module, 1517
SMTPNotSupportedError, 1519
SMTPRecipientsRefused, 1519
SMTPResponseException, 1519
SMTPSenderRefused, 1519
SMTPServerDisconnected, 1518
SMTPUTF8 (*in module email.policy*), 1292
Snapshot (*class in tracemalloc*), 1969
SND_ALIAS (*in module winsound*), 2264
SND_ASYNC (*in module winsound*), 2264
SND_FILENAME (*in module winsound*), 2263
SND_LOOP (*in module winsound*), 2264
SND_MEMORY (*in module winsound*), 2264
SND_NODEFAULT (*in module winsound*), 2264
SND_NOSTOP (*in module winsound*), 2264
SND_NOWAIT (*in module winsound*), 2265
SND_PURGE (*in module winsound*), 2264
sni_callback (*ssl.SSLContext attribute*), 1225
sniff() (*csv.Sniffer method*), 640
Sniffer (*class in csv*), 640

SO (*in module curses.ascii*), 904
SO_INCOMING_CPU (*in module socket*), 1183
sock_accept() (*asyncio.loop method*), 1125
SOCK_CLOEXEC (*in module socket*), 1179
sock_connect() (*asyncio.loop method*), 1125
SOCK_DGRAM (*in module socket*), 1179
SOCK_MAX_SIZE (*in module test.support*), 1904
SOCK_NONBLOCK (*in module socket*), 1179
SOCK_RAW (*in module socket*), 1179
SOCK_RDM (*in module socket*), 1179
sock_recv() (*asyncio.loop method*), 1124
sock_recv_into() (*asyncio.loop method*), 1124
sock_recvfrom() (*asyncio.loop method*), 1124
sock_recvfrom_into() (*asyncio.loop method*), 1125
sock_sendall() (*asyncio.loop method*), 1125
sock_sendfile() (*asyncio.loop method*), 1126
sock_sendto() (*asyncio.loop method*), 1125
SOCK_SEQPACKET (*in module socket*), 1179
SOCK_STREAM (*in module socket*), 1179
socket
    module, 1175, 1443
    object, 1175
socket (*class in socket*), 1184
socket (*socketserver.BaseServer attribute*), 1532
socket() (*imaplib.IMAP4 method*), 1515
socket() (*in module socket*), 1243
socket_type (*socketserver.BaseServer attribute*), 1532
SocketHandler (*class in logging.handlers*), 863
socketpair() (*in module socket*), 1185
sockets (*asyncio.Server attribute*), 1134
socketserver
    module, 1529
SocketType (*in module socket*), 1186
soft deprecated, **2335**
SOFT_KEYWORD (*in module token*), 2211
softkwlist (*in module keyword*), 2212
SOH (*in module curses.ascii*), 903
SOL_ALG (*in module socket*), 1182
SOL_RDS (*in module socket*), 1182
SOMAXCONN (*in module socket*), 1180
sort() (*imaplib.IMAP4 method*), 1515
sort() (*list method*), 48
sort_stats() (*pstats.Stats method*), 1949
sortdict() (*in module test.support*), 1906
sorted()
    built-in function, 27
--sort-keys
    json.tool command line option, 1340
sortTestMethodsUsing (*unittest.TestLoader attribute*), 1827
source (*doctest.Example attribute*), 1795
source (*pdb command*), 1941
source (*shlex.shlex attribute*), 1660
SOURCE_DATE_EPOCH, 2220, 2222
source_from_cache() (*in module importlib.util*), 2145
source_hash() (*in module importlib.util*), 2146
SOURCE_SUFFIXES (*in module importlib.machinery*), 2138
source_to_code() (*importlib.abc.InspectLoader static method*), 2134
SourceFileLoader (*class in importlib.machinery*), 2140
sourcehook() (*shlex.shlex method*), 1658
SourcelessFileLoader (*class in importlib.machinery*), 2141
SourceLoader (*class in importlib.abc*), 2135
SP (*in module curses.ascii*), 905
space
    in printf-style formatting, 61, 78
    in string formatting, 125
--spacing
    calendar command line option, 261
span() (*re.Match method*), 150
sparse (*tarfile.TarInfo attribute*), 627
spawn() (*in module pty*), 2273
spawn_python() (*in module test.support.script_helper*), 1914
spawnl() (*in module os*), 747
spawnle() (*in module os*), 747
spawnlp() (*in module os*), 747
spawnlpe() (*in module os*), 747
spawnv() (*in module os*), 747
spawnve() (*in module os*), 747
spawnvp() (*in module os*), 747
spawnvpe() (*in module os*), 747
spec_from_file_location() (*in module importlib.util*), 2146
spec_from_loader() (*in module importlib.util*), 2145
special
    method, 2335
special method, **2335**
SpecialFileError, 620
specified_attributes (*xml.parsers.expat.xmlparser attribute*), 1435
speed() (*in module turtle*), 1624
Spinbox (*class in tkinter.ttk*), 1691
splice() (*in module os*), 710
SPLICE_F_MORE (*in module os*), 710
SPLICE_F_MOVE (*in module os*), 710
SPLICE_F_NONBLOCK (*in module os*), 710
split() (*BaseExceptionGroup method*), 118
split() (*bytearray method*), 71
split() (*bytes method*), 71
split() (*in module os.path*), 487
split() (*in module re*), 143
split() (*in module shlex*), 1656

split() (*re.Pattern method*), 147
split() (*str method*), 57
splitdrive() (*in module os.path*), 487
splitext() (*in module os.path*), 488
splitlines() (*bytearray method*), 75
splitlines() (*bytes method*), 75
splitlines() (*str method*), 58
SplitResult (*class in urllib.parse*), 1483
SplitResultBytes (*class in urllib.parse*), 1484
splitroot() (*in module os.path*), 487
SpooledTemporaryFile (*class in tempfile*), 501
sprintf-style formatting, 60, 77
sqlite3
    module, 553
SQLITE_DBCONFIG_DEFENSIVE (*in module sqlite3*), 559
SQLITE_DBCONFIG_DQS_DDL (*in module sqlite3*), 559
SQLITE_DBCONFIG_DQS_DML (*in module sqlite3*), 559
SQLITE_DBCONFIG_ENABLE_FKEY (*in module sqlite3*), 559
SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER (*in module sqlite3*), 559
SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION (*in module sqlite3*), 559
SQLITE_DBCONFIG_ENABLE_QPSG (*in module sqlite3*), 559
SQLITE_DBCONFIG_ENABLE_TRIGGER (*in module sqlite3*), 559
SQLITE_DBCONFIG_ENABLE_VIEW (*in module sqlite3*), 559
SQLITE_DBCONFIG_LEGACY_ALTER_TABLE (*in module sqlite3*), 559
SQLITE_DBCONFIG_LEGACY_FILE_FORMAT (*in module sqlite3*), 559
SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE (*in module sqlite3*), 559
SQLITE_DBCONFIG_RESET_DATABASE (*in module sqlite3*), 559
SQLITE_DBCONFIG_TRIGGER_EQP (*in module sqlite3*), 559
SQLITE_DBCONFIG_TRUSTED_SCHEMA (*in module sqlite3*), 559
SQLITE_DBCONFIG_WRITABLE_SCHEMA (*in module sqlite3*), 559
SQLITE_DENY (*in module sqlite3*), 558
sqlite_errorcode (*sqlite3.Error attribute*), 575
sqlite_errorname (*sqlite3.Error attribute*), 575
SQLITE_IGNORE (*in module sqlite3*), 558
SQLITE_OK (*in module sqlite3*), 558
sqlite_version (*in module sqlite3*), 558
sqlite_version_info (*in module sqlite3*), 558
sqrt() (*decimal.Context method*), 377
sqrt() (*decimal.Decimal method*), 370
sqrt() (*in module cmath*), 356
sqrt() (*in module math*), 351
SSL, 1204
ssl
    module, 1204
ssl_version (*ftplib.FTP_TLS attribute*), 1506
SSLCertVerificationError, 1207
SSLContext (*class in ssl*), 1222
SSLEOFError, 1207
SSLError, 1207
SSLErrorNumber (*class in ssl*), 1216
SSLKEYLOGFILE, 1206
SSLObject (*class in ssl*), 1237
sslobject_class (*ssl.SSLContext attribute*), 1228
SSLSession (*class in ssl*), 1239
SSLSocket (*class in ssl*), 1217
sslsocket_class (*ssl.SSLContext attribute*), 1227
SSLSyscallError, 1207
SSLv3 (*ssl.TLSVersion attribute*), 1217
SSLWantReadError, 1207
SSLWantWriteError, 1207
SSLZeroReturnError, 1207
st() (*in module turtle*), 1631
ST_ATIME (*in module stat*), 493
st_atime (*os.stat_result attribute*), 727
st_atime_ns (*os.stat_result attribute*), 727
st_birthtime (*os.stat_result attribute*), 727
st_birthtime_ns (*os.stat_result attribute*), 727
st_blksize (*os.stat_result attribute*), 728
st_blocks (*os.stat_result attribute*), 728
st_creator (*os.stat_result attribute*), 728
ST_CTIME (*in module stat*), 493
st_ctime (*os.stat_result attribute*), 727
st_ctime_ns (*os.stat_result attribute*), 727
ST_DEV (*in module stat*), 493
st_dev (*os.stat_result attribute*), 726
st_file_attributes (*os.stat_result attribute*), 728
st_flags (*os.stat_result attribute*), 728
st_fstype (*os.stat_result attribute*), 728
st_gen (*os.stat_result attribute*), 728
ST_GID (*in module stat*), 493
st_gid (*os.stat_result attribute*), 726
ST_INO (*in module stat*), 493
st_ino (*os.stat_result attribute*), 726
ST_MODE (*in module stat*), 493
st_mode (*os.stat_result attribute*), 726
ST_MTIME (*in module stat*), 493
st_mtime (*os.stat_result attribute*), 727
st_mtime_ns (*os.stat_result attribute*), 727
ST_NLINK (*in module stat*), 493
st_nlink (*os.stat_result attribute*), 726
st_rdev (*os.stat_result attribute*), 728
st_reparse_tag (*os.stat_result attribute*), 728
st_rsize (*os.stat_result attribute*), 728
ST_SIZE (*in module stat*), 493

st_size (*os.stat_result attribute*), 726
st_type (*os.stat_result attribute*), 728
ST_UID (*in module stat*), 493
st_uid (*os.stat_result attribute*), 726
stack (*traceback.TracebackException attribute*), 2080
stack viewer, 1709
stack() (*in module inspect*), 2106
stack_effect() (*in module dis*), 2229
stack_size() (*in module _thread*), 1063
stack_size() (*in module threading*), 957
stackable
    streams, 191
StackSummary (*class in traceback*), 2081
stamp() (*in module turtle*), 1623
standard_b64decode() (*in module base64*), 1366
standard_b64encode() (*in module base64*), 1366
standend() (*curses.window method*), 887
standout() (*curses.window method*), 887
STAR (*in module token*), 2209
STAREQUAL (*in module token*), 2210
starmap() (*in module itertools*), 428
starmap() (*multiprocessing.pool.Pool method*), 1001
starmap_async() (*multiprocessing.pool.Pool method*), 1001
Starred (*class in ast*), 2173
start (*range attribute*), 50
start (*slice attribute*), 27
start (*UnicodeError attribute*), 114
start() (*in module tracemalloc*), 1967
start() (*logging.handlers.QueueListener method*), 872
start() (*multiprocessing.managers.BaseManager method*), 993
start() (*multiprocessing.Process method*), 978
start() (*re.Match method*), 150
start() (*threading.Thread method*), 959
start() (*tkinter.ttk.Progressbar method*), 1694
start() (*xml.etree.ElementTree.TreeBuilder method*), 1398
start_color() (*in module curses*), 879
start_new_thread() (*in module _thread*), 1062
start_ns() (*xml.etree.ElementTree.TreeBuilder method*), 1398
start_server() (*in module asyncio*), 1089
start_serving() (*asyncio.Server method*), 1133
start_threads() (*in module test.support.threading_helper*), 1915
start_tls() (*asyncio.loop method*), 1123
start_tls() (*asyncio.StreamWriter method*), 1093
start_unix_server() (*in module asyncio*), 1090
startCDATA() (*xml.sax.handler.LexicalHandler method*), 1426
StartCdataSectionHandler() (*xml.parsers.expat.xmlparser method*), 1437
--start-directory
    unittest-discover command line option, 1807
StartDoctypeDeclHandler() (*xml.parsers.expat.xmlparser method*), 1436
startDocument() (*xml.sax.handler.ContentHandler method*), 1423
startDTD() (*xml.sax.handler.LexicalHandler method*), 1426
startElement() (*xml.sax.handler.ContentHandler method*), 1424
StartElementHandler() (*xml.parsers.expat.xmlparser method*), 1436
startElementNS() (*xml.sax.handler.ContentHandler method*), 1424
STARTF_FORCEOFFFEEDBACK (*in module subprocess*), 1044
STARTF_FORCEONFEEDBACK (*in module subprocess*), 1044
STARTF_USESHOWWINDOW (*in module subprocess*), 1044
STARTF_USESTDHANDLES (*in module subprocess*), 1044
startfile() (*in module os*), 749
StartNamespaceDeclHandler() (*xml.parsers.expat.xmlparser method*), 1437
startPrefixMapping() (*xml.sax.handler.ContentHandler method*), 1423
StartResponse (*class in wsgiref.types*), 1455
startswith() (*bytearray method*), 69
startswith() (*bytes method*), 69
startswith() (*str method*), 59
startTest() (*unittest.TestResult method*), 1829
startTestRun() (*unittest.TestResult method*), 1829
starttls() (*imaplib.IMAP4 method*), 1515
starttls() (*smtplib.SMTP method*), 1522
STARTUPINFO (*class in subprocess*), 1043
stat
    module, 491, 726
stat() (*in module os*), 725
stat() (*os.DirEntry method*), 725
stat() (*pathlib.Path method*), 470
stat() (*poplib.POP3 method*), 1509
stat_result (*class in os*), 726
state() (*tkinter.ttk.Widget method*), 1689
statement, **2335**
    assert, 109
    del, 47, 89
    except, 107
    if, 35
    import, 31, 2111
    raise, 107
    try, 107
    while, 35

static type checker, **2335**
static_order() (*graphlib.TopologicalSorter method*), 341
staticmethod()
    built-in function, 27
Statistic (*class in tracemalloc*), 1970
StatisticDiff (*class in tracemalloc*), 1971
statistics
    module, 402
statistics() (*tracemalloc.Snapshot method*), 1970
StatisticsError, 414
Stats (*class in pstats*), 1949
status (*http.client.HTTPResponse attribute*), 1498
status (*urllib.response.addinfourl attribute*), 1476
status() (*imaplib.IMAP4 method*), 1515
statvfs() (*in module os*), 729
STD_ERROR_HANDLE (*in module subprocess*), 1044
STD_INPUT_HANDLE (*in module subprocess*), 1044
STD_OUTPUT_HANDLE (*in module subprocess*), 1044
stderr (*asyncio.subprocess.Process attribute*), 1106
stderr (*in module sys*), 2014
stderr (*subprocess.CalledProcessError attribute*), 1033
stderr (*subprocess.CompletedProcess attribute*), 1032
stderr (*subprocess.Popen attribute*), 1042
stderr (*subprocess.TimeoutExpired attribute*), 1033
stdev (*statistics.NormalDist attribute*), 414
stdev() (*in module statistics*), 410
stdin (*asyncio.subprocess.Process attribute*), 1106
stdin (*in module sys*), 2014
stdin (*subprocess.Popen attribute*), 1042
stdlib_module_names (*in module sys*), 2015
stdout (*asyncio.subprocess.Process attribute*), 1106
STDOUT (*in module subprocess*), 1032
stdout (*in module sys*), 2014
stdout (*subprocess.CalledProcessError attribute*), 1033
stdout (*subprocess.CompletedProcess attribute*), 1032
stdout (*subprocess.Popen attribute*), 1042
stdout (*subprocess.TimeoutExpired attribute*), 1033
stem (*pathlib.PurePath attribute*), 462
step (*pdb command*), 1940
step (*range attribute*), 50
step (*slice attribute*), 27
step() (*tkinter.ttk.Progressbar method*), 1694
stls() (*poplib.POP3 method*), 1509
stop (*range attribute*), 50
stop (*slice attribute*), 27
stop() (*asyncio.loop method*), 1114
stop() (*in module tracemalloc*), 1968
stop() (*logging.handlers.QueueListener method*), 872
stop() (*tkinter.ttk.Progressbar method*), 1694
stop() (*unittest.TestResult method*), 1829
stop_here() (*bdb.Bdb method*), 1930
STOP_ITERATION (*monitoring event*), 2019
StopAsyncIteration, 112
StopIteration, 112
stopListening() (*in module logging.config*), 846
stopTest() (*unittest.TestResult method*), 1829
stopTestRun() (*unittest.TestResult method*), 1829
storbinary() (*ftplib.FTP method*), 1503
Store (*class in ast*), 2173
store() (*imaplib.IMAP4 method*), 1515
STORE_ACTIONS (*optparse.Option attribute*), 2316
STORE_ATTR (*opcode*), 2237
STORE_DEREF (*opcode*), 2241
STORE_FAST (*opcode*), 2241
STORE_GLOBAL (*opcode*), 2237
STORE_NAME (*opcode*), 2236
STORE_SLICE (*opcode*), 2233
STORE_SUBSCR (*opcode*), 2233
storlines() (*ftplib.FTP method*), 1503
str (*built-in class*), 51
    (see also string), 51
str() (*in module locale*), 1607
str_digits_check_threshold (*sys.int_info attribute*), 2006
strcoll() (*in module locale*), 1607
StreamError, 620
StreamHandler (*class in logging*), 858
StreamReader (*class in asyncio*), 1091
StreamReader (*class in codecs*), 199
streamreader (*codecs.CodecInfo attribute*), 192
StreamReaderWriter (*class in codecs*), 200
StreamRecoder (*class in codecs*), 200
StreamRequestHandler (*class in socketserver*), 1534
streams, 191
    stackable, 191
StreamWriter (*class in asyncio*), 1092
StreamWriter (*class in codecs*), 198
streamwriter (*codecs.CodecInfo attribute*), 192
StrEnum (*class in enum*), 331
strerror (*OSError attribute*), 111
strerror() (*in module os*), 698
strftime() (*datetime.date method*), 219
strftime() (*datetime.datetime method*), 230
strftime() (*datetime.time method*), 235
strftime() (*in module time*), 779
strict
    error handler's name, 194
strict (*csv.Dialect attribute*), 642
STRICT (*enum.FlagBoundary attribute*), 335
strict (*in module email.policy*), 1293
strict_domain (*http.cookiejar.DefaultCookiePolicy attribute*), 1555
strict_errors() (*in module codecs*), 195
strict_ns_domain (*http.cookiejar.DefaultCookiePolicy attribute*), 1555
strict_ns_set_initial_dollar (*http.cookiejar.DefaultCookiePolicy attribute*),

1555
strict_ns_set_path (*http.cookiejar.DefaultCookiePolicy attribute*), 1555
strict_ns_unverifiable (*http.cookiejar.DefaultCookiePolicy attribute*), 1555
strict_rfc2965_unverifiable (*http.cookiejar.DefaultCookiePolicy attribute*), 1555
STRIDED (*inspect.BufferFlags attribute*), 2110
STRIDED_RO (*inspect.BufferFlags attribute*), 2110
STRIDES (*inspect.BufferFlags attribute*), 2110
strides (*memoryview attribute*), 86
string
    format() (*built-in function*), 15
    formatting, printf, 60
    interpolation, printf, 60
    methods, 52
    module, 121
    object, 51
    str (*built-in class*), 51
    str() (*built-in function*), 28
    text sequence type, 51
STRING (*in module token*), 2208
string (*re.Match attribute*), 150
string_at() (*in module ctypes*), 948
StringIO (*class in io*), 773
stringprep
    module, 175
strip() (*bytearray method*), 71
strip() (*bytes method*), 71
strip() (*str method*), 59
strip_dirs() (*pstats.Stats method*), 1949
stripspaces (*curses.textpad.Textbox attribute*), 902
strong reference, **2335**
strptime() (*datetime.datetime class method*), 223
strptime() (*in module time*), 781
strsignal() (*in module signal*), 1258
struct
    module, 183, 1199
Struct (*class in struct*), 190
struct_time (*class in time*), 781
Structure (*class in ctypes*), 952
structures
    C, 183
strxfrm() (*in module locale*), 1607
STX (*in module curses.ascii*), 903
Style (*class in tkinter.ttk*), 1701
Sub (*class in ast*), 2174
SUB (*in module curses.ascii*), 905
sub() (*in module operator*), 448
sub() (*in module re*), 144
sub() (*re.Pattern method*), 147
subdirs (*filecmp.dircmp attribute*), 499
SubElement() (*in module xml.etree.ElementTree*), 1391
subgroup() (*BaseExceptionGroup method*), 118
submit() (*concurrent.futures.Executor method*), 1023
submodule_search_locations (*importlib.machinery.ModuleSpec attribute*), 2143
subn() (*in module re*), 145
subn() (*re.Pattern method*), 147
subnet_of() (*ipaddress.IPv4Network method*), 1581
subnet_of() (*ipaddress.IPv6Network method*), 1583
subnets() (*ipaddress.IPv4Network method*), 1581
subnets() (*ipaddress.IPv6Network method*), 1583
Subnormal (*class in decimal*), 380
suboffsets (*memoryview attribute*), 86
subpad() (*curses.window method*), 887
subprocess
    module, 1030
subprocess_exec() (*asyncio.loop method*), 1130
subprocess_shell() (*asyncio.loop method*), 1132
SubprocessError, 1033
SubprocessProtocol (*class in asyncio*), 1147
SubprocessTransport (*class in asyncio*), 1143
subscribe() (*imaplib.IMAP4 method*), 1516
subscript
    assignment, 47
    operation, 45
Subscript (*class in ast*), 2177
subsequent_indent (*textwrap.TextWrapper attribute*), 172
substitute() (*string.Template method*), 130
subTest() (*unittest.TestCase method*), 1814
subtract() (*collections.Counter method*), 265
subtract() (*decimal.Context method*), 377
subtype (*email.headerregistry.ContentTypeHeader attribute*), 1298
subwin() (*curses.window method*), 887
successful() (*multiprocessing.pool.AsyncResult method*), 1002
suffix (*pathlib.PurePath attribute*), 462
suffix_map (*in module mimetypes*), 1363
suffix_map (*mimetypes.MimeTypes attribute*), 1364
suffixes (*pathlib.PurePath attribute*), 462
suiteClass (*unittest.TestLoader attribute*), 1827
sum()
    built-in function, 28
summarize() (*doctest.DocTestRunner method*), 1798
summarize_address_range() (*in module ipaddress*), 1586
--summary
    trace command line option, 1960
sumprod() (*in module math*), 349
SUNDAY (*in module calendar*), 258
super (*built-in class*), 28
super (*pyclbr.Class attribute*), 2219

supernet() (*ipaddress.IPv4Network method*), 1581
supernet() (*ipaddress.IPv6Network method*), 1583
supernet_of() (*ipaddress.IPv4Network method*), 1581
supernet_of() (*ipaddress.IPv6Network method*), 1583
supports_bytes_environ (*in module os*), 698
supports_dir_fd (*in module os*), 729
supports_effective_ids (*in module os*), 730
supports_fd (*in module os*), 730
supports_follow_symlinks (*in module os*), 730
supports_unicode_filenames (*in module os.path*), 488
SupportsAbs (*class in typing*), 1757
SupportsBytes (*class in typing*), 1757
SupportsComplex (*class in typing*), 1757
SupportsFloat (*class in typing*), 1757
SupportsIndex (*class in typing*), 1757
SupportsInt (*class in typing*), 1757
SupportsRound (*class in typing*), 1757
suppress() (*in module contextlib*), 2058
SuppressCrashReport (*class in test.support*), 1912
surrogateescape
    error handler's name, 194
surrogatepass
    error handler's name, 195
SW_HIDE (*in module subprocess*), 1044
SWAP (*opcode*), 2231
swap_attr() (*in module test.support*), 1907
swap_item() (*in module test.support*), 1908
swapcase() (*bytearray method*), 75
swapcase() (*bytes method*), 75
swapcase() (*str method*), 59
Symbol (*class in symtable*), 2207
SymbolTable (*class in symtable*), 2205
SymbolTableType (*class in symtable*), 2204
symlink() (*in module os*), 730
symlink_to() (*pathlib.Path method*), 476
symmetric_difference() (*frozenset method*), 87
symmetric_difference_update() (*frozenset method*), 88
symtable
    module, 2204
symtable() (*in module symtable*), 2204
SYMTYPE (*in module tarfile*), 621
SYN (*in module curses.ascii*), 904
sync() (*dbm.dumb.dumbdbm method*), 552
sync() (*dbm.gnu.gdbm method*), 550
sync() (*in module os*), 731
sync() (*shelve.Shelf method*), 542
syncdown() (*curses.window method*), 887
synchronized() (*in module multiprocessing.sharedctypes*), 991
SyncManager (*class in multiprocessing.managers*), 994
syncok() (*curses.window method*), 887
syncup() (*curses.window method*), 887
SyntaxErr, 1411
SyntaxError, 112
SyntaxWarning, 117
sys
    module, 23, 1991
sys_version (*http.server.BaseHTTPRequestHandler attribute*), 1539
sysconf() (*in module os*), 757
sysconf_names (*in module os*), 757
sysconfig
    module, 2023
syslog
    module, 2281
syslog() (*in module syslog*), 2282
SysLogHandler (*class in logging.handlers*), 865
sys.monitoring
    module, 2017
system() (*in module os*), 749
system() (*in module platform*), 910
system_alias() (*in module platform*), 910
system_must_validate_cert() (*in module test.support*), 1908
SystemError, 113
SystemExit, 113
systemId (*xml.dom.DocumentType attribute*), 1406
SystemRandom (*class in random*), 397
SystemRandom (*class in secrets*), 685
SystemRoot, 1038

T

-T
    trace command line option, 1959
-t
    calendar command line option, 261
    tarfile command line option, 632
    trace command line option, 1959
    unittest-discover command line option, 1807
    zipfile command line option, 617
T_FMT (*in module locale*), 1604
T_FMT_AMPM (*in module locale*), 1604
--tab
    json.tool command line option, 1341
TAB (*in module curses.ascii*), 903
tab() (*tkinter.ttk.Notebook method*), 1693
TabError, 113
tabnanny
    module, 2216
tabs() (*tkinter.ttk.Notebook method*), 1693
tabsize (*textwrap.TextWrapper attribute*), 171
tabular
    data, 637

tag (*xml.etree.ElementTree.Element attribute*), 1393
tag_bind() (*tkinter.ttk.Treeview method*), 1700
tag_configure() (*tkinter.ttk.Treeview method*), 1701
tag_has() (*tkinter.ttk.Treeview method*), 1701
tagName (*xml.dom.Element attribute*), 1407
tail (*xml.etree.ElementTree.Element attribute*), 1393
take_snapshot() (*in module tracemalloc*), 1968
takewhile() (*in module itertools*), 428
tan() (*in module cmath*), 356
tan() (*in module math*), 352
tanh() (*in module cmath*), 357
tanh() (*in module math*), 353
tar_filter() (*in module tarfile*), 629
TarError, 620
tarfile
    module, 618
TarFile (*class in tarfile*), 622
tarfile command line option
    -c, 632
    --create, 632
    -e, 632
    --extract, 632
    --filter, 633
    -l, 632
    --list, 632
    -t, 632
    --test, 632
    -v, 633
    --verbose, 633
target (*xml.dom.ProcessingInstruction attribute*), 1410
TarInfo (*class in tarfile*), 626
tarinfo (*tarfile.FilterError attribute*), 620
Task (*class in asyncio*), 1084
task_done() (*asyncio.Queue method*), 1109
task_done() (*multiprocessing.JoinableQueue method*), 983
task_done() (*queue.Queue method*), 1056
TaskGroup (*class in asyncio*), 1074
tau (*in module cmath*), 358
tau (*in module math*), 354
tb_locals (*unittest.TestResult attribute*), 1829
tbreak (*pdb command*), 1939
tcdrain() (*in module termios*), 2270
tcflow() (*in module termios*), 2270
tcflush() (*in module termios*), 2270
tcgetattr() (*in module termios*), 2270
tcgetpgrp() (*in module os*), 711
tcgetwinsize() (*in module termios*), 2271
Tcl() (*in module tkinter*), 1666
TCPServer (*class in socketserver*), 1529
TCSADRAIN (*in module termios*), 2270
TCSAFLUSH (*in module termios*), 2270
TCSANOW (*in module termios*), 2270
tcsendbreak() (*in module termios*), 2270
tcsetattr() (*in module termios*), 2270
tcsetpgrp() (*in module os*), 711
tcsetwinsize() (*in module termios*), 2271
tearDown() (*unittest.TestCase method*), 1814
tearDownClass() (*unittest.TestCase method*), 1814
tee() (*in module itertools*), 429
teleport() (*in module turtle*), 1620
tell() (*io.IOBase method*), 765
tell() (*io.TextIOBase method*), 771
tell() (*io.TextIOWrapper method*), 773
tell() (*mmap.mmap method*), 1267
tell() (*sqlite3.Blob method*), 575
tell() (*wave.Wave_read method*), 1590
tell() (*wave.Wave_write method*), 1591
TEMP, 503
temp_cwd() (*in module test.support.os_helper*), 1917
temp_dir() (*in module test.support.os_helper*), 1918
temp_umask() (*in module test.support.os_helper*), 1918
tempdir (*in module tempfile*), 504
tempfile
    module, 500
Template (*class in string*), 130
template (*string.Template attribute*), 131
temporary
    file, 500
    file name, 500
temporary (*bdb.Breakpoint attribute*), 1928
TemporaryDirectory (*class in tempfile*), 502
TemporaryFile() (*in module tempfile*), 500
teredo (*ipaddress.IPv6Address attribute*), 1577
TERM, 879
termattrs() (*in module curses*), 879
terminal_size (*class in os*), 712
terminate() (*asyncio.subprocess.Process method*), 1106
terminate() (*asyncio.SubprocessTransport method*), 1146
terminate() (*multiprocessing.pool.Pool method*), 1001
terminate() (*multiprocessing.Process method*), 979
terminate() (*subprocess.Popen method*), 1041
terminator (*logging.StreamHandler attribute*), 858
termios
    module, 2270
termname() (*in module curses*), 879
test
    module, 1900
--test
    tarfile command line option, 632
    zipfile command line option, 617
test (*doctest.DocTestFailure attribute*), 1802
test (*doctest.UnexpectedException attribute*), 1802
TEST_DATA_DIR (*in module test.support*), 1904
TEST_HOME_DIR (*in module test.support*), 1904
TEST_HTTP_URL (*in module test.support*), 1905

TEST_SUPPORT_DIR (*in module test.support*), 1904
TestCase (*class in unittest*), 1813
TestFailed, 1903
testfile() (*in module doctest*), 1790
TESTFN (*in module test.support.os_helper*), 1916
TESTFN_NONASCII (*in module test.support.os_helper*), 1916
TESTFN_UNDECODABLE (*in module test.support.os_helper*), 1916
TESTFN_UNENCODABLE (*in module test.support.os_helper*), 1916
TESTFN_UNICODE (*in module test.support.os_helper*), 1916
TestLoader (*class in unittest*), 1825
testMethodPrefix (*unittest.TestLoader attribute*), 1827
testmod() (*in module doctest*), 1791
testNamePatterns (*unittest.TestLoader attribute*), 1827
test.regrtest
    module, 1902
TestResult (*class in unittest*), 1828
TestResults (*class in doctest*), 1797
testsource() (*in module doctest*), 1801
testsRun (*unittest.TestResult attribute*), 1828
TestSuite (*class in unittest*), 1824
test.support
    module, 1903
test.support.bytecode_helper
    module, 1915
test.support.import_helper
    module, 1918
test.support.os_helper
    module, 1916
test.support.script_helper
    module, 1913
test.support.socket_helper
    module, 1913
test.support.threading_helper
    module, 1915
test.support.warnings_helper
    module, 1919
testzip() (*zipfile.ZipFile method*), 610
Text (*class in typing*), 1768
text (*SyntaxError attribute*), 113
text (*traceback.TracebackException attribute*), 2080
text (*xml.etree.ElementTree.Element attribute*), 1393
text encoding, **2335**
text file, **2335**
text mode, 23
text_encoding() (*in module io*), 762
text_factory (*sqlite3.Connection attribute*), 570
Textbox (*class in curses.textpad*), 902
TextCalendar (*class in calendar*), 255
textdomain() (*in module gettext*), 1593
textdomain() (*in module locale*), 1610
textinput() (*in module turtle*), 1641
TextIO (*class in typing*), 1758
TextIOBase (*class in io*), 770
TextIOWrapper (*class in io*), 771
TextTestResult (*class in unittest*), 1830
TextTestRunner (*class in unittest*), 1830
textwrap
    module, 169
TextWrapper (*class in textwrap*), 171
TFD_CLOEXEC (*in module os*), 739
TFD_NONBLOCK (*in module os*), 738
TFD_TIMER_ABSTIME (*in module os*), 739
TFD_TIMER_CANCEL_ON_SET (*in module os*), 739
theme_create() (*tkinter.ttk.Style method*), 1704
theme_names() (*tkinter.ttk.Style method*), 1705
theme_settings() (*tkinter.ttk.Style method*), 1704
theme_use() (*tkinter.ttk.Style method*), 1705
THOUSEP (*in module locale*), 1605
Thread (*class in threading*), 959
thread() (*imaplib.IMAP4 method*), 1516
thread_info (*in module sys*), 2015
thread_time() (*in module time*), 783
thread_time_ns() (*in module time*), 783
ThreadedChildWatcher (*class in asyncio*), 1159
threading
    module, 955
threading_cleanup() (*in module test.support.threading_helper*), 1915
threading_setup() (*in module test.support.threading_helper*), 1915
ThreadingHTTPServer (*class in http.server*), 1538
ThreadingMixIn (*class in socketserver*), 1530
ThreadingMock (*class in unittest.mock*), 1852
ThreadingTCPServer (*class in socketserver*), 1531
ThreadingUDPServer (*class in socketserver*), 1531
ThreadingUnixDatagramServer (*class in socketserver*), 1531
ThreadingUnixStreamServer (*class in socketserver*), 1531
ThreadPool (*class in multiprocessing.pool*), 1007
ThreadPoolExecutor (*class in concurrent.futures*), 1025
threads
    POSIX, 1062
threadsafety (*in module sqlite3*), 558
THURSDAY (*in module calendar*), 258
ticket_lifetime_hint (*ssl.SSLSession attribute*), 1239
tigetflag() (*in module curses*), 879
tigetnum() (*in module curses*), 880
tigetstr() (*in module curses*), 880
TILDE (*in module token*), 2210

tilt() (*in module turtle*), 1633
tiltangle() (*in module turtle*), 1633
time
    module, 774
time (*class in datetime*), 232
time (*ssl.SSLSession attribute*), 1239
time (*uuid.UUID attribute*), 1526
time() (*asyncio.loop method*), 1116
time() (*datetime.datetime method*), 226
time() (*in module time*), 782
Time2Internaldate() (*in module imaplib*), 1511
time_hi_version (*uuid.UUID attribute*), 1526
time_low (*uuid.UUID attribute*), 1526
time_mid (*uuid.UUID attribute*), 1526
time_ns() (*in module time*), 783
timedelta (*class in datetime*), 212
TimedRotatingFileHandler (*class in logging.handlers*), 862
timegm() (*in module calendar*), 258
timeit
    module, 1953
timeit command line option
    -h, 1956
    --help, 1956
    -n, 1956
    --number, 1956
    -p, 1956
    --process, 1956
    -r, 1956
    --repeat, 1956
    -s, 1956
    --setup, 1956
    -u, 1956
    --unit, 1956
    -v, 1956
    --verbose, 1956
timeit() (*in module timeit*), 1954
timeit() (*timeit.Timer method*), 1955
timeout, 1179
Timeout (*class in asyncio*), 1079
timeout (*socketserver.BaseServer attribute*), 1532
timeout (*ssl.SSLSession attribute*), 1239
timeout (*subprocess.TimeoutExpired attribute*), 1033
timeout() (*curses.window method*), 887
timeout() (*in module asyncio*), 1078
timeout_at() (*in module asyncio*), 1079
TIMEOUT_MAX (*in module _thread*), 1063
TIMEOUT_MAX (*in module threading*), 958
TimeoutError, 116, 980, 1030, 1111
TimeoutExpired, 1033
Timer (*class in threading*), 968
Timer (*class in timeit*), 1954
timerfd_create() (*in module os*), 736
timerfd_gettime() (*in module os*), 738
timerfd_gettime_ns() (*in module os*), 738
timerfd_settime() (*in module os*), 737
timerfd_settime_ns() (*in module os*), 738
TimerHandle (*class in asyncio*), 1132
times() (*in module os*), 750
TIMESTAMP (*py_compile.PycInvalidationMode attribute*), 2220
timestamp() (*datetime.datetime method*), 227
timetuple() (*datetime.date method*), 218
timetuple() (*datetime.datetime method*), 227
timetz() (*datetime.datetime method*), 226
timezone (*class in datetime*), 243
timezone (*in module time*), 786
--timing
    trace command line option, 1960
title() (*bytearray method*), 75
title() (*bytes method*), 75
title() (*in module turtle*), 1644
title() (*str method*), 59
Tk, 1663
Tk (*class in tkinter*), 1665
tk (*tkinter.Tk attribute*), 1665
Tk Option Data Types, 1674
Tkinter, 1663
tkinter
    module, 1663
tkinter.colorchooser
    module, 1677
tkinter.commondialog
    module, 1682
tkinter.dnd
    module, 1685
tkinter.filedialog
    module, 1679
tkinter.font
    module, 1677
tkinter.messagebox
    module, 1682
tkinter.scrolledtext
    module, 1684
tkinter.simpledialog
    module, 1679
tkinter.ttk
    module, 1686
TLS, 1204
TLSv1 (*ssl.TLSVersion attribute*), 1217
TLSv1_1 (*ssl.TLSVersion attribute*), 1217
TLSv1_2 (*ssl.TLSVersion attribute*), 1217
TLSv1_3 (*ssl.TLSVersion attribute*), 1217
TLSVersion (*class in ssl*), 1217
tm_gmtoff (*time.struct_time attribute*), 782
tm_hour (*time.struct_time attribute*), 782
tm_isdst (*time.struct_time attribute*), 782
tm_mday (*time.struct_time attribute*), 782

tm_min (*time.struct_time attribute*), 782
tm_mon (*time.struct_time attribute*), 782
tm_sec (*time.struct_time attribute*), 782
tm_wday (*time.struct_time attribute*), 782
tm_yday (*time.struct_time attribute*), 782
tm_year (*time.struct_time attribute*), 782
tm_zone (*time.struct_time attribute*), 782
TMP, 503
TMPDIR, 503
TO_BOOL (*opcode*), 2232
to_bytes() (*int method*), 39
to_eng_string() (*decimal.Context method*), 377
to_eng_string() (*decimal.Decimal method*), 370
to_integral() (*decimal.Decimal method*), 370
to_integral_exact() (*decimal.Context method*), 377
to_integral_exact() (*decimal.Decimal method*), 370
to_integral_value() (*decimal.Decimal method*), 370
to_sci_string() (*decimal.Context method*), 377
to_thread() (*in module asyncio*), 1082
ToASCII() (*in module encodings.idna*), 208
tobuf() (*tarfile.TarInfo method*), 626
tobytes() (*array.array method*), 296
tobytes() (*memoryview method*), 81
today() (*datetime.date class method*), 215
today() (*datetime.datetime class method*), 221
tofile() (*array.array method*), 296
tok_name (*in module token*), 2208
token
    module, 2208
Token (*class in contextvars*), 1059
token (*shlex.shlex attribute*), 1660
token_bytes() (*in module secrets*), 686
token_hex() (*in module secrets*), 686
token_urlsafe() (*in module secrets*), 686
TokenError, 2214
tokenize
    module, 2212
tokenize command line option
    -e, 2214
    --exact, 2214
    -h, 2214
    --help, 2214
tokenize() (*in module tokenize*), 2213
tolist() (*array.array method*), 296
tolist() (*memoryview method*), 82
TOMLDecodeError, 664
tomllib
    module, 664
toordinal() (*datetime.date method*), 218
toordinal() (*datetime.datetime method*), 227
top() (*curses.panel.Panel method*), 908
top() (*poplib.POP3 method*), 1509
top_panel() (*in module curses.panel*), 907
--top-level-directory
    unittest-discover command line option, 1807
TopologicalSorter (*class in graphlib*), 339
toprettyxml() (*xml.dom.minidom.Node method*), 1415
toreadonly() (*memoryview method*), 82
tostring() (*in module xml.etree.ElementTree*), 1391
tostringlist() (*in module xml.etree.ElementTree*), 1391
total() (*collections.Counter method*), 266
total_changes (*sqlite3.Connection attribute*), 570
total_nframe (*tracemalloc.Traceback attribute*), 1972
total_ordering() (*in module functools*), 439
total_seconds() (*datetime.timedelta method*), 214
touch() (*pathlib.Path method*), 476
touchline() (*curses.window method*), 887
touchwin() (*curses.window method*), 887
tounicode() (*array.array method*), 296
ToUnicode() (*in module encodings.idna*), 208
towards() (*in module turtle*), 1624
toxml() (*xml.dom.minidom.Node method*), 1414
tparm() (*in module curses*), 880
trace
    module, 1959
--trace
    trace command line option, 1959
Trace (*class in trace*), 1960
Trace (*class in tracemalloc*), 1971
trace command line option
    -C, 1960
    -c, 1959
    --count, 1959
    --coverdir, 1960
    -f, 1960
    --file, 1960
    -g, 1960
    --help, 1959
    --ignore-dir, 1960
    --ignore-module, 1960
    -l, 1959
    --listfuncs, 1959
    -m, 1960
    --missing, 1960
    --no-report, 1960
    -R, 1960
    -r, 1959
    --report, 1959
    -s, 1960
    --summary, 1960
    -T, 1959
    -t, 1959

--timing, 1960
--trace, 1959
--trackcalls, 1959
--version, 1959
trace function, 957, 2003, 2011
trace() (*in module inspect*), 2107
trace_dispatch() (*bdb.Bdb method*), 1929
traceback
module, 2076
object, 1996, 2076
Traceback (*class in inspect*), 2105
Traceback (*class in tracemalloc*), 1972
traceback (*tracemalloc.Statistic attribute*), 1971
traceback (*tracemalloc.StatisticDiff attribute*), 1971
traceback (*tracemalloc.Trace attribute*), 1971
traceback_limit (*tracemalloc.Snapshot attribute*), 1970
traceback_limit (*wsgiref.handlers.BaseHandler attribute*), 1454
TracebackException (*class in traceback*), 2079
tracebacklimit (*in module sys*), 2015
TracebackType (*class in types*), 309
tracemalloc
module, 1961
tracer() (*in module turtle*), 1639
traces (*tracemalloc.Snapshot attribute*), 1970
--trackcalls
trace command line option, 1959
transfercmd() (*ftplib.FTP method*), 1503
transient_internet() (*in module test.support.socket_helper*), 1913
translate() (*bytearray method*), 69
translate() (*bytes method*), 69
translate() (*in module fnmatch*), 509
translate() (*in module glob*), 507
translate() (*str method*), 60
translation() (*in module gettext*), 1595
transport (*asyncio.StreamWriter attribute*), 1092
Transport (*class in asyncio*), 1143
Transport Layer Security, 1204
Traversable (*class in importlib.abc*), 2137
Traversable (*class in importlib.resources.abc*), 2154
TraversableResources (*class in importlib.abc*), 2138
TraversableResources (*class in importlib.resources.abc*), 2155
TreeBuilder (*class in xml.etree.ElementTree*), 1398
Treeview (*class in tkinter.ttk*), 1697
triangular() (*in module random*), 395
tries (*doctest.DocTestRunner attribute*), 1799
triple-quoted string, **2335**
True, 35, 43
true, 35
True (*built-in variable*), 33
truediv() (*in module operator*), 448
trunc() (*in module math*), 37, 350
truncate() (*in module os*), 731
truncate() (*io.IOBase method*), 765
truth
value, 35
truth() (*in module operator*), 447
try
statement, 107
Try (*class in ast*), 2185
TryStar (*class in ast*), 2186
ttk, 1686
tty
I/O control, 2270
module, 2271
ttyname() (*in module os*), 711
TUESDAY (*in module calendar*), 258
tuple
object, 47, 49
tuple (*built-in class*), 49
Tuple (*class in ast*), 2172
Tuple (*in module typing*), 1767
turtle
module, 1611
Turtle (*class in turtle*), 1644
turtledemo
module, 1649
turtles() (*in module turtle*), 1643
TurtleScreen (*class in turtle*), 1644
turtlesize() (*in module turtle*), 1632
type, **2336**
Boolean, 7
built-in function, 102
object, 29
operations on dictionary, 89
operations on list, 47
union, 99
--type
calendar command line option, 261
type (*built-in class*), 29
Type (*class in typing*), 1767
type (*optparse.Option attribute*), 2303
type (*socket.socket attribute*), 1199
type (*tarfile.TarInfo attribute*), 627
type (*urllib.request.Request attribute*), 1462
type alias, **2336**
type hint, **2336**
TYPE_ALIAS (*symtable.SymbolTableType attribute*), 2205
type_check_only() (*in module typing*), 1763
TYPE_CHECKER (*optparse.Option attribute*), 2315
TYPE_CHECKING (*in module typing*), 1766
type_comment (*ast.arg attribute*), 2196
type_comment (*ast.Assign attribute*), 2180

type_comment (*ast.For attribute*), 2184
type_comment (*ast.FunctionDef attribute*), 2195
type_comment (*ast.With attribute*), 2187
TYPE_COMMENT (*in module token*), 2211
TYPE_IGNORE (*in module token*), 2211
TYPE_PARAMETERS (*symtable.SymbolTableType attribute*), 2205
TYPE_VARIABLE (*symtable.SymbolTableType attribute*), 2205
typeahead() (*in module curses*), 880
TypeAlias (*class in ast*), 2182
TypeAlias (*in module typing*), 1733
TypeAliasType (*class in typing*), 1749
typecode (*array.array attribute*), 294
typecodes (*in module array*), 294
TYPED_ACTIONS (*optparse.Option attribute*), 2316
typed_subpart_iterator() (*in module email.iterators*), 1330
TypedDict (*class in typing*), 1753
TypeError, 114
TypeGuard (*in module typing*), 1741
TypeIs (*in module typing*), 1740
types
    built-in, 35
    immutable sequence, 47
    module, 102, 305
    mutable sequence, 47
    operations on integer, 38
    operations on mapping, 89
    operations on numeric, 37
    operations on sequence, 45, 47
TYPES (*optparse.Option attribute*), 2315
types_map (*in module mimetypes*), 1363
types_map (*mimetypes.MimeTypes attribute*), 1364
types_map_inv (*mimetypes.MimeTypes attribute*), 1364
TypeVar (*class in ast*), 2193
TypeVar (*class in typing*), 1743
TypeVarTuple (*class in ast*), 2194
TypeVarTuple (*class in typing*), 1745
typing
    module, 1719
TZ, 783, 784
tzinfo (*class in datetime*), 236
tzinfo (*datetime.datetime attribute*), 224
tzinfo (*datetime.time attribute*), 233
tzname (*in module time*), 786
tzname() (*datetime.datetime method*), 227
tzname() (*datetime.time method*), 235
tzname() (*datetime.timezone method*), 243
tzname() (*datetime.tzinfo method*), 237
TZPATH (*in module zoneinfo*), 253
tzset() (*in module time*), 783

U

-u
    timeit command line option, 1956
    uuid command line option, 1528
U (*in module re*), 141
UAdd (*class in ast*), 2174
ucd_3_2_0 (*in module unicodedata*), 174
udata (*select.kevent attribute*), 1249
UDPServer (*class in socketserver*), 1529
UF_APPEND (*in module stat*), 496
UF_COMPRESSED (*in module stat*), 496
UF_DATAVAULT (*in module stat*), 496
UF_HIDDEN (*in module stat*), 496
UF_IMMUTABLE (*in module stat*), 495
UF_NODUMP (*in module stat*), 495
UF_NOUNLINK (*in module stat*), 496
UF_OPAQUE (*in module stat*), 496
UF_SETTABLE (*in module stat*), 495
UF_TRACKED (*in module stat*), 496
UID (*class in plistlib*), 668
uid (*tarfile.TarInfo attribute*), 627
uid() (*imaplib.IMAP4 method*), 1516
uidl() (*poplib.POP3 method*), 1509
ulp() (*in module math*), 350
umask() (*in module os*), 698
unalias (*pdb command*), 1943
uname (*tarfile.TarInfo attribute*), 627
uname() (*in module os*), 698
uname() (*in module platform*), 910
UNARY_INVERT (*opcode*), 2232
UNARY_NEGATIVE (*opcode*), 2232
UNARY_NOT (*opcode*), 2232
UnaryOp (*class in ast*), 2174
UnboundLocalError, 114
unbuffered I/O, 23
UNC paths
    and os.makedirs(), 719
uncancel() (*asyncio.Task method*), 1087
UNCHECKED_HASH (*py_compile.PycInvalidationMode attribute*), 2220
unconsumed_tail (*zlib.Decompress attribute*), 590
unctrl() (*in module curses*), 880
unctrl() (*in module curses.ascii*), 906
Underflow (*class in decimal*), 380
undisplay (*pdb command*), 1942
undo() (*in module turtle*), 1623
undobufferentries() (*in module turtle*), 1636
undoc_header (*cmd.Cmd attribute*), 1653
unescape() (*in module html*), 1373
unescape() (*in module xml.sax.saxutils*), 1427
UnexpectedException, 1802
unexpectedSuccesses (*unittest.TestResult attribute*), 1828
unfreeze() (*in module gc*), 2089

unget_wch() (*in module curses*), 880
ungetch() (*in module curses*), 880
ungetch() (*in module msvcrt*), 2252
ungetmouse() (*in module curses*), 880
ungetwch() (*in module msvcrt*), 2252
unhexlify() (*in module binascii*), 1370
Unicode, 173, 191
    database, 173
UNICODE (*in module re*), 141
unicodedata
    module, 173
UnicodeDecodeError, 114
UnicodeEncodeError, 114
UnicodeError, 114
UnicodeTranslateError, 114
UnicodeWarning, 117
unidata_version (*in module unicodedata*), 174
unified_diff() (*in module difflib*), 160
uniform() (*in module random*), 395
UnimplementedFileMode, 1493
Union
    object, 99
union
    type, 99
Union (*class in ctypes*), 952
Union (*in module typing*), 1734
union() (*frozenset method*), 87
UnionType (*class in types*), 309
UNIQUE (*enum.EnumCheck attribute*), 334
unique() (*in module enum*), 337
--unit
    timeit command line option, 1956
unittest
    module, 1803
unittest command line option
    -b, 1806
    --buffer, 1806
    -c, 1806
    --catch, 1806
    --durations, 1806
    -f, 1806
    --failfast, 1806
    -k, 1806
    --locals, 1806
unittest-discover command line option
    -p, 1807
    --pattern, 1807
    -s, 1807
    --start-directory, 1807
    -t, 1807
    --top-level-directory, 1807
    -v, 1807
    --verbose, 1807
unittest.mock
    module, 1835
universal newlines, **2336**
    bytearray.splitlines method, 75
    bytes.splitlines method, 75
    csv.reader function, 637
    importlib.abc.InspectLoader.get_source method, 2134
    io.IncrementalNewlineDecoder class, 773
    io.TextIOWrapper class, 771
    open() built-in function, 22
    str.splitlines method, 58
    subprocess module, 1034
UNIX
    file control, 2274
    I/O control, 2274
unix_dialect (*class in csv*), 640
unix_shell (*in module test.support*), 1903
UnixDatagramServer (*class in socketserver*), 1529
UnixStreamServer (*class in socketserver*), 1529
unknown (*uuid.SafeUUID attribute*), 1525
unknown_decl() (*html.parser.HTMLParser method*), 1376
unknown_open() (*urllib.request.BaseHandler method*), 1465
unknown_open() (*urllib.request.UnknownHandler method*), 1470
UnknownHandler (*class in urllib.request*), 1462
UnknownProtocol, 1493
UnknownTransferEncoding, 1493
unlink() (*in module os*), 731
unlink() (*in module test.support.os_helper*), 1918
unlink() (*multiprocessing.shared_memory.SharedMemory method*), 1018
unlink() (*pathlib.Path method*), 478
unlink() (*xml.dom.minidom.Node method*), 1414
unload() (*in module test.support.import_helper*), 1919
unlock() (*mailbox.Babyl method*), 1350
unlock() (*mailbox.Mailbox method*), 1344
unlock() (*mailbox.Maildir method*), 1347
unlock() (*mailbox.mbox method*), 1348
unlock() (*mailbox.MH method*), 1349
unlock() (*mailbox.MMDF method*), 1351
unlockpt() (*in module os*), 711
UNNAMED_SECTION (*in module configparser*), 662
Unpack (*in module typing*), 1742
unpack() (*in module struct*), 184
unpack() (*struct.Struct method*), 190
unpack_archive() (*in module shutil*), 518
UNPACK_EX (*opcode*), 2237
unpack_from() (*in module struct*), 184
unpack_from() (*struct.Struct method*), 190
UNPACK_SEQUENCE (*opcode*), 2236

unparse() (*in module ast*), 2199
unparsedEntityDecl() (*xml.sax.handler.DTDHandler method*), 1425
UnparsedEntityDeclHandler() (*xml.parsers.expat.xmlparser method*), 1436
Unpickler (*class in pickle*), 528
UnpicklingError, 526
unquote() (*in module email.utils*), 1327
unquote() (*in module urllib.parse*), 1484
unquote_plus() (*in module urllib.parse*), 1485
unquote_to_bytes() (*in module urllib.parse*), 1485
unraisablehook() (*in module sys*), 2016
unregister() (*in module atexit*), 2075
unregister() (*in module codecs*), 192
unregister() (*in module faulthandler*), 1934
unregister() (*select.devpoll method*), 1245
unregister() (*select.epoll method*), 1246
unregister() (*selectors.BaseSelector method*), 1251
unregister() (*select.poll method*), 1247
unregister_archive_format() (*in module shutil*), 518
unregister_dialect() (*in module csv*), 638
unregister_unpack_format() (*in module shutil*), 519
unsafe (*uuid.SafeUUID attribute*), 1525
unselect() (*imaplib.IMAP4 method*), 1516
unset() (*test.support.os_helper.EnvironmentVarGuard method*), 1917
unsetenv() (*in module os*), 698
unshare() (*in module os*), 698
UnstructuredHeader (*class in email.headerregistry*), 1296
unsubscribe() (*imaplib.IMAP4 method*), 1516
UnsupportedOperation, 457, 762
until (*pdb command*), 1940
untokenize() (*in module tokenize*), 2213
untouchwin() (*curses.window method*), 887
unused_data (*bz2.BZ2Decompressor attribute*), 598
unused_data (*lzma.LZMADecompressor attribute*), 603
unused_data (*zlib.Decompress attribute*), 589
unverifiable (*urllib.request.Request attribute*), 1463
unwrap() (*in module inspect*), 2103
unwrap() (*in module urllib.parse*), 1482
unwrap() (*ssl.SSLSocket method*), 1220
up (*pdb command*), 1939
up() (*in module turtle*), 1626
update() (*collections.Counter method*), 266
update() (*dict method*), 91
update() (*frozenset method*), 88
update() (*hashlib.hash method*), 673
update() (*hmac.HMAC method*), 684
update() (*http.cookies.Morsel method*), 1547
update() (*in module turtle*), 1639
update() (*mailbox.Mailbox method*), 1344
update() (*mailbox.Maildir method*), 1347
update() (*trace.CoverageResults method*), 1961
update_abstractmethods() (*in module abc*), 2074
update_authenticated() (*urllib.request.HTTPPasswordMgrWithPriorAuth method*), 1468
update_lines_cols() (*in module curses*), 880
update_panels() (*in module curses.panel*), 907
update_visible() (*mailbox.BabylMessage method*), 1358
update_wrapper() (*in module functools*), 444
upgrade_dependencies() (*venv.EnvBuilder method*), 1980
upper() (*bytearray method*), 76
upper() (*bytes method*), 76
upper() (*str method*), 60
urandom() (*in module os*), 759
URL, 1477, 1487, 1538
    parsing, 1477
    relative, 1477
url (*http.client.HTTPResponse attribute*), 1498
url (*urllib.error.HTTPError attribute*), 1486
url (*urllib.response.addinfourl attribute*), 1476
url (*xmlrpc.client.ProtocolError attribute*), 1563
url2pathname() (*in module urllib.request*), 1459
urlcleanup() (*in module urllib.request*), 1474
urldefrag() (*in module urllib.parse*), 1481
urlencode() (*in module urllib.parse*), 1485
URLError, 1486
urljoin() (*in module urllib.parse*), 1481
urllib
    module, 1457
urllib.error
    module, 1486
urllib.parse
    module, 1477
urllib.request
    module, 1457, 1492
urllib.response
    module, 1476
urllib.robotparser
    module, 1487
urlopen() (*in module urllib.request*), 1457
URLopener (*class in urllib.request*), 1474
urlparse() (*in module urllib.parse*), 1477
urlretrieve() (*in module urllib.request*), 1473
urlsafe_b64decode() (*in module base64*), 1366
urlsafe_b64encode() (*in module base64*), 1366
urlsplit() (*in module urllib.parse*), 1480
urlunparse() (*in module urllib.parse*), 1480
urlunsplit() (*in module urllib.parse*), 1481
urn (*uuid.UUID attribute*), 1526
US (*in module curses.ascii*), 905
use_default_colors() (*in module curses*), 881

use_env() (*in module curses*), 880
use_rawinput (*cmd.Cmd attribute*), 1653
use_tool_id() (*in module sys.monitoring*), 2018
UseForeignDTD() (*xml.parsers.expat.xmlparser method*), 1434
USER, 873
user
    effective id, 693
    id, 695
    id, setting, 698
user() (*poplib.POP3 method*), 1508
USER_BASE (*in module site*), 2113
user_call() (*bdb.Bdb method*), 1930
user_exception() (*bdb.Bdb method*), 1930
user_line() (*bdb.Bdb method*), 1930
user_return() (*bdb.Bdb method*), 1930
USER_SITE (*in module site*), 2113
--user-base
    site command line option, 2114
usercustomize
    module, 2113
UserDict (*class in collections*), 279
UserList (*class in collections*), 279
USERNAME, 483, 694, 873
username (*email.headerregistry.Address attribute*), 1300
USERPROFILE, 483
userptr() (*curses.panel.Panel method*), 908
--user-site
    site command line option, 2114
UserString (*class in collections*), 280
UserWarning, 116
USTAR_FORMAT (*in module tarfile*), 621
USub (*class in ast*), 2174
UTC, 775
utc (*datetime.timezone attribute*), 243
UTC (*in module datetime*), 210
utcfromtimestamp() (*datetime.datetime class method*), 222
utcnow() (*datetime.datetime class method*), 221
utcoffset() (*datetime.datetime method*), 227
utcoffset() (*datetime.time method*), 235
utcoffset() (*datetime.timezone method*), 243
utcoffset() (*datetime.tzinfo method*), 236
utctimetuple() (*datetime.datetime method*), 227
utf8 (*email.policy.EmailPolicy attribute*), 1291
utf8() (*poplib.POP3 method*), 1509
utf8_enabled (*imaplib.IMAP4 attribute*), 1516
utf8_mode (*sys.flags attribute*), 1998
utime() (*in module os*), 731
uuid
    module, 1524
--uuid
    uuid command line option, 1528
UUID (*class in uuid*), 1525
uuid command line option
    -h, 1528
    --help, 1528
    -N, 1528
    -n, 1528
    --name, 1528
    --namespace, 1528
    -u, 1528
    --uuid, 1528
uuid1, 1527
uuid1() (*in module uuid*), 1527
uuid3, 1527
uuid3() (*in module uuid*), 1527
uuid4, 1527
uuid4() (*in module uuid*), 1527
uuid5, 1527
uuid5() (*in module uuid*), 1527

V

-v
    python--m-sqlite3-[-h]-[-v]-[filename]-[sql] command line option, 577
    tarfile command line option, 633
    timeit command line option, 1956
    unittest-discover command line option, 1807
v4_int_to_packed() (*in module ipaddress*), 1586
v6_int_to_packed() (*in module ipaddress*), 1586
valid_signals() (*in module signal*), 1258
validator() (*in module wsgiref.validate*), 1450
value
    truth, 35
value (*ctypes._SimpleCData attribute*), 949
value (*enum.Enum attribute*), 327
value (*http.cookiejar.Cookie attribute*), 1556
value (*http.cookies.Morsel attribute*), 1547
value (*StopIteration attribute*), 112
value (*xml.dom.Attr attribute*), 1409
Value() (*in module multiprocessing*), 990
Value() (*in module multiprocessing.sharedctypes*), 991
Value() (*multiprocessing.managers.SyncManager method*), 995
value_decode() (*http.cookies.BaseCookie method*), 1546
value_encode() (*http.cookies.BaseCookie method*), 1546
ValueError, 114
valuerefs() (*weakref.WeakValueDictionary method*), 299
values
    Boolean, 43
Values (*class in optparse*), 2302
values() (*contextvars.Context method*), 1060

values() (*dict method*), 91
values() (*email.message.EmailMessage method*), 1274
values() (*email.message.Message method*), 1314
values() (*mailbox.Mailbox method*), 1343
values() (*types.MappingProxyType method*), 310
ValuesView (*class in collections.abc*), 284
ValuesView (*class in typing*), 1770
var (*contextvars.Token attribute*), 1059
variable annotation, **2336**
variance (*statistics.NormalDist attribute*), 414
variance() (*in module statistics*), 410
variant (*uuid.UUID attribute*), 1526
vars()
    built-in function, 30
VBAR (*in module token*), 2209
vbar (*tkinter.scrolledtext.ScrolledText attribute*), 1685
VBAREQUAL (*in module token*), 2210
VC_ASSEMBLY_PUBLICKEYTOKEN (*in module msvcrt*), 2253
Vec2D (*class in turtle*), 1645
venv
    module, 1975
--verbose
    tarfile command line option, 633
    timeit command line option, 1956
    unittest-discover command line option, 1807
VERBOSE (*in module re*), 141
verbose (*in module tabnanny*), 2217
verbose (*in module test.support*), 1903
verbose (*sys.flags attribute*), 1998
verify() (*in module enum*), 338
verify() (*smtplib.SMTP method*), 1520
VERIFY_ALLOW_PROXY_CERTS (*in module ssl*), 1211
verify_client_post_handshake() (*ssl.SSLSocket method*), 1221
verify_code (*ssl.SSLCertVerificationError attribute*), 1207
VERIFY_CRL_CHECK_CHAIN (*in module ssl*), 1211
VERIFY_CRL_CHECK_LEAF (*in module ssl*), 1211
VERIFY_DEFAULT (*in module ssl*), 1210
verify_flags (*ssl.SSLContext attribute*), 1229
verify_message (*ssl.SSLCertVerificationError attribute*), 1207
verify_mode (*ssl.SSLContext attribute*), 1230
verify_request() (*socketserver.BaseServer method*), 1533
VERIFY_X509_PARTIAL_CHAIN (*in module ssl*), 1211
VERIFY_X509_STRICT (*in module ssl*), 1211
VERIFY_X509_TRUSTED_FIRST (*in module ssl*), 1211
VerifyFlags (*class in ssl*), 1211
VerifyMode (*class in ssl*), 1210
--version
    python--m-sqlite3-[-h]-[-v]-[filename]-[sql] command line option, 577
    trace command line option, 1959
version (*email.headerregistry.MIMEVersionHeader attribute*), 1298
version (*http.client.HTTPResponse attribute*), 1497
version (*http.cookiejar.Cookie attribute*), 1556
version (*http.cookies.Morsel attribute*), 1546
version (*in module curses*), 888
version (*in module marshal*), 546
version (*in module sqlite3*), 559
version (*in module sys*), 2016
version (*ipaddress.IPv4Address attribute*), 1574
version (*ipaddress.IPv4Network attribute*), 1579
version (*ipaddress.IPv6Address attribute*), 1576
version (*ipaddress.IPv6Network attribute*), 1582
version (*sys.thread_info attribute*), 2015
version (*urllib.request.URLopener attribute*), 1475
version (*uuid.UUID attribute*), 1526
version() (*in module ensurepip*), 1974
version() (*in module platform*), 910
version() (*ssl.SSLSocket method*), 1221
version_info (*in module sqlite3*), 559
version_info (*in module sys*), 2016
version_string() (*http.server.BaseHTTPRequestHandler method*), 1541
vformat() (*string.Formatter method*), 122
virtual
    Environments, 1975
virtual environment, **2336**
virtual machine, **2337**
visit() (*ast.NodeVisitor method*), 2201
visit_Constant() (*ast.NodeVisitor method*), 2201
vline() (*curses.window method*), 888
voidcmd() (*ftplib.FTP method*), 1502
volume (*zipfile.ZipInfo attribute*), 616
vonmisesvariate() (*in module random*), 396
VT (*in module curses.ascii*), 904

W

-w
    calendar command line option, 261
W_OK (*in module os*), 714
wait() (*asyncio.Barrier method*), 1102
wait() (*asyncio.Condition method*), 1100
wait() (*asyncio.Event method*), 1098
wait() (*asyncio.subprocess.Process method*), 1105
wait() (*in module asyncio*), 1081
wait() (*in module concurrent.futures*), 1029
wait() (*in module multiprocessing.connection*), 1004
wait() (*in module os*), 750
wait() (*multiprocessing.pool.AsyncResult method*), 1002

wait() (*subprocess.Popen method*), 1040
wait() (*threading.Barrier method*), 968
wait() (*threading.Condition method*), 964
wait() (*threading.Event method*), 967
wait3() (*in module os*), 751
wait4() (*in module os*), 752
wait_closed() (*asyncio.Server method*), 1134
wait_closed() (*asyncio.StreamWriter method*), 1093
wait_for() (*asyncio.Condition method*), 1100
wait_for() (*in module asyncio*), 1080
wait_for() (*threading.Condition method*), 965
wait_process() (*in module test.support*), 1908
wait_threads_exit() (*in module test.support.threading_helper*), 1915
wait_until_any_call_with() (*unittest.mock.ThreadingMock method*), 1852
wait_until_called() (*unittest.mock.ThreadingMock method*), 1852
waitid() (*in module os*), 750
waitpid() (*in module os*), 751
waitstatus_to_exitcode() (*in module os*), 753
walk() (*email.message.EmailMessage method*), 1277
walk() (*email.message.Message method*), 1317
walk() (*in module ast*), 2201
walk() (*in module os*), 732
walk() (*pathlib.Path method*), 474
walk_packages() (*in module pkgutil*), 2123
walk_stack() (*in module traceback*), 2078
walk_tb() (*in module traceback*), 2079
want (*doctest.Example attribute*), 1795
warn() (*in module warnings*), 2041
warn_default_encoding (*sys.flags attribute*), 1998
warn_explicit() (*in module warnings*), 2041
Warning, 116, 575
WARNING (*in module logging*), 830
WARNING (*in module tkinter.messagebox*), 1684
warning() (*in module logging*), 839
warning() (*logging.Logger method*), 828
warning() (*xml.sax.handler.ErrorHandler method*), 1426
warnings, 2036
    module, 2036
WarningsRecorder (*class in test.support.warnings_helper*), 1921
warnoptions (*in module sys*), 2017
wasSuccessful() (*unittest.TestResult method*), 1829
WatchedFileHandler (*class in logging.handlers*), 859
wave
    module, 1589
Wave_read (*class in wave*), 1590
Wave_write (*class in wave*), 1591
WCONTINUED (*in module os*), 752
WCOREDUMP() (*in module os*), 754
WeakKeyDictionary (*class in weakref*), 298
WeakMethod (*class in weakref*), 299
weakref
    module, 297
WeakSet (*class in weakref*), 299
WeakValueDictionary (*class in weakref*), 299
webbrowser
    module, 1443
WEDNESDAY (*in module calendar*), 258
weekday (*calendar.IllegalWeekdayError attribute*), 259
weekday() (*datetime.date method*), 218
weekday() (*datetime.datetime method*), 228
weekday() (*in module calendar*), 258
weekheader() (*in module calendar*), 258
weibullvariate() (*in module random*), 396
WEXITED (*in module os*), 752
WEXITSTATUS() (*in module os*), 754
wfile (*http.server.BaseHTTPRequestHandler attribute*), 1539
wfile (*socketserver.DatagramRequestHandler attribute*), 1534
whatis (*pdb command*), 1941
when() (*asyncio.Timeout method*), 1079
when() (*asyncio.TimerHandle method*), 1132
where (*pdb command*), 1939
which() (*in module shutil*), 515
whichdb() (*in module dbm*), 546
while
    statement, 35
While (*class in ast*), 2184
whitespace (*in module string*), 122
whitespace (*shlex.shlex attribute*), 1659
whitespace_split (*shlex.shlex attribute*), 1659
Widget (*class in tkinter.ttk*), 1689
--width
    calendar command line option, 261
width (*sys.hash_info attribute*), 2004
width (*textwrap.TextWrapper attribute*), 171
width() (*in module turtle*), 1626
WIFCONTINUED() (*in module os*), 754
WIFEXITED() (*in module os*), 754
WIFSIGNALED() (*in module os*), 754
WIFSTOPPED() (*in module os*), 754
win32_edition() (*in module platform*), 911
win32_is_iot() (*in module platform*), 911
win32_ver() (*in module platform*), 911
WinDLL (*class in ctypes*), 940
window manager (*widgets*), 1673
window() (*curses.panel.Panel method*), 908
window_height() (*in module turtle*), 1643
window_width() (*in module turtle*), 1643
Windows ini file, 645
WindowsError, 115
WindowsPath (*class in pathlib*), 467

WindowsProactorEventLoopPolicy (*class in asyncio*), 1157
WindowsRegistryFinder (*class in importlib.machinery*), 2139
WindowsSelectorEventLoopPolicy (*class in asyncio*), 1157
winerror (*OSError attribute*), 111
WinError() (*in module ctypes*), 948
WINFUNCTYPE() (*in module ctypes*), 944
winreg
    module, 2254
WinSock, 1244
winsound
    module, 2263
winver (*in module sys*), 2017
With (*class in ast*), 2187
WITH_EXCEPT_START (*opcode*), 2236
with_hostmask (*ipaddress.IPv4Interface attribute*), 1585
with_hostmask (*ipaddress.IPv4Network attribute*), 1580
with_hostmask (*ipaddress.IPv6Interface attribute*), 1586
with_hostmask (*ipaddress.IPv6Network attribute*), 1583
with_name() (*pathlib.PurePath method*), 465
with_netmask (*ipaddress.IPv4Interface attribute*), 1585
with_netmask (*ipaddress.IPv4Network attribute*), 1580
with_netmask (*ipaddress.IPv6Interface attribute*), 1586
with_netmask (*ipaddress.IPv6Network attribute*), 1583
with_prefixlen (*ipaddress.IPv4Interface attribute*), 1585
with_prefixlen (*ipaddress.IPv4Network attribute*), 1580
with_prefixlen (*ipaddress.IPv6Interface attribute*), 1585
with_prefixlen (*ipaddress.IPv6Network attribute*), 1583
with_pymalloc() (*in module test.support*), 1906
with_segments() (*pathlib.PurePath method*), 466
with_stem() (*pathlib.PurePath method*), 465
with_suffix() (*pathlib.PurePath method*), 466
with_traceback() (*BaseException method*), 108
withitem (*class in ast*), 2187
WNOHANG (*in module os*), 753
WNOWAIT (*in module os*), 753
wordchars (*shlex.shlex attribute*), 1659
World Wide Web, 1443, 1477, 1487
wrap() (*in module textwrap*), 169
wrap() (*textwrap.TextWrapper method*), 173
wrap_bio() (*ssl.SSLContext method*), 1227
wrap_future() (*in module asyncio*), 1139
wrap_socket() (*ssl.SSLContext method*), 1227
wrapper() (*in module curses*), 881
WrapperDescriptorType (*in module types*), 307
wraps() (*in module functools*), 445
WRITABLE (*in module _tkinter*), 1677
WRITABLE (*inspect.BufferFlags attribute*), 2110
writable() (*bz2.BZ2File method*), 596
writable() (*io.IOBase method*), 765
WRITE (*inspect.BufferFlags attribute*), 2111
write() (*asyncio.StreamWriter method*), 1092
write() (*asyncio.WriteTransport method*), 1145
write() (*codecs.StreamWriter method*), 199
write() (*code.InteractiveInterpreter method*), 2116
write() (*configparser.ConfigParser method*), 661
write() (*email.generator.BytesGenerator method*), 1285
write() (*email.generator.Generator method*), 1286
write() (*in module os*), 711
write() (*in module turtle*), 1630
write() (*io.BufferedIOBase method*), 767
write() (*io.BufferedWriter method*), 769
write() (*io.RawIOBase method*), 766
write() (*io.TextIOBase method*), 771
write() (*mmap.mmap method*), 1267
write() (*sqlite3.Blob method*), 574
write() (*ssl.MemoryBIO method*), 1239
write() (*ssl.SSLSocket method*), 1218
write() (*xml.etree.ElementTree.ElementTree method*), 1396
write() (*zipfile.ZipFile method*), 610
write_byte() (*mmap.mmap method*), 1267
write_bytes() (*pathlib.Path method*), 473
write_docstringdict() (*in module turtle*), 1647
write_eof() (*asyncio.StreamWriter method*), 1092
write_eof() (*asyncio.WriteTransport method*), 1145
write_eof() (*ssl.MemoryBIO method*), 1239
write_history_file() (*in module readline*), 178
write_results() (*trace.CoverageResults method*), 1961
write_text() (*pathlib.Path method*), 473
write_through (*io.TextIOWrapper attribute*), 772
writeframes() (*wave.Wave_write method*), 1591
writeframesraw() (*wave.Wave_write method*), 1591
writeheader() (*csv.DictWriter method*), 643
writelines() (*asyncio.StreamWriter method*), 1092
writelines() (*asyncio.WriteTransport method*), 1145
writelines() (*codecs.StreamWriter method*), 199
writelines() (*io.IOBase method*), 765
writepy() (*zipfile.PyZipFile method*), 613
writer() (*in module csv*), 638
writerow() (*csv.csvwriter method*), 643
writerows() (*csv.csvwriter method*), 643
writestr() (*zipfile.ZipFile method*), 611
WriteTransport (*class in asyncio*), 1142
writev() (*in module os*), 711
writexml() (*xml.dom.minidom.Node method*), 1414
WrongDocumentErr, 1411

wsgi_file_wrapper (*wsgiref.handlers.BaseHandler attribute*), 1454
wsgi_multiprocess (*wsgiref.handlers.BaseHandler attribute*), 1453
wsgi_multithread (*wsgiref.handlers.BaseHandler attribute*), 1453
wsgi_run_once (*wsgiref.handlers.BaseHandler attribute*), 1453
WSGIApplication (*in module wsgiref.types*), 1455
WSGIEnvironment (*in module wsgiref.types*), 1455
wsgiref
    module, 1446
wsgiref.handlers
    module, 1451
wsgiref.headers
    module, 1448
wsgiref.simple_server
    module, 1449
wsgiref.types
    module, 1455
wsgiref.util
    module, 1446
wsgiref.validate
    module, 1450
WSGIRequestHandler (*class in wsgiref.simple_server*), 1450
WSGIServer (*class in wsgiref.simple_server*), 1449
wShowWindow (*subprocess.STARTUPINFO attribute*), 1043
WSTOPPED (*in module os*), 752
WSTOPSIG() (*in module os*), 754
wstring_at() (*in module ctypes*), 948
WTERMSIG() (*in module os*), 755
WUNTRACED (*in module os*), 753
WWW, 1443, 1477, 1487
    server, 1538

X

-x
    compileall command line option, 2222
X (*in module re*), 141
X509 certificate, 1231
X_OK (*in module os*), 714
xatom() (*imaplib.IMAP4 method*), 1516
XATTR_CREATE (*in module os*), 740
XATTR_REPLACE (*in module os*), 740
XATTR_SIZE_MAX (*in module os*), 740
xcor() (*in module turtle*), 1625
XHTML, 1373
XHTML_NAMESPACE (*in module xml.dom*), 1403
xml
    module, 1379
XML() (*in module xml.etree.ElementTree*), 1392
XML_ERROR_ABORTED (*in module xml.parsers.expat.errors*), 1442
XML_ERROR_AMPLIFICATION_LIMIT_BREACH (*in module xml.parsers.expat.errors*), 1442
XML_ERROR_ASYNC_ENTITY (*in module xml.parsers.expat.errors*), 1440
XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF (*in module xml.parsers.expat.errors*), 1440
XML_ERROR_BAD_CHAR_REF (*in module xml.parsers.expat.errors*), 1440
XML_ERROR_BINARY_ENTITY_REF (*in module xml.parsers.expat.errors*), 1440
XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_DUPLICATE_ATTRIBUTE (*in module xml.parsers.expat.errors*), 1440
XML_ERROR_ENTITY_DECLARED_IN_PE (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_EXTERNAL_ENTITY_HANDLING (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_FEATURE_REQUIRES_XML_DTD (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_FINISHED (*in module xml.parsers.expat.errors*), 1442
XML_ERROR_INCOMPLETE_PE (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_INCORRECT_ENCODING (*in module xml.parsers.expat.errors*), 1440
XML_ERROR_INVALID_ARGUMENT (*in module xml.parsers.expat.errors*), 1442
XML_ERROR_INVALID_TOKEN (*in module xml.parsers.expat.errors*), 1440
XML_ERROR_JUNK_AFTER_DOC_ELEMENT (*in module xml.parsers.expat.errors*), 1440
XML_ERROR_MISPLACED_XML_PI (*in module xml.parsers.expat.errors*), 1440
XML_ERROR_NO_BUFFER (*in module xml.parsers.expat.errors*), 1442
XML_ERROR_NO_ELEMENTS (*in module xml.parsers.expat.errors*), 1440
XML_ERROR_NO_MEMORY (*in module xml.parsers.expat.errors*), 1440
XML_ERROR_NOT_STANDALONE (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_NOT_SUSPENDED (*in module xml.parsers.expat.errors*), 1442
XML_ERROR_PARAM_ENTITY_REF (*in module xml.parsers.expat.errors*), 1440
XML_ERROR_PARTIAL_CHAR (*in module xml.parsers.expat.errors*), 1440
XML_ERROR_PUBLICID (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_RECURSIVE_ENTITY_REF (*in module xml.parsers.expat.errors*), 1440

XML_ERROR_RESERVED_NAMESPACE_URI (*in module xml.parsers.expat.errors*), 1442
XML_ERROR_RESERVED_PREFIX_XML (*in module xml.parsers.expat.errors*), 1442
XML_ERROR_RESERVED_PREFIX_XMLNS (*in module xml.parsers.expat.errors*), 1442
XML_ERROR_SUSPEND_PE (*in module xml.parsers.expat.errors*), 1442
XML_ERROR_SUSPENDED (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_SYNTAX (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_TAG_MISMATCH (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_TEXT_DECL (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_UNBOUND_PREFIX (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_UNCLOSED_CDATA_SECTION (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_UNCLOSED_TOKEN (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_UNDECLARING_PREFIX (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_UNDEFINED_ENTITY (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_UNEXPECTED_STATE (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_UNKNOWN_ENCODING (*in module xml.parsers.expat.errors*), 1441
XML_ERROR_XML_DECL (*in module xml.parsers.expat.errors*), 1441
XML_NAMESPACE (*in module xml.dom*), 1402
xmlcharrefreplace
    error handler's name, 194
xmlcharrefreplace_errors() (*in module codecs*), 196
XmlDeclHandler() (*xml.parsers.expat.xmlparser method*), 1436
xml.dom
    module, 1401
xml.dom.minidom
    module, 1412
xml.dom.pulldom
    module, 1417
xml.etree.ElementInclude
    module, 1393
xml.etree.ElementTree
    module, 1381
XMLFilterBase (*class in xml.sax.saxutils*), 1427
XMLGenerator (*class in xml.sax.saxutils*), 1427
XMLID() (*in module xml.etree.ElementTree*), 1392
XMLNS_NAMESPACE (*in module xml.dom*), 1403
XMLParser (*class in xml.etree.ElementTree*), 1399
xml.parsers.expat
    module, 1432
xml.parsers.expat.errors
    module, 1440
xml.parsers.expat.model
    module, 1439
XMLParserType (*in module xml.parsers.expat*), 1432
XMLPullParser (*class in xml.etree.ElementTree*), 1400
XMLReader (*class in xml.sax.xmlreader*), 1428
xmlrpc.client
    module, 1558
xmlrpc.server
    module, 1566
xml.sax
    module, 1419
xml.sax.handler
    module, 1421
xml.sax.saxutils
    module, 1426
xml.sax.xmlreader
    module, 1428
xor() (*in module operator*), 448
xview() (*tkinter.ttk.Treeview method*), 1701

Y

ycor() (*in module turtle*), 1625
year
    calendar command line option, 261
year (*datetime.date attribute*), 216
year (*datetime.datetime attribute*), 224
Year 2038, 775
yeardatescalendar() (*calendar.Calendar method*), 255
yeardays2calendar() (*calendar.Calendar method*), 255
yeardayscalendar() (*calendar.Calendar method*), 255
YES (*in module tkinter.messagebox*), 1684
YESEXPR (*in module locale*), 1605
YESNO (*in module tkinter.messagebox*), 1684
YESNOCANCEL (*in module tkinter.messagebox*), 1684
Yield (*class in ast*), 2196
YIELD_VALUE (*opcode*), 2235
YieldFrom (*class in ast*), 2196
yiq_to_rgb() (*in module colorsys*), 1592
yview() (*tkinter.ttk.Treeview method*), 1701

Z

z
    in string formatting, 125
z85decode() (*in module base64*), 1368
z85encode() (*in module base64*), 1367
Zen of Python, **2337**
ZeroDivisionError, 114

zfill() (*bytearray method*), 76
zfill() (*bytes method*), 76
zfill() (*str method*), 60
zip()
    built-in function, 30
ZIP_BZIP2 (*in module zipfile*), 607
ZIP_DEFLATED (*in module zipfile*), 607
zip_longest() (*in module itertools*), 429
ZIP_LZMA (*in module zipfile*), 607
ZIP_STORED (*in module zipfile*), 607
zipapp
    module, 1985
zipapp command line option
    -c, 1986
    --compress, 1986
    -h, 1986
    --help, 1986
    --info, 1986
    -m, 1986
    --main, 1986
    -o, 1986
    --output, 1986
    -p, 1986
    --python, 1986
zipfile
    module, 606
ZipFile (*class in zipfile*), 607
zipfile command line option
    -c, 617
    --create, 617
    -e, 617
    --extract, 617
    -l, 617
    --list, 617
    --metadata-encoding, 617
    -t, 617
    --test, 617
zipimport
    module, 2119
zipimporter (*class in zipimport*), 2120
ZipImportError, 2119
ZipInfo (*class in zipfile*), 606
zlib
    module, 587
ZLIB_RUNTIME_VERSION (*in module zlib*), 590
ZLIB_VERSION (*in module zlib*), 590
zoneinfo
    module, 248
ZoneInfo (*class in zoneinfo*), 250
ZoneInfoNotFoundError, 254
zscore() (*statistics.NormalDist method*), 415