

Glossary

>>>

The default Python prompt of the interactive shell. Often seen for code examples which can be executed interactively in the interpreter.

...

The default Python prompt of the interactive shell when entering the code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.

2to3

A tool that tries to convert Python 2.x code to Python 3.x code by handling most of the incompatibilities which can be detected by parsing the source and traversing the parse tree.

2to3 is available in the standard library as [lib2to3](#); a standalone entry point is provided as `Tools/scripts/2to3`. See [2to3 - Automated Python 2 to 3 code translation](#).

abstract base class

Abstract base classes complement [duck-typing](#) by providing a way to define interfaces when other techniques like [hasattr\(\)](#) would be clumsy or subtly wrong (for example with [magic methods](#)). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by [isinstance\(\)](#) and [issubclass\(\)](#); see the [abc](#) module documentation. Python comes with many built-in ABCs for data structures (in the [collections.abc](#) module), numbers (in the [numbers](#) module), streams (in the [io](#) module), import finders and loaders (in the [importlib.abc](#) module). You can create your own ABCs with the [abc](#) module.

annotation

A label associated with a variable, a class attribute or a function parameter or return value, used by convention as a [type hint](#).

Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions are stored in the `__annotations__` special attribute of modules, classes, and functions, respectively.

See [variable annotation](#), [function annotation](#), [PEP 484](#) and [PEP 526](#), which describe this functionality.

argument

A value passed to a [function](#) (or [method](#)) when calling the function. There are two kinds of argument:

- *keyword argument*: an argument preceded by an identifier (e.g. `name=`) in a function call or passed as a value in a dictionary preceded by `**`. For example, 3 and 5 are both

keyword arguments in the following calls to `complex()`:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *positional argument*: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an [iterable](#) preceded by `*`. For example, 3 and 5 are both positional arguments in the following calls:

```
complex(3, 5)
complex(*(3, 5))
```

Arguments are assigned to the named local variables in a function body. See the [Calls](#) section for the rules governing this assignment. Syntactically, any expression can be used to represent an argument; the evaluated value is assigned to the local variable.

See also the [parameter](#) glossary entry, the FAQ question on [the difference between arguments and parameters](#), and [PEP 362](#).

asynchronous context manager

An object which controls the environment seen in an `async with` statement by defining `__aenter__()` and `__aexit__()` methods. Introduced by [PEP 492](#).

asynchronous generator

A function which returns an [asynchronous generator iterator](#). It looks like a coroutine function defined with `async def` except that it contains `yield` expressions for producing a series of values usable in an `async for` loop.

Usually refers to an asynchronous generator function, but may refer to an *asynchronous generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

An asynchronous generator function may contain `await` expressions as well as `async for`, and `async with` statements.

asynchronous generator iterator

An object created by a [asynchronous generator](#) function.

This is an [asynchronous iterator](#) which when called using the `__anext__()` method returns an awaitable object which will execute the body of the asynchronous generator function until the next `yield` expression.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See [PEP 492](#) and [PEP 525](#).

asynchronous iterable

An object, that can be used in an `async for` statement. Must return an [asynchronous iterator](#) from its `__aiter__()` method. Introduced by [PEP 492](#).

asynchronous iterator

An object that implements the `__aiter__()` and `__anext__()` methods. `__anext__` must return an [awaitable](#) object. `async for` resolves the awaitables returned by an asynchronous iterator's `__anext__()` method until it raises a [StopAsyncIteration](#) exception. Introduced by [PEP 492](#).

attribute

A value associated with an object which is referenced by name using dotted expressions. For example, if an object `o` has an attribute `a` it would be referenced as `o.a`.

awaitable

An object that can be used in an `await` expression. Can be a [coroutine](#) or an object with an `__await__()` method. See also [PEP 492](#).

BDFL

Benevolent Dictator For Life, a.k.a. [Guido van Rossum](#), Python's creator.

binary file

A [file object](#) able to read and write [bytes-like objects](#). Examples of binary files are files opened in binary mode ('rb', 'wb' or 'rb+'), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

See also [text file](#) for a file object able to read and write `str` objects.

bytes-like object

An object that supports the [Buffer Protocol](#) and can export a C-contiguous buffer. This includes all `bytes`, `bytearray`, and `array.array` objects, as well as many common [memoryview](#) objects. Bytes-like objects can be used for various operations that work with binary data; these include compression, saving to a binary file, and sending over a socket.

Some operations need the binary data to be mutable. The documentation often refers to these as “read-write bytes-like objects”. Example mutable buffer objects include `bytearray` and a [memoryview](#) of a `bytearray`. Other operations require the binary data to be stored in immutable objects (“read-only bytes-like objects”); examples of these include `bytes` and a [memoryview](#) of a `bytes` object.

bytecode

Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This “intermediate language” is said to run on a [virtual machine](#) that executes the

machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

A list of bytecode instructions can be found in the documentation for [the dis module](#).

class

A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

class variable

A variable defined in a class and intended to be modified only at class level (i.e., not in an instance of the class).

coercion

The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer 3, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a [TypeError](#). Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

complex number

An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of -1), often written *i* in mathematics or *j* in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a *j* suffix, e.g., `3+1j`. To get access to complex equivalents of the [math](#) module, use [cmath](#). Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

context manager

An object which controls the environment seen in a [with](#) statement by defining `__enter__()` and `__exit__()` methods. See [PEP 343](#).

context variable

A variable which can have different values depending on its context. This is similar to Thread-Local Storage in which each execution thread may have a different value for a variable. However, with context variables, there may be several contexts in one execution thread and the main usage for context variables is to keep track of variables in concurrent asynchronous tasks. See [contextvars](#).

contiguous

A buffer is considered contiguous exactly if it is either *C-contiguous* or *Fortran contiguous*. Zero-dimensional buffers are C and Fortran contiguous. In one-dimensional arrays, the items must be laid out in memory next to each other, in order of increasing indexes starting from zero. In multidimensional C-contiguous arrays, the last index varies the fastest when visiting

items in order of memory address. However, in Fortran contiguous arrays, the first index varies the fastest.

coroutine

Coroutines is a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the [async def](#) statement. See also [PEP 492](#).

coroutine function

A function which returns a [coroutine](#) object. A coroutine function may be defined with the [async def](#) statement, and may contain [await](#), [async for](#), and [async with](#) keywords. These were introduced by [PEP 492](#).

CPython

The canonical implementation of the Python programming language, as distributed on [python.org](#). The term “CPython” is used when necessary to distinguish this implementation from others such as Jython or IronPython.

decorator

A function returning another function, usually applied as a function transformation using the [@wrapper](#) syntax. Common examples for decorators are [classmethod\(\)](#) and [staticmethod\(\)](#).

The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```
def f(...):  
    ...  
f = staticmethod(f)  
  
@staticmethod  
def f(...):  
    ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for [function definitions](#) and [class definitions](#) for more about decorators.

descriptor

Any object which defines the methods [__get__\(\)](#), [__set__\(\)](#), or [__delete__\(\)](#). When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

For more information about descriptors' methods, see [Implementing Descriptors](#).

dictionary

An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

dictionary view

The objects returned from `dict.keys()`, `dict.values()`, and `dict.items()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See [Dictionary view objects](#).

docstring

A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

duck-typing

A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. (Note, however, that duck-typing can be complemented with [abstract base classes](#).) Instead, it typically employs `hasattr()` tests or [EAFP](#) programming.

EAFP

Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the [LBYL](#) style common to many other languages such as C.

expression

A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also [statements](#) which cannot be used as expressions, such as `while`. Assignments are also statements, not expressions.

extension module

A module written in C or C++, using Python's C API to interact with the core and with user code.

f-string

String literals prefixed with `'f'` or `'F'` are commonly called "f-strings" which is short for [formatted string literals](#). See also [PEP 498](#).

file object

An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

There are actually three categories of file objects: raw [binary files](#), buffered [binary files](#) and [text files](#). Their interfaces are defined in the [io](#) module. The canonical way to create a file object is by using the [open\(\)](#) function.

file-like object

A synonym for [file object](#).

finder

An object that tries to find the [loader](#) for a module that is being imported.

Since Python 3.3, there are two types of finder: [meta path finders](#) for use with `sys.meta_path`, and [path entry finders](#) for use with `sys.path_hooks`.

See [PEP 302](#), [PEP 420](#) and [PEP 451](#) for much more detail.

floor division

Mathematical division that rounds down to nearest integer. The floor division operator is `//`. For example, the expression `11 // 4` evaluates to 2 in contrast to the 2.75 returned by float true division. Note that `(-11) // 4` is -3 because that is -2.75 rounded *downward*. See [PEP 238](#).

function

A series of statements which returns some value to a caller. It can also be passed zero or more [arguments](#) which may be used in the execution of the body. See also [parameter](#), [method](#), and the [Function definitions](#) section.

function annotation

An [annotation](#) of a function parameter or return value.

Function annotations are usually used for [type hints](#): for example, this function is expected to take two [int](#) arguments and is also expected to have an [int](#) return value:

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

Function annotation syntax is explained in section [Function definitions](#).

See [variable annotation](#) and [PEP 484](#), which describe this functionality.

__future__

A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter.

By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it becomes the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection

The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles. The garbage collector can be controlled using the `gc` module.

generator

A function which returns a [generator iterator](#). It looks like a normal function except that it contains `yield` expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function.

Usually refers to a generator function, but may refer to a *generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

generator iterator

An object created by a [generator](#) function.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

generator expression

An expression that returns an iterator. It looks like a normal expression followed by a for clause defining a loop variable, range, and an optional if clause. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))      # sum of squares 0, 1, 4, ... 81
285
```

generic function

A function composed of multiple functions implementing the same operation for different types. Which implementation should be used during a call is determined by the dispatch algorithm.

See also the [single dispatch](#) glossary entry, the `functools.singledispatch()` decorator, and [PEP 443](#).

GIL

See [global interpreter lock](#).

global interpreter lock

The mechanism used by the [CPython](#) interpreter to assure that only one thread executes Python [bytecode](#) at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as [dict](#)) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

Past efforts to create a “free-threaded” interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case. It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.

hash-based pyc

A bytecode cache file that uses the hash rather than the last-modified time of the corresponding source file to determine its validity. See [Cached bytecode invalidation](#).

hashable

An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a [__hash__\(\)](#) method), and can be compared to other objects (it needs an [__eq__\(\)](#) method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

Most of Python’s immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not; immutable containers (such as tuples and frozensets) are only hashable if their elements are hashable. Objects which are instances of user-defined classes are hashable by default. They all compare unequal (except with themselves), and their hash value is derived from their [id\(\)](#).

IDLE

An Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment which ships with the standard distribution of Python.

immutable

An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

import path

A list of locations (or [path entries](#)) that are searched by the [path based finder](#) for modules to import. During import, this list of locations usually comes from `sys.path`, but for subpackages it may also come from the parent package's `__path__` attribute.

importing

The process by which Python code in one module is made available to Python code in another module.

importer

An object that both finds and loads a module; both a [finder](#) and [loader](#) object.

interactive

Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch python with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`).

interpreted

Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also [interactive](#).

interpreter shutdown

When asked to shut down, the Python interpreter enters a special phase where it gradually releases all allocated resources, such as modules and various critical internal structures. It also makes several calls to the [garbage collector](#). This can trigger the execution of code in user-defined destructors or weakref callbacks. Code executed during the shutdown phase can encounter various exceptions as the resources it relies on may not function anymore (common examples are library modules or the warnings machinery).

The main reason for interpreter shutdown is that the `__main__` module or the script being run has finished executing.

iterable

An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as [list](#), [str](#), and [tuple](#)) and some non-sequence types like [dict](#), [file objects](#), and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements [Sequence](#) semantics.

Iterables can be used in a [for](#) loop and in many other places where a sequence is needed ([zip\(\)](#), [map\(\)](#), ...). When an iterable object is passed as an argument to the built-in function [iter\(\)](#), it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call [iter\(\)](#) or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary

unnamed variable to hold the iterator for the duration of the loop. See also [iterator](#), [sequence](#), and [generator](#).

iterator

An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a [StopIteration](#) exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise [StopIteration](#) again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a [list](#)) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

More information can be found in [Iterator Types](#).

key function

A key function or collation function is a callable that returns a value used for sorting or ordering. For example, `locale.strxfrm()` is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, and `itertools.groupby()`.

There are several ways to create a key function. For example, the `str.lower()` method can serve as a key function for case insensitive sorts. Alternatively, a key function can be built from a `lambda` expression such as `lambda r: (r[0], r[2])`. Also, the `operator` module provides three key function constructors: `attrgetter()`, `itemgetter()`, and `methodcaller()`. See the [Sorting HOW TO](#) for examples of how to create and use key functions.

keyword argument

See [argument](#).

lambda

An anonymous inline function consisting of a single [expression](#) which is evaluated when the function is called. The syntax to create a lambda function is `lambda [parameters]: expression`

LBYL

Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the [EAFP](#) approach and is characterized by the presence of many `if` statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between “the looking” and “the leaping”. For example, the code, `if key in mapping: return mapping[key]` can fail if another thread removes *key* from *mapping* after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

list

A built-in Python [sequence](#). Despite its name it is more akin to an array in other languages than to a linked list since access to elements is $O(1)$.

list comprehension

A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ['{:04x}'.format(x) for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

loader

An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a [finder](#). See [PEP 302](#) for details and [importlib.abc.Loader](#) for an [abstract base class](#).

magic method

An informal synonym for [special method](#).

mapping

A container object that supports arbitrary key lookups and implements the methods specified in the [Mapping](#) or [MutableMapping](#) [abstract base classes](#). Examples include [dict](#), [collections.defaultdict](#), [collections.OrderedDict](#) and [collections.Counter](#).

meta path finder

A [finder](#) returned by a search of `sys.meta_path`. Meta path finders are related to, but different from [path entry finders](#).

See [importlib.abc.MetaPathFinder](#) for the methods that meta path finders implement.

metaclass

The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in [Metaclasses](#).

method

A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first [argument](#) (which is usually called `self`). See [function](#) and [nested scope](#).

method resolution order

Method Resolution Order is the order in which base classes are searched for a member during lookup. See [The Python 2.3 Method Resolution Order](#) for details of the algorithm used by the Python interpreter since the 2.3 release.

module

An object that serves as an organizational unit of Python code. Modules have a namespace containing arbitrary Python objects. Modules are loaded into Python by the process of [importing](#).

See also [package](#).

module spec

A namespace containing the import-related information used to load a module. An instance of [importlib.machinery.ModuleSpec](#).

MRO

See [method resolution order](#).

mutable

Mutable objects can change their value but keep their [id\(\)](#). See also [immutable](#).

named tuple

Any tuple-like class whose indexable elements are also accessible using named attributes (for example, [time.localtime\(\)](#) returns a tuple-like object where the *year* is accessible either with an index such as `t[0]` or with a named attribute like `t.tm_year`).

A named tuple can be a built-in type such as [time.struct_time](#), or it can be created with a regular class definition. A full featured named tuple can also be created with the factory function [collections.namedtuple\(\)](#). The latter approach automatically provides extra features such as a self-documenting representation like `Employee(name='jones', title='programmer')`.

namespace

The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions [builtins.open](#) and [os.open\(\)](#) are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing [random.seed\(\)](#) or [itertools.islice\(\)](#) makes it clear that those functions are implemented by the [random](#) and [itertools](#) modules, respectively.

namespace package

A [PEP 420 package](#) which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a [regular package](#) because they have no `__init__.py` file.

See also [module](#).

nested scope

The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes by default work only for reference and not for assignment. Local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace. The [nonlocal](#) allows writing to outer scopes.

new-style class

Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattr__()`, class methods, and static methods.

object

Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any [new-style class](#).

package

A Python [module](#) which can contain submodules or recursively, subpackages. Technically, a package is a Python module with an `__path__` attribute.

See also [regular package](#) and [namespace package](#).

parameter

A named entity in a [function](#) (or method) definition that specifies an [argument](#) (or in some cases, arguments) that the function can accept. There are five kinds of parameter:

- *positional-or-keyword*: specifies an argument that can be passed either [positionally](#) or as a [keyword argument](#). This is the default kind of parameter, for example *foo* and *bar* in the following:

```
def func(foo, bar=None): ...
```

- *positional-only*: specifies an argument that can be supplied only by position. Python has no syntax for defining positional-only parameters. However, some built-in functions have positional-only parameters (e.g. [abs\(\)](#)).
- *keyword-only*: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare `*` in the parameter list of the function definition before them, for example *kw_only1* and *kw_only2* in the following:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `*`, for example `args` in the following:

```
def func(*args, **kwargs): ...
```

- *var-keyword*: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `**`, for example `kwargs` in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

See also the [argument](#) glossary entry, the FAQ question on [the difference between arguments and parameters](#), the `inspect.Parameter` class, the [Function definitions](#) section, and [PEP 362](#).

path entry

A single location on the [import path](#) which the [path based finder](#) consults to find modules for importing.

path entry finder

A [finder](#) returned by a callable on `sys.path_hooks` (i.e. a [path entry hook](#)) which knows how to locate modules given a [path entry](#).

See `importlib.abc.PathEntryFinder` for the methods that path entry finders implement.

path entry hook

A callable on the `sys.path_hook` list which returns a [path entry finder](#) if it knows how to find modules on a specific [path entry](#).

path based finder

One of the default [meta path finders](#) which searches an [import path](#) for modules.

path-like object

An object representing a file system path. A path-like object is either a [str](#) or [bytes](#) object representing a path, or an object implementing the `os.PathLike` protocol. An object that supports the `os.PathLike` protocol can be converted to a [str](#) or [bytes](#) file system path by calling the `os.fspath()` function; `os.fsdecode()` and `os.fsencode()` can be used to guarantee a [str](#) or [bytes](#) result instead, respectively. Introduced by [PEP 519](#).

PEP

Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. PEPs should provide a concise technical specification and a rationale for proposed features.

PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

See [PEP 1](#).

portion

A set of files in a single directory (possibly stored in a zip file) that contribute to a namespace package, as defined in [PEP 420](#).

positional argument

See [argument](#).

provisional API

A provisional API is one which has been deliberately excluded from the standard library's backwards compatibility guarantees. While major changes to such interfaces are not expected, as long as they are marked provisional, backwards incompatible changes (up to and including removal of the interface) may occur if deemed necessary by core developers. Such changes will not be made gratuitously – they will occur only if serious fundamental flaws are uncovered that were missed prior to the inclusion of the API.

Even for provisional APIs, backwards incompatible changes are seen as a “solution of last resort” - every attempt will still be made to find a backwards compatible resolution to any identified problems.

This process allows the standard library to continue to evolve over time, without locking in problematic design errors for extended periods of time. See [PEP 411](#) for more details.

provisional package

See [provisional API](#).

Python 3000

Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated “Py3k”.

Pythonic

An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a [for](#) statement. Many other languages don't have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
for i in range(len(food)):
    print(food[i])
```

As opposed to the cleaner, Pythonic method:

```
for piece in food:
    print(piece)
```

qualified name

A dotted name showing the “path” from a module’s global scope to a class, function or method defined in that module, as defined in [PEP 3155](#). For top-level functions and classes, the qualified name is the same as the object’s name:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

When used to refer to modules, the *fully qualified name* means the entire dotted path to the module, including any parent packages, e.g. `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count

The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Reference counting is generally not visible to Python code, but it is a key element of the CPython implementation. The `sys` module defines a `getrefcount()` function that programmers can call to return the reference count for a particular object.

regular package

A traditional [package](#), such as a directory containing an `__init__.py` file.

See also [namespace package](#).

`__slots__`

A declaration inside a class that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequence

An [iterable](#) which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence. Some built-in sequence types are [list](#), [str](#), [tuple](#), and [bytes](#). Note that [dict](#) also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary [immutable](#) keys rather than integers.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`.

single dispatch

A form of [generic function](#) dispatch where the implementation is chosen based on the type of a single argument.

slice

An object usually containing a portion of a [sequence](#). A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses [slice](#) objects internally.

special method

A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in [Special method names](#).

statement

A statement is part of a suite (a “block” of code). A statement is either an [expression](#) or one of several constructs with a keyword, such as `if`, `while` or `for`.

struct sequence

A tuple with named elements. Struct sequences expose an interface similar to [named tuple](#) in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `_make()` or `_asdict()`. Examples of struct sequences include `sys.float_info` and the return value of `os.stat()`.

text encoding

A codec which encodes Unicode strings to bytes.

text file

A [file object](#) able to read and write [str](#) objects. Often, a text file actually accesses a byte-oriented datastream and handles the [text encoding](#) automatically. Examples of text files are files opened in text mode (`'r'` or `'w'`), `sys.stdin`, `sys.stdout`, and instances of `io.StringIO`.

See also [binary file](#) for a file object able to read and write [bytes-like objects](#).

triple-quoted string

A string which is bound by three instances of either a quotation mark (") or an apostrophe ('). While they don't provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

type

The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

type alias

A synonym for a type, created by assigning the type to an identifier.

Type aliases are useful for simplifying [type hints](#). For example:

```
from typing import List, Tuple

def remove_gray_shades(
    colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

could be made more readable like this:

```
from typing import List, Tuple

Color = Tuple[int, int, int]

def remove_gray_shades(colors: List[Color]) -> List[Color]:
    pass
```

See [typing](#) and [PEP 484](#), which describe this functionality.

type hint

An [annotation](#) that specifies the expected type for a variable, a class attribute, or a function parameter or return value.

Type hints are optional and are not enforced by Python but they are useful to static type analysis tools, and aid IDEs with code completion and refactoring.

Type hints of global variables, class attributes, and functions, but not local variables, can be accessed using `typing.get_type_hints()`.

See [typing](#) and [PEP 484](#), which describe this functionality.

universal newlines

A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old

Macintosh convention `'\r'`. See [PEP 278](#) and [PEP 3116](#), as well as `bytes.splitlines()` for an additional use.

variable annotation

An [annotation](#) of a variable or a class attribute.

When annotating a variable or a class attribute, assignment is optional:

```
class C:
    field: 'annotation'
```

Variable annotations are usually used for [type hints](#): for example this variable is expected to take `int` values:

```
count: int = 0
```

Variable annotation syntax is explained in section [Annotated assignment statements](#).

See [function annotation](#), [PEP 484](#) and [PEP 526](#), which describe this functionality.

virtual environment

A cooperatively isolated runtime environment that allows Python users and applications to install and upgrade Python distribution packages without interfering with the behaviour of other Python applications running on the same system.

See also [venv](#).

virtual machine

A computer defined entirely in software. Python's virtual machine executes the [bytecode](#) emitted by the bytecode compiler.

Zen of Python

Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing “import this” at the interactive prompt.