

---

# **The Python Library Reference**

***Release 3.11.4***

**Guido van Rossum and the Python development team**

**August 16, 2023**

**Python Software Foundation  
Email: [docs@python.org](mailto:docs@python.org)**



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Notes on availability	3
1.1.1	WebAssembly platforms	4
<b>2</b>	<b>Built-in Functions</b>	<b>5</b>
<b>3</b>	<b>Built-in Constants</b>	<b>31</b>
3.1	Constants added by the <code>site</code> module	32
<b>4</b>	<b>Built-in Types</b>	<b>33</b>
4.1	Truth Value Testing	33
4.2	Boolean Operations — <code>and</code> , <code>or</code> , <code>not</code>	33
4.3	Comparisons	34
4.4	Numeric Types — <code>int</code> , <code>float</code> , <code>complex</code>	34
4.4.1	Bitwise Operations on Integer Types	36
4.4.2	Additional Methods on Integer Types	36
4.4.3	Additional Methods on Float	39
4.4.4	Hashing of numeric types	40
4.5	Iterator Types	41
4.5.1	Generator Types	42
4.6	Sequence Types — <code>list</code> , <code>tuple</code> , <code>range</code>	42
4.6.1	Common Sequence Operations	42
4.6.2	Immutable Sequence Types	44
4.6.3	Mutable Sequence Types	44
4.6.4	Lists	45
4.6.5	Tuples	46
4.6.6	Ranges	46
4.7	Text Sequence Type — <code>str</code>	48
4.7.1	String Methods	49
4.7.2	<code>printf</code> -style String Formatting	57
4.8	Binary Sequence Types — <code>bytes</code> , <code>bytearray</code> , <code>memoryview</code>	60
4.8.1	Bytes Objects	60
4.8.2	Bytearray Objects	61
4.8.3	Bytes and Bytearray Operations	62
4.8.4	<code>printf</code> -style Bytes Formatting	73
4.8.5	Memory Views	75
4.9	Set Types — <code>set</code> , <code>frozenset</code>	82
4.10	Mapping Types — <code>dict</code>	85
4.10.1	Dictionary view objects	88
4.11	Context Manager Types	90

4.12	Type Annotation Types — Generic Alias, Union . . . . .	91
4.12.1	Generic Alias Type . . . . .	91
4.12.2	Union Type . . . . .	95
4.13	Other Built-in Types . . . . .	96
4.13.1	Modules . . . . .	97
4.13.2	Classes and Class Instances . . . . .	97
4.13.3	Functions . . . . .	97
4.13.4	Methods . . . . .	97
4.13.5	Code Objects . . . . .	98
4.13.6	Type Objects . . . . .	98
4.13.7	The Null Object . . . . .	98
4.13.8	The Ellipsis Object . . . . .	98
4.13.9	The NotImplemented Object . . . . .	98
4.13.10	Boolean Values . . . . .	99
4.13.11	Internal Objects . . . . .	99
4.14	Special Attributes . . . . .	99
4.15	Integer string conversion length limitation . . . . .	100
4.15.1	Affected APIs . . . . .	101
4.15.2	Configuring the limit . . . . .	101
4.15.3	Recommended configuration . . . . .	102
<b>5</b>	<b>Built-in Exceptions</b> . . . . .	<b>103</b>
5.1	Exception context . . . . .	103
5.2	Inheriting from built-in exceptions . . . . .	104
5.3	Base classes . . . . .	104
5.4	Concrete exceptions . . . . .	105
5.4.1	OS exceptions . . . . .	110
5.5	Warnings . . . . .	112
5.6	Exception groups . . . . .	113
5.7	Exception hierarchy . . . . .	115
<b>6</b>	<b>Text Processing Services</b> . . . . .	<b>117</b>
6.1	<code>string</code> — Common string operations . . . . .	117
6.1.1	String constants . . . . .	117
6.1.2	Custom String Formatting . . . . .	118
6.1.3	Format String Syntax . . . . .	119
6.1.4	Template strings . . . . .	126
6.1.5	Helper functions . . . . .	128
6.2	<code>re</code> — Regular expression operations . . . . .	128
6.2.1	Regular Expression Syntax . . . . .	129
6.2.2	Module Contents . . . . .	134
6.2.3	Regular Expression Objects . . . . .	140
6.2.4	Match Objects . . . . .	142
6.2.5	Regular Expression Examples . . . . .	144
6.3	<code>difflib</code> — Helpers for computing deltas . . . . .	150
6.3.1	SequenceMatcher Objects . . . . .	155
6.3.2	SequenceMatcher Examples . . . . .	157
6.3.3	Differ Objects . . . . .	158
6.3.4	Differ Example . . . . .	159
6.3.5	A command-line interface to <code>difflib</code> . . . . .	160
6.4	<code>textwrap</code> — Text wrapping and filling . . . . .	161
6.5	<code>unicodedata</code> — Unicode Database . . . . .	165
6.6	<code>stringprep</code> — Internet String Preparation . . . . .	167
6.7	<code>readline</code> — GNU readline interface . . . . .	168

6.7.1	Init file . . . . .	169
6.7.2	Line buffer . . . . .	169
6.7.3	History file . . . . .	169
6.7.4	History list . . . . .	170
6.7.5	Startup hooks . . . . .	170
6.7.6	Completion . . . . .	171
6.7.7	Example . . . . .	171
6.8	rlcompleter — Completion function for GNU readline . . . . .	173
6.8.1	Completer Objects . . . . .	173
<b>7</b>	<b>Binary Data Services</b>	<b>175</b>
7.1	struct — Interpret bytes as packed binary data . . . . .	175
7.1.1	Functions and Exceptions . . . . .	176
7.1.2	Format Strings . . . . .	176
7.1.3	Applications . . . . .	180
7.1.4	Classes . . . . .	181
7.2	codecs — Codec registry and base classes . . . . .	182
7.2.1	Codec Base Classes . . . . .	185
7.2.2	Encodings and Unicode . . . . .	192
7.2.3	Standard Encodings . . . . .	193
7.2.4	Python Specific Encodings . . . . .	196
7.2.5	encodings.idna — Internationalized Domain Names in Applications . . . . .	198
7.2.6	encodings.mbcs — Windows ANSI codepage . . . . .	199
7.2.7	encodings.utf_8_sig — UTF-8 codec with BOM signature . . . . .	199
<b>8</b>	<b>Data Types</b>	<b>201</b>
8.1	datetime — Basic date and time types . . . . .	201
8.1.1	Aware and Naive Objects . . . . .	201
8.1.2	Constants . . . . .	202
8.1.3	Available Types . . . . .	202
8.1.4	timedelta Objects . . . . .	203
8.1.5	date Objects . . . . .	207
8.1.6	datetime Objects . . . . .	212
8.1.7	time Objects . . . . .	223
8.1.8	tzinfo Objects . . . . .	227
8.1.9	timezone Objects . . . . .	233
8.1.10	strptime() and strptime() Behavior . . . . .	234
8.2	zoneinfo — IANA time zone support . . . . .	239
8.2.1	Using ZoneInfo . . . . .	239
8.2.2	Data sources . . . . .	240
8.2.3	The ZoneInfo class . . . . .	241
8.2.4	Functions . . . . .	243
8.2.5	Globals . . . . .	244
8.2.6	Exceptions and warnings . . . . .	244
8.3	calendar — General calendar-related functions . . . . .	244
8.4	collections — Container datatypes . . . . .	249
8.4.1	ChainMap objects . . . . .	250
8.4.2	Counter objects . . . . .	252
8.4.3	deque objects . . . . .	255
8.4.4	defaultdict objects . . . . .	259
8.4.5	namedtuple() Factory Function for Tuples with Named Fields . . . . .	261
8.4.6	OrderedDict objects . . . . .	264
8.4.7	UserDict objects . . . . .	267
8.4.8	UserList objects . . . . .	267

8.4.9	UserString objects	267
8.5	<code>collections.abc</code> — Abstract Base Classes for Containers	268
8.5.1	Collections Abstract Base Classes	269
8.5.2	Collections Abstract Base Classes – Detailed Descriptions	271
8.5.3	Examples and Recipes	273
8.6	<code>heapq</code> — Heap queue algorithm	274
8.6.1	Basic Examples	275
8.6.2	Priority Queue Implementation Notes	276
8.6.3	Theory	277
8.7	<code>bisect</code> — Array bisection algorithm	278
8.7.1	Performance Notes	279
8.7.2	Searching Sorted Lists	279
8.7.3	Examples	280
8.8	<code>array</code> — Efficient arrays of numeric values	281
8.9	<code>weakref</code> — Weak references	284
8.9.1	Weak Reference Objects	289
8.9.2	Example	290
8.9.3	Finalizer Objects	290
8.9.4	Comparing finalizers with <code>__del__()</code> methods	291
8.10	<code>types</code> — Dynamic type creation and names for built-in types	292
8.10.1	Dynamic Type Creation	293
8.10.2	Standard Interpreter Types	293
8.10.3	Additional Utility Classes and Functions	298
8.10.4	Coroutine Utility Functions	298
8.11	<code>copy</code> — Shallow and deep copy operations	299
8.12	<code>pprint</code> — Data pretty printer	300
8.12.1	PrettyPrinter Objects	302
8.12.2	Example	303
8.13	<code>reprlib</code> — Alternate <code>repr()</code> implementation	306
8.13.1	Repr Objects	306
8.13.2	Subclassing Repr Objects	307
8.14	<code>enum</code> — Support for enumerations	308
8.14.1	Module Contents	309
8.14.2	Data Types	310
8.14.3	Utilities and Decorators	320
8.14.4	Notes	321
8.15	<code>graphlib</code> — Functionality to operate with graph-like structures	322
8.15.1	Exceptions	324
<b>9</b>	<b>Numeric and Mathematical Modules</b>	<b>325</b>
9.1	<code>numbers</code> — Numeric abstract base classes	325
9.1.1	The numeric tower	325
9.1.2	Notes for type implementors	326
9.2	<code>math</code> — Mathematical functions	328
9.2.1	Number-theoretic and representation functions	329
9.2.2	Power and logarithmic functions	332
9.2.3	Trigonometric functions	333
9.2.4	Angular conversion	334
9.2.5	Hyperbolic functions	334
9.2.6	Special functions	335
9.2.7	Constants	335
9.3	<code>cmath</code> — Mathematical functions for complex numbers	336
9.3.1	Conversions to and from polar coordinates	337
9.3.2	Power and logarithmic functions	338

9.3.3	Trigonometric functions	338
9.3.4	Hyperbolic functions	338
9.3.5	Classification functions	339
9.3.6	Constants	339
9.4	<code>decimal</code> — Decimal fixed point and floating point arithmetic	340
9.4.1	Quick-start Tutorial	341
9.4.2	Decimal objects	345
9.4.3	Context objects	352
9.4.4	Constants	358
9.4.5	Rounding modes	359
9.4.6	Signals	359
9.4.7	Floating Point Notes	361
9.4.8	Working with threads	363
9.4.9	Recipes	363
9.4.10	Decimal FAQ	366
9.5	<code>fractions</code> — Rational numbers	369
9.6	<code>random</code> — Generate pseudo-random numbers	372
9.6.1	Bookkeeping functions	372
9.6.2	Functions for bytes	373
9.6.3	Functions for integers	373
9.6.4	Functions for sequences	374
9.6.5	Real-valued distributions	375
9.6.6	Alternative Generator	376
9.6.7	Notes on Reproducibility	376
9.6.8	Examples	377
9.6.9	Recipes	379
9.7	<code>statistics</code> — Mathematical statistics functions	380
9.7.1	Averages and measures of central location	381
9.7.2	Measures of spread	381
9.7.3	Statistics for relations between two inputs	382
9.7.4	Function details	382
9.7.5	Exceptions	389
9.7.6	<code>NormalDist</code> objects	390
<b>10</b>	<b>Functional Programming Modules</b>	<b>395</b>
10.1	<code>itertools</code> — Functions creating iterators for efficient looping	395
10.1.1	Itertool functions	397
10.1.2	Itertools Recipes	406
10.2	<code>functools</code> — Higher-order functions and operations on callable objects	413
10.2.1	<code>partial</code> Objects	422
10.3	<code>operator</code> — Standard operators as functions	423
10.3.1	Mapping Operators to Functions	427
10.3.2	In-place Operators	428
<b>11</b>	<b>File and Directory Access</b>	<b>431</b>
11.1	<code>pathlib</code> — Object-oriented filesystem paths	431
11.1.1	Basic use	432
11.1.2	Pure paths	433
11.1.3	Concrete paths	441
11.1.4	Correspondence to tools in the <code>os</code> module	449
11.2	<code>os.path</code> — Common pathname manipulations	450
11.3	<code>fileinput</code> — Iterate over lines from multiple input streams	456
11.4	<code>stat</code> — Interpreting <code>stat()</code> results	459
11.5	<code>filecmp</code> — File and Directory Comparisons	464

11.5.1	The <code>dircmp</code> class	465
11.6	<code>tempfile</code> — Generate temporary files and directories	467
11.6.1	Examples	470
11.6.2	Deprecated functions and variables	471
11.7	<code>glob</code> — Unix style pathname pattern expansion	471
11.8	<code>fnmatch</code> — Unix filename pattern matching	473
11.9	<code>linecache</code> — Random access to text lines	474
11.10	<code>shutil</code> — High-level file operations	475
11.10.1	Directory and files operations	475
11.10.2	Archiving operations	481
11.10.3	Querying the size of the output terminal	485
<b>12</b>	<b>Data Persistence</b>	<b>487</b>
12.1	<code>pickle</code> — Python object serialization	487
12.1.1	Relationship to other Python modules	487
12.1.2	Data stream format	488
12.1.3	Module Interface	489
12.1.4	What can be pickled and unpickled?	493
12.1.5	Pickling Class Instances	493
12.1.6	Custom Reduction for Types, Functions, and Other Objects	499
12.1.7	Out-of-band Buffers	500
12.1.8	Restricting Globals	502
12.1.9	Performance	503
12.1.10	Examples	503
12.2	<code>copyreg</code> — Register <code>pickle</code> support functions	504
12.2.1	Example	504
12.3	<code>shelve</code> — Python object persistence	505
12.3.1	Restrictions	506
12.3.2	Example	507
12.4	<code>marshal</code> — Internal Python object serialization	507
12.5	<code>dbm</code> — Interfaces to Unix “databases”	509
12.5.1	<code>dbm.gnu</code> — GNU’s reinterpretation of <code>dbm</code>	510
12.5.2	<code>dbm.ndbm</code> — Interface based on <code>ndbm</code>	512
12.5.3	<code>dbm.dumb</code> — Portable DBM implementation	512
12.6	<code>sqlite3</code> — DB-API 2.0 interface for SQLite databases	513
12.6.1	Tutorial	514
12.6.2	Reference	516
12.6.3	How-to guides	535
12.6.4	Explanation	542
<b>13</b>	<b>Data Compression and Archiving</b>	<b>543</b>
13.1	<code>zlib</code> — Compression compatible with <b>gzip</b>	543
13.2	<code>gzip</code> — Support for <b>gzip</b> files	547
13.2.1	Examples of usage	549
13.2.2	Command Line Interface	550
13.3	<code>bz2</code> — Support for <b>bzip2</b> compression	550
13.3.1	(De)compression of files	550
13.3.2	Incremental (de)compression	552
13.3.3	One-shot (de)compression	553
13.3.4	Examples of usage	553
13.4	<code>lzma</code> — Compression using the LZMA algorithm	554
13.4.1	Reading and writing compressed files	555
13.4.2	Compressing and decompressing data in memory	556
13.4.3	Miscellaneous	558



13.4.4	Specifying custom filter chains	558
13.4.5	Examples	559
13.5	<code>zipfile</code> — Work with ZIP archives	560
13.5.1	ZipFile Objects	561
13.5.2	Path Objects	566
13.5.3	PyZipFile Objects	567
13.5.4	ZipInfo Objects	568
13.5.5	Command-Line Interface	570
13.5.6	Decompression pitfalls	570
13.6	<code>tarfile</code> — Read and write tar archive files	571
13.6.1	TarFile Objects	574
13.6.2	TarInfo Objects	578
13.6.3	Extraction filters	580
13.6.4	Command-Line Interface	584
13.6.5	Examples	585
13.6.6	Supported tar formats	586
13.6.7	Unicode issues	586
<b>14</b>	<b>File Formats</b>	<b>589</b>
14.1	<code>csv</code> — CSV File Reading and Writing	589
14.1.1	Module Contents	589
14.1.2	Dialects and Formatting Parameters	593
14.1.3	Reader Objects	594
14.1.4	Writer Objects	594
14.1.5	Examples	595
14.2	<code>configparser</code> — Configuration file parser	596
14.2.1	Quick Start	597
14.2.2	Supported Datatypes	598
14.2.3	Fallback Values	599
14.2.4	Supported INI File Structure	600
14.2.5	Interpolation of values	601
14.2.6	Mapping Protocol Access	602
14.2.7	Customizing Parser Behaviour	603
14.2.8	Legacy API Examples	607
14.2.9	ConfigParser Objects	609
14.2.10	RawConfigParser Objects	613
14.2.11	Exceptions	614
14.3	<code>tomllib</code> — Parse TOML files	614
14.3.1	Examples	615
14.3.2	Conversion Table	616
14.4	<code>netrc</code> — <code>netrc</code> file processing	616
14.4.1	<code>netrc</code> Objects	617
14.5	<code>plistlib</code> — Generate and parse Apple <code>.plist</code> files	617
14.5.1	Examples	619
<b>15</b>	<b>Cryptographic Services</b>	<b>621</b>
15.1	<code>hashlib</code> — Secure hashes and message digests	621
15.1.1	Hash algorithms	621
15.1.2	Usage	622
15.1.3	Constructors	622
15.1.4	Attributes	623
15.1.5	Hash Objects	623
15.1.6	SHAKE variable length digests	624
15.1.7	File hashing	624

15.1.8	Key derivation . . . . .	625
15.1.9	BLAKE2 . . . . .	626
15.2	hmac — Keyed-Hashing for Message Authentication . . . . .	633
15.3	secrets — Generate secure random numbers for managing secrets . . . . .	635
15.3.1	Random numbers . . . . .	635
15.3.2	Generating tokens . . . . .	635
15.3.3	Other functions . . . . .	636
15.3.4	Recipes and best practices . . . . .	636
<b>16</b>	<b>Generic Operating System Services</b> . . . . .	<b>639</b>
16.1	os — Miscellaneous operating system interfaces . . . . .	639
16.1.1	File Names, Command Line Arguments, and Environment Variables . . . . .	640
16.1.2	Python UTF-8 Mode . . . . .	640
16.1.3	Process Parameters . . . . .	641
16.1.4	File Object Creation . . . . .	648
16.1.5	File Descriptor Operations . . . . .	648
16.1.6	Files and Directories . . . . .	659
16.1.7	Process Management . . . . .	682
16.1.8	Interface to the scheduler . . . . .	695
16.1.9	Miscellaneous System Information . . . . .	697
16.1.10	Random numbers . . . . .	699
16.2	io — Core tools for working with streams . . . . .	700
16.2.1	Overview . . . . .	700
16.2.2	Text Encoding . . . . .	701
16.2.3	High-level Module Interface . . . . .	702
16.2.4	Class hierarchy . . . . .	703
16.2.5	Performance . . . . .	713
16.3	time — Time access and conversions . . . . .	714
16.3.1	Functions . . . . .	715
16.3.2	Clock ID Constants . . . . .	723
16.3.3	Timezone Constants . . . . .	724
16.4	argparse — Parser for command-line options, arguments and sub-commands . . . . .	725
16.4.1	Core Functionality . . . . .	725
16.4.2	Quick Links for add_argument() . . . . .	726
16.4.3	Example . . . . .	726
16.4.4	ArgumentParser objects . . . . .	728
16.4.5	The add_argument() method . . . . .	736
16.4.6	The parse_args() method . . . . .	747
16.4.7	Other utilities . . . . .	750
16.4.8	Upgrading optparse code . . . . .	758
16.4.9	Exceptions . . . . .	759
16.5	getopt — C-style parser for command line options . . . . .	759
16.6	logging — Logging facility for Python . . . . .	761
16.6.1	Logger Objects . . . . .	762
16.6.2	Logging Levels . . . . .	766
16.6.3	Handler Objects . . . . .	767
16.6.4	Formatter Objects . . . . .	768
16.6.5	Filter Objects . . . . .	770
16.6.6	LogRecord Objects . . . . .	771
16.6.7	LogRecord attributes . . . . .	772
16.6.8	LoggerAdapter Objects . . . . .	774
16.6.9	Thread Safety . . . . .	774
16.6.10	Module-Level Functions . . . . .	774
16.6.11	Module-Level Attributes . . . . .	779

16.6.12	Integration with the warnings module	779
16.7	<code>logging.config</code> — Logging configuration	780
16.7.1	Configuration functions	780
16.7.2	Security considerations	782
16.7.3	Configuration dictionary schema	782
16.7.4	Configuration file format	789
16.8	<code>logging.handlers</code> — Logging handlers	791
16.8.1	<code>StreamHandler</code>	792
16.8.2	<code>FileHandler</code>	792
16.8.3	<code>NullHandler</code>	793
16.8.4	<code>WatchedFileHandler</code>	793
16.8.5	<code>BaseRotatingHandler</code>	794
16.8.6	<code>RotatingFileHandler</code>	795
16.8.7	<code>TimedRotatingFileHandler</code>	796
16.8.8	<code>SocketHandler</code>	797
16.8.9	<code>DatagramHandler</code>	798
16.8.10	<code>SysLogHandler</code>	799
16.8.11	<code>NTEventLogHandler</code>	801
16.8.12	<code>SMTPHandler</code>	801
16.8.13	<code>MemoryHandler</code>	802
16.8.14	<code>HTTPHandler</code>	803
16.8.15	<code>QueueHandler</code>	803
16.8.16	<code>QueueListener</code>	804
16.9	<code>getpass</code> — Portable password input	806
16.10	<code>curses</code> — Terminal handling for character-cell displays	806
16.10.1	Functions	807
16.10.2	Window Objects	814
16.10.3	Constants	821
16.11	<code>curses.textpad</code> — Text input widget for <code>curses</code> programs	834
16.11.1	Textbox objects	835
16.12	<code>curses.ascii</code> — Utilities for ASCII characters	836
16.13	<code>curses.panel</code> — A panel stack extension for <code>curses</code>	840
16.13.1	Functions	840
16.13.2	Panel Objects	840
16.14	<code>platform</code> — Access to underlying platform's identifying data	841
16.14.1	Cross Platform	841
16.14.2	Java Platform	843
16.14.3	Windows Platform	843
16.14.4	macOS Platform	844
16.14.5	Unix Platforms	844
16.14.6	Linux Platforms	844
16.15	<code>errno</code> — Standard <code>errno</code> system symbols	845
16.16	<code>ctypes</code> — A foreign function library for Python	852
16.16.1	<code>ctypes</code> tutorial	852
16.16.2	<code>ctypes</code> reference	870
<b>17</b>	<b>Concurrent Execution</b>	<b>887</b>
17.1	<code>threading</code> — Thread-based parallelism	887
17.1.1	Thread-Local Data	890
17.1.2	Thread Objects	890
17.1.3	Lock Objects	893
17.1.4	<code>RLock</code> Objects	894
17.1.5	Condition Objects	895
17.1.6	Semaphore Objects	897

17.1.7	Event Objects	898
17.1.8	Timer Objects	899
17.1.9	Barrier Objects	899
17.1.10	Using locks, conditions, and semaphores in the <code>with</code> statement	901
17.2	<code>multiprocessing</code> — Process-based parallelism	901
17.2.1	Introduction	901
17.2.2	Reference	908
17.2.3	Programming guidelines	937
17.2.4	Examples	941
17.3	<code>multiprocessing.shared_memory</code> — Shared memory for direct access across processes	946
17.4	The <code>concurrent</code> package	952
17.5	<code>concurrent.futures</code> — Launching parallel tasks	952
17.5.1	Executor Objects	952
17.5.2	<code>ThreadPoolExecutor</code>	954
17.5.3	<code>ProcessPoolExecutor</code>	955
17.5.4	Future Objects	957
17.5.5	Module Functions	958
17.5.6	Exception classes	959
17.6	<code>subprocess</code> — Subprocess management	959
17.6.1	Using the <code>subprocess</code> Module	960
17.6.2	Security Considerations	968
17.6.3	Popen Objects	969
17.6.4	Windows Popen Helpers	971
17.6.5	Older high-level API	973
17.6.6	Replacing Older Functions with the <code>subprocess</code> Module	975
17.6.7	Legacy Shell Invocation Functions	978
17.6.8	Notes	979
17.7	<code>sched</code> — Event scheduler	980
17.7.1	Scheduler Objects	980
17.8	<code>queue</code> — A synchronized queue class	981
17.8.1	Queue Objects	982
17.8.2	<code>SimpleQueue</code> Objects	984
17.9	<code>contextvars</code> — Context Variables	985
17.9.1	Context Variables	985
17.9.2	Manual Context Management	986
17.9.3	<code>asyncio</code> support	988
17.10	<code>_thread</code> — Low-level threading API	989
<b>18</b>	<b>Networking and Interprocess Communication</b>	<b>993</b>
18.1	<code>asyncio</code> — Asynchronous I/O	993
18.1.1	Runners	994
18.1.2	Coroutines and Tasks	996
18.1.3	Streams	1013
18.1.4	Synchronization Primitives	1021
18.1.5	Subprocesses	1027
18.1.6	Queues	1032
18.1.7	Exceptions	1035
18.1.8	Event Loop	1036
18.1.9	Futures	1059
18.1.10	Transports and Protocols	1063
18.1.11	Policies	1077
18.1.12	Platform Support	1080
18.1.13	Extending	1082
18.1.14	High-level API Index	1083

18.1.15	Low-level API Index	1086
18.1.16	Developing with asyncio	1092
18.2	socket — Low-level networking interface	1095
18.2.1	Socket families	1096
18.2.2	Module contents	1098
18.2.3	Socket Objects	1110
18.2.4	Notes on socket timeouts	1118
18.2.5	Example	1119
18.3	ssl — TLS/SSL wrapper for socket objects	1122
18.3.1	Functions, Constants, and Exceptions	1123
18.3.2	SSL Sockets	1136
18.3.3	SSL Contexts	1140
18.3.4	Certificates	1148
18.3.5	Examples	1150
18.3.6	Notes on non-blocking sockets	1153
18.3.7	Memory BIO Support	1154
18.3.8	SSL session	1156
18.3.9	Security considerations	1156
18.3.10	TLS 1.3	1157
18.4	select — Waiting for I/O completion	1158
18.4.1	/dev/poll Polling Objects	1160
18.4.2	Edge and Level Trigger Polling (epoll) Objects	1161
18.4.3	Polling Objects	1162
18.4.4	Kqueue Objects	1163
18.4.5	Kevent Objects	1163
18.5	selectors — High-level I/O multiplexing	1165
18.5.1	Introduction	1165
18.5.2	Classes	1166
18.5.3	Examples	1168
18.6	signal — Set handlers for asynchronous events	1169
18.6.1	General rules	1169
18.6.2	Module contents	1170
18.6.3	Examples	1177
18.6.4	Note on SIGPIPE	1177
18.6.5	Note on Signal Handlers and Exceptions	1178
18.7	mmap — Memory-mapped file support	1179
18.7.1	MADV_* Constants	1183
18.7.2	MAP_* Constants	1183
<b>19</b>	<b>Internet Data Handling</b>	<b>1185</b>
19.1	email — An email and MIME handling package	1185
19.1.1	email.message: Representing an email message	1186
19.1.2	email.parser: Parsing email messages	1194
19.1.3	email.generator: Generating MIME documents	1198
19.1.4	email.policy: Policy Objects	1201
19.1.5	email.errors: Exception and Defect classes	1208
19.1.6	email.headerregistry: Custom Header Objects	1209
19.1.7	email.contentmanager: Managing MIME Content	1215
19.1.8	email: Examples	1217
19.1.9	email.message.Message: Representing an email message using the compat32 API	1224
19.1.10	email.mime: Creating email and MIME objects from scratch	1232
19.1.11	email.header: Internationalized headers	1235
19.1.12	email.charset: Representing character sets	1237
19.1.13	email.encoders: Encoders	1239

19.1.14	<code>email.utils</code> : Miscellaneous utilities	1240
19.1.15	<code>email.iterators</code> : Iterators	1243
19.2	<code>json</code> — JSON encoder and decoder	1244
19.2.1	Basic Usage	1246
19.2.2	Encoders and Decoders	1248
19.2.3	Exceptions	1250
19.2.4	Standard Compliance and Interoperability	1251
19.2.5	Command Line Interface	1252
19.3	<code>mailbox</code> — Manipulate mailboxes in various formats	1254
19.3.1	Mailbox objects	1254
19.3.2	Message objects	1262
19.3.3	Exceptions	1270
19.3.4	Examples	1271
19.4	<code>mimetypes</code> — Map filenames to MIME types	1272
19.4.1	MimeTypes Objects	1274
19.5	<code>base64</code> — Base16, Base32, Base64, Base85 Data Encodings	1275
19.5.1	Security Considerations	1278
19.6	<code>binascii</code> — Convert between binary and ASCII	1278
19.7	<code>quopri</code> — Encode and decode MIME quoted-printable data	1281
<b>20</b>	<b>Structured Markup Processing Tools</b>	<b>1283</b>
20.1	<code>html</code> — HyperText Markup Language support	1283
20.2	<code>html.parser</code> — Simple HTML and XHTML parser	1283
20.2.1	Example HTML Parser Application	1284
20.2.2	HTMLParser Methods	1285
20.2.3	Examples	1286
20.3	<code>html.entities</code> — Definitions of HTML general entities	1288
20.4	XML Processing Modules	1289
20.4.1	XML vulnerabilities	1289
20.4.2	The <code>defusedxml</code> Package	1290
20.5	<code>xml.etree.ElementTree</code> — The ElementTree XML API	1290
20.5.1	Tutorial	1291
20.5.2	XPath support	1296
20.5.3	Reference	1298
20.5.4	XInclude support	1301
20.5.5	Reference	1302
20.6	<code>xml.dom</code> — The Document Object Model API	1310
20.6.1	Module Contents	1311
20.6.2	Objects in the DOM	1311
20.6.3	Conformance	1320
20.7	<code>xml.dom.minidom</code> — Minimal DOM implementation	1321
20.7.1	DOM Objects	1322
20.7.2	DOM Example	1323
20.7.3	minidom and the DOM standard	1325
20.8	<code>xml.dom.pulldom</code> — Support for building partial DOM trees	1325
20.8.1	DOMEventStream Objects	1327
20.9	<code>xml.sax</code> — Support for SAX2 parsers	1327
20.9.1	SAXException Objects	1329
20.10	<code>xml.sax.handler</code> — Base classes for SAX handlers	1329
20.10.1	ContentHandler Objects	1331
20.10.2	DTDHandler Objects	1334
20.10.3	EntityResolver Objects	1334
20.10.4	ErrorHandler Objects	1334
20.10.5	LexicalHandler Objects	1334

20.11	<code>xml.sax.saxutils</code> — SAX Utilities	1335
20.12	<code>xml.sax.xmlreader</code> — Interface for XML parsers	1336
20.12.1	XMLReader Objects	1337
20.12.2	IncrementalParser Objects	1338
20.12.3	Locator Objects	1338
20.12.4	InputSource Objects	1339
20.12.5	The <code>Attributes</code> Interface	1339
20.12.6	The <code>AttributesNS</code> Interface	1340
20.13	<code>xml.parsers.expat</code> — Fast XML parsing using Expat	1340
20.13.1	XMLParser Objects	1341
20.13.2	ExpatError Exceptions	1345
20.13.3	Example	1346
20.13.4	Content Model Descriptions	1346
20.13.5	Expat error constants	1347
<b>21</b>	<b>Internet Protocols and Support</b>	<b>1351</b>
21.1	<code>webbrowser</code> — Convenient web-browser controller	1351
21.1.1	Browser Controller Objects	1354
21.2	<code>wsgiref</code> — WSGI Utilities and Reference Implementation	1354
21.2.1	<code>wsgiref.util</code> — WSGI environment utilities	1354
21.2.2	<code>wsgiref.headers</code> — WSGI response header tools	1356
21.2.3	<code>wsgiref.simple_server</code> — a simple WSGI HTTP server	1357
21.2.4	<code>wsgiref.validate</code> — WSGI conformance checker	1358
21.2.5	<code>wsgiref.handlers</code> — server/gateway base classes	1359
21.2.6	<code>wsgiref.types</code> — WSGI types for static type checking	1363
21.2.7	Examples	1363
21.3	<code>urllib</code> — URL handling modules	1365
21.4	<code>urllib.request</code> — Extensible library for opening URLs	1365
21.4.1	Request Objects	1370
21.4.2	OpenerDirector Objects	1372
21.4.3	BaseHandler Objects	1373
21.4.4	HTTPRedirectHandler Objects	1374
21.4.5	HTTPCookieProcessor Objects	1375
21.4.6	ProxyHandler Objects	1375
21.4.7	HTTPPasswordMgr Objects	1375
21.4.8	HTTPPasswordMgrWithPriorAuth Objects	1376
21.4.9	AbstractBasicAuthHandler Objects	1376
21.4.10	HTTPBasicAuthHandler Objects	1376
21.4.11	ProxyBasicAuthHandler Objects	1376
21.4.12	AbstractDigestAuthHandler Objects	1376
21.4.13	HTTPDigestAuthHandler Objects	1377
21.4.14	ProxyDigestAuthHandler Objects	1377
21.4.15	HTTPHandler Objects	1377
21.4.16	HTTPSHandler Objects	1377
21.4.17	FileHandler Objects	1377
21.4.18	DataHandler Objects	1377
21.4.19	FTPHandler Objects	1377
21.4.20	CacheFTPHandler Objects	1378
21.4.21	UnknownHandler Objects	1378
21.4.22	HTTPErrorProcessor Objects	1378
21.4.23	Examples	1378
21.4.24	Legacy interface	1381
21.4.25	<code>urllib.request</code> Restrictions	1383
21.5	<code>urllib.response</code> — Response classes used by <code>urllib</code>	1384

21.6	<code>urllib.parse</code> — Parse URLs into components	1384
21.6.1	URL Parsing	1385
21.6.2	URL parsing security	1389
21.6.3	Parsing ASCII Encoded Bytes	1390
21.6.4	Structured Parse Results	1390
21.6.5	URL Quoting	1391
21.7	<code>urllib.error</code> — Exception classes raised by <code>urllib.request</code>	1393
21.8	<code>urllib.robotparser</code> — Parser for <code>robots.txt</code>	1394
21.9	<code>http</code> — HTTP modules	1395
21.9.1	HTTP status codes	1396
21.9.2	HTTP methods	1398
21.10	<code>http.client</code> — HTTP protocol client	1398
21.10.1	HTTPConnection Objects	1401
21.10.2	HTTPResponse Objects	1403
21.10.3	Examples	1405
21.10.4	HTTPMessage Objects	1406
21.11	<code>ftplib</code> — FTP protocol client	1406
21.11.1	FTP Objects	1408
21.11.2	FTP_TLS Objects	1411
21.12	<code>poplib</code> — POP3 protocol client	1412
21.12.1	POP3 Objects	1413
21.12.2	POP3 Example	1415
21.13	<code>imaplib</code> — IMAP4 protocol client	1415
21.13.1	IMAP4 Objects	1417
21.13.2	IMAP4 Example	1422
21.14	<code>smtplib</code> — SMTP protocol client	1422
21.14.1	SMTP Objects	1424
21.14.2	SMTP Example	1428
21.15	<code>uuid</code> — UUID objects according to <b>RFC 4122</b>	1429
21.15.1	Example	1432
21.16	<code>socketserver</code> — A framework for network servers	1432
21.16.1	Server Creation Notes	1433
21.16.2	Server Objects	1435
21.16.3	Request Handler Objects	1437
21.16.4	Examples	1437
21.17	<code>http.server</code> — HTTP servers	1441
21.17.1	Security Considerations	1447
21.18	<code>http.cookies</code> — HTTP state management	1447
21.18.1	Cookie Objects	1448
21.18.2	Morsel Objects	1449
21.18.3	Example	1450
21.19	<code>http.cookiejar</code> — Cookie handling for HTTP clients	1451
21.19.1	CookieJar and FileCookieJar Objects	1453
21.19.2	FileCookieJar subclasses and co-operation with web browsers	1455
21.19.3	CookiePolicy Objects	1455
21.19.4	DefaultCookiePolicy Objects	1456
21.19.5	Cookie Objects	1458
21.19.6	Examples	1459
21.20	<code>xmlrpc</code> — XMLRPC server and client modules	1460
21.21	<code>xmlrpc.client</code> — XML-RPC client access	1460
21.21.1	ServerProxy Objects	1462
21.21.2	DateTime Objects	1463
21.21.3	Binary Objects	1464
21.21.4	Fault Objects	1464



21.21.5	ProtocolError Objects	1465
21.21.6	MultiCall Objects	1466
21.21.7	Convenience Functions	1467
21.21.8	Example of Client Usage	1467
21.21.9	Example of Client and Server Usage	1468
21.22	xmlrpc.server — Basic XML-RPC servers	1468
21.22.1	SimpleXMLRPCServer Objects	1469
21.22.2	CGIXMLRPCRequestHandler	1472
21.22.3	Documenting XMLRPC server	1473
21.22.4	DocXMLRPCServer Objects	1473
21.22.5	DocCGIXMLRPCRequestHandler	1474
21.23	ipaddress — IPv4/IPv6 manipulation library	1474
21.23.1	Convenience factory functions	1474
21.23.2	IP Addresses	1475
21.23.3	IP Network definitions	1479
21.23.4	Interface objects	1486
21.23.5	Other Module Level Functions	1487
21.23.6	Custom Exceptions	1488
<b>22</b>	<b>Multimedia Services</b>	<b>1489</b>
22.1	wave — Read and write WAV files	1489
22.1.1	Wave_read Objects	1490
22.1.2	Wave_write Objects	1491
22.2	colorsys — Conversions between color systems	1492
<b>23</b>	<b>Internationalization</b>	<b>1493</b>
23.1	gettext — Multilingual internationalization services	1493
23.1.1	GNU <b>gettext</b> API	1493
23.1.2	Class-based API	1494
23.1.3	Internationalizing your programs and modules	1498
23.1.4	Acknowledgements	1501
23.2	locale — Internationalization services	1501
23.2.1	Background, details, hints, tips and caveats	1508
23.2.2	For extension writers and programs that embed Python	1509
23.2.3	Access to message catalogs	1509
<b>24</b>	<b>Program Frameworks</b>	<b>1511</b>
24.1	turtle — Turtle graphics	1511
24.1.1	Introduction	1511
24.1.2	Tutorial	1512
24.1.3	How to...	1514
24.1.4	Turtle graphics reference	1516
24.1.5	Methods of RawTurtle/Turtle and corresponding functions	1519
24.1.6	Methods of TurtleScreen/Screen and corresponding functions	1536
24.1.7	Public classes	1543
24.1.8	Explanation	1545
24.1.9	Help and configuration	1545
24.1.10	turtledemo — Demo scripts	1548
24.1.11	Changes since Python 2.6	1549
24.1.12	Changes since Python 3.0	1549
24.2	cmd — Support for line-oriented command interpreters	1549
24.2.1	Cmd Objects	1550
24.2.2	Cmd Example	1552
24.3	shlex — Simple lexical analysis	1554

24.3.1	shlex Objects	1556
24.3.2	Parsing Rules	1558
24.3.3	Improved Compatibility with Shells	1559
<b>25</b>	<b>Graphical User Interfaces with Tk</b>	<b>1561</b>
25.1	tkinter — Python interface to Tcl/Tk	1561
25.1.1	Architecture	1562
25.1.2	Tkinter Modules	1563
25.1.3	Tkinter Life Preserver	1564
25.1.4	Threading model	1567
25.1.5	Handy Reference	1568
25.1.6	File Handlers	1573
25.2	tkinter.colorchooser — Color choosing dialog	1574
25.3	tkinter.font — Tkinter font wrapper	1574
25.4	Tkinter Dialogs	1576
25.4.1	tkinter.simpledialog — Standard Tkinter input dialogs	1576
25.4.2	tkinter.filedialog — File selection dialogs	1576
25.4.3	tkinter.commondialog — Dialog window templates	1578
25.5	tkinter.messagebox — Tkinter message prompts	1579
25.6	tkinter.scrolledtext — Scrolled Text Widget	1579
25.7	tkinter.dnd — Drag and drop support	1580
25.8	tkinter.ttk — Tk themed widgets	1581
25.8.1	Using Tk	1581
25.8.2	Tk Widgets	1582
25.8.3	Widget	1582
25.8.4	Combobox	1585
25.8.5	Spinbox	1586
25.8.6	Notebook	1587
25.8.7	Progressbar	1589
25.8.8	Separator	1590
25.8.9	Sizegrip	1590
25.8.10	Treeview	1590
25.8.11	Ttk Styling	1596
25.9	tkinter.tix — Extension widgets for Tk	1599
25.9.1	Using Tix	1600
25.9.2	Tix Widgets	1600
25.9.3	Tix Commands	1603
25.10	IDLE	1604
25.10.1	Menus	1605
25.10.2	Editing and Navigation	1608
25.10.3	Startup and Code Execution	1612
25.10.4	Help and Preferences	1615
25.10.5	idlelib	1616
<b>26</b>	<b>Development Tools</b>	<b>1617</b>
26.1	typing — Support for type hints	1617
26.1.1	Relevant PEPs	1618
26.1.2	Type aliases	1618
26.1.3	NewType	1619
26.1.4	Annotating callable objects	1620
26.1.5	Generics	1622
26.1.6	Annotating tuples	1622
26.1.7	The type of class objects	1623
26.1.8	User-defined generic types	1624

26.1.9	The <code>Any</code> type . . . . .	1626
26.1.10	Nominal vs structural subtyping . . . . .	1627
26.1.11	Module contents . . . . .	1628
26.1.12	Deprecation Timeline of Major Features . . . . .	1664
26.2	<code>pydoc</code> — Documentation generator and online help system . . . . .	1665
26.3	Python Development Mode . . . . .	1666
26.3.1	Effects of the Python Development Mode . . . . .	1666
26.3.2	<code>ResourceWarning</code> Example . . . . .	1667
26.3.3	Bad file descriptor error example . . . . .	1668
26.4	<code>doctest</code> — Test interactive Python examples . . . . .	1669
26.4.1	Simple Usage: Checking Examples in Docstrings . . . . .	1671
26.4.2	Simple Usage: Checking Examples in a Text File . . . . .	1672
26.4.3	How It Works . . . . .	1673
26.4.4	Basic API . . . . .	1680
26.4.5	Unittest API . . . . .	1682
26.4.6	Advanced API . . . . .	1684
26.4.7	Debugging . . . . .	1689
26.4.8	Soapbox . . . . .	1692
26.5	<code>unittest</code> — Unit testing framework . . . . .	1693
26.5.1	Basic example . . . . .	1694
26.5.2	Command-Line Interface . . . . .	1695
26.5.3	Test Discovery . . . . .	1696
26.5.4	Organizing test code . . . . .	1697
26.5.5	Re-using old test code . . . . .	1699
26.5.6	Skipping tests and expected failures . . . . .	1699
26.5.7	Distinguishing test iterations using subtests . . . . .	1701
26.5.8	Classes and functions . . . . .	1702
26.5.9	Class and Module Fixtures . . . . .	1722
26.5.10	Signal Handling . . . . .	1724
26.6	<code>unittest.mock</code> — mock object library . . . . .	1724
26.6.1	Quick Guide . . . . .	1725
26.6.2	The Mock Class . . . . .	1727
26.6.3	The patchers . . . . .	1743
26.6.4	<code>MagicMock</code> and magic method support . . . . .	1753
26.6.5	Helpers . . . . .	1756
26.7	<code>unittest.mock</code> — getting started . . . . .	1765
26.7.1	Using Mock . . . . .	1765
26.7.2	Patch Decorators . . . . .	1770
26.7.3	Further Examples . . . . .	1772
26.8	<code>2to3</code> — Automated Python 2 to 3 code translation . . . . .	1785
26.8.1	Using <code>2to3</code> . . . . .	1785
26.8.2	Fixers . . . . .	1787
26.8.3	<code>lib2to3</code> — <code>2to3</code> 's library . . . . .	1791
26.9	<code>test</code> — Regression tests package for Python . . . . .	1791
26.9.1	Writing Unit Tests for the <code>test</code> package . . . . .	1792
26.9.2	Running tests using the command-line interface . . . . .	1793
26.10	<code>test.support</code> — Utilities for the Python test suite . . . . .	1794
26.11	<code>test.support.socket_helper</code> — Utilities for socket tests . . . . .	1803
26.12	<code>test.support.script_helper</code> — Utilities for the Python execution tests . . . . .	1804
26.13	<code>test.support.bytecode_helper</code> — Support tools for testing correct bytecode generation . . . . .	1805
26.14	<code>test.support.threading_helper</code> — Utilities for threading tests . . . . .	1805
26.15	<code>test.support.os_helper</code> — Utilities for os tests . . . . .	1806
26.16	<code>test.support.import_helper</code> — Utilities for import tests . . . . .	1808
26.17	<code>test.support.warnings_helper</code> — Utilities for warnings tests . . . . .	1810

<b>27</b>	<b>Debugging and Profiling</b>	<b>1813</b>
27.1	Audit events table . . . . .	1813
27.2	bdb — Debugger framework . . . . .	1817
27.3	faulthandler — Dump the Python traceback . . . . .	1823
27.3.1	Dumping the traceback . . . . .	1823
27.3.2	Fault handler state . . . . .	1824
27.3.3	Dumping the tracebacks after a timeout . . . . .	1824
27.3.4	Dumping the traceback on a user signal . . . . .	1824
27.3.5	Issue with file descriptors . . . . .	1825
27.3.6	Example . . . . .	1825
27.4	pdb — The Python Debugger . . . . .	1825
27.4.1	Debugger Commands . . . . .	1828
27.5	The Python Profilers . . . . .	1833
27.5.1	Introduction to the profilers . . . . .	1833
27.5.2	Instant User’s Manual . . . . .	1833
27.5.3	profile and cProfile Module Reference . . . . .	1835
27.5.4	The Stats Class . . . . .	1837
27.5.5	What Is Deterministic Profiling? . . . . .	1839
27.5.6	Limitations . . . . .	1840
27.5.7	Calibration . . . . .	1840
27.5.8	Using a custom timer . . . . .	1841
27.6	timeit — Measure execution time of small code snippets . . . . .	1841
27.6.1	Basic Examples . . . . .	1841
27.6.2	Python Interface . . . . .	1842
27.6.3	Command-Line Interface . . . . .	1844
27.6.4	Examples . . . . .	1845
27.7	trace — Trace or track Python statement execution . . . . .	1846
27.7.1	Command-Line Usage . . . . .	1847
27.7.2	Programmatic Interface . . . . .	1848
27.8	tracemalloc — Trace memory allocations . . . . .	1849
27.8.1	Examples . . . . .	1850
27.8.2	API . . . . .	1854
<b>28</b>	<b>Software Packaging and Distribution</b>	<b>1861</b>
28.1	distutils — Building and installing Python modules . . . . .	1861
28.2	ensurepip — Bootstrapping the pip installer . . . . .	1862
28.2.1	Command line interface . . . . .	1862
28.2.2	Module API . . . . .	1863
28.3	venv — Creation of virtual environments . . . . .	1863
28.3.1	Creating virtual environments . . . . .	1864
28.3.2	How venvs work . . . . .	1866
28.3.3	API . . . . .	1867
28.3.4	An example of extending EnvBuilder . . . . .	1869
28.4	zipapp — Manage executable Python zip archives . . . . .	1873
28.4.1	Basic Example . . . . .	1874
28.4.2	Command-Line Interface . . . . .	1874
28.4.3	Python API . . . . .	1875
28.4.4	Examples . . . . .	1876
28.4.5	Specifying the Interpreter . . . . .	1876
28.4.6	Creating Standalone Applications with zipapp . . . . .	1877
28.4.7	The Python Zip Application Archive Format . . . . .	1879
<b>29</b>	<b>Python Runtime Services</b>	<b>1881</b>
29.1	sys — System-specific parameters and functions . . . . .	1881

29.2	<code>sysconfig</code> — Provide access to Python’s configuration information	1902
29.2.1	Configuration variables	1903
29.2.2	Installation paths	1903
29.2.3	Other functions	1905
29.2.4	Using <code>sysconfig</code> as a script	1906
29.3	<code>builtins</code> — Built-in objects	1907
29.4	<code>__main__</code> — Top-level code environment	1907
29.4.1	<code>__name__ == '__main__'</code>	1908
29.4.2	<code>__main__.py</code> in Python Packages	1910
29.4.3	<code>import __main__</code>	1911
29.5	<code>warnings</code> — Warning control	1913
29.5.1	Warning Categories	1913
29.5.2	The Warnings Filter	1914
29.5.3	Temporarily Suppressing Warnings	1916
29.5.4	Testing Warnings	1917
29.5.5	Updating Code For New Versions of Dependencies	1917
29.5.6	Available Functions	1918
29.5.7	Available Context Managers	1919
29.6	<code>dataclasses</code> — Data Classes	1919
29.6.1	Module contents	1920
29.6.2	Post-init processing	1926
29.6.3	Class variables	1926
29.6.4	Init-only variables	1927
29.6.5	Frozen instances	1927
29.6.6	Inheritance	1927
29.6.7	Re-ordering of keyword-only parameters in <code>__init__()</code>	1928
29.6.8	Default factory functions	1928
29.6.9	Mutable default values	1928
29.6.10	Descriptor-typed fields	1929
29.7	<code>contextlib</code> — Utilities for <code>with</code> -statement contexts	1930
29.7.1	Utilities	1930
29.7.2	Examples and Recipes	1939
29.7.3	Single use, reusable and reentrant context managers	1943
29.8	<code>abc</code> — Abstract Base Classes	1945
29.9	<code>atexit</code> — Exit handlers	1950
29.9.1	<code>atexit</code> Example	1951
29.10	<code>traceback</code> — Print or retrieve a stack traceback	1952
29.10.1	<code>TracebackException</code> Objects	1954
29.10.2	<code>StackSummary</code> Objects	1956
29.10.3	<code>FrameSummary</code> Objects	1956
29.10.4	Traceback Examples	1957
29.11	<code>__future__</code> — Future statement definitions	1959
29.12	<code>gc</code> — Garbage Collector interface	1960
29.13	<code>inspect</code> — Inspect live objects	1964
29.13.1	Types and members	1964
29.13.2	Retrieving source code	1969
29.13.3	Introspecting callables with the Signature object	1970
29.13.4	Classes and functions	1974
29.13.5	The interpreter stack	1977
29.13.6	Fetching attributes statically	1979
29.13.7	Current State of Generators and Coroutines	1980
29.13.8	Code Objects Bit Flags	1981
29.13.9	Command Line Interface	1982
29.14	<code>site</code> — Site-specific configuration hook	1982

29.14.1	Readline configuration . . . . .	1984
29.14.2	Module contents . . . . .	1984
29.14.3	Command Line Interface . . . . .	1985
<b>30</b>	<b>Custom Python Interpreters</b>	<b>1987</b>
30.1	code — Interpreter base classes . . . . .	1987
30.1.1	Interactive Interpreter Objects . . . . .	1988
30.1.2	Interactive Console Objects . . . . .	1988
30.2	codeop — Compile Python code . . . . .	1989
<b>31</b>	<b>Importing Modules</b>	<b>1991</b>
31.1	zipimport — Import modules from Zip archives . . . . .	1991
31.1.1	zipimporter Objects . . . . .	1992
31.1.2	Examples . . . . .	1993
31.2	pkgutil — Package extension utility . . . . .	1993
31.3	modulefinder — Find modules used by a script . . . . .	1997
31.3.1	Example usage of ModuleFinder . . . . .	1997
31.4	runpy — Locating and executing Python modules . . . . .	1998
31.5	importlib — The implementation of import . . . . .	2000
31.5.1	Introduction . . . . .	2000
31.5.2	Functions . . . . .	2001
31.5.3	importlib.abc — Abstract base classes related to import . . . . .	2003
31.5.4	importlib.machinery — Importers and path hooks . . . . .	2009
31.5.5	importlib.util — Utility code for importers . . . . .	2014
31.5.6	Examples . . . . .	2017
31.6	importlib.resources — Resources . . . . .	2020
31.7	Deprecated functions . . . . .	2021
31.8	importlib.resources.abc — Abstract base classes for resources . . . . .	2023
31.9	Using importlib.metadata . . . . .	2024
31.9.1	Overview . . . . .	2025
31.9.2	Functional API . . . . .	2026
31.9.3	Distributions . . . . .	2029
31.9.4	Distribution Discovery . . . . .	2029
31.9.5	Extending the search algorithm . . . . .	2029
31.10	The initialization of the sys.path module search path . . . . .	2030
31.10.1	Virtual environments . . . . .	2031
31.10.2	_pth files . . . . .	2031
31.10.3	Embedded Python . . . . .	2031
<b>32</b>	<b>Python Language Services</b>	<b>2033</b>
32.1	ast — Abstract Syntax Trees . . . . .	2033
32.1.1	Abstract Grammar . . . . .	2033
32.1.2	Node classes . . . . .	2036
32.1.3	ast Helpers . . . . .	2065
32.1.4	Compiler Flags . . . . .	2069
32.1.5	Command-Line Usage . . . . .	2069
32.2	symtable — Access to the compiler’s symbol tables . . . . .	2070
32.2.1	Generating Symbol Tables . . . . .	2070
32.2.2	Examining Symbol Tables . . . . .	2070
32.3	token — Constants used with Python parse trees . . . . .	2072
32.4	keyword — Testing for Python keywords . . . . .	2076
32.5	tokenize — Tokenizer for Python source . . . . .	2076
32.5.1	Tokenizing Input . . . . .	2077
32.5.2	Command-Line Usage . . . . .	2078

32.5.3	Examples	2079
32.6	tabnanny — Detection of ambiguous indentation	2081
32.7	pyclbr — Python module browser support	2081
32.7.1	Function Objects	2082
32.7.2	Class Objects	2083
32.8	py_compile — Compile Python source files	2083
32.8.1	Command-Line Interface	2085
32.9	compileall — Byte-compile Python libraries	2085
32.9.1	Command-line use	2085
32.9.2	Public functions	2087
32.10	dis — Disassembler for Python bytecode	2089
32.10.1	Bytecode analysis	2090
32.10.2	Analysis functions	2091
32.10.3	Python Bytecode Instructions	2093
32.10.4	Opcode collections	2105
32.11	pickletools — Tools for pickle developers	2105
32.11.1	Command line usage	2106
32.11.2	Programmatic Interface	2106
<b>33</b>	<b>MS Windows Specific Services</b>	<b>2109</b>
33.1	msvcrt — Useful routines from the MS VC++ runtime	2109
33.1.1	File Operations	2109
33.1.2	Console I/O	2110
33.1.3	Other Functions	2111
33.2	winreg — Windows registry access	2111
33.2.1	Functions	2111
33.2.2	Constants	2117
33.2.3	Registry Handle Objects	2119
33.3	winsound — Sound-playing interface for Windows	2120
<b>34</b>	<b>Unix Specific Services</b>	<b>2123</b>
34.1	posix — The most common POSIX system calls	2123
34.1.1	Large File Support	2123
34.1.2	Notable Module Contents	2124
34.2	pwd — The password database	2124
34.3	grp — The group database	2125
34.4	termios — POSIX style tty control	2126
34.4.1	Example	2127
34.5	tty — Terminal control functions	2127
34.6	pty — Pseudo-terminal utilities	2128
34.6.1	Example	2129
34.7	fcntl — The fcntl and ioctl system calls	2129
34.8	resource — Resource usage information	2132
34.8.1	Resource Limits	2132
34.8.2	Resource Usage	2135
34.9	syslog — Unix syslog library routines	2136
34.9.1	Examples	2138
<b>35</b>	<b>Superseded Modules</b>	<b>2139</b>
35.1	aifc — Read and write AIFF and AIFC files	2139
35.2	asynchat — Asynchronous socket command/response handler	2141
35.2.1	asynchat Example	2143
35.3	asyncore — Asynchronous socket handler	2144
35.3.1	asyncore Example basic HTTP client	2147



35.3.2	asyncore Example basic echo server . . . . .	2148
35.4	audioop — Manipulate raw audio data . . . . .	2148
35.5	cgi — Common Gateway Interface support . . . . .	2152
35.5.1	Introduction . . . . .	2152
35.5.2	Using the cgi module . . . . .	2153
35.5.3	Higher Level Interface . . . . .	2154
35.5.4	Functions . . . . .	2156
35.5.5	Caring about security . . . . .	2157
35.5.6	Installing your CGI script on a Unix system . . . . .	2157
35.5.7	Testing your CGI script . . . . .	2158
35.5.8	Debugging CGI scripts . . . . .	2158
35.5.9	Common problems and solutions . . . . .	2159
35.6	cgitb — Traceback manager for CGI scripts . . . . .	2159
35.7	chunk — Read IFF chunked data . . . . .	2160
35.8	crypt — Function to check Unix passwords . . . . .	2161
35.8.1	Hashing Methods . . . . .	2162
35.8.2	Module Attributes . . . . .	2162
35.8.3	Module Functions . . . . .	2162
35.8.4	Examples . . . . .	2163
35.9	imghdr — Determine the type of an image . . . . .	2164
35.10	imp — Access the import internals . . . . .	2165
35.10.1	Examples . . . . .	2169
35.11	mailcap — Mailcap file handling . . . . .	2170
35.12	msilib — Read and write Microsoft Installer files . . . . .	2171
35.12.1	Database Objects . . . . .	2172
35.12.2	View Objects . . . . .	2173
35.12.3	Summary Information Objects . . . . .	2173
35.12.4	Record Objects . . . . .	2174
35.12.5	Errors . . . . .	2174
35.12.6	CAB Objects . . . . .	2174
35.12.7	Directory Objects . . . . .	2175
35.12.8	Features . . . . .	2175
35.12.9	GUI classes . . . . .	2176
35.12.10	Precomputed tables . . . . .	2177
35.13	nis — Interface to Sun's NIS (Yellow Pages) . . . . .	2177
35.14	nntplib — NNTP protocol client . . . . .	2178
35.14.1	NNTP Objects . . . . .	2180
35.14.2	Utility functions . . . . .	2185
35.15	optparse — Parser for command line options . . . . .	2185
35.15.1	Background . . . . .	2186
35.15.2	Tutorial . . . . .	2188
35.15.3	Reference Guide . . . . .	2195
35.15.4	Option Callbacks . . . . .	2205
35.15.5	Extending optparse . . . . .	2209
35.15.6	Exceptions . . . . .	2212
35.16	ossaudiodev — Access to OSS-compatible audio devices . . . . .	2213
35.16.1	Audio Device Objects . . . . .	2214
35.16.2	Mixer Device Objects . . . . .	2216
35.17	pipes — Interface to shell pipelines . . . . .	2217
35.17.1	Template Objects . . . . .	2218
35.18	smtpd — SMTP Server . . . . .	2219
35.18.1	SMTPServer Objects . . . . .	2219
35.18.2	DebuggingServer Objects . . . . .	2220
35.18.3	PureProxy Objects . . . . .	2220



35.18.4	SMTPChannel Objects	2220
35.19	sndhdr — Determine type of sound file	2222
35.20	spwd — The shadow password database	2223
35.21	sunau — Read and write Sun AU files	2224
35.21.1	AU_read Objects	2225
35.21.2	AU_write Objects	2226
35.22	telnetlib — Telnet client	2227
35.22.1	Telnet Objects	2228
35.22.2	Telnet Example	2230
35.23	uu — Encode and decode uuencode files	2230
35.24	xdrlib — Encode and decode XDR data	2231
35.24.1	Packer Objects	2231
35.24.2	Unpacker Objects	2233
35.24.3	Exceptions	2234
<b>36</b>	<b>Security Considerations</b>	<b>2235</b>
<b>A</b>	<b>Glossary</b>	<b>2237</b>
<b>B</b>	<b>About these documents</b>	<b>2251</b>
B.1	Contributors to the Python Documentation	2251
<b>C</b>	<b>History and License</b>	<b>2253</b>
C.1	History of the software	2253
C.2	Terms and conditions for accessing or otherwise using Python	2254
C.2.1	PSF LICENSE AGREEMENT FOR PYTHON 3.11.4	2254
C.2.2	BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0	2255
C.2.3	CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1	2256
C.2.4	CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2	2257
C.2.5	ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.11.4 DOCUMENTATION	2257
C.3	Licenses and Acknowledgements for Incorporated Software	2258
C.3.1	Mersenne Twister	2258
C.3.2	Sockets	2259
C.3.3	Asynchronous socket services	2259
C.3.4	Cookie management	2260
C.3.5	Execution tracing	2260
C.3.6	UUencode and UUdecode functions	2261
C.3.7	XML Remote Procedure Calls	2261
C.3.8	test_epoll	2262
C.3.9	Select kqueue	2262
C.3.10	SipHash24	2263
C.3.11	strtod and dtoa	2263
C.3.12	OpenSSL	2264
C.3.13	expat	2267
C.3.14	libffi	2268
C.3.15	zlib	2268
C.3.16	cfuhash	2269
C.3.17	libmpdec	2269
C.3.18	W3C C14N test suite	2270
C.3.19	Audioop	2271
<b>D</b>	<b>Copyright</b>	<b>2273</b>
	<b>Bibliography</b>	<b>2275</b>

<b>Python Module Index</b>	<b>2277</b>
<b>Index</b>	<b>2281</b>

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](#).



## 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, which builds 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.

## 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			
<b>A</b> <code>abs()</code> <code>aiter()</code> <code>all()</code> <code>anext()</code> <code>any()</code> <code>ascii()</code>  <b>B</b> <code>bin()</code> <code>bool()</code> <code>breakpoint()</code> <code>bytearray()</code> <code>bytes()</code>  <b>C</b> <code>callable()</code> <code>chr()</code> <code>classmethod()</code> <code>compile()</code> <code>complex()</code>  <b>D</b> <code>delattr()</code> <code>dict()</code> <code>dir()</code> <code>divmod()</code>	<b>E</b> <code>enumerate()</code> <code>eval()</code> <code>exec()</code>  <b>F</b> <code>filter()</code> <code>float()</code> <code>format()</code> <code>frozenset()</code>  <b>G</b> <code>getattr()</code> <code>globals()</code>  <b>H</b> <code>hasattr()</code> <code>hash()</code> <code>help()</code> <code>hex()</code>  <b>I</b> <code>id()</code> <code>input()</code> <code>int()</code> <code>isinstance()</code> <code>issubclass()</code> <code>iter()</code>	<b>L</b> <code>len()</code> <code>list()</code> <code>locals()</code>  <b>M</b> <code>map()</code> <code>max()</code> <code>memoryview()</code> <code>min()</code>  <b>N</b> <code>next()</code>  <b>O</b> <code>object()</code> <code>oct()</code> <code>open()</code> <code>ord()</code>  <b>P</b> <code>pow()</code> <code>print()</code> <code>property()</code>	<b>R</b> <code>range()</code> <code>repr()</code> <code>reversed()</code> <code>round()</code>  <b>S</b> <code>set()</code> <code>setattr()</code> <code>slice()</code> <code>sorted()</code> <code>staticmethod()</code> <code>str()</code> <code>sum()</code> <code>super()</code>  <b>T</b> <code>tuple()</code> <code>type()</code>  <b>V</b> <code>vars()</code>  <b>Z</b> <code>zip()</code>  <b>_</b> <code>__import__()</code>

**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.

New 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.

New 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** (*x=False*)

Return a Boolean value, i.e. one of `True` or `False`. *x* is converted using the standard *truth testing procedure*. If *x* 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 Values*).

Changed in version 3.7: *x* is now a positional-only parameter.

**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`.

New 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  $0 \leq x < 256$ . 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  $0 \leq x < 256$ , 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  $0 \leq x < 256$ . *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.

New 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.

Changed in version 3.11: 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*.

New 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` (*real*=0, *imag*=0)

**class** `complex` (*string*)

Return a complex number with the value *real* + *imag*\*1j or convert a string or number to a complex number. If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If *imag* is omitted, it defaults to zero and the constructor serves as a numeric conversion like `int` and `float`. If both arguments are omitted, returns `0j`.

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__()`.

---

**Note:** When converting from a string, the string must not contain whitespace around the central + or − operator. For example, `complex('1+2j')` is fine, but `complex('1 + 2j')` raises `ValueError`.

---

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__', 'calcsiz', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
```

(continues on next page)

(continued from previous page)

```

...     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  $0 \leq \text{abs}(a \% b) < \text{abs}(b)$ .

### **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**(*expression*, *globals*=None, *locals*=None)

The arguments are a string and optional globals and locals. If provided, *globals* must be a dictionary. If provided, *locals* can be any mapping object.

The *expression* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* dictionaries 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* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed with the *globals* and *locals* in the environment where `eval()` is called. Note, `eval()` does not have access to the *nested scopes* (non-locals) in the enclosing environment.

The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. 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.

**exec** (*object*, *globals*=None, *locals*=None, */*, *\**, *closure*=None)

This function supports dynamic execution of Python code. *object* 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).<sup>1</sup> 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. If `exec` gets two separate objects as *globals* and *locals*, the code will be executed as if it were embedded 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 dictionary, respectively, which may be useful to pass around for use as the second and third argument to `exec()`.

---

---

**Note:** The default *locals* act as described for function `locals()` below: modifications to the default *locals* dictionary should not be attempted. Pass an explicit *locals* dictionary if you need to see effects of the code on *locals* after function `exec()` returns.

---

---

<sup>1</sup> 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.

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

**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** (*x=0.0*)

Return a floating point number constructed from a number or string *x*.

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"
digitpart     ::= digit (["_"] digit)*
number        ::= [digitpart] "." digitpart | digitpart ["."]
exponent      ::= ("e" | "E") ["+" | "-"] digitpart
floatnumber   ::= number [exponent]
floatvalue    ::= [sign] (floatnumber | infinity | nan)
```

Here `digit` is a Unicode decimal digit (character in the Unicode general category `Nd`). 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.

Examples:

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

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: *x* is now a positional-only parameter.

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. See `__hash__` for details.

---

**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` (`x=0`)

**class** `int` (`x, base=10`)

Return an integer object constructed from a number or string `x`, or return 0 if no arguments are given. If `x` defines `__int__()`, `int(x)` returns `x.__int__()`. If `x` defines `__index__()`, it returns `x.__index__()`. If `x` defines `__trunc__()`, it returns `x.__trunc__()`. For floating point numbers, this truncates towards zero.

If `x` is not a number or if `base` is given, then `x` 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: `x` is now a positional-only parameter.

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 `x` 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** ()

Update and return a dictionary representing the current local symbol table. Free variables are returned by *locals()* when it is called in function blocks, but not in class blocks. Note that at the module level, *locals()* and *globals()* are the same dictionary.

---

**Note:** The contents of this dictionary should not be modified; changes may not affect the values of local and free variables used by the interpreter.

---

**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)`.

New in version 3.4: The *default* keyword-only argument.

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)`.

New in version 3.4: The *default* keyword-only argument.

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 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)
```

(continues on next page)

(continued from previous page)

```

>>> 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.

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

    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.”

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*.

**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=1*)

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

**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, they can be called as regular functions (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.

### **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).

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, \*\*kws*)

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()*. Note, the locals dictionary is only useful for reads since updates to the locals dictionary are ignored.

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).

**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))
```

(continues on next page)

(continued from previous page)

```
>>> 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.





## 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).

## 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.<sup>1</sup> 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
<code>x or y</code>	if <code>x</code> is true, then <code>x</code> , else <code>y</code>	(1)
<code>x and y</code>	if <code>x</code> is false, then <code>x</code> , else <code>y</code>	(2)
<code>not x</code>	if <code>x</code> is false, then <code>True</code> , else <code>False</code>	(3)

---

<sup>1</sup> Additional information on these special methods may be found in the Python Reference Manual (customization).

Notes:

- (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
<code>&lt;</code>	strictly less than
<code>&lt;=</code>	less than or equal
<code>&gt;</code>	strictly greater than
<code>&gt;=</code>	greater than or equal
<code>==</code>	equal
<code>!=</code>	not equal
<code>is</code>	object identity
<code>is not</code>	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.<sup>2</sup>

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

Notes:

- (1) Also referred to as integer division. The resultant value 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 <https://www.unicode.org/Public/14.0.0/ucd/extracted/DerivedNumericType.txt> for a complete list of code points with the `Nd` property.

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

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

Operation	Result
<code>math.trunc(x)</code>	<i>x</i> truncated to <i>Integral</i>
<code>round(x[, n])</code>	<i>x</i> rounded to <i>n</i> digits, rounding half to even. If <i>n</i> is omitted, it defaults to 0.
<code>math.floor(x)</code>	the greatest <i>Integral</i> $\leq x$
<code>math.ceil(x)</code>	the least <i>Integral</i> $\geq x$

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
<code>x   y</code>	bitwise <i>or</i> of <i>x</i> and <i>y</i>	(4)
<code>x ^ y</code>	bitwise <i>exclusive or</i> of <i>x</i> and <i>y</i>	(4)
<code>x &amp; y</code>	bitwise <i>and</i> of <i>x</i> and <i>y</i>	(4)
<code>x &lt;&lt; n</code>	<i>x</i> shifted left by <i>n</i> bits	(1)(2)
<code>x &gt;&gt; n</code>	<i>x</i> shifted right by <i>n</i> bits	(1)(3)
<code>~x</code>	the bits of <i>x</i> 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  $2^{k-1} \leq \text{abs}(x) < 2^k$ . Equivalently, when  $\text{abs}(x)$  is small enough to have a correctly rounded logarithm, then  $k = 1 + \text{int}(\log(\text{abs}(x), 2))$ . 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
```

New 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")
```

New 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)
```

New 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
```

New 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 exactly equal to the original integer and with a positive denominator. The integer ratio of integers (whole numbers) is always the integer as the numerator and 1 as the denominator.

New in version 3.8.

### 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 and with 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 be 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 - 1)`, 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

    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.
```

(continues on next page)

(continued from previous page)

```

        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 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.

### 4.5.1 Generator Types

Python's *generators* 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.

## 4.6 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.

### 4.6.1 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.<sup>3</sup>

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

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.)

---

<sup>3</sup> They must have since the parser can't tell the type of the operands.

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]]
```

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  $s$ :  $\text{len}(s) + i$  or  $\text{len}(s) + j$  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  $i \leq k < j$ . If  $i$  or  $j$  is greater than  $\text{len}(s)$ , use  $\text{len}(s)$ . If  $i$  is omitted or `None`, use 0. If  $j$  is omitted or `None`, use  $\text{len}(s)$ . 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  $0 \leq n < (j-i)/k$ . 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  $\text{len}(s)$  if they are greater. When  $k$  is negative,  $i$  and  $j$  are reduced to  $\text{len}(s) - 1$  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.

## 4.6.2 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*.

## 4.6.3 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
<code>s[i] = x</code>	item <i>i</i> of <i>s</i> is replaced by <i>x</i>	
<code>s[i:j] = t</code>	slice of <i>s</i> from <i>i</i> to <i>j</i> is replaced by the contents of the iterable <i>t</i>	
<code>del s[i:j]</code>	same as <code>s[i:j] = []</code>	
<code>s[i:j:k] = t</code>	the elements of <code>s[i:j:k]</code> are replaced by those of <i>t</i>	(1)
<code>del s[i:j:k]</code>	removes the elements of <code>s[i:j:k]</code> from the list	
<code>s.append(x)</code>	appends <i>x</i> to the end of the sequence (same as <code>s[len(s):len(s)] = [x]</code> )	
<code>s.clear()</code>	removes all items from <i>s</i> (same as <code>del s[:]</code> )	(5)
<code>s.copy()</code>	creates a shallow copy of <i>s</i> (same as <code>s[:]</code> )	(5)
<code>s.extend(t)</code> or <code>s += t</code>	extends <i>s</i> with the contents of <i>t</i> (for the most part the same as <code>s[len(s):len(s)] = t</code> )	
<code>s *= n</code>	updates <i>s</i> with its contents repeated <i>n</i> times	(6)
<code>s.insert(i, x)</code>	inserts <i>x</i> into <i>s</i> at the index given by <i>i</i> (same as <code>s[i:i] = [x]</code> )	
<code>s.pop()</code> or <code>s.pop(i)</code>	retrieves the item at <i>i</i> and also removes it from <i>s</i>	(2)
<code>s.remove(x)</code>	remove the first item from <i>s</i> where <code>s[i]</code> is equal to <i>x</i>	(3)
<code>s.reverse()</code>	reverses the items of <i>s</i> in place	(4)

Notes:

- (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.  
New 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*.

## 4.6.4 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.

## 4.6.5 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.

## 4.6.6 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))
[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).

New in version 3.3: 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.

## 4.7 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.

## 4.7.1 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 of the Unicode Standard.

New 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'
```

New 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 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.

New 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.

Call `keyword.iskeyword()` 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)
```

(continues on next page)

(continued from previous page)

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

**str.islower()**

Return `True` if all cased characters<sup>4</sup> 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<sup>4</sup> 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)`.

---

<sup>4</sup> Cased characters are those with general category property being one of “Lu” (Letter, uppercase), “Ll” (Letter, lowercase), or “Lt” (Letter, titlecase).

`str.lower()`

Return a copy of the string with all the cased characters<sup>4</sup> converted to lowercase.

The lowercasing algorithm used is described in section 3.13 of the Unicode Standard.

`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'
```

New 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'
```

New in version 3.9.

`str.replace(old, new[, count])`

Return a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

`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 (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). 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', '']
```

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
<code>\n</code>	Line Feed
<code>\r</code>	Carriage Return
<code>\r\n</code>	Carriage Return + Line Feed
<code>\v</code> or <code>\x0b</code>	Line Tabulation
<code>\f</code> or <code>\x0c</code>	Form Feed
<code>\x1c</code>	File Separator
<code>\x1d</code>	Group Separator
<code>\x1e</code>	Record Separator
<code>\x85</code>	Next Line (C1 Control Code)
<code>\u2028</code>	Line Separator
<code>\u2029</code>	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()
[]
```

(continues on next page)

(continued from previous page)

```
>>> "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<sup>Page 52, 4</sup> 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 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'
```

## 4.7.2 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()` in the C language.

If *format* requires a single argument, *values* may be a single non-tuple object.<sup>5</sup> 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).

<sup>5</sup> To format only a tuple you should therefore provide a singleton tuple whose only element is the tuple to be formatted.

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:

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 <code>repr()</code> ).	(5)
's'	String (converts any Python object using <code>str()</code> ).	(5)
'a'	String (converts any Python object using <code>ascii()</code> ).	(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.

## 4.8 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.

### 4.8.1 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  $0 \leq x < 256$  (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'
```

New 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*).

## 4.8.2 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'
```

New 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)`.

### 4.8.3 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.

---

New 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.

---

New 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.

New 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!'.rstrip(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'
>>> 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 (for example, `b'1<>2<>3'.split(b'<>')` returns `[b'1', b'2', b'3']`). 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'']
```

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.

New 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\nnde fg\rkl\r\n'.splitlines()
[b'ab c', b'', b'de fg', b'kl']
>>> b'ab c\nnde 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'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.

## 4.8.4 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.<sup>Page 57, 5</sup> 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 <code>__bytes__()</code> ).	(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 <code>repr(obj).encode('ascii', 'backslashreplace')</code> ).	(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.	