4. Object-Oriented Python - Python in a Nutshell, 3rd Edition

Special Methods

A class may define or inherit special methods (i.e., methods whose names begin and end with double underscores, AKA "dunder" or "magic" methods). Each special method relates to a specific operation. Python implicitly calls a special method whenever you perform the related operation on an instance object. In most cases, the method's return value is the operation's result, and attempting an operation when its related method is not present raises an exception.

Throughout this section, we point out the cases in which these general rules do not apply. In the following, `x` is the instance of class `C` on which you perform the operation, and `y` is the other operand, if any. The parameter `self` of each method also refers to the instance object `x`. In the following sections, whenever we mention calls to `x.__whatever__(...)`, keep in mind that the exact call happening is rather, pedantically speaking, `x.__class__.__whatever__(x, ...)`.

General-Purpose Special Methods

Some special methods relate to general-purpose operations. A class that defines or inherits these methods allows its instances to control such operations. These operations can be divided into the following categories:

Initialization and finalization

> A class can control its instances' initialization (a very common requirement) via the special methods `__new__` and `__init__`, and/or their finalization (a rare requirement) via `__del__`.

Representation as string

> A class can control how Python renders its instances as strings via special methods `__repr__`, `__str__`, `__format__`, (v3 only) `__bytes__`, and (v2 only) `__unicode__`.

Comparison, hashing, and use in a Boolean context

> A class can control how its instances compare with other objects (methods `__lt__`, `__le__`, `__gt__`, `__ge__`, `__eq__`, `__ne__`), how dictionaries use them as keys and sets use them as members (`__hash__`), and whether they evaluate to true or false in Boolean contexts (`__nonzero__` in v2, `__bool__` in v3).

Attribute reference, binding, and unbinding

> A class can control access to its instances' attributes (reference, binding, unbinding) via special methods `__getattribute__`, `__getattr__`, `__setattr__`, and `__delattr__`.

Callable instances

> An instance is callable, just like a function object, if its class has the special method `__call__`.

Table 4-1 documents the general-purpose special methods.

Table 4-1. General-purpose special methods

__bytes__	`__bytes__(self)` In v3, calling `bytes(x)` calls `x.__bytes__()`, if present. If a class supplies both special methods `__bytes__` and `__str__`, the two should return equivalent strings (of bytes and text type, respectively).
__call__	`__call__(self[,args...])` When you call `x([args...])`, Python translates the operation into a call to `x.__call__([args...])`. The arguments for the call operation correspond to the parameters for the `__call__` method, minus the first. The first parameter, conventionally called `self`, refers to `x`, and Python supplies it implicitly and automatically, just as in any other call to a bound method.
__dir__	`__dir__(self)` When you call `dir(x)`, Python translates the operation into a call to `x.__dir__()`, which must return a sorted list of `x`'s attributes. If `x`'s class does not have a `__dir__`, then `dir(x)` does its own introspection to return a list of `x`'s attributes, striving to produce relevant, rather than complete, information.
__del__	`__del__(self)` Just before `x` disappears because of garbage collection, Python calls `x.__del__()` to let `x` finalize itself. If `__del__` is absent, Python performs no special finalization upon garbage-collecting `x` (this is the usual case: very few classes need to define `__del__`). Python ignores the return value of `__del__` and performs no implicit call to `__del__` methods of class `C`'s superclasses. `C.__del__` must explicitly perform any needed finalization, including if need be, by delegation. For example, when class `C` has a base class `B` to finalize, the code in `C.__del__` must call `super(C, self).__del__()` (or, in v3, just `super().__del__()`). Note that the `__del__` method has no direct connection with the `del` statement, as covered in "del Statements". `__del__` is generally not the best approach when you need timely and guaranteed finalization. For such needs, use the `try`/`finally` statement covered in "try/finally" (or, even better, the `with` statement, covered in "The with Statement"). Instances of classes defining `__del__` cannot participate in cyclic-garbage collection, covered in "Garbage Collection". Therefore, you should be particularly careful to avoid reference loops involving such instances, and define `__del__` only when there is no feasible alternative.
__delattr__	`__delattr__(self, name)` At every request to unbind attribute `x.y` (typically, a `del` statement `del x.y`), Python calls `x.__delattr__('y')`. All the considerations discussed later for `__setattr__` also apply to `__delattr__`. Python ignores the return value of `__delattr__`. If `__delattr__` is absent, Python translates `del x.y` into `del x.__dict__['y']`.

`__eq__`, `__ge__`, `__gt__`, `__le__`, `__lt__`, `__ne__`	`__eq__(self, other)` `__ge__(self, other)` `__gt__(self, other)` `__le__(self, other)` `__lt__(self, other)` `__ne__(self, other)` The comparisons `x==y`, `x>=y`, `x>y`, `x<=y`, `x<y`, and `x!=y`, respectively, call the special methods listed here, which should return `False` or `True`. Each method may return `NotImplemented` to tell Python to handle the comparison in alternative ways (e.g., Python may then try `y>x` in lieu of `x<y`). Best practice is to define only one inequality comparison method (normally `__lt__`) plus `__eq__`, and decorate the class with `functools.total_ordering` (covered in Table 7-4) to avoid boilerplate, and any risk of logical contradictions in your comparisons.
`__getattr__`	`__getattr__(self, name)` When the attribute `x.y` can't be found by the usual steps (i.e., when `AttributeError` would normally be raised), Python calls `x.__getattr__('y')` instead. Python does not call `__getattr__` for attributes found by normal means (i.e., as keys in `x.__dict__`, or via `x.__class__`). If you want Python to call `__getattr__` on every attribute reference, keep the attributes elsewhere (e.g., in another dictionary referenced by an attribute with a private name), or else override `__getattribute__` instead. `__getattr__` should raise `AttributeError` if it cannot find `y`.
`__getattribute__`	`__getattribute__(self, name)` At every request to access attribute `x.y`, Python calls `x.__getattribute__('y')`, which must get and return the attribute value or else raise `AttributeError`. The normal semantics of attribute access (using `x.__dict__`, `C.__slots__`, `C`'s class attributes, `x.__getattr__`) are all due to `object.__getattribute__`. When class `C` overrides `__getattribute__`, it must implement all of the attribute access semantics it wants to offer. Most often, the most convenient way to implement attribute access semantics is by delegating (e.g., calling `object.__getattribute__(self, ...)` as part of the operation of your override of `__getattribute__`). **Overriding `__getattribute__` slows attribute access** When a class overrides `__getattribute__`, attribute accesses on instances of the class become slow, since the overriding code executes on every such attribute access.

__hash__	`__hash__(self)` Calling `hash(x)` calls `x.__hash__()` (and so do other contexts that need to know `x`'s hash value, namely, using `x` as a dictionary key, such as `D[x]` where `D` is a dictionary, or using `x` as a set member). `__hash__` must return an `int` such that `x==y` implies `hash(x)==hash(y)`, and must always return the same value for a given object. When `__hash__` is absent, calling `hash(x)` calls `id(x)` instead, as long as `__eq__` is also absent. Other contexts that need to know `x`'s hash value behave the same way. Any `x` such that `hash(x)` returns a result, rather than raising an exception, is known as a *hashable object*. When `__hash__` is absent, but `__eq__` is present, calling `hash(x)` raises an exception (and so do other contexts that need to know `x`'s hash value). In this case, `x` is not hashable and therefore cannot be a dictionary key or set member. You normally define `__hash__` only for immutable objects that also define `__eq__`. Note that if there exists any `y` such that `x==y`, even if `y` is of a different type, and both `x` and `y` are hashable, you *must* ensure that `hash(x)==hash(y)`.
__init__	`__init__(self[,args...])` When a call `C([args...])` creates instance `x` of class `C`, Python calls `x.__init__([args...])` to let `x` initialize itself. If `__init__` is absent (i.e., it's inherited from `object`), you must call class `C` without arguments, `C()`, and `x` has no instance-specific attributes upon creation. Python performs no implicit call to `__init__` methods of class `C`'s superclasses. `C.__init__` must explicitly perform any needed initialization, including, if need be, by delegation. For example, when class `C` has a base class `B` to initialize without arguments, the code in `C.__init__` must explicitly call `super(C,self).__init__()` (or, in v3, just `super().__init__()`). However, `__init__`'s inheritance works just like for any other method or attribute: that is, if class `C` itself does not override `__init__`, it just inherits it from the first superclass in its `__mro__` to override `__init__`, like for every other attribute. `__init__` must return `None`; otherwise, calling the class raises a `TypeError`.

__new__

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

When you call `C([args...])`, Python gets the new instance `x` that you are creating by invoking `C.__new__(C,[args...])`. Every class has the class method `__new__` (most often simply inheriting it from `object`), which can return any value `x`. In other words, `__new__` is not constrained to return a new instance of `C`, although normally it's expected to do so. If, and only if, the value `x` that `__new__` returns is indeed an instance of `C` or of any subclass of `C` (whether a new or previously existing one), Python continues after calling `__new__` by implicitly calling `__init__` on `x` (with the same `[args...]` that were originally passed to `__new__`).

Initialize immutables in __new__, all others in __init__

Since you could perform most kinds of initialization of new instances in either `__init__` or `__new__`, you may wonder where best to place them. Simple: put the initialization in `__init__` only, unless you have a specific reason to put it in `__new__`. (If a type is immutable, its instances cannot be changed in `__init__` for initialization purposes, so this is a special case in which `__new__` does have to perform all initialization.) This tip makes life simpler, since `__init__` is an instance method, while `__new__` is a specialized class method.

__nonzero__

`__nonzero__(self)`

When evaluating `x` as true or false (see "Boolean Values")—for example, on a call to `bool(x)`—v2 calls `x.__nonzero__()`, which should return `True` or `False`. When `__nonzero__` is not present, Python calls `__len__` instead, and takes `x` as false when `x.__len__()` returns `0` (so, to check if a container is nonempty, avoid coding `if len(container)>0:`; just code `if container:` instead). When neither `__nonzero__` nor `__len__` is present, Python always considers `x` true.

In v3, this special method is spelled, more readably, as `__bool__`.

__repr__

`__repr__(self)`

Calling `repr(x)` (which also happens implicitly in the interactive interpreter when `x` is the result of an expression statement) calls `x.__repr__()` to get and return a complete string representation of `x`. If `__repr__` is absent, Python uses a default string representation. `__repr__` should return a string with unambiguous information on `x`. Ideally, when feasible, the string should be an expression such that `eval(repr(x))==x` (but don't go crazy aiming for that goal).

__setattr__

`__setattr__(self, name, value)`

At every request to bind attribute `x.y` (typically, an assignment statement `x.y=value`, but also, for example, `setattr(x, 'y', value)`), Python calls `x.__setattr__('y', value)`. Python always calls `__setattr__` for *any* attribute binding on `x`—a major difference from `__getattr__` (`__setattr__` is closer to `__getattribute__` in this sense). To avoid recursion, when `x.__setattr__` binds `x`'s attributes, it must modify `x.__dict__` directly (e.g., via `x.__dict__[name]=value`); even better, `__setattr__` can delegate the setting to the superclass (by calling `super(C, x).__setattr__('y', value)` or, in v3, just `super().__setattr__('y', value)`). Python ignores the return value of `__setattr__`. If `__setattr__` is absent (i.e., inherited from `object`), and `C.y` is not an overriding descriptor, Python usually translates `x.y=z` into `x.__dict__['y']=z`.

__str__	`__str__(self)` The `str(x)` built-in type and the `print(x)` function call `x.__str__()` to get an informal, concise string representation of `x`. If `__str__` is absent, Python calls `x.__repr__` instead. `__str__` should return a conveniently human-readable string, even if it entails some approximation.
__unicode__	`__unicode__(self)` In v2, calling `unicode(x)` calls `x.__unicode__()`, if present, in preference to `x.__str__()`. If a class supplies both special methods `__unicode__` and `__str__`, the two should return equivalent strings (of Unicode and plain-string type, respectively).
__format__	`__format__(self, format_string='')` Calling `format(x)` calls `x.__format__()`, and calling `format(x, format_string)` calls `x.__format__(format_string)`. The class is responsible for interpreting the format string (each class may define its own small "language" of format specifications, inspired by those implemented by built-in types as covered in "String Formatting"). If `__format__` is inherited from `object`, it delegates to `__str__` and does not accept a nonempty format string.

Special Methods for Containers

An instance can be a *container* (a sequence, mapping, or set—mutually exclusive concepts). For maximum usefulness, containers should provide special methods `__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `__contains__`, and `__iter__`, plus nonspecial methods discussed in the following sections. In many cases, suitable implementations of the nonspecial methods can be had by extending the appropriate *abstract base class*, from module `collections`, such as `Sequence`, `MutableSequence`, and so on, as covered in "Abstract Base Classes".

Sequences

In each item-access special method, a sequence that has `L` items should accept any integer `key` such that `-L<=key<L`. For compatibility with built-in sequences, a negative index `key`, `0>key>=-L`, should be equivalent to `key+L`. When `key` has an invalid type, indexing should raise `TypeError`. When `key` is a value of a valid type but out of range, indexing should raise `IndexError`. For sequence classes that do not define `__iter__`, the `for` statement relies on these requirements, as do built-in functions that take iterable arguments. Every item-access special method of a sequence should also, if at all practical, accept as its index argument an instance of the built-in type `slice` whose `start`, `step`, and `stop` attributes are `int`s or `None`; the *slicing* syntax relies on this requirement, as covered in "Container slicing".

A sequence should also allow concatenation (with another sequence of the same type) by `+`, and repetition by `*` (multiplication by an integer). A sequence should therefore have special methods `__add__`, `__mul__`, `__radd__`, and `__rmul__`, covered in "Special Methods for Numeric Objects"; mutable sequences should also have equivalent in-place methods `__iadd__` and `__imul__`. A sequence should be meaningfully comparable to another sequence of the same type, implementing *lexicographic* comparison like lists and tuples do. (Inheriting from ABCs `Sequence` or `MutableSequence`, alas, does not suffice to fulfill these requirements; such inheritance only supplies `__iadd__`.)

Every sequence should have the nonspecial methods covered in "List methods": `count` and `index` in any case, and, if mutable, then also `append`, `insert`, `extend`, `pop`, `remove`, `reverse`, and `sort`, with the same signatures and semantics as the corresponding methods of lists. (Inheriting from ABCs `Sequence` or `MutableSequence` does suffice to fulfill these requirements, except for `sort`.)

An immutable sequence should be hashable if, and only if, all of its items are. A sequence type may constrain its items in some ways (for example, accepting only string items), but that is not mandatory.

Mappings

A mapping's item-access special methods should raise `KeyError`, rather than `IndexError`, when they receive an invalid `key` argument value of a valid type. Any mapping should define the nonspecial methods covered in "Dictionary Methods": `copy`, `get`, `items`, `keys`, `values`, and, in v2, `iteritems`, `iterkeys`, and `itervalues`. In v2, special method `__iter__` should be equivalent to `iterkeys` (in v3, it should be equivalent to `keys`, which, in v3, has the semantics `iterkeys` has in v2). A mutable mapping should also define methods `clear`, `pop`, `popitem`, `setdefault`, and `update`. (Inheriting from ABCs `Mapping` or `MutableMapping` does fulfill these requirements, except for `copy`.)

An immutable mapping should be hashable if all of its items are. A mapping type may constrain its keys in some ways (for example, accepting only hashable keys, or, even more specifically, accepting, say, only string keys), but that is not mandatory. Any mapping should be meaningfully comparable to another mapping of the same type (at least for equality and inequality; not necessarily for ordering comparisons).

Sets

Sets are a peculiar kind of container—containers that are neither sequences nor mappings, and cannot be indexed, but do have a length (number of elements) and are iterable. Sets also support many operators (`&`, `|`, `^`, `-`, as well as membership tests and comparisons) and equivalent nonspecial methods (`intersection`, `union`, and so on). If you implement a set-like container, it should be polymorphic to Python built-in sets, covered in "Sets". (Inheriting from ABCs `Set` or `MutableSet` does fulfill these requirements.)

An immutable set-like type should be hashable if all of its elements are. A set-like type may constrain its elements in some ways (for example, accepting only hashable elements, or, even more specifically, accepting, say, only integer elements), but that is not mandatory.

Container slicing

When you reference, bind, or unbind a slicing such as `x[i:j]` or `x[i:j:k]` on a container `x` (in practice, this is only used with sequences), Python calls `x`'s applicable item-access special method, passing as `key` an object of a built-in type called a *slice object*. A slice object has the attributes `start`, `stop`, and `step`. Each attribute is `None` if you omit the corresponding value in the slice syntax. For example, `del x[:3]` calls `x.__delitem__(y)`, where `y` is a slice object such that `y.stop` is `3`, `y.start` is `None`, and `y.step` is `None`. It is up to container object `x` to appropriately interpret slice object arguments passed to `x`'s special methods. The method `indices` of slice objects can help: call it with your container's length as its only argument, and it returns a tuple of three nonnegative indices suitable as `start`, `stop`, and `step` for a loop indexing each item in the slice. A common idiom in a sequence class's `__getitem__` special method, to fully support slicing, is, for example:

```
def __getitem__(self, index):
  # Recursively specialcase
  slicing
  if isinstance(index, slice):
    return self.__class__(self[x]
                          for x in range(*self.indices(len(self
))))
  # Check index, dealing with negative indices
  too
  if not isinstance(index, numbers.Integral): raise TypeError
  if index < 0: index += len(self)
  if not (0 <= index < len(self)): raise IndexError
  # Index is now a correct integral number, within
  range(len(self))
  ...rest of __getitem__, dealing with single-item access...
```

This idiom uses generator-expression (genexp) syntax and assumes that your class's `__init__` method can be called with an iterable argument to create a suitable new instance of the class.

Container methods

The special methods `__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__`, and `__contains__` expose container functionality (see Table 4-2).

Table 4-2. Container methods

__contains__	`__contains__(self,item)`
	
	The Boolean test `y in x` calls `x.__contains__(y)`. When `x` is a sequence, or set-like, `__contains__` should return `True` when `y` equals the value of an item in `x`. When `x` is a mapping, `__contains__` should return `True` when `y` equals the value of a key in `x`. Otherwise, `__contains__` should return `False`. When `__contains__` is absent, Python performs `y in x` as follows, taking time proportional to `len(x)`:
	
	`for z in x:    if y==z: return Truereturn False`

__delitem__	`__delitem__(self,key)` For a request to unbind an item or slice of `x` (typically `del x[key]`), Python calls `x.__delitem__(key)`. A container `x` should have `__delitem__` only if `x` is mutable so that items (and possibly slices) can be removed.
__getitem__	`__getitem__(self,key)` When you access `x[key]` (i.e., when you index or slice container `x`), Python calls `x.__getitem__(key)`. All (non-set-like) containers should have `__getitem__`.

__iter__	`__iter__(self)` For a request to loop on all items of `x` (typically `for item in x`), Python calls `x.__iter__()` to get an iterator on `x`. The built-in function `iter(x)` also calls `x.__iter__()`. When `__iter__` is absent, `iter(x)` synthesizes and returns an iterator object that wraps `x` and yields `x[0]`, `x[1]`, and so on, until one of these indexings raises `IndexError` to indicate the end of the container. However, it is best to ensure that all of the container classes you code have `__iter__`.
__len__	`__len__(self)` Calling `len(x)` calls `x.__len__()` (and so do other built-in functions that need to know how many items are in container `x`). `__len__` should return an `int`, the number of items in `x`. Python also calls `x.__len__()` to evaluate `x` in a Boolean context, when `__nonzero__` (`__bool__` in v3) is absent; in this case, a container is taken as false if and only if the container is empty (i.e., the container's length is `0`). All containers should have `__len__`, unless it's just too expensive for the container to determine how many items it contains.
__setitem__	`__setitem__(self,key,value)` For a request to bind an item or slice of `x` (typically an assignment `x[key]=value`), Python calls `x.__setitem__(key,value)`. A container `x` should have `__setitem__` only if `x` is mutable so that items, and possibly slices, can be added and/or rebound.

Abstract Base Classes

Abstract base classes (ABCs) are an important pattern in object-oriented (OO) design: they're classes that cannot be directly instantiated, but exist only to be extended by concrete classes (the more usual kind of classes, the ones that can be instantiated).

One recommended approach to OO design is to never extend a concrete class: if two concrete classes have so much in common that you're tempted to have one of them inherit from the other, proceed instead by making an *abstract* base class that subsumes all they do have in common, and have each concrete class extend that ABC. This approach avoids many of the subtle traps and pitfalls of inheritance.

Python offers rich support for ABCs, enough to make them a first-class part of Python's object model.

abc

The standard library module `abc` supplies metaclass `ABCMeta` and, in v3, class `ABC` (subclassing `ABC` makes `ABCMeta` the metaclass, and has no other effect).

When you use `abc.ABCMeta` as the metaclass for any class `C`, this makes `C` an ABC, and supplies the class method *`C`*`.register`, callable with a single argument: that argument can be any existing class (or built-in type) `X`.

Calling *`C`*`.register(`*`X`*`)` makes `X` a *virtual* subclass of `C`, meaning that `issubclass(`*`X`*`,`*`C`*`)` returns `True`, but `C` does not appear in *`X`*`.__mro__`, nor does `X` inherit any of `C`'s methods or other attributes.

Of course, it's also possible to have a new class `Y` inherit from `C` in the normal way, in which case `C` does appear in *`Y`*`.__mro__`, and `Y` inherits all of `C`'s methods, as usual in subclassing.

An ABC `C` can also optionally override class method `__subclasshook__`, which `issubclass(`*`X`*`,`*`C`*`)` calls with the single argument `X`, `X` being any class or type. When *`C`*`.__subclasshook__(`*`X`*`)` returns `True`, then so does `issubclass(`*`X`*`,`*`C`*`)`; when *`C`*`.__subclasshook__(`*`X`*`)` returns `False`, then so does `issubclass(`*`X`*`,`*`C`*`)`; when

`C.__subclasshook__(X)` returns `NotImplemented`, then `issubclass(X,C)` proceeds in the usual way.

The module `abc` also supplies the decorator `abstractmethod` (and `abstractproperty`, but the latter is deprecated in v3, where you can just apply both the `abstractmethod` and `property` decorators to get the same effect). Abstract methods and properties can have implementations (available to subclasses via the `super` built-in)—however, the point of making methods and properties abstract is that you can instantiate any nonvirtual subclass `X` of an ABC `C` only if `X` overrides every abstract property and method of `C`.

ABCs in the collections module

`collections` supplies many ABCs. Since Python 3.4, the ABCs are in `collections.abc` (but, for backward compatibility, can still be accessed directly in `collections` itself: the latter access will cease working in some future release of v3).

Some just characterize any class defining or inheriting a specific abstract method, as listed in Table 4-3:

Table 4-3.

Callable	Any class with `__call__`
Container	Any class with `__contains__`
Hashable	Any class with `__hash__`
Iterable	Any class with `__iter__`
Sized	Any class with `__len__`

The other ABCs in `collections` extend one or more of the preceding ones, add more abstract methods, and supply *mixin* methods implemented in terms of the abstract methods (when you extend any ABC in a concrete class, you must override the abstract methods; you can optionally override some or all of the mixin methods, if that helps improve performance, but you don't have to—you can just inherit them, if this results in performance that's sufficient for your purposes).

Here is the set of ABCs directly extending the preceding ones:

ABC	Extends	Abstract methods	Mixin methods
Iterator	`Iterable`	`__next__` (in v2, `next`)	`__iter__`
Mapping	`Container`, `Iterable`, `Sized`	`__getitem__`, `__iter__`, `__len__`	`__contains__`, `__eq__`, `__ne__`, `get`, `items`, `keys`, `values`
MappingView	`Sized`		`__len__`
Sequence	`Container`, `Iterable`, `Sized`	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `count`, `index`
Set	`Container`, `Iterable`, `Sized`	`__contains__`, `__iter`, `__len__`	`__and__`, `__eq__`, `__ge__`, `__gt__`, `__le__`, `__lt__`, `__ne__`, `__or__`, `__sub__`, `__xor__`, `isdisjoint`

And lastly, the set of ABCs further extending the previous ones:

ABC	Extends	Abstract methods	Mixin methods
ItemsView	`MappingView`, `Set`		`__contains__`, `__iter__`
KeysView	`MappingView`, `Set`		`__contains__`, `__iter__`
MutableMapping	`Mapping`	`__delitem__`, `__getitem__`, `__iter__`, `__len__`, `__setitem__`	`Mapping`'s methods, plus `clear`, `pop`, `popitem`, `setdefault`, `update`
MutableSequence	`Sequence`	`__delitem__`, `__getitem__`, `__len__`, `__setitem__`, `insert`	`Sequence`'s methods, plus `__iadd__`, `append`, `extend`, `pop`, `remove`, `reverse`
MutableSet	`Set`	`__contains__`, `__iter`, `__len__`, `add`, `discard`	`Set`'s methods, plus `__iand__`, `__ior__`, `__isub__`, `__ixor__`, `clear`, `pop`, `remove`
ValuesView	`MappingView`		`__contains__`, `__iter__`

See the online docs for further details and usage examples.

The numbers module

`numbers` supplies a hierarchy (also known as a *tower*) of ABCs representing various kinds of numbers. `numbers` supplies the following ABCs:

Number	The root of the hierarchy: numbers of *any* kind (need not support any given operation)
Complex	Extends `Number`; must support (via the appropriate special methods) conversions to `complex` and `bool`, `+`, `-`, `*`, `/`, `==`, `!=`, `abs()`; and, directly, the method `conjugate()` and properties `real` and `imag`
Real	Extends `Complex`; additionally, must support (via the appropriate special methods) conversion to `float`, `math.trunc()`, `round()`, `math.floor()`, `math.ceil()`, `divmod()`, `//`, `%`, `<`, `<=`, `>`, `>=`
Rational	Extends `Real`; additionally, must support the properties `numerator` and `denominator`
Integral	Extends `Rational`; additionally, must support (via the appropriate special methods) conversion to `int`, `**`, and bitwise operations `<<`, `>>`, `&`, `^`, `|`, `~`

See the online docs for notes on implementing your own numeric types.

Special Methods for Numeric Objects

An instance may support numeric operations by means of many special methods. Some classes that are not numbers also support some of the special methods in Table 4-4 in order to overload operators such as `+` and `*`. In particular, sequences should have special methods `__add__`, `__mul__`, `__radd__`, and `__rmul__`, as mentioned in "Sequences".

Table 4-4.

__abs__, __invert__, __neg__, __pos__	`__abs__(self) __invert__(self) __neg__(self)`
	`__pos__(self)`
	
	The unary operators `abs(x)`, `~x`, `-x`, and `+x`, respectively, call these methods.

__add__, __mod__, __mul__, __sub__	`) )`
	`__add__(self,other__mod__(self, other__mul__(self,other)`
	`__sub__(self,other)`
	
	The operators `x+y`, `x%y`, `x*y`, and `x-y`, and `x/y`, respectively, call these methods, usually for arithmetic computations.

__div__, __floordiv__, __truediv__	`) )`
	`__div__(self,other__floordiv__(self, other__truediv__(self,other)`
	
	The operators `x/y` and `x//y` call these methods, usually for arithmetic divisions. In v2, operator `/` calls `__truediv__`, if present, instead of `__div__`, in situations where division is nontruncating, as covered in “Arithmetic Operations”. In v3, there is no `__div__`, only `__truediv__` and `__floordiv__`.

__and__, __lshift__, __or__, __rshift__, __xor__	`) )`
	`__and__(self,other__lshift__(self, other__or__(self,other`
	`) )`
	`__rshift__(self, other__xor__(self,other)`
	
	The operators `x&y`, `x<<y`, `x|y`, `x>>y`, and `x^y`, respectively, call these methods, usually for bitwise operations.

__complex__, __float__, __int__, __long__	`__complex__(self) __float__(self) __int__(self)`
	`__long__(self)`
	
	The built-in types `complex(x)`, `float(x)`, `int(x)`, and (in v2 only) `long(x)`, respectively, call these methods.

__divmod__	`__divmod__(self,other)` The built-in function `divmod(x,y)` calls `x.__divmod__(y)`. `__divmod__` should return a pair `(quotient,remainder)` equal to `(x//y,x%y)`.
__hex__, __oct__	`__hex__(self) __oct__(self)` In v2 only, the built-in function `hex(x)` calls `x.__hex__()`, and built-in function `oct(x)` calls `x.__oct__()`. Each of these special methods should return a string representing the value of `x`, in base 16 and 8, respectively. In v3, these special methods don’t exist: the built-in functions `hex` and `oct` operate directly on the result of calling the special method `__index__` on their operand.

__iadd__, __idiv__, __ifloordiv__, __imod__, __imul__, __isub__, __itruediv__	`) )`
	`__iadd__(self,other__idiv__(self, other__ifloordiv__(self,other`
	`) ) )`
	`__imod__(self, other__imul__(self,other__isub__(self, other`
	`)`
	`__itruediv__(self,other)`
	
	The augmented assignments `x+=y`, `x/=y`, `x//=y`, `x%=y`, `x*=y`, `x-=y`, and `x/=y`, respectively, call these methods. Each method should modify `x` in place and return `self`. Define these methods when `x` is mutable (i.e., when `x` can change in place).

__iand__, __ilshift__, __ior__, __irshift__, __ixor__	`) )`
	`__iand__(self,other__ilshift__(self, other__ior__(self,other`
	`) )`
	`__irshift__(self, other__ixor__(self,other)`
	
	The augmented assignments `x&=y`, `x<<=y`, `x|=y`, `x>>=y`, and `x^=y`, respectively, call these methods. Each method should modify `x` in place and return `self`.

__index__	`__index__(self)` Like `__int__`, but meant to be supplied only by types that are alternative implementations of integers (in other words, all of the type's instances can be exactly mapped into integers). For example, out of all built-in types, only `int` (and, in v2, `long`) supply `__index__`; `float` and `str` don't, although they do supply `__int__`. Sequence indexing and slicing internally use `__index__` to get the needed integer indices.
__ipow__	`__ipow__(self,other)` The augmented assignment `x**=y` calls `x.__ipow__(y)`. `__ipow__` should modify `x` in place and return `self`.
__pow__	`__pow__(self,other[,modulo])` `x**y` and `pow(x,y)` both call `x.__pow__(y)`, while `pow(x,y,z)` calls `x.__pow__(y,z)`. `x.__pow__(y,z)` should return a value equal to the expression `x.__pow__(y)%z`.

__radd__, __rdiv__, __rmod__, __rmul__, __rsub__	`) )`
	`__radd__(self,other__rdiv__(self, other__rmod__(self,other`
	`) )`
	`__rmul__(self, other__rsub__(self,other)`
	
	The operators `y+x`, `y/x`, `y%x`, `y*x`, and `y-x`, respectively, call these methods on `x` when `y` doesn't have a needed method `__add__`, `__div__`, and so on, or when that method returns `NotImplemented`.

__rand__, __rlshift__, __ror__, __rrshift__, __rxor__	`) )`
	`__rand__(self,other__rlshift__(self, other__ror__(self,other`
	`) )`
	`__rrshift__(self, other__rxor__(self,other)`
	
	The operators `y&x`, `y<<x`, `y|x`, `y>>x`, and `x^y`, respectively, call these methods on `x` when `y` doesn't have a needed method `__and__`, `__lshift__`, and so on, or when that method returns `NotImplemented`.

__rdivmod__	`__rdivmod__(self,other)` The built-in function `divmod(y,x)` calls `x.__rdivmod__(y)` when `y` doesn't have `__divmod__`, or when that method returns `NotImplemented`. `__rdivmod__` should return a pair `(remainder,quotient)`.
__rpow__	`__rpow__(self,other)` `y**x` and `pow(y,x)` call `x.__rpow__(y)` when `y` doesn't have `__pow__`, or when that method returns `NotImplemented`. There is no three-argument form in this case.