3. Data model

3.1. Objects, values and types

Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a "stored program computer", code is also represented by objects.)

Every object has an identity, a type and a value. An object's *identity* never changes once it has been created; you may think of it as the object's address in memory. The `is` operator compares the identity of two objects; the `id()` function returns an integer representing its identity.

CPython implementation detail: For CPython, `id(x)` is the memory address where `x` is stored.

An object's type determines the operations that the object supports (e.g., "does it have a length?") and also defines the possible values for objects of that type. The `type()` function returns an object's type (which is an object itself). Like its identity, an object's *type* is also unchangeable. [1]

The *value* of some objects can change. Objects whose value can change are said to be *mutable*; objects whose value is unchangeable once they are created are called *immutable*. (The value of an immutable container object that contains a reference to a mutable object can change when the latter's value is changed; however the container is still considered immutable, because the collection of objects it contains cannot be changed. So, immutability is not strictly the same as having an unchangeable value, it is more subtle.) An object's mutability is determined by its type; for instance, numbers, strings and tuples are immutable, while dictionaries and lists are mutable.

Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected. An implementation is allowed to postpone garbage collection or omit it altogether — it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable.

CPython implementation detail: CPython currently uses a reference-counting scheme with (optional) delayed detection of cyclically linked garbage, which collects most objects as soon as they become unreachable, but is not guaranteed to collect garbage containing circular references. See the documentation of the `gc` module for information on controlling the collection of cyclic garbage. Other implementations act differently and CPython may change. Do not depend on immediate finalization of objects when they become unreachable (so you should always close files explicitly).

Note that the use of the implementation's tracing or debugging facilities may keep objects alive that would normally be collectable. Also note that catching an exception with a `try`…`except` statement may keep objects alive.

Some objects contain references to "external" resources such as open files or windows. It is understood that these resources are freed when the object is garbage-collected, but since garbage collection is not guaranteed to happen, such objects also provide an explicit way to release the external resource, usually a `close()` method. Programs are strongly recommended to explicitly close such objects. The `try`…`finally` statement and the `with` statement provide convenient ways to do this.

Some objects contain references to other objects; these are called *containers*. Examples of containers are tuples, lists and dictionaries. The references are part of a container's value. In most cases, when we talk about the value of a container, we imply the values, not the identities of the contained objects; however, when we talk about the mutability of a container, only the identities of the immediately contained objects are implied. So, if an immutable container (like a tuple) contains a reference to a mutable object, its value changes if that mutable object is changed.

Types affect almost all aspects of object behavior. Even the importance of object identity is affected in some sense: for immutable types, operations that compute new values may actually return a reference to any existing object with the same type and value, while for mutable objects this is not allowed. For example, after `a = 1; b = 1`, *a* and *b* may or may not refer to the same object with the value one, depending on the implementation. This is because `int` is an immutable type, so the reference to `1` can be reused. This behaviour depends on the implementation used, so should not be relied upon, but is something to be aware of when making use of object identity tests. However, after `c = []; d = []`, *c* and *d* are guaranteed to refer to two different, unique, newly created empty lists. (Note that `e = f = []` assigns the *same* object to both *e* and *f*.)

3.2. The standard type hierarchy

Below is a list of the types that are built into Python. Extension modules (written in C, Java, or other languages, depending on the implementation) can define additional types. Future versions of Python may add types to the type hierarchy (e.g., rational numbers, efficiently stored arrays of integers, etc.), although such additions will often be provided via the standard library instead.

Some of the type descriptions below contain a paragraph listing 'special attributes.' These are attributes that provide access to the implementation and are not intended for general use. Their definition may change in the future.

3.2.1. None

This type has a single value. There is a single object with this value. This object is accessed through the built-in name `None`. It is used to signify the absence of a value in many situations, e.g., it is returned from functions that don't explicitly return anything. Its truth value is false.

3.2.2. NotImplemented

This type has a single value. There is a single object with this value. This object is accessed through the built-in name `NotImplemented`. Numeric methods and rich comparison methods should return this value if they do not implement the operation for the operands provided. (The interpreter will then try the reflected operation, or some other fallback, depending on the operator.) It should not be evaluated in a boolean context.

See Implementing the arithmetic operations for more details.

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.

3.2.3. Ellipsis

This type has a single value. There is a single object with this value. This object is accessed through the literal `...` or the built-in name `Ellipsis`. Its truth value is true.

3.2.4. `numbers.Number`

These are created by numeric literals and returned as results by arithmetic operators and arithmetic built-in functions. Numeric objects are immutable; once created their value never changes. Python numbers are of course strongly related to mathematical numbers, but subject to the limitations of numerical representation in computers.

The string representations of the numeric classes, computed by `__repr__()` and `__str__()`, have the following properties:

- They are valid numeric literals which, when passed to their class constructor, produce an object having the value of the original numeric.
- The representation is in base 10, when possible.
- Leading zeros, possibly excepting a single zero before a decimal point, are not shown.
- Trailing zeros, possibly excepting a single zero after a decimal point, are not shown.
- A sign is shown only when the number is negative.

Python distinguishes between integers, floating-point numbers, and complex numbers:

3.2.4.1. `numbers.Integral`

These represent elements from the mathematical set of integers (positive and negative).

> **Note:** The rules for integer representation are intended to give the most meaningful interpretation of shift and mask operations involving negative integers.

There are two types of integers:

Integers (`int`)

: These represent numbers in an unlimited range, subject to available (virtual) memory only. For the purpose of shift and mask operations, a binary representation is assumed, and negative numbers are represented in a variant of 2's complement which gives the illusion of an infinite string of sign bits extending to the left.

Booleans (`bool`)

: These represent the truth values False and True. The two objects representing the values `False` and `True` are the only Boolean objects. The Boolean type is a subtype of the integer type, and Boolean values behave like the values 0 and 1, respectively, in almost all contexts, the exception being that when converted to a string, the strings `"False"` or `"True"` are returned, respectively.

3.2.4.2. `numbers.Real` (`float`)

These represent machine-level double precision floating-point numbers. You are at the mercy of the underlying machine architecture (and C or Java implementation) for the accepted range and handling of overflow. Python does not support single-precision floating-point numbers; the savings in processor and memory usage that are usually the reason for using these are dwarfed by the overhead of using objects in Python, so there is no reason to complicate the language with two kinds of floating-point numbers.

3.2.4.3. `numbers.Complex` (`complex`)

These represent complex numbers as a pair of machine-level double precision floating-point numbers. The same caveats apply as for floating-point numbers. The real and imaginary parts of a complex number `z` can be retrieved through the read-only attributes `z.real` and `z.imag`.

3.2.5. Sequences

These represent finite ordered sets indexed by non-negative numbers. The built-in function `len()` returns the number of items of a sequence. When the length of a sequence is *n*, the index set contains the numbers 0, 1, …, *n*-1. Item *i* of sequence *a* is selected by `a[i]`. Some sequences, including built-in sequences, interpret negative subscripts by adding the sequence length. For example, `a[-2]` equals `a[n-2]`, the second to last item of sequence a with length `n`.

Sequences also support slicing: `a[i:j]` selects all items with index *k* such that *i* `<=` *k* `<` *j*. When used as an expression, a slice is a sequence of the same type. The comment above about negative indexes also applies to negative slice positions.

Some sequences also support "extended slicing" with a third "step" parameter: `a[i:j:k]` selects all items of *a* with index *x* where `x = i + n*k`, *n* `>=` `0` and *i* `<=` *x* `<` *j*.

Sequences are distinguished according to their mutability:

3.2.5.1. Immutable sequences

An object of an immutable sequence type cannot change once it is created. (If the object contains references to other objects, these other objects may be mutable and may be changed; however, the collection of objects directly referenced by an immutable object cannot change.)

The following types are immutable sequences:

Strings

A string is a sequence of values that represent Unicode code points. All the code points in the range `U+0000 - U+10FFFF` can be represented in a string. Python doesn't have a `char` type; instead, every code point in the string is represented as a string object with length `1`. The built-in function `ord()` converts a code point from its string form to an integer in the range `0 - 10FFFF`; `chr()` converts an integer in the range `0 - 10FFFF` to the corresponding length `1` string object. `str.encode()` can be used to convert a `str` to `bytes` using the given text encoding, and `bytes.decode()` can be used to achieve the opposite.

Tuples

The items of a tuple are arbitrary Python objects. Tuples of two or more items are formed by comma-separated lists of expressions. A tuple of one item (a ‘singleton’) can be formed by affixing a comma to an expression (an expression by itself does not create a tuple, since parentheses must be usable for grouping of expressions). An empty tuple can be formed by an empty pair of parentheses.

Bytes

A bytes object is an immutable array. The items are 8-bit bytes, represented by integers in the range 0 <= x < 256. Bytes literals (like `b'abc'`) and the built-in `bytes()` constructor can be used to create bytes objects. Also, bytes objects can be decoded to strings via the `decode()` method.

3.2.5.2. Mutable sequences

Mutable sequences can be changed after they are created. The subscription and slicing notations can be used as the target of assignment and `del` (delete) statements.

> **Note:** The `collections` and `array` module provide additional examples of mutable sequence types.

There are currently two intrinsic mutable sequence types:

Lists

The items of a list are arbitrary Python objects. Lists are formed by placing a comma-separated list of expressions in square brackets. (Note that there are no special cases needed to form lists of length 0 or 1.)

Byte Arrays

A bytearray object is a mutable array. They are created by the built-in `bytearray()` constructor. Aside from being mutable (and hence unhashable), byte arrays otherwise provide the same interface and functionality as immutable `bytes` objects.

3.2.6. Set types

These represent unordered, finite sets of unique, immutable objects. As such, they cannot be indexed by any subscript. However, they can be iterated over, and the built-in function `len()` returns the number of items in a set. Common uses for sets are fast membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.

For set elements, the same immutability rules apply as for dictionary keys. Note that numeric types obey the normal rules for numeric comparison: if two numbers compare equal (e.g., `1` and `1.0`), only one of them can be contained in a set.

There are currently two intrinsic set types:

Sets

These represent a mutable set. They are created by the built-in `set()` constructor and can be modified afterwards by several methods, such as `add()`.

Frozen sets

These represent an immutable set. They are created by the built-in `frozenset()` constructor. As a frozenset is immutable and hashable, it can be used again as an element of another set, or as a dictionary key.

3.2.7. Mappings

These represent finite sets of objects indexed by arbitrary index sets. The subscript notation `a[k]` selects the item indexed by `k` from the mapping `a`; this can be used in expressions and as the target of assignments or `del` statements. The built-in function `len()` returns the number of items in a mapping.

There is currently a single intrinsic mapping type:

3.2.7.1. Dictionaries

These represent finite sets of objects indexed by nearly arbitrary values. The only types of values not acceptable as keys are values containing lists or dictionaries or other mutable types that are compared by value rather than by object identity, the reason being that the efficient implementation of dictionaries requires a key's hash value to remain constant. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (e.g., `1` and `1.0`) then they can be used interchangeably to index the same dictionary entry.

Dictionaries preserve insertion order, meaning that keys will be produced in the same order they were added sequentially over the dictionary. Replacing an existing key does not change the order, however removing a key and re-inserting it will add it to the end instead of keeping its old place.

Dictionaries are mutable; they can be created by the `{}` notation (see section Dictionary displays).

The extension modules `dbm.ndbm` and `dbm.gnu` provide additional examples of mapping types, as does the `collections` module.

> *Changed in version 3.7:* Dictionaries did not preserve insertion order in versions of Python before 3.6. In CPython 3.6, insertion order was preserved, but it was considered an implementation detail at that time rather than a language guarantee.

3.2.8. Callable types

These are the types to which the function call operation (see section Calls) can be applied:

3.2.8.1. User-defined functions

A user-defined function object is created by a function definition (see section Function definitions). It should be called with an argument list containing the same number of items as the function's formal parameter list.

3.2.8.1.1. Special read-only attributes

Attribute	Meaning
`function.__globals__`	A reference to the `dictionary` that holds the function's global variables – the global namespace of the module in which the function was defined.

Attribute	Meaning
`function.__closure__`	`None` or a `tuple` of cells that contain bindings for the names specified in the `co_freevars` attribute of the function's `code object`. A cell object has the attribute `cell_contents`. This can be used to get the value of the cell, as well as set the value.

3.2.8.1.2. Special writable attributes

Most of these attributes check the type of the assigned value:

Attribute	Meaning
`function.__doc__`	The function's documentation string, or `None` if unavailable.
`function.__name__`	The function's name. See also: `__name__ attributes`.
`function.__qualname__`	The function's qualified name. See also: `__qualname__ attributes`. *Added in version 3.3.*
`function.__module__`	The name of the module the function was defined in, or `None` if unavailable.
`function.__defaults__`	A `tuple` containing default parameter values for those parameters that have defaults, or `None` if no parameters have a default value.
`function.__code__`	The code object representing the compiled function body.
`function.__dict__`	The namespace supporting arbitrary function attributes. See also: `__dict__ attributes`.
`function.__annotations__`	A `dictionary` containing annotations of parameters. The keys of the dictionary are the parameter names, and `'return'` for the return annotation, if provided. See also: Annotations Best Practices.
`function.__kwdefaults__`	A `dictionary` containing defaults for keyword-only parameters.
`function.__type_params__`	A `tuple` containing the type parameters of a generic function. *Added in version 3.12.*

Function objects also support getting and setting arbitrary attributes, which can be used, for example, to attach metadata to functions. Regular attribute dot-notation is used to get and set such attributes.

CPython implementation detail: CPython's current implementation only supports function attributes on user-defined functions. Function attributes on built-in functions may be supported in the future.

Additional information about a function's definition can be retrieved from its code object (accessible via the `__code__` attribute).

3.2.8.2. Instance methods

An instance method object combines a class, a class instance and any callable object (normally a user-defined function).

Special read-only attributes:

`method.__self__`	Refers to the class instance object to which the method is bound
`method.__func__`	Refers to the original function object
`method.__doc__`	The method's documentation (same as `method.__func__.__doc__`). A `string` if the original function had a docstring, else `None`.
`method.__name__`	The name of the method (same as `method.__func__.__name__`)
`method.__module__`	The name of the module the method was defined in, or `None` if unavailable.

Methods also support accessing (but not setting) the arbitrary function attributes on the underlying function object.

User-defined method objects may be created when getting an attribute of a class (perhaps via an instance of that class), if that attribute is a user-defined function object or a `classmethod` object.

When an instance method object is created by retrieving a user-defined function object from a class via one of its instances, its `__self__` attribute is the instance, and the method object is said to be *bound*. The new method's `__func__` attribute is the original function object.

When an instance method object is created by retrieving a `classmethod` object from a class or instance, its `__self__` attribute is the class itself, and its `__func__` attribute is the function object underlying the class method.

When an instance method object is called, the underlying function (`__func__`) is called, inserting the class instance (`__self__`) in front of the argument list. For instance, when `C` is a class which contains a definition for a function `f()`, and `x` is an instance of `C`, calling `x.f(1)` is equivalent to calling `C.f(x, 1)`.

When an instance method object is derived from a `classmethod` object, the "class instance" stored in `__self__` will actually be the class itself, so that calling either `x.f(1)` or `C.f(1)` is equivalent to calling `f(C,1)` where `f` is the underlying function.

It is important to note that user-defined functions which are attributes of a class instance are not converted to bound methods; this *only* happens when the function is an attribute of the class.

3.2.8.3. Generator functions

A function or method which uses the `yield` statement (see section The yield statement) is called a *generator function*. Such a function, when called, always returns an iterator object which can be used to execute the body of the function: calling the iterator's `iterator.__next__()` method will cause the function to execute until it provides a value using the `yield` statement. When the function executes a `return` statement or falls off the end, a `StopIteration` exception is raised and the iterator will have reached the end of the set of values to be returned.

3.2.8.4. Coroutine functions

A function or method which is defined using `async def` is called a *coroutine function*. Such a function, when called, returns a coroutine object. It may contain `await` expressions, as well as `async with` and `async for` statements. See also the Coroutine Objects section.

3.2.8.5. Asynchronous generator functions

A function or method which is defined using `async def` and which uses the `yield` statement is called a *asynchronous generator function*. Such a function, when called, returns an asynchronous iterator object which can be used in an `async for` statement to execute the body of the function.

Calling the asynchronous iterator's `aiterator.__anext__` method will return an awaitable which when awaited will execute until it provides a value using the `yield` expression. When the function executes an empty `return` statement or falls off the end, a `StopAsyncIteration` exception is raised and the asynchronous iterator will have reached the end of the set of values to be yielded.

3.2.8.6. Built-in functions

A built-in function object is a wrapper around a C function. Examples of built-in functions are `len()` and `math.sin()` (`math` is a standard built-in module). The number and type of the arguments are determined by the C function. Special read-only attributes:

- `__doc__` is the function's documentation string, or `None` if unavailable. See `function.__doc__`.
- `__name__` is the function's name. See `function.__name__`.
- `__self__` is set to `None` (but see the next item).
- `__module__` is the name of the module the function was defined in or `None` if unavailable. See `function.__module__`.

3.2.8.7. Built-in methods

This is really a different disguise of a built-in function, this time containing an object passed to the C function as an implicit extra argument. An example of a built-in method is `alist.append()`, assuming *alist* is a list object. In this case, the special read-only attribute `__self__` is set to the object denoted by *alist*. (The attribute has the same semantics as it does with `other instance methods`.)

3.2.8.8. Classes

Classes are callable. These objects normally act as factories for new instances of themselves, but variations are possible for class types that override `__new__()`. The arguments of the call are passed to `__new__()`

and, in the typical case, to `__init__()` to initialize the new instance.

3.2.8.9. Class Instances

Instances of arbitrary classes can be made callable by defining a `__call__()` method in their class.

3.2.9. Modules

Modules are a basic organizational unit of Python code, and are created by the import system as invoked either by the `import` statement, or by calling functions such as `importlib.import_module()` and built-in `__import__()`. A module object has a namespace implemented by a `dictionary` object (this is the dictionary referenced by the `__globals__` attribute of functions defined in the module). Attribute references are translated to lookups in this dictionary, e.g., `m.x` is equivalent to `m.__dict__["x"]`. A module object does not contain the code object used to initialize the module (since it isn't needed once the initialization is done).

Attribute assignment updates the module's namespace dictionary, e.g., `m.x = 1` is equivalent to `m.__dict__["x"] = 1`.

3.2.9.1. Import-related attributes on module objects

Module objects have the following attributes that relate to the import system. When a module is created using the machinery associated with the import system, these attributes are filled in based on the module's spec, before the loader executes and loads the module.

To create a module dynamically rather than using the import system, it's recommended to use `importlib.util.module_from_spec()`, which will set the various import-controlled attributes to appropriate values. It's also possible to use the `types.ModuleType` constructor to create modules directly, but this technique is more error-prone, as most attributes must be manually set on the module object after it has been created when using this approach.

> **Caution:** With the exception of `__name__`, it is **strongly** recommended that you rely on `__spec__` and its attributes instead of any of the other individual attributes listed in this subsection. Note that updating an attribute on `__spec__` will not update the corresponding attribute on the module itself:
>
> ```
> >>> import typing
> >>> typing.__name__, typing.__spec__.name
> ('typing', 'typing')
> >>> typing.__spec__.name = 'spelling'
> >>> typing.__name__, typing.__spec__.name
> ('typing', 'spelling')
> >>> typing.__name__ = 'keyboard_smashing'
> >>> typing.__name__, typing.__spec__.name
> ('keyboard_smashing', 'spelling')
> ```

`module.`**`__name__`**

The name used to uniquely identify the module in the import system. For a directly executed module, this will be set to `"__main__"`.

This attribute must be set to the fully qualified name of the module. It is expected to match the value of `module.__spec__.name`.

`module.__spec__`

A record of the module's import-system-related state.

Set to the `module spec` that was used when importing the module. See Module specs for more details.

> *Added in version 3.4.*

`module.__package__`

The package a module belongs to.

If the module is top-level (that is, not a part of any specific package) then the attribute should be set to `''` (the empty string). Otherwise, it should be set to the name of the module's package (which can be equal to `module.__name__` if the module itself is a package). See **PEP 366** for further details.

This attribute is used instead of `__name__` to calculate explicit relative imports for main modules. It defaults to `None` for modules created dynamically using the `types.ModuleType` constructor; use `importlib.util.module_from_spec()` instead to ensure the attribute is set to a `str`.

It is **strongly** recommended that you use `module.__spec__.parent` instead of `module.__package__`. `__package__` is now only used as a fallback if `__spec__.parent` is not set, and this fallback path is deprecated.

> *Changed in version 3.4:* This attribute now defaults to `None` for modules created dynamically using the `types.ModuleType` constructor. Previously the attribute was optional.

> *Changed in version 3.6:* The value of `__package__` is expected to be the same as `__spec__.parent`. `__package__` is now only used as a fallback during import resolution if `__spec__.parent` is not defined.

> *Changed in version 3.10:* `ImportWarning` is raised if an import resolution falls back to `__package__` instead of `__spec__.parent`.

> *Changed in version 3.12:* Raise `DeprecationWarning` instead of `ImportWarning` when falling back to `__package__` during import resolution.

> *Deprecated since version 3.13, will be removed in version 3.15:* `__package__` will cease to be set or taken into consideration by the import system or standard library.

`module.__loader__`

The loader object that the import machinery used to load the module.

This attribute is mostly useful for introspection, but can be used for additional loader-specific functionality, for example getting data associated with a loader.

`__loader__` defaults to `None` for modules created dynamically using the `types.ModuleType` constructor; use `importlib.util.module_from_spec()` instead to ensure the attribute is set to a loader object.

It is **strongly** recommended that you use `module.__spec__.loader` instead of `module.__loader__`.

> *Changed in version 3.4:* This attribute now defaults to `None` for modules created dynamically using the `types.ModuleType` constructor. Previously the attribute was optional.

> *Deprecated since version 3.12, will be removed in version 3.16:* Setting `__loader__` on a module while failing to set `__spec__.loader` is deprecated. In Python 3.16, `__loader__` will cease to be set or taken into consideration by the import system or the standard library.

`module.`**`__path__`**

A (possibly empty) sequence of strings enumerating the locations where the package's submodules will be found. Non-package modules should not have a `__path__` attribute. See __path__ attributes on modules for more details.

It is **strongly** recommended that you use `module.__spec__.submodule_search_locations` instead of `module.__path__`.

`module.`**`__file__`**

`module.`**`__cached__`**

`__file__` and `__cached__` are both optional attributes that may or may not be set. Both attributes should be a `str` when they are available.

`__file__` indicates the pathname of the file from which the module was loaded (if loaded from a file), or the pathname of the shared library file for extension modules loaded dynamically from a shared library. It might be missing for certain types of modules, such as C modules that are statically linked into the interpreter, and the import system may opt to leave it unset if it has no semantic meaning (for example, a module loaded from a database).

If `__file__` is set then the `__cached__` attribute might also be set, which is the path to any compiled version of the code (for example, a byte-compiled file). The file does not need to exist to set this attribute; the path can simply point to where the compiled file *would* exist (see **PEP 3147**).

Note that `__cached__` may be set even if `__file__` is not set. However, that scenario is quite atypical. Ultimately, the loader is what makes use of the module spec provided by the finder (from which `__file__` and `__cached__` are derived). So if a loader can load from a cached module but otherwise does not load from a file, that atypical scenario may be appropriate.

It is **strongly** recommended that you use `module.__spec__.cached` instead of `module.__cached__`.

> *Deprecated since version 3.13, will be removed in version 3.15:* Setting `__cached__` on a module while failing to set `__spec__.cached` is deprecated. In Python 3.15, `__cached__` will cease to be set or taken into consideration by the import system or standard library.

3.2.9.2. Other writable attributes on module objects

As well as the import-related attributes listed above, module objects also have the following writable attributes:

`module.__doc__`

The module's documentation string, or `None` if unavailable. See also: `__doc__ attributes`.

`module.__annotations__`

A dictionary containing variable annotations collected during module body execution. For best practices on working with `__annotations__`, please see Annotations Best Practices.

3.2.9.3. Module dictionaries

Module objects also have the following special read-only attribute:

`module.__dict__`

The module's namespace as a dictionary object. Uniquely among the attributes listed here, `__dict__` cannot be accessed as a global variable from within a module; it can only be accessed as an attribute on module objects.

CPython implementation detail: Because of the way CPython clears module dictionaries, the module dictionary will be cleared when the module falls out of scope even if the dictionary still has live references. To avoid this, copy the dictionary or keep the module around while using its dictionary directly.

3.2.10. Custom classes

Custom class types are typically created by class definitions (see section Class definitions). A class has a namespace implemented by a dictionary object. Class attribute references are translated to lookups in this dictionary, e.g., `C.x` is translated to `C.__dict__["x"]` (although there are a number of hooks which allow for other means of locating attributes). When the attribute name is not found there, the attribute search continues in the base classes. This search of the base classes uses the C3 method resolution order which behaves correctly even in the presence of 'diamond' inheritance structures where there are multiple inheritance paths leading back to a common ancestor. Additional details on the C3 MRO used by Python can be found at The Python 2.3 Method Resolution Order.

When a class attribute reference (for class `C`, say) would yield a class method object, it is transformed into an instance method object whose `__self__` attribute is `C`. When it would yield a `staticmethod` object, it is transformed into the object wrapped by the static method object. See section Implementing Descriptors for another way in which attributes retrieved from a class may differ from those actually contained in its `__dict__`.

Class attribute assignments update the class's dictionary, never the dictionary of a base class.

A class object can be called (see above) to yield a class instance (see below).

3.2.10.1. Special attributes

Attribute	Meaning
`type.__name__`	The class's name. See also: `__name__ attributes`.

Attribute	Meaning
type.**__qualname__**	The class's qualified name. See also: `__qualname__ attributes`.
type.**__module__**	The name of the module in which the class was defined.
type.**__dict__**	A `mapping proxy` providing a read-only view of the class's namespace. See also: `__dict__ attributes`.
type.**__bases__**	A `tuple` containing the class's bases. In most cases, for a class defined as `class X(A, B, C)`, `X.__bases__` will be exactly equal to `(A, B, C)`.
type.**__doc__**	The class's documentation string, or `None` if undefined. Not inherited by subclasses.
type.**__annotations__**	A dictionary containing variable annotations collected during class body execution. For best practices on working with `__annotations__`, please see Annotations Best Practices. **Caution:** Accessing the `__annotations__` attribute of a class object directly may yield incorrect results in the presence of metaclasses. In addition, the attribute may not exist for some classes. Use `inspect.get_annotations()` to retrieve class annotations safely.
type.**__type_params__**	A `tuple` containing the type parameters of a generic class. *Added in version 3.12.*
type.**__static_attributes__**	A `tuple` containing names of attributes of this class which are assigned through `self.X` from any function in its body. *Added in version 3.13.*
type.**__firstlineno__**	The line number of the first line of the class definition, including decorators. Setting the `__module__` attribute removes the `__firstlineno__` item from the type's dictionary. *Added in version 3.13.*
type.**__mro__**	The `tuple` of classes that are considered when looking for base classes during method resolution.

3.2.10.2. Special methods

In addition to the special attributes described above, all Python classes also have the following two methods available:

type.**mro**()

This method can be overridden by a metaclass to customize the method resolution order for its instances. It is called at class instantiation, and its result is stored in `__mro__`.

type.**__subclasses__**()

Each class keeps a list of weak references to its immediate subclasses. This method returns a list of all those references still alive. The list is in definition order. Example:

```
>>> class A: pass
>>> class B(A): pass
>>> A.__subclasses__()
[<class 'B'>]
```

3.2.11. Class instances

A class instance is created by calling a class object (see above). A class instance has a namespace implemented as a dictionary which is the first place in which attribute references are searched. When an attribute is not found there, and the instance's class has an attribute by that name, the search continues with the class attributes. If a class attribute is found that is a user-defined function object, it is transformed into an instance method object whose `__self__` attribute is the instance. Static method and class method objects are also transformed; see above under "Classes". See section Implementing Descriptors for another way in which attributes of a class retrieved via its instances may differ from the objects actually stored in the class's `__dict__`. If no class attribute is found, and the object's class has a `__getattr__()` method, that is called to satisfy the lookup.

Attribute assignments and deletions update the instance's dictionary, never a class's dictionary. If the class has a `__setattr__()` or `__delattr__()` method, this is called instead of updating the instance dictionary directly.

Class instances can pretend to be numbers, sequences, or mappings if they have methods with certain special names. See section Special method names.

3.2.11.1. Special attributes

object.**__class__**

The class to which a class instance belongs.

object.**__dict__**

A dictionary or other mapping object used to store an object's (writable) attributes. Not all instances have a `__dict__` attribute; see the section on __slots__ for more details.

3.2.12. I/O objects (also known as file objects)

A file object represents an open file. Various shortcuts are available to create file objects: the `open()` built-in function, and also `os.popen()`, `os.fdopen()`, and the `makefile()` method of socket objects (and perhaps by other functions or methods provided by extension modules).

The objects `sys.stdin`, `sys.stdout` and `sys.stderr` are initialized to file objects corresponding to the interpreter's standard input, output and error streams; they are all open in text mode and therefore follow the interface defined by the `io.TextIOBase` abstract class.

3.2.13. Internal types

A few types used internally by the interpreter are exposed to the user. Their definitions may change with future versions of the interpreter, but they are mentioned here for completeness.

3.2.13.1. Code objects

Code objects represent *byte-compiled* executable Python code, or bytecode. The difference between a code object and a function object is that the function object contains an explicit reference to the function's globals (the module in which it was defined), while a code object contains no context; also the default argument values are stored in the function object, not in the code object (because they represent values calculated at run-time). Unlike function objects, code objects are immutable and contain no references (directly or indirectly) to mutable objects.

3.2.13.1.1. Special read-only attributes

`codeobject.`**`co_name`**	The function name
`codeobject.`**`co_qualname`**	The fully qualified function name *Added in version 3.11.*
`codeobject.`**`co_argcount`**	The total number of positional parameters (including positional-only parameters and parameters with default values) that the function has
`codeobject.`**`co_posonlyargcount`**	The number of positional-only parameters (including arguments with default values) that the function has
`codeobject.`**`co_kwonlyargcount`**	The number of keyword-only parameters (including arguments with default values) that the function has
`codeobject.`**`co_nlocals`**	The number of local variables used by the function (including parameters)
`codeobject.`**`co_varnames`**	A `tuple` containing the names of the local variables in the function (starting with the parameter names)
`codeobject.`**`co_cellvars`**	A `tuple` containing the names of local variables that are referenced from at least one nested scope inside the function
`codeobject.`**`co_freevars`**	A `tuple` containing the names of free (closure) variables that a nested scope references in an outer scope. See also `function.__closure__`.

	Note: references to global and builtin names are *not* included.
`codeobject.co_code`	A string representing the sequence of bytecode instructions in the function
`codeobject.co_consts`	A `tuple` containing the literals used by the bytecode in the function
`codeobject.co_names`	A `tuple` containing the names used by the bytecode in the function
`codeobject.co_filename`	The name of the file from which the code was compiled
`codeobject.co_firstlineno`	The line number of the first line of the function
`codeobject.co_lnotab`	A string encoding the mapping from bytecode offsets to line numbers. For details, see the source code of the interpreter. *Deprecated since version 3.12:* This attribute of code objects is deprecated, and may be removed in Python 3.15.
`codeobject.co_stacksize`	The required stack size of the code object
`codeobject.co_flags`	An `integer` encoding a number of flags for the interpreter.

The following flag bits are defined for `co_flags`: bit `0x04` is set if the function uses the `*arguments` syntax to accept an arbitrary number of positional arguments; bit `0x08` is set if the function uses the `**keywords` syntax to accept arbitrary keyword arguments; bit `0x20` is set if the function is a generator. See Code Objects Bit Flags for details on the semantics of each flags that might be present.

Future feature declarations (for example, `from __future__ import division`) also use bits in `co_flags` to indicate whether a code object was compiled with a particular feature enabled. See `compiler_flag`.

Other bits in `co_flags` are reserved for internal use.

If a code object represents a function, the first item in `co_consts` is the documentation string of the function, or `None` if undefined.

3.2.13.1.2. Methods on code objects

`codeobject.co_positions()`

Returns an iterable over the source code positions of each bytecode instruction in the code object.

The iterator returns `tuple`s containing the `(start_line, end_line, start_column, end_column)`. The *i-th* tuple corresponds to the position of the source code that compiled to the *i-th* code unit. Column information is 0-indexed utf-8 byte offsets on the given source line.

This positional information can be missing. A non-exhaustive lists of cases where this may happen:

- Running the interpreter with `-X no_debug_ranges`.
- Loading a pyc file compiled while using `-X no_debug_ranges`.

- Position tuples corresponding to artificial instructions.
- Line and column numbers that can't be represented due to implementation specific limitations.

When this occurs, some or all of the tuple elements can be `None`.

> *Added in version 3.11.*

> **Note:** This feature requires storing column positions in code objects which may result in a small increase of disk usage of compiled Python files or interpreter memory usage. To avoid storing the extra information and/or deactivate printing the extra traceback information, the `-X no_debug_ranges` command line flag or the `PYTHONNODEBUGRANGES` environment variable can be used.

`codeobject.`**`co_lines()`**

Returns an iterator that yields information about successive ranges of bytecodes. Each item yielded is a `(start, end, lineno)` `tuple`:

- `start` (an `int`) represents the offset (inclusive) of the start of the bytecode range
- `end` (an `int`) represents the offset (exclusive) of the end of the bytecode range
- `lineno` is an `int` representing the line number of the bytecode range, or `None` if the bytecodes in the given range have no line number

The items yielded will have the following properties:

- The first range yielded will have a `start` of 0.
- The `(start, end)` ranges will be non-decreasing and consecutive. That is, for any pair of `tuple`s, the `start` of the second will be equal to the `end` of the first.
- No range will be backwards: `end >= start` for all triples.
- The last `tuple` yielded will have `end` equal to the size of the bytecode.

Zero-width ranges, where `start == end`, are allowed. Zero-width ranges are used for lines that are present in the source code, but have been eliminated by the bytecode compiler.

> *Added in version 3.10.*

> **See also:**
> **PEP 626 - Precise line numbers for debugging and other tools.**
> The PEP that introduced the `co_lines()` method.

`codeobject.`**`replace`**`(**kwargs)`

Return a copy of the code object with new values for the specified fields.

Code objects are also supported by the generic function `copy.replace()`.

> *Added in version 3.8.*

3.2.13.2. Frame objects

Frame objects represent execution frames. They may occur in traceback objects, and are also passed to registered trace functions.

3.2.13.2.1. Special read-only attributes

`frame.f_back`	Points to the previous stack frame (towards the caller), or `None` if this is the bottom stack frame
`frame.f_code`	The code object being executed in this frame. Accessing this attribute raises an auditing event `object.__getattr__` with arguments `obj` and `"f_code"`.
`frame.f_locals`	The mapping used by the frame to look up local variables. If the frame refers to an optimized scope, this may return a write-through proxy object. *Changed in version 3.13:* Return a proxy for optimized scopes.
`frame.f_globals`	The dictionary used by the frame to look up global variables
`frame.f_builtins`	The dictionary used by the frame to look up built-in (intrinsic) names
`frame.f_lasti`	The "precise instruction" of the frame object (this is an index into the bytecode string of the code object)

3.2.13.2.2. Special writable attributes

`frame.f_trace`	If not `None`, this is a function called for various events during code execution (this is used by debuggers). Normally an event is triggered for each new source line (see `f_trace_lines`).
`frame.f_trace_lines`	Set this attribute to `False` to disable triggering a tracing event for each source line.
`frame.f_trace_opcodes`	Set this attribute to `True` to allow per-opcode events to be requested. Note that this may lead to undefined interpreter behaviour if exceptions raised by the trace function escape to the function being traced.
`frame.f_lineno`	The current line number of the frame – writing to this from within a trace function jumps to the given line (only for the bottom-most frame). A debugger can implement a Jump command (aka Set Next Statement) by writing to this attribute.

3.2.13.2.3. Frame object methods

Frame objects support one method:

`frame.clear()`

This method clears all references to local variables held by the frame. Also, if the frame belonged to a generator, the generator is finalized. This helps break reference cycles involving frame objects (for example when catching an exception and storing its traceback for later use).

`RuntimeError` is raised if the frame is currently executing or suspended.

> *Added in version 3.4.*

> *Changed in version 3.13:* Attempting to clear a suspended frame raises `RuntimeError` (as has always been the case for executing frames).

3.2.13.3. Traceback objects

Traceback objects represent the stack trace of an exception. A traceback object is implicitly created when an exception occurs, and may also be explicitly created by calling `types.TracebackType`.

> *Changed in version 3.7:* Traceback objects can now be explicitly instantiated from Python code.

For implicitly created tracebacks, when the search for an exception handler unwinds the execution stack, at each unwound level a traceback object is inserted in front of the current traceback. When an exception handler is entered, the stack trace is made available to the program. (See section The try statement.) It is accessible as the third item of the tuple returned by `sys.exc_info()`, and as the `__traceback__` attribute of the caught exception.

When the program contains no suitable handler, the stack trace is written (nicely formatted) to the standard error stream; if the interpreter is interactive, it is also made available to the user as `sys.last_traceback`.

For explicitly created tracebacks, it is up to the creator of the traceback to determine how the `tb_next` attributes should be linked to form a full stack trace.

Special read-only attributes:

`traceback.`**`tb_frame`**	Points to the execution frame of the current level. Accessing this attribute raises an auditing event `object.__getattr__` with arguments `obj` and `"tb_frame"`.
`traceback.`**`tb_lineno`**	Gives the line number where the exception occurred
`traceback.`**`tb_lasti`**	Indicates the “precise instruction”.

The line number and last instruction in the traceback may differ from the line number of its frame object if the exception occurred in a `try` statement with no matching except clause or with a `finally` clause.

`traceback.`**`tb_next`**

The special writable attribute `tb_next` is the next level in the stack trace (towards the frame where the exception occurred), or `None` if there is no next level.

> *Changed in version 3.7:* This attribute is now writable

3.2.13.4. Slice objects

Slice objects are used to represent slices for `__getitem__()` methods. They are also created by the built-in `slice()` function.

Special read-only attributes: `start` is the lower bound; `stop` is the upper bound; `step` is the step value; each is `None` if omitted. These attributes can have any type.

Slice objects support one method:

`slice.indices(self, length)`

This method takes a single integer argument *length* and computes information about the slice that the slice object would describe if applied to a sequence of *length* items. It returns a tuple of three integers; respectively these are the *start* and *stop* indices and the *step* or stride length of the slice. Missing or out-of-bounds indices are handled in a manner consistent with regular slices.

3.2.13.5. Static method objects

Static method objects provide a way of defeating the transformation of function objects to method objects described above. A static method object is a wrapper around any other object, usually a user-defined method object. When a static method object is retrieved from a class or a class instance, the object actually returned is the wrapped object, which is not subject to any further transformation. Static method objects are also callable. Static method objects are created by the built-in `staticmethod()` constructor.

3.2.13.6. Class method objects

A class method object, like a static method object, is a wrapper around another object that alters the way in which that object is retrieved from classes and class instances. The behaviour of class method objects upon such retrieval is described above, under "instance methods". Class method objects are created by the built-in `classmethod()` constructor.

3.3. Special method names

A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. This is Python's approach to *operator overloading*, allowing classes to define their own behavior with respect to language operators. For instance, if a class defines a method named `__getitem__()`, and `x` is an instance of this class, then `x[i]` is roughly equivalent to `type(x).__getitem__(x, i)`. Except where mentioned, attempts to execute an operation raise an exception when no appropriate method is defined (typically `AttributeError` or `TypeError`).

Setting a special method to `None` indicates that the corresponding operation is not available. For example, if a class sets `__iter__()` to `None`, the class is not iterable, so calling `iter()` on its instances will raise a `TypeError` (without falling back to `__getitem__()`). [2]

When implementing a class that emulates any built-in type, it is important that the emulation only be implemented to the degree that it makes sense for the object being modelled. For example, some sequences may work well with retrieval of individual elements, but extracting a slice may not make sense. (One example of this is the `NodeList` interface in the W3C's Document Object Model.)

3.3.1. Basic customization

`object.__new__(cls[, ...])`

Called to create a new instance of class *cls*. `__new__()` is a static method (special-cased so you need not declare it as such) that takes the class of which an instance was requested as its first argument. The remaining arguments are those passed to the object constructor expression (the call to the class). The return value of `__new__()` should be the new object instance (usually an instance of *cls*).

Typical implementations create a new instance of the class by invoking the superclass's `__new__()` method using `super().__new__(cls[, ...])` with appropriate arguments and then modifying the newly created instance as necessary before returning it.

If `__new__()` is invoked during object construction and it returns an instance of *cls*, then the new instance's `__init__()` method will be invoked like `__init__(self[, ...])`, where *self* is the new instance and the remaining arguments are the same as were passed to the object constructor.

If `__new__()` does not return an instance of *cls*, then the new instance's `__init__()` method will not be invoked.

`__new__()` is intended mainly to allow subclasses of immutable types (like int, str, or tuple) to customize instance creation. It is also commonly overridden in custom metaclasses in order to customize class creation.

`object.__init__(self[, ...])`

Called after the instance has been created (by `__new__()`), but before it is returned to the caller. The arguments are those passed to the class constructor expression. If a base class has an `__init__()` method, the derived class's `__init__()` method, if any, must explicitly call it to ensure proper initialization of the base class part of the instance; for example: `super().__init__([args...])`.

Because `__new__()` and `__init__()` work together in constructing objects (`__new__()` to create it, and `__init__()` to customize it), no non-`None` value may be returned by `__init__()`; doing so will cause a `TypeError` to be raised at runtime.

`object.__del__(self)`

Called when the instance is about to be destroyed. This is also called a finalizer or (improperly) a destructor. If a base class has a `__del__()` method, the derived class's `__del__()` method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance.

It is possible (though not recommended!) for the `__del__()` method to postpone destruction of the instance by creating a new reference to it. This is called object *resurrection*. It is implementation-dependent whether `__del__()` is called a second time when a resurrected object is about to be destroyed; the current CPython implementation only calls it once.

It is not guaranteed that `__del__()` methods are called for objects that still exist when the interpreter exits. `weakref.finalize` provides a straightforward way to register a cleanup function to be called when an object is garbage collected.

Note: `del x` doesn't directly call `x.__del__()` — the former decrements the reference count for `x` by one, and the latter is only called when `x`'s reference count reaches zero.

CPython implementation detail: It is possible for a reference cycle to prevent the reference count of an object from going to zero. In this case, the cycle will be later detected and deleted by the cyclic garbage collector. A common cause of reference cycles is when an exception has been caught in a local variable. The frame's locals then reference the exception, which references its own traceback, which references the locals of all frames caught in the traceback.

See also: Documentation for the `gc` module.

Warning: Due to the precarious circumstances under which `__del__()` methods are invoked, exceptions that occur during their execution are ignored, and a warning is printed to `sys.stderr` instead. In particular:

- `__del__()` can be invoked when arbitrary code is being executed, including from any arbitrary thread. If `__del__()` needs to take a lock or invoke any other blocking resource, it may deadlock as the resource may already be taken by the code that gets interrupted to execute `__del__()`.
- `__del__()` can be executed during interpreter shutdown. As a consequence, the global variables it needs to access (including other modules) may already have been deleted or set to `None`. Python guarantees that globals whose name begins with a single underscore are deleted from their module before other globals are deleted; if no other references to such globals exist, this may help in assuring that imported modules are still available at the time when the `__del__()` method is called.

`object.`**`__repr__`**`(`*`self`*`)`

Called by the `repr()` built-in function to compute the "official" string representation of an object. If at all possible, this should look like a valid Python expression that could be used to recreate an object with the same value (given an appropriate environment). If this is not possible, a string of the form `<...some useful description...>` should be returned. The return value must be a string object. If a class defines `__repr__()` but not `__str__()`, then `__repr__()` is also used when an "informal" string representation of instances of that class is required.

This is typically used for debugging, so it is important that the representation is information-rich and unambiguous. A default implementation is provided by the `object` class itself.

`object.`**`__str__`**`(`*`self`*`)`

Called by `str(object)`, the default `__format__()` implementation, and the built-in function `print()`, to compute the "informal" or nicely printable string representation of an object. The return value must be a str object.

This method differs from `object.__repr__()` in that there is no expectation that `__str__()` return a valid Python expression: a more convenient or concise representation can be used.

The default implementation defined by the built-in type `object` calls `object.__repr__()`.

`object.`**`__bytes__`**`(`*`self`*`)`

Called by `bytes` to compute a byte-string representation of an object. This should return a `bytes` object. The `object` class itself does not provide this method.

`object.__format__(self, format_spec)`

Called by the `format()` built-in function, and by extension, evaluation of formatted string literals and the `str.format()` method, to produce a "formatted" string representation of an object. The *format_spec* argument is a string that contains a description of the formatting options desired. The interpretation of the *format_spec* argument is up to the type implementing `__format__()`, however most classes will either delegate formatting to one of the built-in types, or use a similar formatting option syntax.

See Format Specification Mini-Language for a description of the standard formatting syntax.

The return value must be a string object.

The default implementation by the `object` class should be given an empty *format_spec* string. It delegates to `__str__()`.

> *Changed in version 3.4:* The __format__ method of `object` itself raises a `TypeError` if passed any non-empty string.

> *Changed in version 3.7:* `object.__format__(x, '')` is now equivalent to `str(x)` rather than `format(str(x), '')`.

`object.__lt__(self, other)`
`object.__le__(self, other)`
`object.__eq__(self, other)`
`object.__ne__(self, other)`
`object.__gt__(self, other)`
`object.__ge__(self, other)`

These are the so-called "rich comparison" methods. The correspondence between operator symbols and method names is as follows: `x<y` calls `x.__lt__(y)`, `x<=y` calls `x.__le__(y)`, `x==y` calls `x.__eq__(y)`, `x!=y` calls `x.__ne__(y)`, `x>y` calls `x.__gt__(y)`, and `x>=y` calls `x.__ge__(y)`.

A rich comparison method may return the singleton `NotImplemented` if it does not implement the operation for a given pair of arguments. By convention, `False` and `True` are returned for a successful comparison. However, these methods can return any value, so if the comparison operator is used in a Boolean context (e.g., in the condition of an `if` statement), Python will call `bool()` on the value to determine if the result is true or false.

By default, `object` implements `__eq__()` by using `is`, returning `NotImplemented` in the case of a false comparison: `True if x is y else NotImplemented`. For `__ne__()`, by default it delegates to `__eq__()` and inverts the result unless it is `NotImplemented`. There are no other implied relationships among the comparison operators or default implementations; for example, the truth of `(x<y or x==y)` does not imply `x<=y`. To automatically generate ordering operations from a single root operation, see `functools.total_ordering()`.

By default, the `object` class provides implementations consistent with Value comparisons: equality compares according to object identity, and order comparisons raise `TypeError`. Each default method may generate these results directly, but may also return `NotImplemented`.

See the paragraph on `__hash__()` for some important notes on creating hashable objects which support custom comparison operations and are usable as dictionary keys.

There are no swapped-argument versions of these methods (to be used when the left argument does not support the operation but the right argument does); rather, `__lt__()` and `__gt__()` are each other's reflection, `__le__()` and `__ge__()` are each other's reflection, and `__eq__()` and `__ne__()` are their own reflection. If the operands are of different types, and the right operand's type is a direct or indirect subclass of the left operand's type, the reflected method of the right operand has priority, otherwise the left operand's method has priority. Virtual subclassing is not considered.

When no appropriate method returns any value other than `NotImplemented`, the `==` and `!=` operators will fall back to `is` and `is not`, respectively.

`object.`**`__hash__`**`(`*`self`*`)`

Called by built-in function `hash()` and for operations on members of hashed collections including `set`, `frozenset`, and `dict`. The `__hash__()` method should return an integer. The only required property is that objects which compare equal have the same hash value; it is advised to mix together the hash values of the components of the object that also play a part in comparison of objects by packing them into a tuple and hashing the tuple. Example:

```
def __hash__(self):
    return hash((self.name, self.nick, self.color))
```

Note: `hash()` truncates the value returned from an object's custom `__hash__()` method to the size of a `Py_ssize_t`. This is typically 8 bytes on 64-bit builds and 4 bytes on 32-bit builds. If an object's `__hash__()` must interoperate on builds of different bit sizes, be sure to check the width on all supported builds. An easy way to do this is with `python -c "import sys; print(sys.hash_info.width)"`.

If a class does not define an `__eq__()` method it should not define a `__hash__()` operation either; if it defines `__eq__()` but not `__hash__()`, its instances will not be usable as items in hashable collections. If a class defines mutable objects and implements an `__eq__()` method, it should not implement `__hash__()`, since the implementation of hashable collections requires that a key's hash value is immutable (if the object's hash value changes, it will be in the wrong hash bucket).

User-defined classes have `__eq__()` and `__hash__()` methods by default (inherited from the `object` class); with them, all objects compare unequal (except with themselves) and `x.__hash__()` returns an appropriate value such that `x == y` implies both that `x is y` and `hash(x) == hash(y)`.

A class that overrides `__eq__()` and does not define `__hash__()` will have its `__hash__()` implicitly set to `None`. When the `__hash__()` method of a class is `None`, instances of the class will raise an appropriate `TypeError` when a program attempts to retrieve their hash value, and will also be correctly identified as unhashable when checking `isinstance(obj, collections.abc.Hashable)`.

If a class that overrides `__eq__()` needs to retain the implementation of `__hash__()` from a parent class, the interpreter must be told this explicitly by setting `__hash__ = <ParentClass>.__hash__`.

If a class that does not override `__eq__()` wishes to suppress hash support, it should include `__hash__ = None` in the class definition. A class which defines its own `__hash__()` that explicitly raises a `TypeError` would be incorrectly identified as hashable by an `isinstance(obj, collections.abc.Hashable)` call.

> **Note:** By default, the `__hash__()` values of str and bytes objects are "salted" with an unpredictable random value. Although they remain constant within an individual Python process, they are not predictable between repeated invocations of Python.
>
> This is intended to provide protection against a denial-of-service caused by carefully chosen inputs that exploit the worst case performance of a dict insertion, $O(n^2)$ complexity. See http://ocert.org/advisories/ocert-2011-003.html for details.
>
> Changing hash values affects the iteration order of sets. Python has never made guarantees about this ordering (and it typically varies between 32-bit and 64-bit builds).
>
> See also `PYTHONHASHSEED`.

Changed in version 3.3: Hash randomization is enabled by default.

object.**__bool__**(*self*)

Called to implement truth value testing and the built-in operation `bool()`; should return `False` or `True`. When this method is not defined, `__len__()` is called, if it is defined, and the object is considered true if its result is nonzero. If a class defines neither `__len__()` nor `__bool__()` (which is true of the `object` class itself), all its instances are considered true.

3.3.2. Customizing attribute access

The following methods can be defined to customize the meaning of attribute access (use of, assignment to, or deletion of `x.name`) for class instances.

object.**__getattr__**(*self*, *name*)

Called when the default attribute access fails with an `AttributeError` (either `__getattribute__()` raises an `AttributeError` because *name* is not an instance attribute or an attribute in the class tree for `self`; or `__get__()` of a *name* property raises `AttributeError`). This method should either return the (computed) attribute value or raise an `AttributeError` exception. The `object` class itself does not provide this method.

Note that if the attribute is found through the normal mechanism, `__getattr__()` is not called. (This is an intentional asymmetry between `__getattr__()` and `__setattr__()`.) This is done both for efficiency reasons and because otherwise `__getattr__()` would have no way to access other attributes of the instance. Note that at least for instance variables, you can take total control by not inserting any values in the instance attribute dictionary (but instead inserting them in another object). See the `__getattribute__()` method below for a way to actually get total control over attribute access.

`object.__getattribute__(self, name)`

Called unconditionally to implement attribute accesses for instances of the class. If the class also defines `__getattr__()`, the latter will not be called unless `__getattribute__()` either calls it explicitly or raises an `AttributeError`. This method should return the (computed) attribute value or raise an `AttributeError` exception. In order to avoid infinite recursion in this method, its implementation should always call the base class method with the same name to access any attributes it needs, for example, `object.__getattribute__(self, name)`.

> **Note:** This method may still be bypassed when looking up special methods as the result of implicit invocation via language syntax or built-in functions. See Special method lookup.

For certain sensitive attribute accesses, raises an auditing event `object.__getattr__` with arguments `obj` and `name`.

`object.__setattr__(self, name, value)`

Called when an attribute assignment is attempted. This is called instead of the normal mechanism (i.e. store the value in the instance dictionary). *name* is the attribute name, *value* is the value to be assigned to it.

If `__setattr__()` wants to assign to an instance attribute, it should call the base class method with the same name, for example, `object.__setattr__(self, name, value)`.

For certain sensitive attribute assignments, raises an auditing event `object.__setattr__` with arguments `obj`, `name`, `value`.

`object.__delattr__(self, name)`

Like `__setattr__()` but for attribute deletion instead of assignment. This should only be implemented if `del obj.name` is meaningful for the object.

For certain sensitive attribute deletions, raises an auditing event `object.__delattr__` with arguments `obj` and `name`.

`object.__dir__(self)`

Called when `dir()` is called on the object. An iterable must be returned. `dir()` converts the returned iterable to a list and sorts it.

3.3.2.1. Customizing module attribute access

Special names `__getattr__` and `__dir__` can be also used to customize access to module attributes. The `__getattr__` function at the module level should accept one argument which is the name of an attribute and return the computed value or raise an `AttributeError`. If an attribute is not found on a module object through the normal lookup, i.e. `object.__getattribute__()`, then `__getattr__` is searched in the module `__dict__` before raising an `AttributeError`. If found, it is called with the attribute name and the result is returned.

The `__dir__` function should accept no arguments, and return an iterable of strings that represents the names accessible on module. If present, this function overrides the standard `dir()` search on a module.

For a more fine grained customization of the module behavior (setting attributes, properties, etc.), one can set the `__class__` attribute of a module object to a subclass of `types.ModuleType`. For example:

```
import sys
from types import ModuleType

class VerboseModule(ModuleType):
    def __repr__(self):
        return f'Verbose {self.__name__}'

    def __setattr__(self, attr, value):
        print(f'Setting {attr}...')
        super().__setattr__(attr, value)

sys.modules[__name__].__class__ = VerboseModule
```

Note: Defining module `__getattr__` and setting module `__class__` only affect lookups made using the attribute access syntax – directly accessing the module globals (whether by code within the module, or via a reference to the module's globals dictionary) is unaffected.

Changed in version 3.5: `__class__` module attribute is now writable.

Added in version 3.7: `__getattr__` and `__dir__` module attributes.

See also:

PEP 562 - Module __getattr__ and __dir__
Describes the `__getattr__` and `__dir__` functions on modules.

3.3.2.2. Implementing Descriptors

The following methods only apply when an instance of the class containing the method (a so-called *descriptor* class) appears in an *owner* class (the descriptor must be in either the owner's class dictionary or in the class dictionary for one of its parents). In the examples below, "the attribute" refers to the attribute whose name is the key of the property in the owner class' `__dict__`. The `object` class itself does not implement any of these protocols.

object.**__get__**(*self*, *instance*, *owner=None*)

Called to get the attribute of the owner class (class attribute access) or of an instance of that class (instance attribute access). The optional *owner* argument is the owner class, while *instance* is the instance that the attribute was accessed through, or `None` when the attribute is accessed through the *owner*.

This method should return the computed attribute value or raise an `AttributeError` exception.

PEP 252 specifies that `__get__()` is callable with one or two arguments. Python's own built-in descriptors support this specification; however, it is likely that some third-party tools have descriptors that require both arguments. Python's own `__getattribute__()` implementation always passes in both arguments whether they are required or not.

object.**__set__**(*self*, *instance*, *value*)

Called to set the attribute on an instance *instance* of the owner class to a new value, *value*.

Note, adding `__set__()` or `__delete__()` changes the kind of descriptor to a "data descriptor". See Invoking Descriptors for more details.

object.**__delete__**(*self*, *instance*)

Called to delete the attribute on an instance *instance* of the owner class.

Instances of descriptors may also have the `__objclass__` attribute present:

object.**__objclass__**

The attribute `__objclass__` is interpreted by the `inspect` module as specifying the class where this object was defined (setting this appropriately can assist in runtime introspection of dynamic class attributes). For callables, it may indicate that an instance of the given type (or a subclass) is expected or required as the first positional argument (for example, CPython sets this attribute for unbound methods that are implemented in C).

3.3.2.3. Invoking Descriptors

In general, a descriptor is an object attribute with "binding behavior", one whose attribute access has been overridden by methods in the descriptor protocol: `__get__()`, `__set__()`, and `__delete__()`. If any of those methods are defined for an object, it is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete the attribute from an object's dictionary. For instance, `a.x` has a lookup chain starting with `a.__dict__['x']`, then `type(a).__dict__['x']`, and continuing through the base classes of `type(a)` excluding metaclasses.

However, if the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined and how they were called.

The starting point for descriptor invocation is a binding, `a.x`. How the arguments are assembled depends on `a`:

Direct Call

The simplest and least common call is when user code directly invokes a descriptor method: `x.__get__(a)`.

Instance Binding

If binding to an object instance, `a.x` is transformed into the call: `type(a).__dict__['x'].__get__(a, type(a))`.

Class Binding

If binding to a class, `A.x` is transformed into the call: `A.__dict__['x'].__get__(None, A)`.

Super Binding

A dotted lookup such as `super(A, a).x` searches `a.__class__.__mro__` for a base class `B` following `A` and then returns `B.__dict__['x'].__get__(a, A)`. If not a descriptor, `x` is returned unchanged.

For instance bindings, the precedence of descriptor invocation depends on which descriptor methods are defined. A descriptor can define any combination of `__get__()`, `__set__()` and `__delete__()`. If it does not define `__get__()`, then accessing the attribute will return the descriptor object itself unless there is a value in the object's instance dictionary. If the descriptor defines `__set__()` and/or `__delete__()`, it is a data descriptor; if it defines neither, it is a non-data descriptor. Normally, data descriptors define both `__get__()` and `__set__()`, while non-data descriptors have just the `__get__()` method. Data descriptors with `__get__()` and `__set__()` (and/or `__delete__()`) defined always override a redefinition in an instance dictionary. In contrast, non-data descriptors can be overridden by instances.

Python methods (including those decorated with `@staticmethod` and `@classmethod`) are implemented as non-data descriptors. Accordingly, instances can redefine and override methods. This allows individual instances to acquire behaviors that differ from other instances of the same class.

The `property()` function is implemented as a data descriptor. Accordingly, instances cannot override the behavior of a property.

3.3.2.4. __slots__

__slots__ allow us to explicitly declare data members (like properties) and deny the creation of `__dict__` and *__weakref__* (unless explicitly declared in *__slots__* or available in a parent.)

The space saved over using `__dict__` can be significant. Attribute lookup speed can be significantly improved as well.

`object.`**`__slots__`**

This class variable can be assigned a string, iterable, or sequence of strings with variable names used by instances. *__slots__* reserves space for the declared variables and prevents the automatic creation of `__dict__` and *__weakref__* for each instance.

Notes on using *__slots__*:

- When inheriting from a class without *__slots__*, the `__dict__` and *__weakref__* attribute of the instances will always be accessible.
- Without a `__dict__` variable, instances cannot be assigned new variables not listed in the *__slots__* definition. Attempts to assign to an unlisted variable name raises `AttributeError`. If dynamic assignment of new variables is desired, then add `'__dict__'` to the sequence of strings in the *__slots__* declaration.
- Without a *__weakref__* variable for each instance, classes defining *__slots__* do not support `weak references` to its instances. If weak reference support is needed, then add `'__weakref__'` to the sequence of strings in the *__slots__* declaration.
- *__slots__* are implemented at the class level by creating descriptors for each variable name. As a result, class attributes cannot be used to set default values for instance variables defined by *__slots__*; otherwise,

the class attribute would overwrite the descriptor assignment.

- The action of a *__slots__* declaration is not limited to the class where it is defined. *__slots__* declared in parents are available in child classes. However, instances of a child subclass will get a `__dict__` and *__weakref__* unless the subclass also defines *__slots__* (which should only contain names of any *additional* slots).
- If a class defines a slot also defined in a base class, the instance variable defined by the base class slot is inaccessible (except by retrieving its descriptor directly from the base class). This renders the meaning of the program undefined. In the future, a check may be added to prevent this.
- `TypeError` will be raised if nonempty *__slots__* are defined for a class derived from a `"variable-length" built-in type` such as `int`, `bytes`, and `tuple`.
- Any non-string iterable may be assigned to *__slots__*.
- If a `dictionary` is used to assign *__slots__*, the dictionary keys will be used as the slot names. The values of the dictionary can be used to provide per-attribute docstrings that will be recognised by `inspect.getdoc()` and displayed in the output of `help()`.
- `__class__` assignment works only if both classes have the same *__slots__*.
- Multiple inheritance with multiple slotted parent classes can be used, but only one parent is allowed to have attributes created by slots (the other bases must have empty slot layouts) - violations raise `TypeError`.
- If an iterator is used for *__slots__* then a descriptor is created for each of the iterator's values. However, the *__slots__* attribute will be an empty iterator.

3.3.3. Customizing class creation

Whenever a class inherits from another class, `__init_subclass__()` is called on the parent class. This way, it is possible to write classes which change the behavior of subclasses. This is closely related to class decorators, but where class decorators only affect the specific class they're applied to, `__init_subclass__` solely applies to future subclasses of the class defining the method.

classmethod object.**__init_subclass__**(*cls*)

This method is called whenever the containing class is subclassed. *cls* is then the new subclass. If defined as a normal instance method, this method is implicitly converted to a class method.

Keyword arguments which are given to a new class are passed to the parent class's `__init_subclass__`. For compatibility with other classes using `__init_subclass__`, one should take out the needed keyword arguments and pass the others over to the base class, as in:

```
class Philosopher:
    def __init_subclass__(cls, /, default_name, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.default_name = default_name

class AustralianPhilosopher(Philosopher, default_name="Bruce"):
    pass
```

The default implementation `object.__init_subclass__` does nothing, but raises an error if it is called with any arguments.

Note: The metaclass hint `metaclass` is consumed by the rest of the type machinery, and is never passed to `__init_subclass__` implementations. The actual metaclass (rather than the explicit hint) can be accessed as `type(cls)`.

Added in version 3.6.

When a class is created, `type.__new__()` scans the class variables and makes callbacks to those with a `__set_name__()` hook.

`object.__set_name__(self, owner, name)`

Automatically called at the time the owning class *owner* is created. The object has been assigned to *name* in that class:

```
class A:
    x = C()  # Automatically calls: x.__set_name__(A, 'x')
```

If the class variable is assigned after the class is created, `__set_name__()` will not be called automatically. If needed, `__set_name__()` can be called directly:

```
class A:
   pass

c = C()
A.x = c                  # The hook is not called
c.__set_name__(A, 'x')   # Manually invoke the hook
```

See Creating the class object for more details.

Added in version 3.6.

3.3.3.1. Metaclasses

By default, classes are constructed using `type()`. The class body is executed in a new namespace and the class name is bound locally to the result of `type(name, bases, namespace)`.

The class creation process can be customized by passing the `metaclass` keyword argument in the class definition line, or by inheriting from an existing class that included such an argument. In the following example, both `MyClass` and `MySubclass` are instances of `Meta`:

```
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

class MySubclass(MyClass):
    pass
```

Any other keyword arguments that are specified in the class definition are passed through to all metaclass operations described below.

When a class definition is executed, the following steps occur:

- MRO entries are resolved;
- the appropriate metaclass is determined;
- the class namespace is prepared;
- the class body is executed;
- the class object is created.

3.3.3.2. Resolving MRO entries

`object.__mro_entries__(self, bases)`

If a base that appears in a class definition is not an instance of `type`, then an `__mro_entries__()` method is searched on the base. If an `__mro_entries__()` method is found, the base is substituted with the result of a call to `__mro_entries__()` when creating the class. The method is called with the original bases tuple passed to the *bases* parameter, and must return a tuple of classes that will be used instead of the base. The returned tuple may be empty: in these cases, the original base is ignored.

See also:

`types.resolve_bases()`
: Dynamically resolve bases that are not instances of `type`.

`types.get_original_bases()`
: Retrieve a class's "original bases" prior to modifications by `__mro_entries__()`.

PEP 560
: Core support for typing module and generic types.

3.3.3.3. Determining the appropriate metaclass

The appropriate metaclass for a class definition is determined as follows:

- if no bases and no explicit metaclass are given, then `type()` is used;
- if an explicit metaclass is given and it is *not* an instance of `type()`, then it is used directly as the metaclass;
- if an instance of `type()` is given as the explicit metaclass, or bases are defined, then the most derived metaclass is used.

The most derived metaclass is selected from the explicitly specified metaclass (if any) and the metaclasses (i.e. `type(cls)`) of all specified base classes. The most derived metaclass is one which is a subtype of *all* of these candidate metaclasses. If none of the candidate metaclasses meets that criterion, then the class definition will fail with `TypeError`.

3.3.3.4. Preparing the class namespace

Once the appropriate metaclass has been identified, then the class namespace is prepared. If the metaclass has a `__prepare__` attribute, it is called as `namespace = metaclass.__prepare__(name, bases, **kwds)` (where the additional keyword arguments, if any, come from the class definition). The `__prepare__` method should be implemented as a `classmethod`. The namespace returned by `__prepare__` is passed in to `__new__`, but when the final class object is created the namespace is copied into a new `dict`.

If the metaclass has no `__prepare__` attribute, then the class namespace is initialised as an empty ordered mapping.

See also:

PEP 3115 - Metaclasses in Python 3000
Introduced the `__prepare__` namespace hook

3.3.3.5. Executing the class body

The class body is executed (approximately) as `exec(body, globals(), namespace)`. The key difference from a normal call to `exec()` is that lexical scoping allows the class body (including any methods) to reference names from the current and outer scopes when the class definition occurs inside a function.

However, even when the class definition occurs inside the function, methods defined inside the class still cannot see names defined at the class scope. Class variables must be accessed through the first parameter of instance or class methods, or through the implicit lexically scoped `__class__` reference described in the next section.

3.3.3.6. Creating the class object

Once the class namespace has been populated by executing the class body, the class object is created by calling `metaclass(name, bases, namespace, **kwds)` (the additional keywords passed here are the same as those passed to `__prepare__`).

This class object is the one that will be referenced by the zero-argument form of `super()`. `__class__` is an implicit closure reference created by the compiler if any methods in a class body refer to either `__class__` or `super`. This allows the zero argument form of `super()` to correctly identify the class being defined based on lexical scoping, while the class or instance that was used to make the current call is identified based on the first argument passed to the method.

CPython implementation detail: In CPython 3.6 and later, the `__class__` cell is passed to the metaclass as a `__classcell__` entry in the class namespace. If present, this must be propagated up to the `type.__new__` call in order for the class to be initialised correctly. Failing to do so will result in a `RuntimeError` in Python 3.8.

When using the default metaclass `type`, or any metaclass that ultimately calls `type.__new__`, the following additional customization steps are invoked after creating the class object:

1. The `type.__new__` method collects all of the attributes in the class namespace that define a `__set_name__()` method;
2. Those `__set_name__` methods are called with the class being defined and the assigned name of that particular attribute;

3. The `__init_subclass__()` hook is called on the immediate parent of the new class in its method resolution order.

After the class object is created, it is passed to the class decorators included in the class definition (if any) and the resulting object is bound in the local namespace as the defined class.

When a new class is created by `type.__new__`, the object provided as the namespace parameter is copied to a new ordered mapping and the original object is discarded. The new copy is wrapped in a read-only proxy, which becomes the `__dict__` attribute of the class object.

See also:

PEP 3135 - New super
Describes the implicit `__class__` closure reference

3.3.3.7. Uses for metaclasses

The potential uses for metaclasses are boundless. Some ideas that have been explored include enum, logging, interface checking, automatic delegation, automatic property creation, proxies, frameworks, and automatic resource locking/synchronization.

3.3.4. Customizing instance and subclass checks

The following methods are used to override the default behavior of the `isinstance()` and `issubclass()` built-in functions.

In particular, the metaclass `abc.ABCMeta` implements these methods in order to allow the addition of Abstract Base Classes (ABCs) as "virtual base classes" to any class or type (including built-in types), including other ABCs.

type.**__instancecheck__**(*self*, *instance*)
Return true if *instance* should be considered a (direct or indirect) instance of *class*. If defined, called to implement `isinstance(instance, class)`.

type.**__subclasscheck__**(*self*, *subclass*)
Return true if *subclass* should be considered a (direct or indirect) subclass of *class*. If defined, called to implement `issubclass(subclass, class)`.

Note that these methods are looked up on the type (metaclass) of a class. They cannot be defined as class methods in the actual class. This is consistent with the lookup of special methods that are called on instances, only in this case the instance is itself a class.

See also:

PEP 3119 - Introducing Abstract Base Classes
Includes the specification for customizing `isinstance()` and `issubclass()` behavior through `__instancecheck__()` and `__subclasscheck__()`, with motivation for this functionality in the context of adding Abstract Base Classes (see the `abc` module) to the language.

3.3.5. Emulating generic types

When using type annotations, it is often useful to *parameterize* a generic type using Python's square-brackets notation. For example, the annotation `list[int]` might be used to signify a `list` in which all the elements are of type `int`.

> **See also:**
>
> **PEP 484 - Type Hints**
> Introducing Python's framework for type annotations
>
> **Generic Alias Types**
> Documentation for objects representing parameterized generic classes
>
> **Generics, user-defined generics and `typing.Generic`**
> Documentation on how to implement generic classes that can be parameterized at runtime and understood by static type-checkers.

A class can *generally* only be parameterized if it defines the special class method `__class_getitem__()`.

classmethod object.**__class_getitem__**(*cls*, *key*)

Return an object representing the specialization of a generic class by type arguments found in *key*.

When defined on a class, `__class_getitem__()` is automatically a class method. As such, there is no need for it to be decorated with `@classmethod` when it is defined.

3.3.5.1. The purpose of *__class_getitem__*

The purpose of `__class_getitem__()` is to allow runtime parameterization of standard-library generic classes in order to more easily apply type hints to these classes.

To implement custom generic classes that can be parameterized at runtime and understood by static type-checkers, users should either inherit from a standard library class that already implements `__class_getitem__()`, or inherit from `typing.Generic`, which has its own implementation of `__class_getitem__()`.

Custom implementations of `__class_getitem__()` on classes defined outside of the standard library may not be understood by third-party type-checkers such as mypy. Using `__class_getitem__()` on any class for purposes other than type hinting is discouraged.

3.3.5.2. *__class_getitem__* versus *__getitem__*

Usually, the subscription of an object using square brackets will call the `__getitem__()` instance method defined on the object's class. However, if the object being subscribed is itself a class, the class method `__class_getitem__()` may be called instead. `__class_getitem__()` should return a GenericAlias object if it is properly defined.

Presented with the expression `obj[x]`, the Python interpreter follows something like the following process to decide whether `__getitem__()` or `__class_getitem__()` should be called:

```
from inspect import isclass

def subscribe(obj, x):
    """Return the result of the expression 'obj[x]'"""

    class_of_obj = type(obj)

    # If the class of obj defines __getitem__,
    # call class_of_obj.__getitem__(obj, x)
    if hasattr(class_of_obj, '__getitem__'):
        return class_of_obj.__getitem__(obj, x)

    # Else, if obj is a class and defines __class_getitem__,
    # call obj.__class_getitem__(x)
    elif isclass(obj) and hasattr(obj, '__class_getitem__'):
        return obj.__class_getitem__(x)

    # Else, raise an exception
    else:
        raise TypeError(
            f"'{class_of_obj.__name__}' object is not subscriptable"
        )
```

In Python, all classes are themselves instances of other classes. The class of a class is known as that class's metaclass, and most classes have the `type` class as their metaclass. `type` does not define `__getitem__()`, meaning that expressions such as `list[int]`, `dict[str, float]` and `tuple[str, bytes]` all result in `__class_getitem__()` being called:

```
>>> # list has class "type" as its metaclass, like most classes:
>>> type(list)
<class 'type'>
>>> type(dict) == type(list) == type(tuple) == type(str) == type(bytes)
True
>>> # "list[int]" calls "list.__class_getitem__(int)"
>>> list[int]
list[int]
>>> # list.__class_getitem__ returns a GenericAlias object:
>>> type(list[int])
<class 'types.GenericAlias'>
```

However, if a class has a custom metaclass that defines `__getitem__()`, subscribing the class may result in different behaviour. An example of this can be found in the `enum` module:

```
>>> from enum import Enum
>>> class Menu(Enum):
...     """A breakfast menu"""
...     SPAM = 'spam'
...     BACON = 'bacon'
...
>>> # Enum classes have a custom metaclass:
>>> type(Menu)
<class 'enum.EnumMeta'>
>>> # EnumMeta defines __getitem__,
```

```
>>> # so __class_getitem__ is not called,
>>> # and the result is not a GenericAlias object:
>>> Menu['SPAM']
<Menu.SPAM: 'spam'>
>>> type(Menu['SPAM'])
<enum 'Menu'>
```

See also:

PEP 560 - Core Support for typing module and generic types
Introducing `__class_getitem__()`, and outlining when a subscription results in `__class_getitem__()` being called instead of `__getitem__()`

3.3.6. Emulating callable objects

`object.__call__(self[, args...])`

Called when the instance is "called" as a function; if this method is defined, `x(arg1, arg2, ...)` roughly translates to `type(x).__call__(x, arg1, ...)`. The `object` class itself does not provide this method.

3.3.7. Emulating container types

The following methods can be defined to implement container objects. None of them are provided by the `object` class itself. Containers usually are sequences (such as `lists` or `tuples`) or mappings (like dictionaries), but can represent other containers as well. The first set of methods is used either to emulate a sequence or to emulate a mapping; the difference is that for a sequence, the allowable keys should be the integers *k* for which `0 <= k < N` where *N* is the length of the sequence, or `slice` objects, which define a range of items. It is also recommended that mappings provide the methods `keys()`, `values()`, `items()`, `get()`, `clear()`, `setdefault()`, `pop()`, `popitem()`, `copy()`, and `update()` behaving similar to those for Python's standard `dictionary` objects. The `collections.abc` module provides a `MutableMapping` abstract base class to help create those methods from a base set of `__getitem__()`, `__setitem__()`, `__delitem__()`, and `keys()`. Mutable sequences should provide methods `append()`, `count()`, `index()`, `extend()`, `insert()`, `pop()`, `remove()`, `reverse()` and `sort()`, like Python standard `list` objects. Finally, sequence types should implement addition (meaning concatenation) and multiplication (meaning repetition) by defining the methods `__add__()`, `__radd__()`, `__iadd__()`, `__mul__()`, `__rmul__()` and `__imul__()` described below; they should not define other numerical operators. It is recommended that both mappings and sequences implement the `__contains__()` method to allow efficient use of the `in` operator; for mappings, `in` should search the mapping's keys; for sequences, it should search through the values. It is further recommended that both mappings and sequences implement the `__iter__()` method to allow efficient iteration through the container; for mappings, `__iter__()` should iterate through the object's keys; for sequences, it should iterate through the values.

`object.__len__(self)`

Called to implement the built-in function `len()`. Should return the length of the object, an integer >= 0. Also, an object that doesn't define a `__bool__()` method and whose `__len__()` method returns zero is considered to be false in a Boolean context.

CPython implementation detail: In CPython, the length is required to be at most `sys.maxsize`. If the length is larger than `sys.maxsize` some features (such as `len()`) may raise `OverflowError`. To prevent raising `OverflowError` by truth value testing, an object must define a `__bool__()` method.

object.**__length_hint__**(*self*)

Called to implement `operator.length_hint()`. Should return an estimated length for the object (which may be greater or less than the actual length). The length must be an integer `>=` 0. The return value may also be `NotImplemented`, which is treated the same as if the `__length_hint__` method didn't exist at all. This method is purely an optimization and is never required for correctness.

> *Added in version 3.4.*

Note: Slicing is done exclusively with the following three methods. A call like

```
a[1:2] = b
```

is translated to

```
a[slice(1, 2, None)] = b
```

and so forth. Missing slice items are always filled in with `None`.

object.**__getitem__**(*self*, *key*)

Called to implement evaluation of `self[key]`. For sequence types, the accepted keys should be integers. Optionally, they may support `slice` objects as well. Negative index support is also optional. If *key* is of an inappropriate type, `TypeError` may be raised; if *key* is a value outside the set of indexes for the sequence (after any special interpretation of negative values), `IndexError` should be raised. For mapping types, if *key* is missing (not in the container), `KeyError` should be raised.

Note: `for` loops expect that an `IndexError` will be raised for illegal indexes to allow proper detection of the end of the sequence.

Note: When subscripting a *class*, the special class method `__class_getitem__()` may be called instead of `__getitem__()`. See __class_getitem__ versus __getitem__ for more details.

object.**__setitem__**(*self*, *key*, *value*)

Called to implement assignment to `self[key]`. Same note as for `__getitem__()`. This should only be implemented for mappings if the objects support changes to the values for keys, or if new keys can be added, or for sequences if elements can be replaced. The same exceptions should be raised for improper *key* values as for the `__getitem__()` method.

object.**__delitem__**(*self*, *key*)

Called to implement deletion of `self[key]`. Same note as for `__getitem__()`. This should only be implemented for mappings if the objects support removal of keys, or for sequences if elements can be removed from the sequence. The same exceptions should be raised for improper *key* values as for the `__getitem__()` method.

object.**__missing__**(*self*, *key*)

Called by `dict.__getitem__()` to implement `self[key]` for dict subclasses when key is not in the dictionary.

object.**__iter__**(*self*)

This method is called when an iterator is required for a container. This method should return a new iterator object that can iterate over all the objects in the container. For mappings, it should iterate over the keys of the container.

object.**__reversed__**(*self*)

Called (if present) by the `reversed()` built-in to implement reverse iteration. It should return a new iterator object that iterates over all the objects in the container in reverse order.

If the `__reversed__()` method is not provided, the `reversed()` built-in will fall back to using the sequence protocol (`__len__()` and `__getitem__()`). Objects that support the sequence protocol should only provide `__reversed__()` if they can provide an implementation that is more efficient than the one provided by `reversed()`.

The membership test operators (`in` and `not in`) are normally implemented as an iteration through a container. However, container objects can supply the following special method with a more efficient implementation, which also does not require the object be iterable.

object.**__contains__**(*self*, *item*)

Called to implement membership test operators. Should return true if *item* is in *self*, false otherwise. For mapping objects, this should consider the keys of the mapping rather than the values or the key-item pairs.

For objects that don't define `__contains__()`, the membership test first tries iteration via `__iter__()`, then the old sequence iteration protocol via `__getitem__()`, see this section in the language reference.

3.3.8. Emulating numeric types

The following methods can be defined to emulate numeric objects. Methods corresponding to operations that are not supported by the particular kind of number implemented (e.g., bitwise operations for non-integral numbers) should be left undefined.

object.**__add__**(*self*, *other*)
object.**__sub__**(*self*, *other*)
object.**__mul__**(*self*, *other*)
object.**__matmul__**(*self*, *other*)
object.**__truediv__**(*self*, *other*)
object.**__floordiv__**(*self*, *other*)
object.**__mod__**(*self*, *other*)
object.**__divmod__**(*self*, *other*)
object.**__pow__**(*self*, *other*[, *modulo*])

`object.__lshift__(self, other)`
`object.__rshift__(self, other)`
`object.__and__(self, other)`
`object.__xor__(self, other)`
`object.__or__(self, other)`

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `@`, `/`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`). For instance, to evaluate the expression `x + y`, where *x* is an instance of a class that has an `__add__()` method, `type(x).__add__(x, y)` is called. The `__divmod__()` method should be the equivalent to using `__floordiv__()` and `__mod__()`; it should not be related to `__truediv__()`. Note that `__pow__()` should be defined to accept an optional third argument if the ternary version of the built-in `pow()` function is to be supported.

If one of those methods does not support the operation with the supplied arguments, it should return `NotImplemented`.

`object.__radd__(self, other)`
`object.__rsub__(self, other)`
`object.__rmul__(self, other)`
`object.__rmatmul__(self, other)`
`object.__rtruediv__(self, other)`
`object.__rfloordiv__(self, other)`
`object.__rmod__(self, other)`
`object.__rdivmod__(self, other)`
`object.__rpow__(self, other[, modulo])`
`object.__rlshift__(self, other)`
`object.__rrshift__(self, other)`
`object.__rand__(self, other)`
`object.__rxor__(self, other)`
`object.__ror__(self, other)`

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `@`, `/`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`) with reflected (swapped) operands. These functions are only called if the left operand does not support the corresponding operation [3] and the operands are of different types. [4] For instance, to evaluate the expression `x - y`, where *y* is an instance of a class that has an `__rsub__()` method, `type(y).__rsub__(y, x)` is called if `type(x).__sub__(x, y)` returns `NotImplemented`.

Note that ternary `pow()` will not try calling `__rpow__()` (the coercion rules would become too complicated).

> **Note:** If the right operand's type is a subclass of the left operand's type and that subclass provides a different implementation of the reflected method for the operation, this method will be called before the left operand's non-reflected method. This behavior allows subclasses to override their ancestors' operations.

`object.__iadd__(self, other)`
`object.__isub__(self, other)`

object.**__imul__**(*self*, *other*)
object.**__imatmul__**(*self*, *other*)
object.**__itruediv__**(*self*, *other*)
object.**__ifloordiv__**(*self*, *other*)
object.**__imod__**(*self*, *other*)
object.**__ipow__**(*self*, *other*[, *modulo*])
object.**__ilshift__**(*self*, *other*)
object.**__irshift__**(*self*, *other*)
object.**__iand__**(*self*, *other*)
object.**__ixor__**(*self*, *other*)
object.**__ior__**(*self*, *other*)

These methods are called to implement the augmented arithmetic assignments (`+=`, `-=`, `*=`, `@=`, `/=`, `//=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=`, `|=`). These methods should attempt to do the operation in-place (modifying *self*) and return the result (which could be, but does not have to be, *self*). If a specific method is not defined, or if that method returns `NotImplemented`, the augmented assignment falls back to the normal methods. For instance, if *x* is an instance of a class with an `__iadd__()` method, `x += y` is equivalent to `x = x.__iadd__(y)` . If `__iadd__()` does not exist, or if `x.__iadd__(y)` returns `NotImplemented`, `x.__add__(y)` and `y.__radd__(x)` are considered, as with the evaluation of `x + y`. In certain situations, augmented assignment can result in unexpected errors (see Why does a_tuple[i] += ['item'] raise an exception when the addition works?), but this behavior is in fact part of the data model.

object.**__neg__**(*self*)
object.**__pos__**(*self*)
object.**__abs__**(*self*)
object.**__invert__**(*self*)

Called to implement the unary arithmetic operations (`-`, `+`, `abs()` and `~`).

object.**__complex__**(*self*)
object.**__int__**(*self*)
object.**__float__**(*self*)

Called to implement the built-in functions `complex()`, `int()` and `float()`. Should return a value of the appropriate type.

object.**__index__**(*self*)

Called to implement `operator.index()`, and whenever Python needs to losslessly convert the numeric object to an integer object (such as in slicing, or in the built-in `bin()`, `hex()` and `oct()` functions). Presence of this method indicates that the numeric object is an integer type. Must return an integer.

If `__int__()`, `__float__()` and `__complex__()` are not defined then corresponding built-in functions `int()`, `float()` and `complex()` fall back to `__index__()`.

object.**__round__**(*self*[, *ndigits*])
object.**__trunc__**(*self*)
object.**__floor__**(*self*)
object.**__ceil__**(*self*)

Called to implement the built-in function `round()` and `math` functions `trunc()`, `floor()` and `ceil()`. Unless *ndigits* is passed to `__round__()` all these methods should return the value of the object truncated to an `Integral` (typically an `int`).

The built-in function `int()` falls back to `__trunc__()` if neither `__int__()` nor `__index__()` is defined.

> *Changed in version 3.11:* The delegation of `int()` to `__trunc__()` is deprecated.

3.3.9. With Statement Context Managers

A *context manager* is an object that defines the runtime context to be established when executing a `with` statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. Context managers are normally invoked using the `with` statement (described in section The with statement), but can also be used by directly invoking their methods.

Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files, etc.

For more information on context managers, see Context Manager Types. The `object` class itself does not provide the context manager methods.

`object.__enter__(self)`

Enter the runtime context related to this object. The `with` statement will bind this method's return value to the target(s) specified in the `as` clause of the statement, if any.

`object.__exit__(self, exc_type, exc_value, traceback)`

Exit the runtime context related to this object. The parameters describe the exception that caused the context to be exited. If the context was exited without an exception, all three arguments will be `None`.

If an exception is supplied, and the method wishes to suppress the exception (i.e., prevent it from being propagated), it should return a true value. Otherwise, the exception will be processed normally upon exit from this method.

Note that `__exit__()` methods should not reraise the passed-in exception; this is the caller's responsibility.

See also:

PEP 343 - The "with" statement
The specification, background, and examples for the Python `with` statement.

3.3.10. Customizing positional arguments in class pattern matching

When using a class name in a pattern, positional arguments in the pattern are not allowed by default, i.e. `case MyClass(x, y)` is typically invalid without special support in `MyClass`. To be able to use that kind of pattern, the class needs to define a *__match_args__* attribute.

`object.__match_args__`

This class variable can be assigned a tuple of strings. When this class is used in a class pattern with positional arguments, each positional argument will be converted into a keyword argument, using the corresponding value in *__match_args__* as the keyword. The absence of this attribute is equivalent to setting it to `()`.

For example, if `MyClass.__match_args__` is `("left", "center", "right")` that means that `case MyClass(x, y)` is equivalent to `case MyClass(left=x, center=y)`. Note that the number of arguments in the pattern must be smaller than or equal to the number of elements in *__match_args__*; if it is larger, the pattern match attempt will raise a `TypeError`.

> *Added in version 3.10.*

See also:

PEP 634 - Structural Pattern Matching
The specification for the Python `match` statement.

3.3.11. Emulating buffer types

The buffer protocol provides a way for Python objects to expose efficient access to a low-level memory array. This protocol is implemented by builtin types such as `bytes` and `memoryview`, and third-party libraries may define additional buffer types.

While buffer types are usually implemented in C, it is also possible to implement the protocol in Python.

`object.__buffer__(self, flags)`
Called when a buffer is requested from *self* (for example, by the `memoryview` constructor). The *flags* argument is an integer representing the kind of buffer requested, affecting for example whether the returned buffer is read-only or writable. `inspect.BufferFlags` provides a convenient way to interpret the flags. The method must return a `memoryview` object.

`object.__release_buffer__(self, buffer)`
Called when a buffer is no longer needed. The *buffer* argument is a `memoryview` object that was previously returned by `__buffer__()`. The method must release any resources associated with the buffer. This method should return `None`. Buffer objects that do not need to perform any cleanup are not required to implement this method.

> *Added in version 3.12.*

See also:

PEP 688 - Making the buffer protocol accessible in Python
Introduces the Python `__buffer__` and `__release_buffer__` methods.

`collections.abc.Buffer`
ABC for buffer types.

3.3.12. Special method lookup

For custom classes, implicit invocations of special methods are only guaranteed to work correctly if defined on an object's type, not in the object's instance dictionary. That behaviour is the reason why the following code raises an exception:

```
>>> class C:
...     pass
...
>>> c = C()
>>> c.__len__ = lambda: 5
>>> len(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'C' has no len()
```

The rationale behind this behaviour lies with a number of special methods such as `__hash__()` and `__repr__()` that are implemented by all objects, including type objects. If the implicit lookup of these methods used the conventional lookup process, they would fail when invoked on the type object itself:

```
>>> 1 .__hash__() == hash(1)
True
>>> int.__hash__() == hash(int)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor '__hash__' of 'int' object needs an argument
```

Incorrectly attempting to invoke an unbound method of a class in this way is sometimes referred to as 'metaclass confusion', and is avoided by bypassing the instance when looking up special methods:

```
>>> type(1).__hash__(1) == hash(1)
True
>>> type(int).__hash__(int) == hash(int)
True
```

In addition to bypassing any instance attributes in the interest of correctness, implicit special method lookup generally also bypasses the `__getattribute__()` method even of the object's metaclass:

```
>>> class Meta(type):
...     def __getattribute__(*args):
...         print("Metaclass getattribute invoked")
...         return type.__getattribute__(*args)
...
>>> class C(object, metaclass=Meta):
...     def __len__(self):
...         return 10
...     def __getattribute__(*args):
...         print("Class getattribute invoked")
...         return object.__getattribute__(*args)
...
>>> c = C()
>>> c.__len__()                 # Explicit lookup via instance
Class getattribute invoked
10
>>> type(c).__len__(c)          # Explicit lookup via type
Metaclass getattribute invoked
10
```

```
>>> len(c)                      # Implicit lookup
10
```

Bypassing the `__getattribute__()` machinery in this fashion provides significant scope for speed optimisations within the interpreter, at the cost of some flexibility in the handling of special methods (the special method *must* be set on the class object itself in order to be consistently invoked by the interpreter).

3.4. Coroutines

3.4.1. Awaitable Objects

An awaitable object generally implements an `__await__()` method. Coroutine objects returned from `async def` functions are awaitable.

> **Note:** The generator iterator objects returned from generators decorated with `types.coroutine()` are also awaitable, but they do not implement `__await__()`.

object.**__await__**(*self*)

Must return an iterator. Should be used to implement awaitable objects. For instance, `asyncio.Future` implements this method to be compatible with the `await` expression. The `object` class itself is not awaitable and does not provide this method.

> **Note:** The language doesn't place any restriction on the type or value of the objects yielded by the iterator returned by `__await__`, as this is specific to the implementation of the asynchronous execution framework (e.g. `asyncio`) that will be managing the awaitable object.

Added in version 3.5.

> **See also:** **PEP 492** for additional information about awaitable objects.

3.4.2. Coroutine Objects

Coroutine objects are awaitable objects. A coroutine's execution can be controlled by calling `__await__()` and iterating over the result. When the coroutine has finished executing and returns, the iterator raises `StopIteration`, and the exception's `value` attribute holds the return value. If the coroutine raises an exception, it is propagated by the iterator. Coroutines should not directly raise unhandled `StopIteration` exceptions.

Coroutines also have the methods listed below, which are analogous to those of generators (see Generator-iterator methods). However, unlike generators, coroutines do not directly support iteration.

Changed in version 3.5.2: It is a `RuntimeError` to await on a coroutine more than once.

coroutine.**send**(*value*)

Starts or resumes execution of the coroutine. If *value* is `None`, this is equivalent to advancing the iterator returned by `__await__()`. If *value* is not `None`, this method delegates to the `send()` method of the iterator that caused the coroutine to suspend. The result (return value, `StopIteration`, or other exception) is the same as when iterating over the `__await__()` return value, described above.

`coroutine.throw(value)`
`coroutine.throw(type[, value[, traceback]])`

Raises the specified exception in the coroutine. This method delegates to the `throw()` method of the iterator that caused the coroutine to suspend, if it has such a method. Otherwise, the exception is raised at the suspension point. The result (return value, `StopIteration`, or other exception) is the same as when iterating over the `__await__()` return value, described above. If the exception is not caught in the coroutine, it propagates back to the caller.

> *Changed in version 3.12:* The second signature (type[, value[, traceback]]) is deprecated and may be removed in a future version of Python.

`coroutine.close()`

Causes the coroutine to clean itself up and exit. If the coroutine is suspended, this method first delegates to the `close()` method of the iterator that caused the coroutine to suspend, if it has such a method. Then it raises `GeneratorExit` at the suspension point, causing the coroutine to immediately clean itself up. Finally, the coroutine is marked as having finished executing, even if it was never started.

Coroutine objects are automatically closed using the above process when they are about to be destroyed.

3.4.3. Asynchronous Iterators

An *asynchronous iterator* can call asynchronous code in its `__anext__` method.

Asynchronous iterators can be used in an `async for` statement.

The `object` class itself does not provide these methods.

`object.__aiter__(self)`

Must return an *asynchronous iterator* object.

`object.__anext__(self)`

Must return an *awaitable* resulting in a next value of the iterator. Should raise a `StopAsyncIteration` error when the iteration is over.

An example of an asynchronous iterable object:

```
class Reader:
    async def readline(self):
        ...

    def __aiter__(self):
        return self

    async def __anext__(self):
        val = await self.readline()
        if val == b'':
```

```
            raise StopAsyncIteration
        return val
```

Added in version 3.5.

Changed in version 3.7: Prior to Python 3.7, `__aiter__()` could return an *awaitable* that would resolve to an asynchronous iterator.

Starting with Python 3.7, `__aiter__()` must return an asynchronous iterator object. Returning anything else will result in a `TypeError` error.

3.4.4. Asynchronous Context Managers

An *asynchronous context manager* is a *context manager* that is able to suspend execution in its `__aenter__` and `__aexit__` methods.

Asynchronous context managers can be used in an `async with` statement.

The `object` class itself does not provide these methods.

`object.`**`__aenter__`**`(self)`

Semantically similar to `__enter__()`, the only difference being that it must return an *awaitable*.

`object.`**`__aexit__`**`(self, exc_type, exc_value, traceback)`

Semantically similar to `__exit__()`, the only difference being that it must return an *awaitable*.

An example of an asynchronous context manager class:

```
class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')

    async def __aexit__(self, exc_type, exc, tb):
        await log('exiting context')
```

Added in version 3.5.

Footnotes

[1] It *is* possible in some cases to change an object's type, under certain controlled conditions. It generally isn't a good idea though, since it can lead to some very strange behaviour if it is handled incorrectly.

[2] The `__hash__()`, `__iter__()`, `__reversed__()`, `__contains__()`, `__class_getitem__()` and `__fspath__()` methods have special handling for this. Others will still raise a `TypeError`, but may do so by relying on the behavior that `None` is not callable.

[3] "Does not support" here means that the class has no such method, or the method returns `NotImplemented`. Do not set the method to `None` if you want to force fallback to the right operand's reflected method—that will instead have the opposite effect of explicitly *blocking* such fallback.

[4] For operands of the same type, it is assumed that if the non-reflected method – such as `__add__()` – fails then the overall operation is not supported, which is why the reflected method is not called.