Chapter 4. Object-Oriented Python

Python is an object-oriented (OO) programming language. Unlike some other object-oriented languages, however, Python doesn't force you to use the object-oriented paradigm exclusively: it also supports procedural programming, with modules and functions, so that you can select the best paradigm for each part of your program. The object-oriented paradigm helps you group state (data) and behavior (code) together in handy packets of functionality. Moreover, it offers some useful specialized mechanisms covered in this chapter, like *inheritance* and *special methods*. The simpler procedural approach, based on modules and functions, may be more suitable when you don't need the pluses[1] of object-oriented programming. With Python, you can mix and match paradigms.

In addition to core OO concepts, this chapter covers *abstract base classes*, *decorators*, and *metaclasses*.

Classes and Instances

If you're familiar with object-oriented programming in other OO languages such as C++ or Java, you probably have a good grasp of classes and instances: a *class* is a user-defined type, which you *instantiate* to build *instances*, i.e., objects of that type. Python supports this through its class and instance objects.

Python Classes

A *class* is a Python object with the following characteristics:

- You can call a class object just like you'd call a function. The call, known as *instantiation*, returns an object known as an *instance* of the class; the class is also known as the instance's *type*.

- A class has arbitrarily named attributes that you can bind and reference.
- The values of class attributes can be *descriptors* (including functions), covered in **"Descriptors"**, or ordinary data objects.
- Class attributes bound to functions are also known as *methods* of the class.
- A method can have any one of many Python-defined names with two leading and two trailing underscores (known as *dunder names*, short for "double-underscore names"—the name `__init__`, for example, is pronounced "dunder init"). Python implicitly calls such special methods, when a class supplies them, when various kinds of operations occur on that class or its instances.
- A class can *inherit* from one or more classes, meaning it delegates to other class objects the lookup of some attributes (including regular and dunder methods) that are not in the class itself.

An instance of a class is a Python object with arbitrarily named attributes that you can bind and reference. Every instance object delegates attribute lookup to its class for any attribute not found in the instance itself. The class, in turn, may delegate the lookup to classes from which it inherits, if any.

In Python, classes are objects (values), handled just like other objects. You can pass a class as an argument in a call to a function, and a function can return a class as the result of a call. You can bind a class to a variable, an item in a container, or an attribute of an object. Classes can also be keys into a dictionary. Since classes are perfectly ordinary objects in Python, we often say that classes are *first-class* objects.

The class Statement

The **`class`** statement is the most usual way you create a class object. **`class`** is a single-clause compound statement with the following syntax:

```
class Classname(base-classes, *, **kw):
```

```
    statement(s)
```

Classname is an identifier: a variable that the class statement, when finished, binds (or rebinds) to the just-created class object. Python naming **conventions** advise using title case for class names, such as `Item`, `PrivilegedUser`, `MultiUseFacility`, etc.

*base-classes_* is a comma-delimited series of expressions whose values are class objects. Various programming languages use different names for these class objects: you can call them the *bases, superclasses*, or *parents* of the class. You can say the class created *inherits* from, *derives* from, *extends*, or *subclasses* its base classes; in this book, we generally use *extend*. This class is a *direct subclass* or *descendant* of its base classes. `**kw` can include a named argument `metaclass=` to establish the class's *metaclass*,[2] as covered in **"How Python Determines a Class's Metaclass"**.

Syntactically, including *base-classes* is optional: to indicate that you're creating a class without bases, just omit *base-classes* (and, optionally, also omit the parentheses around it, placing the colon right after the class name). Every class inherits from `object`, whether you specify explicit bases or not.

The subclass relationship between classes is transitive: if *C1* extends *C2*, and *C2* extends *C3*, then *C1* extends *C3*. The built-in function `issubclass(C1, C2)` accepts two class objects: it returns **`True`** when *C1* extends *C2*, and otherwise it returns **`False`**. Any class is a subclass of itself; therefore, `issubclass(C, C)` returns **`True`** for any class *C*. We cover how base classes affect a class's functionality in **"Inheritance"**.

The nonempty sequence of indented statements that follows the **`class`** statement is the *class body*. A class body executes immediately as part of the **`class`** statement's execution. Until the body finishes executing, the new class object does not yet exist, and the *Classname* identifier is not yet bound (or rebound). **"How a Metaclass Creates a Class"** provides more details about what happens when a **`class`** statement executes. Note that the **`class`** statement does not immediately create any instance of the new class, but rather defines the set of attributes shared by all instances when you later create instances by calling the class.

The Class Body

The body of a class is where you normally specify class attributes; these attributes can be descriptor objects (including functions) or ordinary data objects of any type. An attribute of a class can be another class—so, for example, you can have a **`class`** statement "nested" inside another **`class`** statement.

Attributes of class objects

You usually specify an attribute of a class object by binding a value to an identifier within the class body. For example:

```
class C1:
    x = 23
print(C1.x)                             # prints: 23
```

Here, the class object `C1` has an attribute named `x`, bound to the value `23`, and `C1.x` refers to that attribute. Such attributes may also be accessed via instances: `c = C1(); print(c.x)`. However, this isn't always reliable in practice. For example, when the class instance `c` has an `x` attribute, that's what `c.x` accesses, not the class-level one. So, to access a class-level attribute from an instance, using, say, `print(c.__class__.x)` may be best.

You can also bind or unbind class attributes outside the class body. For example:

```
class C2:
    pass
C2.x = 23
print(C2.x)                             # prints: 23
```

Your program is usually more readable if you bind class attributes only with statements inside the class body. However, rebinding them elsewhere may be necessary if you want to carry state information at a class, rather than instance, level; Python lets you do that, if you wish. There is no difference between a class attribute bound in the class body and one bound or rebound outside the body by assigning to an attribute.

As we'll discuss shortly, all class instances share all of the class's attributes.

The **`class`** statement implicitly sets some class attributes. The attribute `__name__` is the *`Classname`* identifier string used in the **`class`** statement. The attribute `__bases__` is the tuple of class objects given (or implied) as the base classes in the **`class`** statement. For example, using the class `C1` we just created:

```
print(C1.__name__, C1.__bases__) # prints: C1 (<class 'object'>,)
```

A class also has an attribute called `__dict__`, which is the read-only mapping that the class uses to hold other attributes (also known, informally, as the class's *namespace*).

In statements directly in a class's body, references to class attributes must use a simple name, not a fully qualified name. For example:

```
class C3:
    x = 23
    y = x + 22                    # must use just x, not C3.x
```

However, in statements within *methods* defined in a class body, references to class attributes must use a fully qualified name, not a simple name. For example:

```
class C4:
```

```
    x = 23
    def amethod(self):
        print(C4.x)            # must use C4.x or self.x, not just x!
```

Attribute references (i.e., expressions like `C.x`) have semantics richer than attribute bindings. We cover such references in detail in **"Attribute Reference Basics"**.

Function definitions in a class body

Most class bodies include some **`def`** statements, since functions (known as *methods* in this context) are important attributes for most class instances. A **`def`** statement in a class body obeys the rules covered in **"Functions"**. In addition, a method defined in a class body has a mandatory first parameter, conventionally always named `self`, that refers to the instance on which you call the method. The `self` parameter plays a special role in method calls, as covered in **"Bound and Unbound Methods"**.

Here's an example of a class that includes a method definition:

```
class C5:
    def hello(self):
        print('Hello')
```

A class can define a variety of special dunder methods relating to specific operations on its instances. We discuss these methods in detail in **"Special Methods"**.

Class-private variables

When a statement in a class body (or in a method in the body) uses an identifier starting (but not ending) with two underscores, such as `__ident`, Python implicitly changes the identifier to `_Classname__ident`, where `Classname` is the name of the class. This implicit change lets a class use "private" names for attributes, methods, global variables, and other purposes, reducing the risk of accidentally duplicating names used elsewhere (particularly in subclasses).

By convention, identifiers starting with a *single* underscore are private to the scope that binds them, whether that scope is or isn't a class. The Python compiler does not enforce this privacy convention: it is up to programmers to respect it.

Class documentation strings

If the first statement in the class body is a string literal, the compiler binds that string as the *documentation string* (or *docstring*) for the class. The docstring for the class is available in the `__doc__` attribute; if the first statement in the class body is *not* a string literal, its value is **None**. See "Docstrings" for more information on documentation strings.

Descriptors

A *descriptor* is an object whose class supplies one or more special methods named `__get__`, `__set__`, or `__delete__`. Descriptors that are class attributes control the semantics of accessing and setting attributes on instances of that class. Roughly speaking, when you access an instance attribute, Python gets the attribute's value by calling `__get__` on the corresponding descriptor, if any. For example:

```
class Const:  # class with an overriding descriptor, see later
    def __init__(self, value):
        self.__dict__['value'] = value
    def __set__(self, *_):
        # silently ignore any attempt at setting
        # (a better design choice might be to raise AttributeError)
        pass
    def __get__(self, *_):
        # always return the constant value
        return self.__dict__['value']
    def __delete__(self, *_):
        # silently ignore any attempt at deleting
```

```
        # (a better design choice might be to raise AttributeError)
        pass

class X:
    c = Const(23)

x = X()
print(x.c)  # prints: 23
x.c = 42    # silently ignored (unless you raise AttributeError)
print(x.c)  # prints: 23
del x.c     # silently ignored again (ditto)
print(x.c)  # prints: 23
```

For more details, see **"Attribute Reference Basics"**.

Overriding and nonoverriding descriptors

When a descriptor's class supplies a special method named `__set__`, the descriptor is known as an *overriding descriptor* (or, using the older, confusing terminology, a *data descriptor*); when the descriptor's class supplies `__get__` and not `__set__`, the descriptor is known as a *nonoverriding descriptor*.

For example, the class of function objects supplies `__get__`, but not `__set__`; therefore, function objects are nonoverriding descriptors. Roughly speaking, when you assign a value to an instance attribute with a corresponding descriptor that is overriding, Python sets the attribute value by calling `__set__` on the descriptor. For more details, see **"Attributes of instance objects"**.

The third dunder method of the descriptor protocol is `__delete__`, called when the **`del`** statement is used on the descriptor instance. If **`del`** is not supported, it is still a good idea to implement `__delete__`, raising a proper `AttributeError` exception; otherwise, the caller will get a mysterious `AttributeError: __delete__` exception.

The **online docs** include many more examples of descriptors and their related methods.

Instances

To create an instance of a class, call the class object as if it were a function. Each call returns a new instance whose type is that class:

```
an_instance = C5()
```

The built-in function `isinstance(i, C)`, with a class as argument *C*, returns **`True`** when *i* is an instance of class *C* or any subclass of *C*. Otherwise, `isinstance` returns **`False`**. If *C* is a tuple of types (**3.10+** or multiple types joined using the | operator), `isinstance` returns **`True`** if *i* is an instance or subclass instance of any of the given types, and **`False`** otherwise.

__init__

When a class defines or inherits a method named `__init__`, calling the class object executes `__init__` on the new instance to perform per instance initialization. Arguments passed in the call must correspond to `__init__`'s parameters, except for the parameter `self`. For example, consider the following class definition:

```
class C6:
    def __init__(self, n):
        self.x = n
```

Here's how you can create an instance of the `C6` class:

```
another_instance = C6(42)
```

As shown in the `C6` class definition, the `__init__` method typically contains statements that bind instance attributes. An `__init__` method must not return a value other than **None**; if it does, Python raises a `TypeError` exception.

The main purpose of `__init__` is to bind, and thus create, the attributes of a newly created instance. You may also bind, rebind, or unbind instance attributes outside `__init__`. However, your code is more readable when you initially bind all class instance attributes in the `__init__` method.

When `__init__` is absent (and not inherited from any base class), you must call the class without arguments, and the new instance has no instance-specific attributes.

Attributes of instance objects

Once you have created an instance, you can access its attributes (data and methods) using the dot (`.`) operator. For example:

```
an_instance.hello()                     # prints: Hello
print(another_instance.x)               # prints: 42
```

Attribute references such as these have fairly rich semantics in Python; we cover them in detail in **"Attribute Reference Basics"**.

You can give an instance object an attribute by binding a value to an attribute reference. For example:

```
class C7:
    pass
z = C7()
z.x = 23
print(z.x)                              # prints: 23
```

Instance object `z` now has an attribute named `x`, bound to the value `23`, and `z.x` refers to that attribute. The `__setattr__` special method, if present, intercepts every attempt to bind an attribute. (We cover `__setattr__` in **Table 4-1**.)

When you attempt to bind to an instance attribute whose name corresponds to an overriding descriptor in the class, the descriptor's `__set__` method intercepts the attempt: if `C7.x` were an overriding descriptor, `z.x=23` would execute `type(z).x.__set__(z, 23)`.

Creating an instance sets two instance attributes. For any instance *z*, `z.__class__` is the class object to which *z* belongs, and `z.__dict__` is the mapping *z* uses to hold its other attributes. For example, for the instance `z` we just created:

```
print(z.__class__.__name__, z.__dict__)  # prints: C7 {'x':23}
```

You may rebind (but not unbind) either or both of these attributes, but this is rarely necessary.

For any instance *z*, any object *x*, and any identifier *S* (except `__class__` and `__dict__`), `z.S=x` is equivalent to `z.__dict__['S']=x` (unless a `__setattr__` special method, or an overriding descriptor's `__set__` special method, intercepts the binding attempt). For example, again referring to the `z` we just created:

```
z.y = 45
z.__dict__['z'] = 67
print(z.x, z.y, z.z)                 # prints: 23 45 67
```

There is no difference between instance attributes created by assigning to attributes and those created by explicitly binding an entry in `z.__dict__`.

The factory function idiom

It's often necessary to create instances of different classes depending on some condition, or avoid creating a new instance if an existing one is available for reuse. A common misconception is that such needs might be met by having `__init__` return a particular object. However, this approach is infeasible: Python raises an exception if `__init__` returns any value other than **None**. The best way to implement flexible object creation is to use a function rather than calling the class object directly. A function used this way is known as a *factory function*.

Calling a factory function is a flexible approach: a function may return an existing reusable instance or create a new instance by calling whatever class is appropriate. Say you have two almost interchangeable classes, `SpecialCase` and `NormalCase`, and want to flexibly generate instances of either one of them, depending on an argument. The following `appropriate_case` factory function, as a "toy" example, allows you to do just that (we'll talk more about the `self` parameter in **"Bound and Unbound Methods"**):

```
class SpecialCase:
    def amethod(self):
        print('special')
class NormalCase:
    def amethod(self):
        print('normal')
def appropriate_case(isnormal=True):
    if isnormal:
        return NormalCase()
    else:
        return SpecialCase()
aninstance = appropriate_case(isnormal=False)
aninstance.amethod()                  # prints: special
```

__new__

Every class has (or inherits) a class method named `__new__` (we cover class methods in **"Class methods"**). When you call `C(*args, **kwds)` to create a new instance of class `C`, Python first calls `C.__new__(C, *args, **kwds)`, and uses `__new__`'s return value *x* as the newly created instance. Then Python calls `C.__init__(x, *args, **kwds)`, but only when *x* is indeed an instance of `C` or any of its subclasses (otherwise, *x*'s state remains as `__new__` had left it). Thus, for example, the statement `x=C(23)` is equivalent to:

```
x = C.__new__(C, 23)
if isinstance(x, C):
    type(x).__init__(x, 23)
```

`object.__new__` creates a new, uninitialized instance of the class it receives as its first argument. It ignores other arguments when that class has an `__init__` method, but it raises an exception when it receives other arguments beyond the first, and the class that's the first argument does not have an `__init__` method. When you override `__new__` within a class body, you do not need to add `__new__=classmethod(__new__)`, nor use an `@classmethod` decorator, as you normally would: Python recognizes the name `__new__` and treats it as special in this context. In those sporadic cases in which you rebind `C.__new__` later, outside the body of class `C`, you do need to use `C.__new__=classmethod(`*`whatever`*`)`.

`__new__` has most of the flexibility of a factory function, as covered in the previous section. `__new__` may choose to return an existing instance or make a new one, as appropriate. When `__new__` does create a new instance, it usually delegates creation to `object.__new__` or the `__new__` method of another superclass of `C`.

The following example shows how to override the class method `__new__` in order to implement a version of the Singleton design pattern:

```
class Singleton:
    _singletons = {}
    def __new__(cls, *args, **kwds):
        if cls not in cls._singletons:
            cls._singletons[cls] = obj = super().__new__(cls)
            obj._initialized = False
        return cls._singletons[cls]
```

(We cover the built-in `super` in **“Cooperative superclass method calling”**.)

Any subclass of `Singleton` (that does not further override `__new__`) has exactly one instance. When the subclass defines `__init__`, it must ensure `__init__` is safe to call repeatedly (at each call of the subclass) on the subclass’s only instance.[3] In this example, we insert the `_initialized` attribute, set to **`False`**, when `__new__` actually creates a new instance. Subclasses’ `__init__` methods can test if `self._initialized` is **`False`** and, if so, set it to **`True`** and continue with the rest of the `__init__` method. When subsequent “creates” of the singleton instance call `__init__` again, `self._initialized` will be **`True`**, indicating the instance is already initialized, and `__init__` can typically just return, avoiding some repetitive work.

Attribute Reference Basics

An *attribute reference* is an expression of the form *`x.name`*, where *`x`* is any expression and *name* is an identifier called the *attribute name*. Many Python objects have attributes, but an attribute reference has special, rich semantics when *x* refers to a class or instance. Methods are attributes, too, so everything we say about attributes in general also applies to callable attributes (i.e., methods).

Say that *x* is an instance of class `C`, which inherits from base class `B`. Both classes and the instance have several attributes (data and methods), as follows:

```
class B:
    a = 23
    b = 45
    def f(self):
        print('method f in class B')
    def g(self):
        print('method g in class B')
class C(B):
    b = 67
```

```
    c = 89
    d = 123
    def g(self):
        print('method g in class C')
    def h(self):
        print('method h in class C')
x = C()
x.d = 77
x.e = 88
```

A few attribute dunder names are special. `C.__name__` is the string `'C'`, the class's name. `C.__bases__` is the tuple `(B,)`, the tuple of *C*'s base classes. `x.__class__` is the class *C* to which *x* belongs. When you refer to an attribute with one of these special names, the attribute reference looks directly into a dedicated slot in the class or instance object and fetches the value it finds there. You cannot unbind these attributes. You may rebind them on the fly, changing the name or base classes of a class or the class of an instance, but this advanced technique is rarely necessary.

Class *C* and instance *x* each have one other special attribute: a mapping named `__dict__` (typically mutable for *x*, but not for *C*). All other attributes of a class or instance,[4] except the few special ones, are held as items in the `__dict__` attribute of the class or instance.

Getting an attribute from a class

When you use the syntax `C.name` to refer to an attribute on a class object *C*, lookup proceeds in two steps:

1. When `'name'` is a key in `C.__dict__`, `C.name` fetches the value *v* from `C.__dict__['name']`. Then, when *v* is a descriptor (i.e., `type(v)` supplies a method named `__get__`), the value of `C.name` is the result of calling `type(v).__get__(v, None, C)`. When *v* is not a descriptor, the value of `C.name` is *v*.
2. When `'name'` is *not* a key in `C.__dict__`, `C.name` delegates the lookup to *C*'s base classes, meaning it loops on *C*'s ancestor classes and tries the *name* lookup on each (in *method resolution order*, as covered in **"Inheritance"**).

Getting an attribute from an instance

When you use the syntax `x.name` to refer to an attribute of instance *x* of class *C*, lookup proceeds in three steps:

1. When `'name'` is in *C* (or in one of *C*'s ancestor classes) as the name of an overriding descriptor *v* (i.e., `type(v)` supplies methods `__get__` and `__set__`), the value of `x.name` is the result of `type(v).__get__(v, x, C)`.
2. Otherwise, when `'name'` is a key in `x.__dict__`, `x.name` fetches and returns the value at `x.__dict__['name']`.
3. Otherwise, `x.name` delegates the lookup to *x*'s class (according to the same two-step lookup process used for `C.name`, as just detailed):
    1. When this finds a descriptor *v*, the overall result of the attribute lookup is, again, `type(v).__get__(v, x, C)`.
    2. When this finds a nondescriptor value *v*, the overall result of the attribute lookup is just *v*.

When these lookup steps do not find an attribute, Python raises an `AttributeError` exception. However, for lookups of `x.name`, when *C* defines or inherits the special method `__getattr__`, Python calls `C.__getattr__(x, 'name')` rather than raising the exception. It's then up to `__getattr__` to return a suitable value or raise the appropriate exception, normally `AttributeError`.

Consider the following attribute references, defined previously:

```
print(x.e, x.d, x.c, x.b, x.a)              # prints: 88 77 89 67 23
```

`x.e` and `x.d` succeed in step 2 of the instance lookup process, since no descriptors are involved and `'e'` and `'d'` are both keys in `x.__dict__`. Therefore, the lookups go no further but rather return `88` and `77`. The other three references must proceed to step 3 of the instance lookup process and look in `x.__class__` (i.e., `C`). `x.c` and `x.b` succeed in step 1 of the class lookup process, since `'c'` and `'b'` are both keys in `C.__dict__`.

Therefore, the lookups go no further but rather return 89 and 67. `x.a` gets all the way to step 2 of the class lookup process, looking in `C.__bases__[0]` (i.e., `B`). `'a'` is a key in `B.__dict__`; therefore, `x.a` finally succeeds and returns 23.

Setting an attribute

Note that the attribute lookup steps happen as just described only when you *refer* to an attribute, not when you *bind* an attribute. When you bind to a class or instance attribute whose name is not special (unless a `__setattr__` method, or the `__set__` method of an overriding descriptor, intercepts the binding of an instance attribute), you affect only the `__dict__` entry for the attribute (in the class or instance, respectively). In other words, for attribute binding, there is no lookup procedure involved, except for the check for overriding descriptors.

Bound and Unbound Methods

The method `__get__` of a function object can return the function object itself, or a *bound method object* that wraps the function; a bound method is associated with the specific instance it's obtained from.

In the code in the previous section, the attributes `f`, `g`, and `h` are functions; therefore, an attribute reference to any one of them returns a method object that wraps the respective function. Consider the following:

```
print(x.h, x.g, x.f, C.h, C.g, C.f)
```

This statement outputs three bound methods, represented by strings like:

```
<bound method C.h of <__main__.C object at 0x8156d5c>>
```

and then three function objects, represented by strings like:

```
<function C.h at 0x102cabae8>
```

BOUND METHODS VERSUS FUNCTION OBJECTS

We get bound methods when the attribute reference is on instance *x*, and function objects when the attribute reference is on class *C*.

Because a bound method is already associated with a specific instance, you can call the method as follows:

```
x.h()                   # prints: method h in class C
```

The key thing to notice here is that you don't pass the method's first argument, `self`, by the usual argument-passing syntax. Rather, a bound method of instance *x* implicitly binds the `self` parameter to object *x*. Thus, the method's body can access the instance's attributes as attributes of `self`, even though we don't pass an explicit argument to the method.

Let's take a closer look at bound methods. When an attribute reference on an instance, in the course of the lookup, finds a function object that's an attribute in the instance's class, the lookup calls the function's `__get__` method to get the attribute's value. The call, in this case, creates and returns a *bound method* that wraps the function.

Note that when the attribute reference's lookup finds a function object directly in `x.__dict__`, the attribute reference operation does *not* create a bound method. In such cases, Python does not treat the function as a descriptor and does not call the function's `__get__` method; rather, the function object itself is the attribute's value. Similarly, Python creates no bound methods for callables that are not ordinary functions, such as built-in (as opposed to Python-coded) functions, since such callables are not descriptors.

A bound method has three read-only attributes in addition to those of the function object it wraps: `im_class` is the class object that supplies the method, `im_func` is the wrapped function, and `im_self` refers to *x*, the instance from which you got the method.

You use a bound method just like its `im_func` function, but calls to a bound method do not explicitly supply an argument corresponding to the first parameter (conventionally named `self`). When you call a bound method, the bound method passes `im_self` as the first argument to `im_func` before other arguments (if any) given at the point of call.

Let's follow, in excruciatingly low-level detail, the conceptual steps involved in a method call with the normal syntax `x.`*`name`*`(`*`arg`*`)`. In the following context:

```
def f(a, b): ...             # a function f with two arguments

class C:
    name = f
x = C()
```

`x` is an instance object of class `C`, `name` is an identifier that names a method of `x`'s (an attribute of `C` whose value is a function, in this case function `f`), and *arg* is any expression. Python first checks if `'name'` is the attribute name in `C` of an overriding descriptor, but it isn't—functions are descriptors, because their type defines the method `__get__`, but *not* overriding ones, because their type does not define the method `__set__`. Python next checks if `'name'` is a key in `x.__dict__`, but it isn't. So, Python finds `name` in `C` (everything would work just the same if `name` were found, by inheritance, in one of `C`'s `__bases__`). Python notices that the attribute's value, function object `f`, is a descriptor. Therefore, Python calls `f.__get__(x, C)`, which returns a bound method object with `im_func` set to `f`, `im_class` set to `C`, and `im_self` set to `x`. Then Python calls this bound method object, with *arg* as the only argument. The bound method inserts `im_self` (i.e., `x`) as the first argument, and *arg* becomes the second one in a call to the bound method's `im_func` (i.e., function `f`). The overall effect is just like calling:

```
x.__class__.__dict__['name'](x, arg)
```

When a bound method's function body executes, it has no special namespace relationship to either its `self` object or any class. Variables referenced are local or global, just like any other function, as covered in **"Namespaces"**. Variables do not implicitly indicate attributes in `self`, nor do they indicate attributes in any class object. When the method needs to refer to, bind, or unbind an attribute of its `self` object, it does so by standard attribute reference syntax (e.g., `self.name`).[5] The lack of implicit scoping may take some getting used to (simply because Python differs in this respect from many, though far from all, other object-oriented languages), but it results in clarity, simplicity, and the removal of potential ambiguities.

Bound method objects are first-class objects: you can use them wherever you can use a callable object. Since a bound method holds references to both the function it wraps and the `self` object on which it executes, it's a powerful and flexible alternative to a closure (covered in **"Nested functions and nested scopes"**). An instance object whose class supplies the special method `__call__` (covered in **Table 4-1**) offers another viable alternative. These constructs let you bundle some behavior (code) and some state (data) into a single callable object. Closures are simplest, but they are somewhat limited in their applicability. Here's the closure from the section on nested functions and nested scopes:

```
def make_adder_as_closure(augend):
    def add(addend, _augend=augend):
        return addend + _augend
    return add
```

Bound methods and callable instances are richer and more flexible than closures. Here's how to implement the same functionality with a bound method:

```
def make_adder_as_bound_method(augend):
    class Adder:
        def __init__(self, augend):
            self.augend = augend
        def add(self, addend):
            return addend+self.augend
    return Adder(augend).add
```

And here's how to implement it with a callable instance (an instance whose class supplies the special method `__call__`):

```
def make_adder_as_callable_instance(augend):
    class Adder:
        def __init__(self, augend):
            self.augend = augend
        def __call__(self, addend):
            return addend+self.augend
    return Adder(augend)
```

From the viewpoint of the code that calls the functions, all of these factory functions are interchangeable, since all of them return callable objects that are polymorphic (i.e., usable in the same ways). In terms of implementation, the closure is simplest; the object-oriented approaches—i.e., the bound method and the callable instance—use more flexible, general, and powerful mechanisms, but there is no need for that extra power in this simple example (since no other state is required beyond the `augend`, which is just as easily carried in the closure as in either of the object-oriented approaches).

Inheritance

When you use an attribute reference `C.`*`name`* on a class object `C`, and `'`*`name`*`'` is not a key in `C.__dict__`, the lookup implicitly proceeds on each class object that is in `C.__bases__` in a specific order (which for historical reasons is known as the *method resolution order*, or MRO, but in fact applies to all attributes, not just methods). `C`'s base classes may in turn have their own bases. The lookup checks direct and indirect ancestors, one by one, in MRO, stopping when `'name'` is found.

Method resolution order

The lookup of an attribute name in a class essentially occurs by visiting ancestor classes in left-to-right, depth-first order. However, in the presence of multiple inheritance (which makes the inheritance graph a general *directed acyclic graph*, or DAG, rather than specifically a tree), this simple approach might lead to some ancestor classes being visited twice. In such cases, the resolution order leaves in the lookup sequence only the *rightmost* occurrence of any given class.

Each class and built-in type has a special read-only class attribute called `__mro__`, which is the tuple of types used for method resolution, in order. You can reference `__mro__` only on classes, not on instances, and, since `__mro__` is a read-only attribute, you cannot rebind or unbind it. For a detailed and highly technical explanation of all aspects of Python's MRO, you may want to study Michele Simionato's essay **"The Python 2.3 Method Resolution Order"**[6] and Guido van Rossum's article on **"The History of Python"**. In particular, note that it *is* quite possible that Python cannot determine *any* unambiguous MRO for a certain class: in this case, Python raises a `TypeError` exception when it executes that **`class`** statement.

Overriding attributes

As we've just seen, the search for an attribute proceeds along the MRO (typically, up the inheritance tree) and stops as soon as the attribute is found. Descendant classes are always examined before their ancestors, so that when a subclass defines an attribute with the same name as one in a superclass, the search finds the definition in the subclass and stops there. This is known as the subclass *overriding* the definition in the superclass. Consider the following code:

```
class B:
    a = 23
    b = 45
    def f(self):
        print('method f in class B')
    def g(self):
        print('method g in class B')
class C(B):
    b = 67
    c = 89
    d = 123
    def g(self):
        print('method g in class C')
    def h(self):
        print('method h in class C')
```

Here, class `C` overrides attributes `b` and `g` of its superclass `B`. Note that, unlike in some other languages, in Python you may override data attributes just as easily as callable attributes (methods).

Delegating to superclass methods

When subclass *`C`* overrides a method *`f`* of its superclass *`B`*, the body of *`C.f`* often wants to delegate some part of its operation to the superclass's implementation of the method. This can sometimes be done using a function object, as follows:

```
class Base:
    def greet(self, name):
        print('Welcome', name)
class Sub(Base):
    def greet(self, name):
        print('Well Met and', end=' ')
        Base.greet(self, name)
x = Sub()
x.greet('Alex')
```

The delegation to the superclass, in the body of `Sub.greet`, uses a function object obtained by attribute reference `Base.greet` on the superclass, and therefore passes all arguments normally, including `self`. (If it seems a bit ugly explicitly using the base class, bear with us; you'll see a better way to do this shortly, in this very section). Delegating to a superclass implementation is a frequent use of such function objects.

One common use of delegation occurs with the special method `__init__`. When Python creates an instance, it does not automatically call the `__init__` methods of any base classes, unlike some other object-oriented languages. It is up to a subclass to initialize its superclasses, using delegation as necessary. For example:

```
class Base:
    def __init__(self):
        self.anattribute = 23
class Derived(Base):
    def __init__(self):
        Base.__init__(self)
        self.anotherattribute = 45
```

If the `__init__` method of class `Derived` didn't explicitly call that of class `Base`, instances of `Derived` would miss that portion of their initialization. Thus, such instances would violate the **Liskov substitution principle (LSP)**, since they'd lack the attribute `anattribute`. This issue does *not* arise if a subclass does not define `__init__`, since in that case it inherits it from the superclass. So, there is *never* any reason to code:

```
class Derived(Base):
    def __init__(self):
        Base.__init__(self)
```

NEVER CODE A METHOD THAT JUST DELEGATES TO THE SUPERCLASS

You should never define a semantically empty `__init__` (i.e., one that just delegates to the superclass). Instead, inherit `__init__` from the superclass. This advice applies to *all* methods, special or not, but for some reason the bad habit of coding such semantically empty methods seems to show up most often for `__init__`.

The preceding code illustrates the concept of delegation to an object's superclass, but it is actually a poor practice, in today's Python, to code these superclasses explicitly by name. If the base class is renamed, all the call sites to it must be updated. Or, worse, if refactoring the class hierarchy introduces a new layer between the `Derived` and `Base` class, the newly inserted class's method will be silently skipped.

The recommended approach is to call methods defined in a superclass using the `super` built-in type. To invoke methods up the inheritance chain, just call `super()`, without arguments:

```
class Derived(Base):
    def __init__(self):
        super().__init__()
        self.anotherattribute = 45
```

Cooperative superclass method calling

Explicitly calling the superclass's version of a method using the superclass's name is also quite problematic in cases of multiple inheritance with so-called "diamond-shaped" graphs. Consider the following code:

```
class A:
    def met(self):
        print('A.met')
class B(A):
    def met(self):
```

```python
        print('B.met')
        A.met(self)
class C(A):
    def met(self):
        print('C.met')
        A.met(self)
class D(B,C):
    def met(self):
        print('D.met')
        B.met(self)
        C.met(self)
```

When we call `D().met()`, `A.met` ends up being called twice. How can we ensure that each ancestor's implementation of the method is called once and only once? The solution is to use `super`:

```python
class A:
    def met(self):
        print('A.met')
class B(A):
    def met(self):
        print('B.met')
        super().met()
class C(A):
    def met(self):
        print('C.met')
        super().met()
class D(B,C):
    def met(self):
        print('D.met')
        super().met()
```

Now, `D().met()` results in exactly one call to each class's version of `met`. If you get into the good habit of always coding superclass calls with `super`, your classes will fit smoothly even in complicated inheritance structures—and there will be no ill effects if the inheritance structure instead turns out to be simple.

The only situation in which you may prefer to use the rougher approach of calling superclass methods through the explicit syntax is when various classes have different and incompatible signatures for the same method. This is an unpleasant situation in many respects; if you do have to deal with it, the explicit syntax may sometimes be the least of the evils. Proper use of multiple inheritance is seriously hampered; but then, even the most fundamental properties of OOP, such as polymorphism between base and subclass instances, are impaired when you give methods of the same name different signatures in a superclass and its subclass.

Dynamic class definition using the type built-in function

In addition to the `type(obj)` use, you can also call `type` with three arguments to define a new class:

```
NewClass = type(name, bases, class_attributes, **kwargs)
```

where *`name`* is the name of the new class (which should match the target variable), *`bases`* is a tuple of immediate superclasses, *`class_attributes`* is a `dict` of class-level methods and attributes to define in the new class, and `**`*`kwargs`* are optional named arguments to pass to the metaclass of one of the base classes.

For example, with a simple hierarchy of `Vehicle` classes (such as `LandVehicle`, `WaterVehicle`, `AirVehicle`, `SpaceVehicle`, etc.), you can dynamically create hybrid classes at runtime, such as:

```
AmphibiousVehicle = type('AmphibiousVehicle',
                         (LandVehicle, WaterVehicle), {})
```

This would be equivalent to defining a multiply inherited class:

```
class AmphibiousVehicle(LandVehicle, WaterVehicle): pass
```

When you call type to create classes at runtime, you do not need to manually define the combinatorial expansion of all combinations of `Vehicle` subclasses, and adding new subclasses does not require massive extension of defined mixed classes.[7] For more notes and examples, see the **online documentation**.

"Deleting" class attributes

Inheritance and overriding provide a simple and effective way to add or modify (override) class attributes (such as methods) noninvasively—i.e., without modifying the base class defining the attributes—by adding or overriding the attributes in subclasses. However, inheritance does not offer a way to delete (hide) base classes' attributes noninvasively. If the subclass simply fails to define (override) an attribute, Python finds the base class's definition. If you need to perform such deletion, possibilities include the following:

- Override the method and raise an exception in the method's body.
- Eschew inheritance, hold the attributes elsewhere than in the subclass's `__dict__`, and define `__getattr__` for selective delegation.
- Override `__getattribute__` to similar effect.

The last of these techniques is demonstrated in **"__getattribute__"**.

CONSIDER USING AGGREGATION INSTEAD OF INHERITANCE

An alternative to inheritance is to use *aggregation*: instead of inheriting from a base class, hold an instance of that base class as a private attribute. You then get complete control over the attribute's life cycle and public interface by providing public methods in the containing class that delegate to the contained attribute (i.e., by calling equivalent methods on the attribute). This way, the containing class has more control over the creation and deletion of the attribute; also, for any unwanted methods that the attribute's class provides, you simply don't write delegating methods in the containing class.

The Built-in object Type

The built-in `object` type is the ancestor of all built-in types and classes. The `object` type defines some special methods (documented in **"Special Methods"**) that implement the default semantics of objects:

`__new__`, `__init__`
: You can create a direct instance of `object` by calling `object()` without any arguments. The call uses `object.__new__` and `object.__init__` to make and return an instance object without attributes (and without even a `__dict__` in which to hold attributes). Such instance objects may be useful as "sentinels," guaranteed to compare unequal to any other distinct object.

`__delattr__`, `__getattr__`, `__getattribute__`, `__setattr__`
: By default, any object handles attribute references (as covered in **"Attribute Reference Basics"**) using these methods of `object`.

`__hash__`, `__repr__`, `__str__`
: Passing an object to `hash`, `repr`, or `str` calls the object's corresponding dunder method.

A subclass of `object` (i.e., any class) may—and often will!—override any of these methods and/or add others.

Class-Level Methods

Python supplies two built-in nonoverriding descriptor types, which give a class two distinct kinds of "class-level methods": *static methods* and *class methods*.

Static methods

A *static method* is a method that you can call on a class, or on any instance of the class, without the special behavior and constraints of ordinary methods regarding the first parameter. A static method may have any signature; it may have no parameters, and the first parameter, if any, plays no special role. You can think of a static method as an ordinary function that you're able to call normally, despite the fact that it happens to be bound to a class attribute.

While it is never *necessary* to define static methods (you can always choose to instead define a normal function, outside the class), some programmers consider them to be an elegant syntax alternative when a function's purpose is tightly bound to some specific class.

To build a static method, call the built-in type `staticmethod` and bind its result to a class attribute. Like all binding of class attributes, this is normally done in the body of the class, but you may also choose to perform it elsewhere. The only argument to `staticmethod` is the function to call when Python calls the static method. The following example shows one way to define and call a static method:

```
class AClass:
    def astatic():
        print('a static method')
    astatic = staticmethod(astatic)

an_instance = AClass()
print(AClass.astatic())             # prints: a static method
print(an_instance.astatic())        # prints: a static method
```

This example uses the same name for the function passed to `staticmethod` and for the attribute bound to `staticmethod`'s result. This naming convention is not mandatory, but it's a good idea, and we recommend you always use it. Python offers a special, simplified syntax to support this style, covered in **"Decorators"**.

Class methods

A *class method* is a method you can call on a class or on any instance of the class. Python binds the method's first parameter to the class on which you call the method, or the class of the instance on which you call the method; it does not bind it to the instance, as for normal bound methods. The first parameter of a class method is conventionally named `cls`.

As with static methods, while it is never *necessary* to define class methods (you can always choose to define a normal function, outside the class, that takes the class object as its first parameter), class methods are an elegant alternative to such functions (particularly since they can usefully be overridden in subclasses, when that is necessary).

To build a class method, call the built-in type `classmethod` and bind its result to a class attribute. Like all binding of class attributes, this is normally done in the body of the class, but you may choose to perform it elsewhere. The only argument to `classmethod` is the function to call when Python calls the class method. Here's one way you can define and call a class method:

```
class ABase:
    def aclassmet(cls):
        print('a class method for', cls.__name__)
    aclassmet = classmethod(aclassmet)
class ADeriv(ABase):
    pass

b_instance = ABase()
d_instance = ADeriv()
print(ABase.aclassmet())        # prints: a class method for ABase
print(b_instance.aclassmet())   # prints: a class method for ABase
print(ADeriv.aclassmet())       # prints: a class method for ADeriv
print(d_instance.aclassmet())   # prints: a class method for ADeriv
```

This example uses the same name for the function passed to `classmethod` and for the attribute bound to `classmethod`'s result. Again, this naming convention is not mandatory, but it's a good idea, and we recommend that you always use it. Python's simplified syntax to support this style is covered in **"Decorators"**.

Properties

Python supplies a built-in overriding descriptor type, usable to give a class's instances *properties*. A property is an instance attribute with special functionality. You reference, bind, or unbind the attribute with the normal syntax (e.g., `print(x.prop)`, `x.prop=23`, `del x.prop`). However, rather than following the usual semantics for attribute reference, binding, and unbinding, these accesses call on instance *x* the methods that you specify as arguments to the built-in type `property`. Here's one way to define a read-only property:

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
    area = property(area, doc='area of the rectangle')
```

Each instance *r* of class `Rectangle` has a synthetic read-only attribute *r*`.area`, which the method *r*`.area()` computes on the fly by multiplying the sides. The docstring `Rectangle.area.__doc__` is `'area of the rectangle'`. The *r*`.area` attribute is read-only (attempts to rebind or unbind it fail) because we specify only a `get` method in the call to `property`, and no `set` or `del` methods.

Properties perform tasks similar to those of the special methods `__getattr__`, `__setattr__`, and `__delattr__` (covered in **"General-Purpose Special Methods"**), but properties are faster and simpler. To build a property, call the built-in type `property` and bind its result to a class attribute. Like all binding of class attributes, this is normally done in the body of the class, but you may choose to do it elsewhere. Within the body of a class *C*, you can use the following syntax:

```
attrib = property(fget=None, fset=None, fdel=None, doc=None)
```

When *x* is an instance of *C* and you reference *x*`.`*attrib*, Python calls on *x* the method you passed as argument `fget` to the property constructor, without arguments. When you assign *x*`.`*attrib* `=` *value*, Python calls the method you passed as argument `fset`, with *value* as the only argument. When you execute **`del`** *x*`.`*attrib*, Python calls the method you passed as argument `fdel`, without arguments. Python uses the argument you passed as `doc` as the docstring of the attribute. All parameters to `property` are optional. When an argument is missing, Python raises an exception when some code attempts that operation. For example, in the `Rectangle` example, we made the property `area` read-only because we passed an argument only for the parameter `fget`, and not for the parameters `fset` and `fdel`.

An elegant syntax to create properties in a class is to use `property` as a *decorator* (see **"Decorators"**):

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    @property
    def area(self):
        """area of the rectangle"""
        return self.width * self.height
```

To use this syntax, you *must* give the getter method the same name as you want the property to have; the method's docstring becomes the docstring of the property. If you want to add a setter and/or a deleter as well, use decorators named (in this example) `area.setter` and `area.deleter`, and name the methods thus decorated the same as the property, too. For example:

```
import math
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    @property
    def area(self):
        """area of the rectangle"""
        return self.width * self.height
    @area.setter
```

```
    def area(self, value):
        scale = math.sqrt(value/self.area)
        self.width *= scale
        self.height *= scale
```

Why properties are important

The crucial importance of properties is that their existence makes it perfectly safe (and indeed advisable) for you to expose public data attributes as part of your class's public interface. Should it ever become necessary, in future versions of your class or other classes that need to be polymorphic to it, to have some code execute when the attribute is referenced, rebound, or unbound, you will be able to change the plain attribute into a property and get the desired effect without any impact on any code that uses your class (aka "client code"). This lets you avoid goofy idioms, such as *accessor* and *mutator* methods, required by OO languages lacking properties. For example, client code can use natural idioms like this:

```
some_instance.widget_count += 1
```

rather than being forced into contorted nests of accessors and mutators like this:

```
some_instance.set_widget_count(some_instance.get_widget_count() + 1)
```

If you're ever tempted to code methods whose natural names are something like `get_`*`this`* or `set_`*`that`*, wrap those methods into properties instead, for clarity.

Properties and inheritance

Inheritance of properties works just like for any other attribute. However, there's a little trap for the unwary: *the methods called upon to access a property are those defined in the class in which the property itself* *is defined*, without intrinsic use of further overriding that may happen in subclasses. Consider this example:

```
class B:
    def f(self):
        return 23
    g = property(f)
class C(B):
    def f(self):
        return 42

c = C()
print(c.g)                # prints: 23, not 42
```

Accessing the property `c.g` calls `B.f`, not `C.f`, as you might expect. The reason is quite simple: the property constructor receives (directly or via the decorator syntax) the *function object* `f` (and that happens at the time the **`class`** statement for `B` executes, so the function object in question is the one also known as `B.f`). The fact that the subclass `C` later redefines the name `f` is therefore irrelevant, since the property performs no lookup for that name, but rather uses the function object it received at creation time. If you need to work around this issue, you can do it by adding the extra level of lookup indirection yourself:

```
class B:
    def f(self):
        return 23
    def _f_getter(self):
        return self.f()
    g = property(_f_getter)
class C(B):
    def f(self):
        return 42

c = C()
print(c.g)                # prints: 42, as expected
```

Here, the function object held by the property is `B._f_getter`, which in turn does perform a lookup for the name `f` (since it calls `self.f()`); therefore, the overriding of `f` has the expected effect. As David Wheeler famously put it, "All problems in computer science can be solved by another level of indirection."[8]

__slots__

Normally, each instance object *x* of any class *C* has a dictionary *x*.`__dict__` that Python uses to let you bind arbitrary attributes on *x*. To save a little memory (at the cost of letting *x* have only a predefined set of attribute names), you can define in class *C* a class attribute named `__slots__`, a sequence (normally a tuple) of strings (normally identifiers). When class *C* has `__slots__`, instance *x* of class *C* has no `__dict__`: trying to bind on *x* an attribute whose name is not in *C*.`__slots__` raises an exception.

Using `__slots__` lets you reduce memory consumption for small instance objects that can do without the powerful and convenient ability to have arbitrarily named attributes. `__slots__` is worth adding only to classes that can have so many instances that saving a few tens of bytes per instance is important—typically classes that could have millions, not mere thousands, of instances alive at the same time. Unlike most other class attributes, however, `__slots__` works as we've just described only if an assignment in the class body binds it as a class attribute. Any later alteration, rebinding, or unbinding of `__slots__` has no effect, nor does inheriting `__slots__` from a base class. Here's how to add `__slots__` to the `Rectangle` class defined earlier to get smaller (though less flexible) instances:

```
class OptimizedRectangle(Rectangle):
    __slots__ = 'width', 'height'
```

There's no need to define a slot for the `area` property: `__slots__` does not constrain properties, only ordinary instance attributes, which would reside in the instance's `__dict__` if `__slots__` wasn't defined.

3.8+ `__slots__` attributes can also be defined using a `dict` with attribute names for the keys and docstrings for the values. `OptimizedRectangle` could be declared more fully as:

```
class OptimizedRectangle(Rectangle):
    __slots__ = {'width': 'rectangle width in pixels',
                 'height': 'rectangle height in pixels'}
```

__getattribute__

All references to instance attributes go through the special method `__getattribute__`. This method comes from `object`, where it implements attribute reference semantics (as documented in **"Attribute Reference Basics"**). You may override `__getattribute__` for purposes such as hiding inherited class attributes for a subclass's instances. For instance, the following example shows one way to implement a list without `append`:

```
class listNoAppend(list):
    def __getattribute__(self, name):
        if name == 'append':
            raise AttributeError(name)
        return list.__getattribute__(self, name)
```

An instance *x* of class `listNoAppend` is almost indistinguishable from a built-in list object, except that its runtime performance is substantially worse, and any reference to `x.append` raises an exception.

Implementing `__getattribute__` can be tricky; it is often easier to use the built-in functions `getattr` and `setattr` and the instance's `__dict__` (if any), or to reimplement `__getattr__` and `__setattr__`. Of course, in some cases (such as the preceding example), there is no alternative.

Per Instance Methods

An instance can have instance-specific bindings for all attributes, including callable attributes (methods). For a method, just like for any other attribute (except those bound to overriding descriptors), an instance-specific binding hides a class-level binding: attribute lookup does not consider the class when it finds a binding directly in the instance. An instance-specific binding for a callable attribute does not perform any of the transformations detailed in **"Bound and Unbound Methods"**: the attribute reference returns exactly the same callable object that was earlier bound directly to the instance attribute.

However, this does not work as you might expect for per instance bindings of the special methods that Python calls implicitly as a result of various operations, as covered in **"Special Methods"**. Such implicit uses of special methods always rely on the *class-level* binding of the special method, if any. For example:

```
def fake_get_item(idx):
    return idx
class MyClass:
    pass
n = MyClass()
n.__getitem__ = fake_get_item
print(n[23])                      # results in:
# Traceback (most recent call last):
#   File "<stdin>", line 1, in ?
# TypeError: unindexable object
```

Inheritance from Built-in Types

A class can inherit from a built-in type. However, a class may directly or indirectly extend multiple built-in types only if those types are specifically designed to allow this level of mutual compatibility. Python does not support unconstrained inheritance from multiple arbitrary built-in types. Normally, a new-style class only extends at most one substantial built-in type. For example, this:

```
class noway(dict, list):
    pass
```

raises a `TypeError` exception, with a detailed explanation of “multiple bases have instance lay-out conflict.” When you see such error messages, it means that you’re trying to inherit, directly or indirectly, from multiple built-in types that are not specifically designed to cooperate at such a deep level.

Special Methods

A class may define or inherit special methods, often referred to as “dunder” methods because, as described earlier, their names have leading and trailing double underscores. 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 discussion, *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*. 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 dunder 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 categories:

Initialization and finalization

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

String representation

A class can control how Python renders its instances as strings via special methods `__repr__`, `__str__`, `__format__`, and `__bytes__`.

Comparison, hashing, and use in a Boolean context

A class can control how its instances compare with other objects (via special methods `__lt__`, `__le__`, `__gt__`, `__ge__`, `__eq__`, and `__ne__`), how dictionaries use them as keys and sets use them as members (via `__hash__`), and whether they evaluate as truthy or falsy in Boolean contexts (via`__bool__`).

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

A class can make its instances callable, just like function objects, via special method `__call__`.

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

Table 4-1. General-purpose special methods

`__bool__`	`__bool__(self)` When evaluating *x* as true or false (see "Boolean Values")—for example, on a call to `bool(x)`—Python calls `x.__bool__()`, which should return **`True`** or **`False`**. When `__bool__` is not present, Python calls `__len__`, and takes *x* as falsy when `x.__len__()` returns `0` (to check that a container is nonempty, avoid coding **`if`** `len(container)>0:`; use **`if`** *`container`*: instead). When neither `__bool__` nor `__len__` is present, Python considers *x* truthy.
`__bytes__`	`__bytes__(self)` Calling `bytes(x)` calls `x.__bytes__()`, if present. If a class supplies both special methods

	`__bytes__` and `__str__`, they should return "equivalent" strings, respectively, of `bytes` and `str` type.
`__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 one. The first parameter, conventionally called `self`, refers to *x*: Python supplies it implicitly, just as in any other call to a bound method.
`__del__`	`__del__(self)` Just before *x* disappears via garbage collection, Python calls *x*`.__del__()` to let *x* finalize itself. If `__del__` is absent, Python does no special finalization on garbage-collecting *x* (this is the most common case: very few classes need to define `__del__`). Python ignores the return value of `__del__` and doesn't implicitly call `__del__` methods of class *C*'s superclasses. *C*`.__del__` must explicitly perform any needed finalization, including, if need be, by delegation. When class *C* has base classes to finalize, *C*`.__del__` must call `super().__del__()`. The `__del__` method has no specific connection with the **`del`** statement, 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__` don't participate in cyclic garbage collection, covered in **"Garbage Collection"**. Be careful to avoid reference loops involving such instances: define `__del__` only when there is no feasible alternative.
`__delattr__`	`__delattr__(self, name)` At every request to unbind attribute *x.y* (typically, **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__`. Absent `__delattr__`, Python turns **del** *x.y* into **del** *x*.`__dict__['`*y*`']`.
`__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. When *x*'s class has no `__dir__`, `dir(`*x*`)` performs introspection to return a sorted list of *x*'s attributes, striving to produce relevant, rather than complete, information.
`__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 8-7), to avoid boilerplate and any risk of logical contradictions in your comparisons.
`__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"). When `__format__` is inherited from `object`, it delegates to `__str__` and does not accept a nonempty format string.
`__getattr__`	`__getattr__(self, `*name*`)` When *x*`.`*y* can't be found by the usual steps (i.e., when an `AttributeError` would usually be raised), Python calls *x*`.__getattr__('`*y*`')`. Python does not call `__getattr__` for attributes found by normal means (as keys in *x*`.__dict__`, or via *x*`.__class__`). If you want Python to call `__getattr__` for *every* attribute, keep the attributes elsewhere (e.g., in another `dict` referenced by an attribute with a private name), or override `__getattribute__` instead. `__getattr__` should raise `AttributeError` if it can't 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 usual semantics of

attribute access (*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 semantics it wants to offer. The typical way to implement attribute access is by delegating (e.g., call `object.__getattribute__(self, ...)` as part of the operation of your override of `__getattribute__`).

OVERRIDING __GETATTRIBUTE__ SLOWS ATTRIBUTE ACCESS

When a class overrides `__getattribute__`, all attribute accesses on instances of the class become slow, as the overriding code executes on every 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)`. (There are few cases, among Python built-ins, where *x*==*y* can hold between objects of different types. The most important ones are equality between different number types: an `int` can equal a `bool`, a `float`, a `fractions.Fraction` instance, or a `decimal.Decimal` instance.)
`__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 *C* without arguments, `C()`, and *x* has no instance-specific attributes on creation. Python performs no implicit call to `__init__` methods of class *C*'s superclasses. `C.__init__` must explicitly perform any 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().__init__()`. `__init__`'s inheritance works just like for any other method or attribute: if *C* itself does not override `__init__`, it inherits it from the first superclass in its `__mro__` to override `__init__`, like every other attribute.

	`__init__` must return `None`; otherwise, calling the class raises `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__` (usually, it just inherits it from `object`), which can return any value *x*. In other words, `__new__` need not return a new instance of `C`, although it's expected to do so. If the value *x* that `__new__` returns is an instance of `C` or of any subclass of `C` (whether a new or a previously existing one), Python then calls `__init__` on *x* (with the same `[*args*...]` originally passed to `__new__`). **INITIALIZE IMMUTABLES IN __NEW__, ALL OTHERS IN __INIT__** You can perform most kinds of initialization of new instances in either `__init__` or `__new__`, so you may wonder where it's best to place them. Best practice is to put the initialization in `__init__` only, unless you have a specific reason to put it in `__new__`. (When a type is immutable, `__init__` cannot change its instances: in this case, `__new__` has to perform all initialization.)
`__repr__`	`__repr__(self)` Calling `repr(*x*)` (which 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*. When feasible, try to make `eval(repr(x))==x` (but, don't go crazy to achieve this goal!).
`__setattr__`	`__setattr__(self, name, value)` At any request to bind attribute `x.y` (usually, an assignment statement `x.y=value`, but also, e.g., `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__` (in this respect, `__setattr__` is closer to `__getattribute__`). To avoid recursion, when `x.__setattr__` binds *x*'s attributes, it must modify `x.__dict__` directly (e.g., via `x.__dict__[name]=value`); or better, `__setattr__` can delegate to the superclass (call `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` (however, `__setattr__` also works fine with `__slots__`).
`__str__`	`__str__(self)` Like print(*x*), `str(x)` calls `x.__str__()` to get an informal, concise string representation of *x*. If `__str__` is absent, Python calls `x.__repr__`. `__str__` should return a convenient human-readable string, even when that entails some approximation.

Special Methods for Containers

An instance can be a *container* (a sequence, mapping, or set—mutually exclusive concepts[9]). For maximum usefulness, containers should provide special methods `__getitem__`, `__contains__`, and `__iter__` (and, if mutable, also `__setitem__` and `__delitem__`), plus nonspecial methods discussed in the following sections. In many cases, you can obtain suitable implementations of the nonspecial methods by extending the appropriate abstract base class from the `collections.abc` module, 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`*.[10] 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 a `TypeError` exception. When *`key`* is a value of a valid type but out of range, indexing should raise an `IndexError` exception. 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"**; in addition, *mutable* sequences should 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 the `Sequence` or `MutableSequence` abstract base class does not suffice to fulfill all of these requirements; inheriting from `MutableSequence`, at most, 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 the `Sequence` or `MutableSequence` abstract base class 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 a `KeyError` exception, 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`, and `values`. A mutable mapping should also define the methods `clear`, `pop`, `popitem`, `setdefault`, and `update`. (Inheriting from the `Mapping` or `MutableMapping` abstract base class fulfills 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, although not necessarily for ordering comparisons).

Sets

Sets are a peculiar kind of container: they are neither sequences nor mappings and cannot be indexed, but they do have a length (number of elements) and are iterable. Sets also support many operators (`&`, `|`, `^`, and `-`, 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 the `Set` or `MutableSet` abstract base class fulfills 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 (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. For example, a common idiom in a sequence class's `__getitem__` special method to fully support slicing is:

```
def __getitem__(self, index):
    # Recursively special-case slicing
    if isinstance(index, slice):
        return self.__class__(self[x]
                              for x in range(*index.indices(len(self))))
    # Check index, and deal with a negative and/or out-of-bounds index
    index = operator.index(index)
    if index < 0:
        index += len(self)
    if not (0 <= index < len(self)):
        raise IndexError
```

```
    # Index is now a correct int, 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 and *x* is iterable, Python performs *y* **in** *x* as follows, taking time proportional to `len(x)`:
	<pre>for z in x:
	    if y==z:
	        return True
	return False</pre>

`__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__` if *x* is mutable and 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 an `IndexError` exception 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 `__bool__` is absent; in this case, a container is falsy 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__` if *x* is mutable, so items, and maybe slices, can be added or rebound.

Abstract Base Classes

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

One recommended approach to OO design (attributed to Arthur J. Riel) is to never extend a concrete class.[11] If two concrete classes have enough in common to tempt you to have one of them inherit from the other, proceed instead by making an *abstract* base class that subsumes all they 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.[12]

The abc module

The standard library module `abc` supplies metaclass `ABCMeta` and class `ABC` (subclassing `abc.ABC` makes `abc.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 single 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 `abc` module also supplies the decorator `abstractmethod` to designate methods that must be implemented in inheriting classes. You can define a property as abstract by using both the `property` and `abstractmethod` decorators, in that order.[13] Abstract methods and properties can have implementations (available to subclasses via the `super` built-in), but the point of making methods and properties abstract is that you can instantiate a 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, in `collections.abc`.[14] Some of these ABCs accept as a virtual subclass any class defining or inheriting a specific abstract method, as listed in Table 4-3.

Table 4-3. Single-method ABCs

ABC	Abstract methods
`Callable`	`__call__`
`Container`	`__contains__`
`Hashable`	`__hash__`
`Iterable`	`__iter__`

ABC	Abstract methods
`Sized`	`__len__`

The other ABCs in `collections.abc` extend one or more of these, adding more abstract methods and/or *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 also override some or all of the mixin methods, when that helps improve performance, but you don't have to—you can just inherit them, when this results in performance that's sufficient for your purposes.)

Table 4-4 details the ABCs in `collections.abc` that directly extend the preceding ones.

Table 4-4. ABCs with additional methods

ABC	Extends	Abstract methods	Mixin methods
`Iterator`	`Iterable`	`__next__`	`__iter__`

`Mapping`	`Container`
	`Iterable`
	`Sized`
		`__getitem__`
		`__iter__`
		`__len__`
			`__contains__`
			`__eq__`
			`__ne__`
			`getitems`
			`keys`
			`values`

`MappingView`	`Sized`		`__len__`

`Sequence`	`Container`
	`Iterable`
	`Sized`
		`__getitem__`
		`__len__`
			`__contains__`
			`__iter__`
			`__reversed__`
			`count`
			`index`

ABC	Extends	Abstract methods	Mixin methods

`Set`	`Container`
	`Iterable`
	`Sized`
		`__contains__`
		`__iter`
		`__len__`
			`__and__`[a]
			`__eq__`
			`__ge__`[b]
			`__gt__`
			`__le__`
			`__lt__`
			`__ne__`
			`__or__`
			`__sub__`
			`__xor__`
			`isdisjoint`

[a] For sets and mutable sets, many dunder methods are equivalent to nonspecial methods in the concrete class `set`; e.g., `__add__` is like `intersection` and `__iadd__` is like `intersection_update`.

[b] For sets, the ordering methods reflect the concept of *subset*: `s1 <= s2` means "`s1` is a subset of or equal to `s2`."

Table 4-5 details the ABCs in this module that further extend the previous ones.

Table 4-5. The remaining ABCs in `collections.abc`

ABC	Extends	Abstract methods	Mixin methods
`ItemsView`	`MappingView` `Set`		`__contains__` `__iter__`
`KeysView`	`MappingView` `Set`		`__contains__` `__iter__`

ABC	Extends	Abstract methods	Mixin methods
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.

ABCs in the numbers module

numbers supplies a hierarchy (also known as a *tower*) of ABCs representing various kinds of numbers. Table 4-6 lists the ABCs in the numbers module.

Table 4-6. ABCs supplied by the numbers module

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

a So, every int or float has a property real equal to its value, and a property imag equal to 0.

b So, every int has a property numerator equal to its value, and a property denominator equal to 1.

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 4-7** in order to overload operators such as + and *. In particular, sequences should have special methods `__add__`, `__mul__`, `__radd__`, and `__rmul__`, as mentioned in **"Sequences"**. When one of the binary methods (such as `__add__`, `__sub__`, etc.) is called with an operand of an unsupported type for that method, the method should return the built-in singleton `NotImplemented`.

Table 4-7. Special methods for numeric objects

`__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*, respectively, call these methods, usually for arithmetic computations.
`__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__`	`__complex__(self)`, `__float__(self)`, `__int__(self)` The built-in types `complex(`*x*`)`, `float(`*x*`)`, and `int(`*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*`)`.
`__floordiv__`, `__truediv__`	`__floordiv__(self, other)`, `__truediv__(self, other)` The operators *x* `//` *y* and *x* `/` *y*, respectively, call these methods, usually for arithmetic division.
`__iadd__`, `__ifloordiv__`, `__imod__`, `__imul__`, `__isub__`, `__itruediv__`, `__imatmul__`	`__iadd__(self, other)`, `__ifloordiv__(self, other)`, `__imod__(self, other)`, `__imul__(self, other)`, `__isub__(self, other)`, `__itruediv__(self, other)`, `__imatmul__(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`. Define these methods when *x* is mutable (i.e., when *x* *can* change in place).
`__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 the built-in types, only `int` supplies `__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`.
`__matmul__`	`__matmul__(self, other)` The operator *x* `@` *y* calls this method, usually for matrix multiplication.
`__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__`, `__rmod__`, `__rmul__`, `__rsub__`, `__rmatmul__`	`__radd__(self, other)`, `__rmod__(self, other)`, `__rmul__(self, other)`, `__rsub__(self, other)`, `__rmatmul__(self, other)` The operators *y* + *x*, *y* / *x*, *y* % *x*, *y* * *x*, *y* - *x*, and *y* @

<table>
<tr><td></td><td>x, respectively, call these methods on x when y doesn't have the needed method __add__, __truediv__, and so on, or when that method returns NotImplemented.</td></tr>
<tr><td>__rand__,
__rlshift__,
__ror__,
__rrshift__,
__rxor__</td><td>__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 the needed method __and__, __lshift__, and so on, or when that method returns NotImplemented.</td></tr>
<tr><td>__rdivmod__</td><td>__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).</td></tr>
<tr><td>__rpow__</td><td>__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.</td></tr>
</table>

Decorators

In Python, you often use *higher-order functions*: callables that accept a function as an argument and return a function as their result. For example, descriptor types such as `staticmethod` and `classmethod`, covered in **“Class-Level Methods”**, can be used, within class bodies, as follows:

```
def f(cls, ...):
    # ...definition of f snipped...
f = classmethod(f)
```

However, having the call to `classmethod` textually *after* the **`def`** statement hurts code readability: while reading *f*'s definition, the reader of the code is not yet aware that *f* is going to become a class method rather than an instance method. The code is more readable if the mention of `classmethod` comes *before* the `def`. For this purpose, use the syntax form known as *decoration*:

```
@classmethod
def f(cls, ...):
    # ...definition of f snipped...
```

The decorator, here `@classmethod`, must be immediately followed by a **`def`** statement and means that *f* = *`classmethod`*(*f*) executes right after the **`def`** statement (for whatever name *f* the **`def`** defines). More generally, *`@expression`* evaluates the expression (which must be a name, possibly qualified, or a call) and binds the result to an internal temporary name (say, *__aux*); any decorator must be immediately followed by a **`def`** (or **`class`**) statement, and means that *f* = *__aux*(*f*) executes right after the **`def`** or **`class`** statement (for whatever name *f* the **`def`** or **`class`** defines). The object bound to *__aux* is known as a *decorator*, and it's said to *decorate* function or class *f*.

Decorators are a handy shorthand for some higher-order functions. You can apply decorators to any **`def`** or **`class`** statement, not just in class bodies. You may code custom decorators, which are just higher-order functions accepting a function or class object as an argument and returning a function or class object as the result. For example, here is a simple example decorator that does not modify the function it decorates, but rather prints the function's docstring to standard output at function definition time:

```python
def showdoc(f):
    if f.__doc__:
        print(f'{f.__name__}: {f.__doc__}')
    else:
        print(f'{f.__name__}: No docstring!')
    return f

@showdoc
def f1():
    """a docstring"""  # prints: f1: a docstring

@showdoc
def f2():
    pass               # prints: f2: No docstring!
```

The standard library module `functools` offers a handy decorator, `wraps`, to enhance decorators built by the common "wrapping" idiom:

```python
import functools

def announce(f):
    @functools.wraps(f)
    def wrap(*a, **k):
        print(f'Calling {f.__name__}')
        return f(*a, **k)
    return wrap
```

Decorating a function *f* with `@announce` causes a line announcing the call to be printed before each call to *f*. Thanks to the `functools.wraps(`*f*`)` decorator, the wrapper adopts the name and docstring of the wrappee: this is useful, for example, when calling the built-in `help` on such a decorated function.

Metaclasses

Any object, even a class object, has a type. In Python, types and classes are also first-class objects. The type of a class object is also known as the class's *metaclass*.[15] An object's behavior is mostly determined by the type of the object. This also holds for classes: a class's behavior is mostly determined by the class's metaclass. Metaclasses are an advanced subject, and you may want to skip the rest of this section. However, fully grasping metaclasses can lead you to a deeper understanding of Python; very occasionally, it can be useful to define your own custom metaclasses.

Alternatives to Custom Metaclasses for Simple Class Customization

While a custom metaclass lets you tweak classes' behaviors in pretty much any way you want, it's often possible to achieve some customizations more simply than by coding a custom metaclass.

When a class `C` has or inherits a class method `__init_subclass__`, Python calls that method whenever you subclass `C`, passing the newly built subclass as the only positional argument. `__init_subclass__` can also have named parameters, in which case Python passes corresponding named arguments found in the class statement that performs the subclassing. As a purely illustrative example:

```
>>> class C:
...     def __init_subclass__(cls, foo=None, **kw):
...         print(cls, kw)
...         cls.say_foo = staticmethod(lambda: f'*{foo}*')
...         super().__init_subclass__(**kw)
...
>>> class D(C, foo='bar'):
...     pass
...
```

```
<class '__main__.D'> {}
```

```
>>> D.say_foo()
```

```
'*bar*'
```

The code in `__init_subclass__` can alter `cls` in any applicable, post-class-creation way; essentially, it works like a class decorator that Python automatically applies to any subclass of `C`.

Another special method used for customization is `__set_name__`, which lets you ensure that instances of descriptors added as class attributes know what class you're adding them to, and under which names. At the end of the **`class`** statement that adds *ca* to class *C* with name *n*, when the type of *ca* has the method `__set_name__`, Python calls `ca.__set_name__(C, n)`. For example:

```
>>> class Attrib:
...     def __set_name__(self, cls, name):
...         print(f'Attribute {name!r} added to {cls}')
...
>>> class AClass:
...     some_name = Attrib()
...
```

```
Attribute 'some_name' added to <class '__main__.AClass'>
```

```
>>>
```

How Python Determines a Class's Metaclass

The **class** statement accepts optional named arguments (after the bases, if any). The most important named argument is `metaclass`, which, if present, identifies the new class's metaclass. Other named arguments are allowed only if a non-`type` metaclass is present, in which case they are passed on to the optional `__prepare__` method of the metaclass (it's entirely up to the `__prepare__` method to make use of such named arguments).[16] When the named argument `metaclass` is absent, Python determines the metaclass by inheritance; for classes with no explicitly specified bases, the metaclass defaults to `type`.

Python calls the `__prepare__` method, if present, as soon as it determines the metaclass, as follows:

```
class M:
    def __prepare__(classname, *classbases, **kwargs):
        return {}
    # ...rest of M snipped...
class X(onebase, another, metaclass=M, foo='bar'):
    # ...body of X snipped...
```

Here, the call is equivalent to `M.__prepare__('X', onebase, another, foo='bar')`. `__prepare__`, if present, must return a mapping (usually just a dictionary), which Python uses as the *d* mapping in which it executes the class body. If `__prepare__` is absent, Python uses a new, initially empty `dict` as *d*.

How a Metaclass Creates a Class

Having determined the metaclass *M*, Python calls *M* with three arguments: the class name (a `str`), the tuple of base classes *t*, and the dictionary (or other mapping resulting from `__prepare__`) *d* in which the class body just finished executing.[17] The call returns the class object *C*, which Python then binds to the class name, completing the execution of the **class** statement. Note that this is in fact an instantiation of type *M*, so the call to *M* executes `M.__init__(C, namestring, t, d)`, where `C` is the return value of `M.__new__(M, namestring, t, d)`, just as in any other instantiation.

After Python creates the class object `C`, the relationship between class `C` and its type (`type(C)`, normally `M`) is the same as that between any object and its type. For example, when you call the class object `C` (to create an instance of `C`), `M.__call__` executes, with class object `C` as the first argument.

Note the benefit, in this context, of the approach described in **"Per Instance Methods"**, whereby special methods are looked up only on the class, not on the instance. Calling `C` to instantiate it must execute the metaclass's `M.__call__`, whether or not `C` has a per instance attribute (method) `__call__` (i.e., independently of whether *instances* of `C` are or aren't callable). This way, the Python object model avoids having to make the relationship between a class and its metaclass an ad hoc special case. Avoiding ad hoc special cases is a key to Python's power: Python has few, simple, general rules, and applies them consistently.

Defining and using your own metaclasses

It's easy to define custom metaclasses: inherit from `type` and override some of its methods. You can also perform most of the tasks for which you might consider creating a metaclass with `__new__`, `__init__`, `__getattribute__`, and so on, without involving metaclasses. However, a custom metaclass can be faster, since special processing is done only at class creation time, which is a rare operation. A custom metaclass lets you define a whole category of classes in a framework that magically acquire whatever interesting behavior you've coded, quite independently of what special methods the classes themselves may choose to define.

To alter a specific class in an explicit way, a good alternative is often to use a class decorator, as mentioned in **"Decorators"**. However, decorators are not inherited, so the decorator must be explicitly applied to each class of interest.[18] Metaclasses, on the other hand, *are* inherited; in fact, when you define a custom metaclass `M`, it's usual to also define an otherwise empty class *C* with metaclass *M*, so that other classes requiring *M* can just inherit from *C*.

Some behavior of class objects can be customized only in metaclasses. The following example shows how to use a metaclass to change the string format of class objects:

```
class MyMeta(type):
    def __str__(cls):
        return f'Beautiful class {cls.__name__!r}'
class MyClass(metaclass=MyMeta):
    pass
x = MyClass()
print(type(x))      # prints: Beautiful class 'MyClass'
```

A substantial custom metaclass example

Suppose that, programming in Python, we miss C's `struct` type: an object that is just a bunch of data attributes, in order, with fixed names (data classes, covered in the following section, fully address this requirement, which makes this example a purely illustrative one). Python lets us easily define a generic `Bunch` class that is similar, apart from the fixed order and names:

```
class Bunch:
    def __init__(self, **fields):
        self.__dict__ = fields
p = Bunch(x=2.3, y=4.5)
print(p)       # prints: <_main__.Bunch object at 0x00AE8B10>
```

A custom metaclass can exploit the fact that attribute names are fixed at class creation time. The code shown in **Example 4-1** defines a metaclass, `MetaBunch`, and a class, `Bunch`, to let us write code like:

```
class Point(Bunch):
```

```
    """A Point has x and y coordinates, defaulting to 0.0,
       and a color, defaulting to 'gray'-and nothing more,
       except what Python and the metaclass conspire to add,
       such as __init__ and __repr__.
    """
    x = 0.0
    y = 0.0
    color = 'gray'
# example uses of class Point
q = Point()
print(q)                    # prints: Point()
p = Point(x=1.2, y=3.4)
print(p)                    # prints: Point(x=1.2, y=3.4)
```

In this code, the `print` calls emit readable string representations of our `Point` instances. `Point` instances are quite memory lean, and their performance is basically the same as for instances of the simple class `Bunch` in the previous example (there is no extra overhead due to implicit calls to special methods). **Example 4-1** is quite substantial, and following all its details requires a grasp of aspects of Python discussed later in this book, such as strings (covered in **Chapter 9**) and module warnings (covered in **"The warnings Module"**). The identifier `mcl` used in **Example 4-1** stands for "metaclass," clearer in this special advanced case than the habitual case of `cls` standing for "class."

Example 4-1. The `MetaBunch` metaclass

```
import warnings
class MetaBunch(type):
    """
    Metaclass for new and improved "Bunch": implicitly defines
    __slots__, __init__, and __repr__ from variables bound in
    class scope.
    A class statement for an instance of MetaBunch (i.e., for a
    class whose metaclass is MetaBunch) must define only
    class-scope data attributes (and possibly special methods, but
    NOT __init__ and __repr__). MetaBunch removes the data
    attributes from class scope, snuggles them instead as items in
    a class-scope dict named __dflts__, and puts in the class a
```

```python
    __slots__ with those attributes' names, an __init__ that takes
    as optional named arguments each of them (using the values in
    __dflts__ as defaults for missing ones), and a __repr__ that
    shows the repr of each attribute that differs from its default
    value (the output of __repr__ can be passed to __eval__ to make
    an equal instance, as per usual convention in the matter, if
    each non-default-valued attribute respects that convention too).
    The order of data attributes remains the same as in the class body.
    """
    def __new__(mcl, classname, bases, classdict):
        """Everything needs to be done in __new__, since
           type.__new__ is where __slots__ are taken into account.
        """
        # Define as local functions the __init__ and __repr__ that
        # we'll use in the new class
        def __init__(self, **kw):
            """__init__ is simple: first, set attributes without
               explicit values to their defaults; then, set those
               explicitly passed in kw.
            """
            for k in self.__dflts__:
                if not k in kw:
                    setattr(self, k, self.__dflts__[k])
            for k in kw:
                setattr(self, k, kw[k])
        def __repr__(self):
            """__repr__ is minimal: shows only attributes that
               differ from default values, for compactness.
            """
            rep = [f'{k}={getattr(self, k)!r}'
                    for k in self.__dflts__
                    if getattr(self, k) != self.__dflts__[k]
                  ]
            return f'{classname}({', '.join(rep)})'
        # Build the newdict that we'll use as class dict for the
        # new class
        newdict = {'__slots__': [], '__dflts__': {},
                   '__init__': __init__, '__repr__' :__repr__,}
        for k in classdict:
            if k.startswith('__') and k.endswith('__'):
                # Dunder methods: copy to newdict, or warn
                # about conflicts
                if k in newdict:
```

```
                    warnings.warn(f'Cannot set attr {k!r}'
                                  f' in bunch-class {classname!r}')
                else:
                    newdict[k] = classdict[k]
            else:
                # Class variables: store name in __slots__, and
                # name and value as an item in __dflts__
                newdict['__slots__'].append(k)
                newdict['__dflts__'][k] = classdict[k]
        # Finally, delegate the rest of the work to type.__new__
        return super().__new__(mcl, classname, bases, newdict)

class Bunch(metaclass=MetaBunch):
    """For convenience: inheriting from Bunch can be used to get
       the new metaclass (same as defining metaclass= yourself).
    """
    pass
```

Data Classes

As the previous `Bunch` class exemplified, a class whose instances are just a bunch of named data items is a great convenience. Python's standard library covers that with the `dataclasses` module.

The main feature of the `dataclasses` module you'll be using is the `dataclass` function: a decorator you apply to any class whose instances you want to be just such a bunch of named data items. As a typical example, consider the following code:

```
import dataclasses
@dataclasses.dataclass
class Point:
    x: float
    y: float
```

Now you can call, say, `pt = Point(0.5, 0.5)` and get a variable with attributes `pt.x` and `pt.y`, each equal to `0.5`. By default, the `dataclass` decorator has imbued the class `Point` with an `__init__` method accepting initial floating-point values for attributes x and y, and a __repr__ method ready to appropriately display any instance of the class:

```
>>> pt
```

```
Point(x=0.5, y=0.5)
```

The dataclass function takes many optional named parameters to let you tweak details of the class it decorates. The parameters you may be explicitly using most often are listed in **Table 4-8**.

Table 4-8. Commonly used dataclass function parameters

Parameter name	Default value and resulting behavior
eq	**True** When **True**, generates an __eq__ method (unless the class defines one)
frozen	**False** When **True**, makes each instance of the class read-only (not allowing rebinding or deletion of attributes)
init	**True** When **True**, generates an __init__ method (unless the class defines one)
kw_only	**False** **3.10+** When **True**, forces arguments to __init__ to be named, not positional
order	**False** When **True**, generates order-comparison special

Parameter name	Default value and resulting behavior
	methods (`__le__`, `__lt__`, and so on) unless the class defines them
`repr`	**`True`** When **`True`**, generates a `__repr__` method (unless the class defines one)
`slots`	**`False`** **3.10+** When **`True`**, adds the appropriate `__slots__` attribute to the class (saving some amount of memory for each instance, but disallowing the addition of other, arbitrary attributes to class instances)

The decorator also adds to the class a `__hash__` method (allowing instances to be keys in a dictionary and members of a set) when that is safe (typically, when you set `frozen` to **`True`**). You may force the addition of `__hash__` even when that's not necessarily safe, but we earnestly recommend that you don't; if you insist, check the **online docs** for details on how to do so.

If you need to tweak each instance of a `dataclass` after the automatically generated `__init__` method has done the core work of assigning each instance attribute, define a method called `__post_init__`, and the decorator will ensure it is called right after `__init__` is done.

Say you wish to add an attribute to `Point` to capture the time when the point was created. This could be added as an attribute assigned in `__post_init__`. Add the attribute `create_time` to the members defined for `Point`, as type `float` with a default value of `0`, and then add an implementation for `__post_init__`:

```
def __post_init__(self):
    self.create_time = time.time()
```

Now if you create the variable `pt = Point(0.5, 0.5)`, printing it out will display the creation timestamp, similar to the following:

```
>>> pt
```

```
Point(x=0.5, y=0.5, create_time=1645122864.3553088)
```

Like regular classes, `dataclasses` can also support additional methods and properties, such as this method that computes the distance between two `Point`s and this property that returns the distance from a `Point` at the origin:

```
def distance_from(self, other):
    dx, dy = self.x - other.x, self.y - other.y
    return math.hypot(dx, dy)

@property
def distance_from_origin(self):
    return self.distance_from(Point(0, 0))
```

For example:

```
>>> pt.distance_from(Point(-1, -1))
```

```
2.1213203435596424
```

```
>>> pt.distance_from_origin
```

```
0.7071067811865476
```

The `dataclasses` module also supplies `asdict` and `astuple` functions, each taking a `dataclass` instance as the first argument and returning, respectively, a `dict` and a `tuple` with the class's fields. Furthermore, the module supplies a `field` function that you may use to customize the treatment of some of a `dataclass`'s fields (i.e., instance attributes), and several other specialized functions and classes needed only for very advanced, esoteric purposes; to learn all about them, check out the **online docs**.

Enumerated Types (Enums)

When programming, you'll often want to create a set of related values that catalog or *enumerate* the possible values for a particular property or program setting,[19] whatever they might be: terminal colors, logging levels, process states, playing card suits, clothing sizes, or just about anything else you can think of. An *enumerated type* (*enum*) is a type that defines a group of such values, with symbolic names that you can use as typed global constants. Python provides the `Enum` class and related subclasses in the `enum` module for defining enums.

Defining an enum gives your code a set of symbolic constants that represent the values in the enumeration. In the absence of enums, constants might be defined as `int`s, as in this code:

```
# colors
RED = 1
GREEN = 2
BLUE = 3

# sizes
XS = 1
```

```
S = 2
M = 3
L = 4
XL = 5
```

However, in this design, there is no mechanism to warn against nonsense expressions like `RED > XL` or `L * BLUE`, since they are all just `int`s. There is also no logical grouping of the colors or sizes.

Instead, you can use an `Enum` subclass to define these values:

```
from enum import Enum, auto

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

class Size(Enum):
    XS = auto()
    S = auto()
    M = auto()
    L = auto()
    XL = auto()
```

Now, code like `Color.RED > Size.S` stands out visually as incorrect, and at runtime raises a Python `TypeError`. Using `auto()` automatically assigns incrementing `int` values beginning with `1` (in most cases, the actual values assigned to enum members are not meaningful).

CALLING ENUM CREATES A CLASS, NOT AN INSTANCE

Surprisingly, when you call `enum.Enum()`, it doesn't return a newly built *instance*, but rather a newly built *subclass*. So, the preceding snippet is equivalent to:

```
from enum import Enum
Color = Enum('Color', ('RED', 'GREEN', 'BLUE'))
Size = Enum('Size', 'XS S M L XL')
```

When you *call* `Enum` (rather than explicitly subclassing it in a class statement), the first argument is the name of the subclass you're building; the second argument gives all the names of that subclass's members, either as a sequence of strings or as a single whitespace-separated (or comma-separated) string.

We recommend that you define `Enum` subclasses using class inheritance syntax, instead of this abbreviated form. The **`class`** form is more visually explicit, so it is easier to see if a member is missing, misspelled, or added later.

The values within an enum are called its *members*. It is conventional to use all uppercase characters to name enum members, treating them much as though they were manifest constants. Typical uses of the members of an enum are assignment and identity checking:

```
while process_state is ProcessState.RUNNING:
    # running process code goes here
    if processing_completed():
        process_state = ProcessState.IDLE
```

You can obtain all members of an `Enum` by iterating over the `Enum` class itself, or from the class's `__members__` attribute. `Enum` members are all global singletons, so comparison with **`is`** and **`is not`** is preferred over `==` or `!=`.

The `enum` module contains several classes[20] to support different forms of enums, listed in **Table 4-9**.

Table 4-9. enum classes

Class	Description
`Enum`	Basic enumeration class; member values can be any Python object, typically `int`s or `str`s, but do not support `int` or `str` methods. Useful for defining enumerated types whose members are an unordered group.
`Flag`	Used to define enums that you can combine with operators `|`, `&`, `^`, and `~`; member values must be defined as `int`s to support these bitwise operations (Python, however, assumes no ordering among them). `Flag` members with a `0` value are falsy; other members are truthy. Useful when you create or check values with bitwise operations (e.g., file permissions). To support bitwise operations, you generally use powers of 2 (1, 2, 4, 8, etc.) as member values.
`IntEnum`	Equivalent to **`class`** `IntEnum(`*`int, Enum`*`)`; member values are `int`s and support all `int` operations, including ordering. Useful when order among values is significant, such as when defining logging levels.
`IntFlag`	Equivalent to **`class`** `IntFlag(`*`int, Flag`*`)`; member values are `int`s (usually, powers of 2) supporting all `int` operations, including comparisons.
`StrEnum`	3.11+ Equivalent to **`class`** `StrEnum(`*`str, Enum`*`)`; member values are `str`s and support all `str` operations.

The `enum` module also defines some support functions, listed in **Table 4-10**.

Table 4-10. `enum` support functions

Support function	Description
`auto`	Autoincrements member values as you define them. Values typically start at `1` and increment by 1; for `Flag`, increments are in powers of 2.
`unique`	Class decorator to ensure that members' values differ from each other.

The following example shows how to define a `Flag` subclass to work with the file permissions in the `st_mode` attribute returned from calling `os.stat` or `Path.stat` (for a description of the `stat` functions, see Chapter 11):

```
import enum
import stat

class Permission(enum.Flag):
    EXEC_OTH = stat.S_IXOTH
    WRITE_OTH = stat.S_IWOTH
    READ_OTH = stat.S_IROTH
    EXEC_GRP = stat.S_IXGRP
    WRITE_GRP = stat.S_IWGRP
    READ_GRP = stat.S_IRGRP
    EXEC_USR = stat.S_IXUSR
    WRITE_USR = stat.S_IWUSR
    READ_USR = stat.S_IRUSR

    @classmethod
    def from_stat(cls, stat_result):
        return cls(stat_result.st_mode & 0o777)

from pathlib import Path

cur_dir = Path.cwd()
dir_perm = Permission.from_stat(cur_dir.stat())
```

```
    if dir_perm & Permission.READ_OTH:
        print(f'{cur_dir} is readable by users outside the owner group')

    # the following raises TypeError: Flag enums do not support order
    # comparisons
    print(Permission.READ_USR > Permission.READ_OTH)
```

Using enums in place of arbitrary `int`s or `str`s can add readability and type integrity to your code. You can find more details on the classes and methods of the `enum` module in the Python **docs**.

1 Or “drawbacks,” according to one reviewer. One developer’s meat is another developer’s poison.

2 When that’s the case, it’s also OK to have other named arguments after `metaclass=`. Such arguments, if any, are passed on to the metaclass.

3 That need arises because `__init__`, on any subclass of `Singleton` that defines this special method, repeatedly executes, each time you instantiate the subclass, on the only instance that exists for each subclass of `Singleton`.

4 Except for instances of a class defining `__slots__`, covered in **“__slots__”**.

5 Some other OO languages, like **Modula-3**, similarly require explicit use of `self`.

6 Many Python releases later, Michele’s essay still applies!

7 One of the authors has used this technique to dynamically combine small mixin test classes to create complex test case classes to test multiple independent product features.

8 To complete the usually truncated famous quote: “except of course for the problem of too many indirections.”

9 Third-party extensions can also define types of containers that are not sequences, not mappings, and not sets.

10 Lower bound included, upper bound excluded—as always, the norm for Python.

11 See, for example, **"Avoid Extending Classes"** by Bill Harlan.

12 For a related concept focused on type checking, see `typing.Protocols`, covered in **"Protocols"**.

13 The `abc` module does include the `abstractproperty` decorator, which combines these two, but `abstractproperty` is deprecated, and new code should use the two decorators as described.

14 For backward compatibility these ABCs were also accessible in the `collections` module until Python 3.9, but the compatibility imports were removed in Python 3.10. New code should import these ABCs from `collections.abc`.

15 Strictly speaking, the type of a class `C` could be said to be the metaclass only of *instances* of `C` rather than of `C` itself, but this subtle semantic distinction is rarely, if ever, observed in practice.

16 Or when a base class has `__init_subclass__`, in which case the named arguments are passed to that method, as covered in **"Alternatives to Custom Metaclasses for Simple Class Customization"**.

17 This is similar to calling `type` with three arguments, as described in **"Dynamic class definition using the type built-in function"**.

18 `__init_subclass__`, covered in **"Alternatives to Custom Metaclasses for Simple Class Customization"**, works much like an "inherited decorator," so it's often an alternative to a custom metaclass.

19 Don't confuse this concept with the unrelated `enumerate` built-in function, covered in **Chapter 8**, which generates `(number, item)` pairs from an iterable.

20 `enum`'s specialized metaclass behaves so differently from the usual `type` metaclass that it's worth pointing out all the differences between `enum.Enum` and ordinary classes. You can read about this in the **"How are Enums different?" section** of Python's online documentation.