6

- The Object-Oriented Approach
- Custom Classes
- Custom Collection Classes

Object-Oriented Programming

In all the previous chapters we used objects extensively, but our style of programming has been strictly procedural. Python is a multiparadigm language—it allows us to program in procedural, object-oriented, and functional style, or in any mixture of styles, since it does not force us to program in any one particular way.

It is perfectly possible to write any program in procedural style, and for very small programs (up to, say, 500 lines), doing so is rarely a problem. But for most programs, and especially for medium-size and large programs, object-oriented programming offers many advantages.

This chapter covers all the fundamental concepts and techniques for doing object-oriented programming in Python. The first section is especially for those who are less experienced and for those coming from a procedural programming background (such as C or Fortran). The section starts by looking at some of the problems that can arise with procedural programming that object-oriented programming can solve. Then it briefly describes Python's approach to object-oriented programming and explains the relevant terminology. After that, the chapter's two main sections begin.

The second section covers the creation of custom data types that hold single items (although the items themselves may have many attributes), and the third section covers the creation of custom collection data types that can hold any number of objects of any types. These sections cover most aspects of object-oriented programming in Python, although we defer some more advanced material to Chapter 8.

The Object-Oriented Approach

In this section we will look at some of the problems of a purely procedural approach by considering a situation where we need to represent circles, potentially lots of them. The minimum data required to represent a circle is its (x, y) position and its radius. One simple approach is to use a 3-tuple for each circle. For example:

```
circle = (11, 60, 8)
```

One drawback of this approach is that it isn't obvious what each element of the tuple represents. We could mean `(x, y, radius)` or, just as easily, `(radius, x, y)`. Another drawback is that we can access the elements by index position only. If we have two functions, `distance_from_origin(x, y)` and `edge_distance_from_origin(x, y, radius)`, we would need to use tuple unpacking to call them with a circle tuple:

```
distance = distance_from_origin(*circle[:2])
distance = edge_distance_from_origin(*circle)
```

Both of these assume that the circle tuples are of the form `(x, y, radius)`. We can solve the problem of knowing the element order and of using tuple unpacking by using a named tuple:

```
import collections
Circle = collections.namedtuple("Circle", "x y radius")
circle = Circle(13, 84, 9)
distance = distance_from_origin(circle.x, circle.y)
```

This allows us to create `Circle` 3-tuples with named attributes which makes function calls much easier to understand, since to access elements we can use their names. Unfortunately, problems remain. For example, there is nothing to stop an invalid circle from being created:

```
circle = Circle(33, 56, -5)
```

It doesn't make sense to have a circle with a negative radius, but the `circle` named tuple is created here without raising an exception—just as it would be if the radius was given as a variable that held a negative number. The error will be noticed only if we call the `edge_distance_from_origin()` function—and then only if that function actually checks for a negative radius. This inability to validate when creating an object is probably the worst aspect of taking a purely procedural approach.

If we want circles to be mutable so that we can move them by changing their coordinates or resize them by changing their radius, we can do so by using the private `collections.namedtuple._replace()` method:

```
circle = circle._replace(radius=12)
```

Just as when we create a `Circle`, there is nothing to stop us from (or warn us about) setting invalid data.

If the circles were going to need lots of changes, we might opt to use a mutable data type such as a list, for the sake of convenience:

```
circle = [36, 77, 8]
```

This doesn't give us any protection from putting in invalid data, and the best we can do about accessing elements by name is to create some constants so that we can write things like `circle[RADIUS] = 5`. But using a list brings additional problems—for example, we can legitimately call `circle.sort()`! Using a dictionary might be an alternative, for example, `circle = dict(x=36, y=77, radius=8)`, but again there is no way to ensure a valid radius and no way to prevent inappropriate methods from being called.

Object-Oriented Concepts and Terminology

What we need is some way to package up the data that is needed to represent a circle, and some way to restrict the methods that can be applied to the data so that only valid operations are possible. Both of these things can be achieved by creating a custom `Circle` data type. We will see how to create a `Circle` data type in later in this section, but first we need to cover some preliminaries and explain some terminology. Don't worry if the terminology is unfamiliar at first; it will become much clearer once we reach the examples.

We use the terms *class*, *type*, and *data type* interchangeably. In Python we can create custom classes that are fully integrated and that can be used just like the built-in data types. We have already encountered many classes, for example, `dict`, `int`, and `str`. We use the term *object*, and occasionally the term *instance*, to refer to an instance of a particular class. For example, `5` is an `int` object and `"oblong"` is a `str` object.

Most classes encapsulate both data and the methods that can be applied to that data. For example, the `str` class holds a string of Unicode characters as its data and supports methods such as `str.upper()`. Many classes also support additional features; for example, we can concatenate two strings (or any two sequences) using the `+` operator and find a sequence's length using the built-in `len()` function. Such features are provided by *special methods*—these are like normal methods except that their names always begin and end with two underscores, and are predefined. For example, if we want to create a class that supports concatenation using the `+` operator and also the `len()` function, we can do so by implementing the `__add__()` and `__len__()` special methods in our class. Conversely, we should never define any method with a name that begins and ends with two underscores unless it is one of the predefined special methods and is

appropriate to our class. This will ensure that we never get conflicts with later versions of Python even if they introduce new predefined special methods.

Objects usually have attributes—methods are callable attributes, and other attributes are data. For example, a `complex` object has `imag` and `real` attributes and lots of methods, including special methods like `__add__()` and `__sub__` (to support the binary + and – operators), and normal methods like `conjugate()`. Data attributes (often referred to simply as "attributes") are normally implemented as *instance variables*, that is, variables that are unique to a particular object. We will see examples of this, and also examples of how to provide data attributes as *properties*. A property is an item of object data that is accessed like an instance variable but where the accesses are handled by methods behind the scenes. As we will see, using properties makes it easy to do data validation.

Inside a method (which is just a function whose first argument is the instance on which it is called to operate), several kinds of variables are potentially accessible. The object's instance variables can be accessed by qualifying their name with the instance itself. Local variables can be created inside the method; these are accessed without qualification. Class variables (sometimes called static variables) can be accessed by qualifying their name with the class name, and global variables, that is, module variables, are accessed without qualification.

Some of the Python literature uses the concept of a *namespace*, a mapping from names to objects. Modules are namespaces—for example, after the statement `import math` we can access objects in the `math` module by qualifying them with their namespace name (e.g., `math.pi` and `math.sin()`). Similarly, classes and objects are also namespaces; for example, if we have `z = complex(1, 2)`, the `z` object's namespace has two attributes which we can access (`z.real` and `z.imag`).

One of the advantages of object orientation is that if we have a class, we can *specialize* it. This means that we make a new class that inherits all the attributes (data and methods) from the original class, usually so that we can add or replace methods or add more instance variables. We can *subclass* (another term for *specialize*), any Python class, whether built-in or from the standard library, or one of our own custom classes.* The ability to subclass is one of the great advantages offered by object-oriented programming since it makes it straightforward to use an existing class that has tried and tested functionality as the basis for a new class that extends the original, adding new data attributes or new functionality in a very clean and direct way. Furthermore, we can pass objects of our new class to functions and methods that were written for the original class and they will work correctly.

We use the term *base class* to refer to a class that is inherited; a base class may be the immediate ancestor, or may be further up the inheritance tree. Another term for base class is *super class*. We use the term *subclass*, *derived*

*Some library classes that are implemented in C cannot be subclassed; such classes specify this in their documentation.

class, or *derived* to describe a class that inherits from (i.e., specializes) another class. In Python every built-in and library class and every class we create is derived directly or indirectly from the ultimate base class—`object`. Figure 6.1 illustrates some of the inheritance terminology.

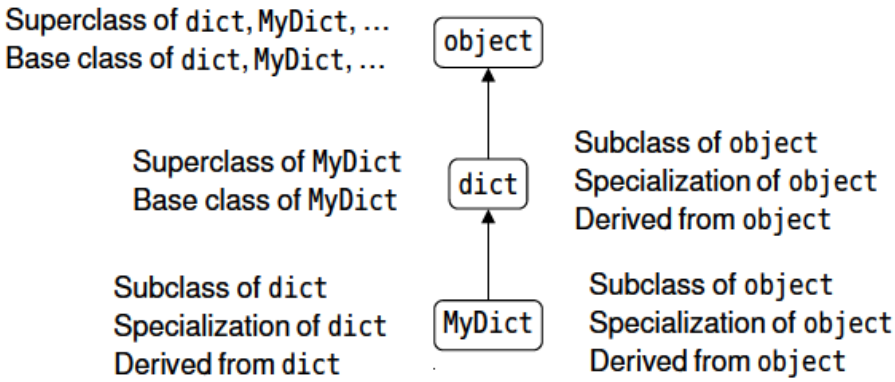

Figure 6.1 *Some object-oriented inheritance terminology*

Any method can be overridden, that is, reimplemented, in a subclass; this is the same as Java (apart from Java's "final" methods).* If we have an object of class `MyDict` (a class that inherits `dict`) and we call a method that is defined by both `dict` and `MyDict`, Python will correctly call the `MyDict` version—this is known as *dynamic method binding*, also called *polymorphism*. If we need to call the base class version of a method inside a reimplemented method we can do so by using the built-in `super()` function.

Python also supports *duck typing*—"if it walks like a duck and quacks like a duck, it is a duck". In other words, if we want to call certain methods on an object, it doesn't matter what class the object is, only that it has the methods we want to call. In the preceding chapter we saw that when we needed a file object we could provide one by calling the built-in `open()` function—or by creating an `io.StringIO` object and providing that instead, since `io.StringIO` objects have the same API (Application Programming Interface), that is, the same methods, as the file objects returned by `open()` in text mode.

Inheritance is used to model is-a relationships, that is, where a class's objects are essentially the same as some other class's objects, but with some variations, such as extra data attributes and extra methods. Another approach is to use *aggregation* (also called *composition*)—this is where a class includes one or more instance variables that are of other classes. Aggregation is used to model has-a relationships. In Python, every class uses inheritance—because all custom classes have `object` as their ultimate base class, and most classes also use aggregation since most classes have instance variables of various types.

*In C++ terminology, all Python methods are virtual.

Some object-oriented languages have two features that Python does not provide. The first is overloading, that is, having methods with the same name but with different parameter lists in the same class. Thanks to Python's versatile argument-handling capabilities this is never a limitation in practice. The second is access control—there are no bulletproof mechanisms for enforcing data privacy. However, if we create attributes (instance variables or methods) that begin with two leading underscores, Python will prevent unintentional accesses so that they can be considered to be private. (This is done by name mangling; we will see an example in Chapter 8.)

Just as we use an uppercase letter as the first letter of custom modules, we will do the same thing for custom classes. We can define as many classes as we like, either directly in a program or in modules—class names don't have to match module names, and modules may contain as many class definitions as we like.

Now that we have seen some of the problems that classes can solve, introduced the necessary terminology, and covered some background matters, we can begin to create some custom classes.

Custom Classes

In earlier chapters we created custom classes: custom exceptions. Here are two new syntaxes for creating custom classes:

```
class className:
    suite

class className(base_classes):
    suite
```

Since the exception subclasses we created did not add any new attributes (no instance data or methods) we used a suite of `pass` (i.e., nothing added), and since the suite was just one statement we put it on the same line as the `class` statement itself. Note that just like `def` statements, `class` is a statement, so we can create classes dynamically if we want to. A class's methods are created using `def` statements in the class's suite. Class instances are created by calling the class with any necessary arguments; for example, `x = complex(4, 8)` creates a complex number and sets `x` to be an object reference to it.

Attributes and Methods

Let's start with a very simple class, `Point`, that holds an (x, y) coordinate. The class is in file `Shape.py`, and its complete implementation (excluding docstrings) is show here:

```
class Point:
```

```
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return math.hypot(self.x, self.y)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __repr__(self):
        return "Point({0.x!r}, {0.y!r})".format(self)

    def __str__(self):
        return "({0.x!r}, {0.y!r})".format(self)
```

Since no base classes are specified, `Point` is a direct subclass of `object`, just as though we had written `class Point(object)`. Before we discuss each of the methods, let's see some examples of their use:

```
import Shape
a = Shape.Point()
repr(a)                     # returns: 'Point(0, 0)'
b = Shape.Point(3, 4)
str(b)                      # returns: '(3, 4)'
b.distance_from_origin()    # returns: 5.0
b.x = -19
str(b)                      # returns: '(-19, 4)'
a == b, a != b              # returns: (False, True)
```

The `Point` class has two data attributes, `self.x` and `self.y`, and five methods (not counting inherited methods), four of which are special methods; they are illustrated in Figure 6.2. Once the `Shape` module is imported, the `Point` class can be used like any other. The data attributes can be accessed directly (e.g., `y = a.y`), and the class integrates nicely with all of Python's other classes by providing support for the equality operator (`==`) and for producing strings in representational and string forms. And Python is smart enough to supply the inequality operator (`!=`) based on the equality operator. (It is also possible to specify each operator individually if we want total control, for example, if they are not exact opposites of each other.)

Python automatically supplies the first argument in method calls—it is an object reference to the object itself (called *this* in C++ and Java). We must include this argument in the parameter list, and by convention the parameter is called `self`. All object attributes (data and method attributes) must be qualified by `self`. This requires a little bit more typing compared with some other languages, but has the advantage of providing absolute clarity: we always know that we are accessing an object attribute if we qualify with `self`.

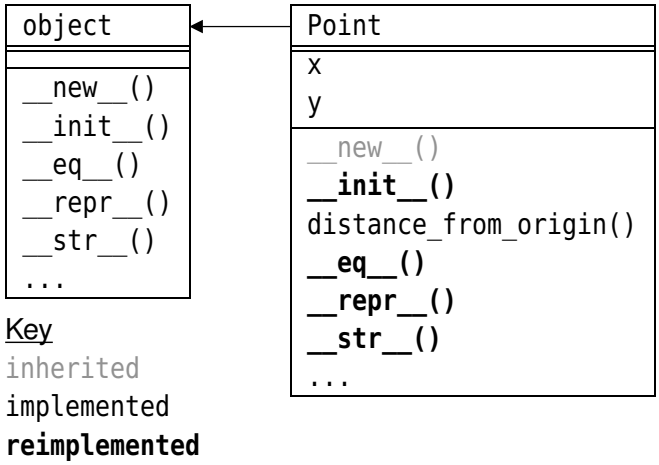

Figure 6.2 *The Point class's inheritance hierarchy*

To create an object, two steps are necessary. First a raw or uninitialized object must be created, and then the object must be initialized, ready for use. Some object-oriented languages (such as C++ and Java) combine these two steps into one, but Python keeps them separate. When an object is created (e.g., `p = Shape.Point()`), first the special method `__new__()` is called to create the object, and then the special method `__init__()` is called to initialize it.

In practice almost every Python class we create will require us to reimplement only the `__init__()` method, since the `object.__new__()` method is almost always sufficient and is automatically called if we don't provide our own `__new__()` method. (Later in this chapter we will show a rare example where we do need to reimplement `__new__()`.) Not having to reimplement methods in a subclass is another benefit of object-oriented programming—if the base class method is sufficient we don't have to reimplement it in our subclass. This works because if we call a method on an object and the object's class does not have an implementation of that method, Python will automatically go through the object's base classes, and their base classes, and so on, until it finds the method—and if the method is not found an `AttributeError` exception is raised.

Alternative FuzzyBool ➤ 256

For example, if we execute `p = Shape.Point()`, Python begins by looking for the method `Point.__new__()`. Since we have not reimplemented this method, Python looks for the method in `Point`'s base classes. In this case there is only one base class, `object`, and this has the required method, so Python calls `object.__new__()` and creates a raw uninitialized object. Then Python looks for the initializer, `__init__()`, and since we have reimplemented it, Python doesn't need to look further and calls `Point.__init__()`. Finally, Python sets `p` to be an object reference to the newly created and initialized object of type `Point`.

Because they are so short and a few pages away, for convenience we will show each method again before discussing it.

```
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

The two instance variables, `self.x` and `self.y`, are created in the initializer, and assigned the values of the `x` and `y` parameters. Since Python will find this initializer when we create a new `Point` object, the `object.__init__()` method will not be called. This is because as soon as Python has found the required method it calls it and doesn't look further.

Object-oriented purists might start the method off with a call to the base class `__init__()` method by calling `super().__init__()`. The effect of calling the `super()` function like this is to call the base class's `__init__()` method. For classes that directly inherit `object` there is no need to do this, and in this book we call base class methods only when necessary—for example, when creating classes that are designed to be subclassed, or when creating classes that don't directly inherit `object`. This is to some extent a matter of coding style—it is perfectly reasonable to always call `super().__init__()` at the start of a custom class's `__init__()` method.

```
    def distance_from_origin(self):
        return math.hypot(self.x, self.y)
```

This is a conventional method that performs a computation based on the object's instance variables. It is quite common for methods to be fairly short and to have only the object they are called on as an argument, since often all the data the method needs is available inside the object.

```
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
```

Methods should not have names that begin and end with two underscores—unless they are one of the predefined special methods. Python provides special methods for all the comparison operators as shown in Table 6.1.

All instances of custom classes support `==` by default, and the comparison returns `False`—unless we compare a custom object with itself. We can override this behavior by reimplementing the `__eq__()` special method as we have done here. Python will supply the `__ne__()` (not equal) inequality operator (`!=`) automatically if we implement `__eq__()` but don't implement `__ne__()`.

By default, all instances of custom classes are hashable, so `hash()` can be called on them and they can be used as dictionary keys and stored in sets. But if we reimplement `__eq__()`, instances are no longer hashable. We will see how to fix this when we discuss the `FuzzyBool` class later on.

Fuzzy-Bool ➤ 254

By implementing this special method we can compare `Point` objects, but if we were to try to compare a `Point` with an object of a different type—say, `int`—we

Table 6.1 *Comparison Special Methods*

Special Method	Usage	Description
`__lt__(self, other)`	`x < y`	Returns `True` if `x` is less than `y`
`__le__(self, other)`	`x <= y`	Returns `True` if `x` is less than or equal to `y`
`__eq__(self, other)`	`x == y`	Returns `True` if `x` is equal to `y`
`__ne__(self, other)`	`x != y`	Returns `True` if `x` is not equal to `y`
`__ge__(self, other)`	`x >= y`	Returns `True` if `x` is greater than or equal to `y`
`__gt__(self, other)`	`x > y`	Returns `True` if `x` is greater than `y`

would get an `AttributeError` exception (since `int`s don't have an `x` attribute). On the other hand, we *can* compare `Point` objects with other objects that coincidentally just happen to have an `x` attribute (thanks to Python's duck typing), but this may lead to surprising results.

If we want to avoid inappropriate comparisons there are a few approaches we can take. One is to use an assertion, for example, `assert isinstance(other, Point)`. Another is to raise a `TypeError` to indicate that comparisons between the two types are not supported, for example, `if not isinstance(other, Point): raise TypeError()`. The third way (which is also the most Pythonically correct) is to do this: `if not isinstance(other, Point): return NotImplemented`. In this third case, if `NotImplemented` is returned, Python will then try calling `other.__eq__(self)` to see whether the `other` type supports the comparison with the `Point` type, and if there is no such method or if that method also returns `NotImplemented`, Python will give up and raise a `TypeError` exception. (Note that only reimplementations of the comparison special methods listed in Table 6.1 may return `NotImplemented`.)

The built-in `isinstance()` function takes an object and a class (or a tuple of classes), and returns `True` if the object is of the given class (or of one of the tuple of classes), or of one of the class's (or one of the tuple of classes') base classes.

```
def __repr__(self):
    return "Point({0.x!r}, {0.y!r})".format(self)
```

str.format() 78 ◀

The built-in `repr()` function calls the `__repr__()` special method for the object it is given and returns the result. The string returned is one of two kinds. One kind is where the string returned can be evaluated using the built-in `eval()` function to produce an object equivalent to the one `repr()` was called on. The other kind is used where this is not possible; we will see an example later on. Here is how we can go from a `Point` object to a string and back to a `Point` object:

```
p = Shape.Point(3, 9)
repr(p)                                  # returns: 'Point(3, 9)'
q = eval(p.__module__ + "." + repr(p))
repr(q)                                  # returns: 'Point(3, 9)'
```

import 195 ➤

We must give the module name when `eval()`-ing if we used `import Shape`. (This would not be necessary if we had done the import differently, for example, `from Shape import Point`.) Python provides every object with a few private attributes, one of which is `__module__`, a string that holds the object's module name, which in this example is `"Shape"`.

Dynamic code execution ➤ 344

At the end of this snippet we have two `Point` objects, `p` and `q`, both with the same attribute values, so they compare as equal. The `eval()` function returns the result of executing the string it is given—which must contain a valid Python statement.

```
def __str__(self):
    return "({0.x!r}, {0.y!r})".format(self)
```

The built-in `str()` function works like the `repr()` function, except that it calls the object's `__str__()` special method. The result is intended to be understandable to human readers and is not expected to be suitable for passing to the `eval()` function. Continuing the previous example, `str(p)` (or `str(q)`) would return the string `'(3, 9)'`.

We have now covered the simple `Point` class—and also covered a lot of behind-the-scenes details that are important to know but which can mostly be left in the background. The `Point` class holds an (x, y) coordinate—a fundamental part of what we need to represent a circle, as we discussed at the beginning of the chapter. In the next subsection we will see how to create a custom `Circle` class, inheriting from `Point` so that we don't have to duplicate the code for the `x` and `y` attributes or for the `distance_from_origin()` method.

Inheritance and Polymorphism

The `Circle` class builds on the `Point` class using inheritance. The `Circle` class adds one additional data attribute (`radius`), and three new methods. It also reimplements a few of `Point`'s methods. Here is the complete class definition:

```
class Circle(Point):

    def __init__(self, radius, x=0, y=0):
        super().__init__(x, y)
        self.radius = radius

    def edge_distance_from_origin(self):
        return abs(self.distance_from_origin() - self.radius)
```

```
    def area(self):
        return math.pi * (self.radius ** 2)

    def circumference(self):
        return 2 * math.pi * self.radius

    def __eq__(self, other):
        return self.radius == other.radius and super().__eq__(other)

    def __repr__(self):
        return "Circle({0.radius!r}, {0.x!r}, {0.y!r})".format(self)

    def __str__(self):
        return repr(self)
```

Inheritance is achieved simply by listing the class (or classes) that we want our class to inherit in the `class` line.* Here we have inherited the `Point` class—the inheritance hierarchy for `Circle` is shown in Figure 6.3.

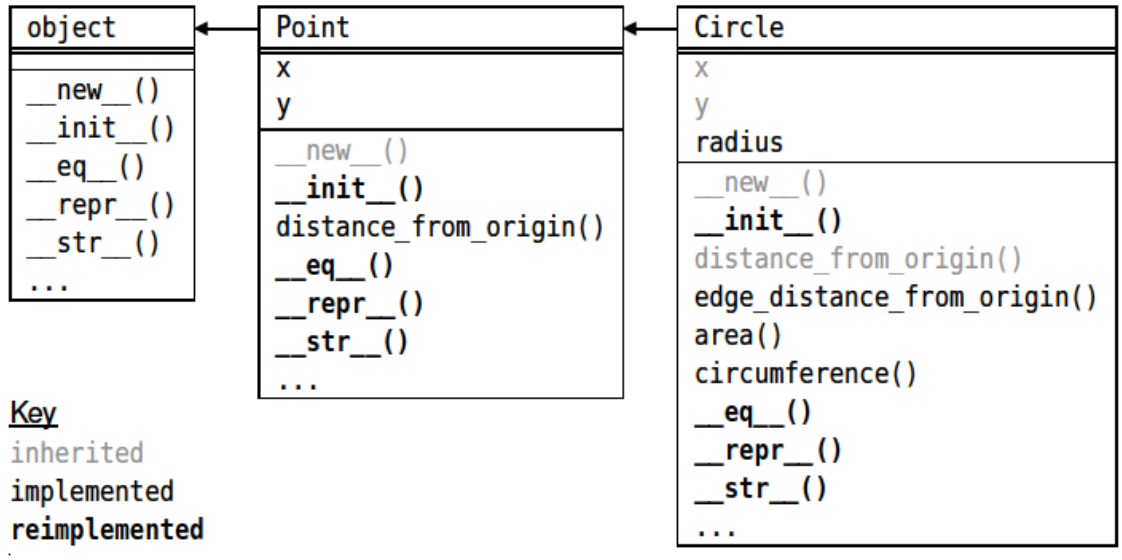

Figure 6.3 *The Circle class's inheritance hierarchy*

Inside the `__init__()` method we use `super()` to call the base class's `__init__()` method—this creates and initializes the `self.x` and `self.y` attributes. Users of the class could supply an invalid radius, such as -2; in the next subsection we will see how to prevent such problems by making attributes more robust using properties.

The `area()` and `circumference()` methods are straightforward. The `edge_distance_from_origin()` method calls the `distance_from_origin()` method as part

* Multiple inheritance, abstract base types, and other advanced object-oriented techniques are covered in Chapter 8.

of its computation. Since the `Circle` class does not provide an implementation of the `distance_from_origin()` method, the one provided by the `Point` base class will be found and used. Contrast this with the reimplementation of the `__eq__()` method. This method compares this circle's radius with the other circle's radius, and if they are equal it then explicitly calls the base class's `__eq__()` method using `super()`. If we did not use `super()` we would have infinite recursion, since `Circle.__eq__()` would then just keep calling itself. Notice also that we don't have to pass the `self` argument in the `super()` calls since Python automatically passes it for us.

Here are a couple of usage examples:

```
p = Shape.Point(28, 45)
c = Shape.Circle(5, 28, 45)
p.distance_from_origin()        # returns: 53.0
c.distance_from_origin()        # returns: 53.0
```

We can call the `distance_from_origin()` method on a `Point` or on a `Circle`, since `Circles` can stand in for `Points`.

Polymorphism means that any object of a given class can be used as though it were an object of any of its class's base classes. This is why when we create a subclass we need to implement only the additional methods we require and have to reimplement only those existing methods we want to replace. And when reimplementing methods, we can use the base class's implementation if necessary by using `super()` inside the reimplementation.

In the `Circle`'s case we have implemented additional methods, such as `area()` and `circumference()`, and reimplemented methods we needed to change. The reimplementations of `__repr__()` and `__str__()` are necessary because without them the base class methods will be used and the strings returned will be of `Points` instead of `Circles`. The reimplementations of `__init__()` and `__eq__()` are necessary because we must account for the fact that `Circles` have an additional attribute, and in both cases we make use of the base class implementations of the methods to minimize the work we must do.

The `Point` and `Circle` classes are as complete as we need them to be. We could provide additional methods, such as other comparison special methods if we wanted to be able to order `Points` or `Circles`. Another thing that we might want to do for which no method is provided is to copy a `Point` or `Circle`. Most Python classes don't provide a `copy()` method (exceptions being `dict.copy()` and `set.copy()`). If we want to copy a `Point` or `Circle` we can easily do so by importing the `copy` module and using the `copy.copy()` function. (There is no need to use `copy.deepcopy()` for `Point` and `Circle` objects since they contain only immutable instance variables.)

Shallow and deep copying
146 ◀

Using Properties to Control Attribute Access

In the previous subsection the `Point` class included a `distance_from_origin()` method, and the `Circle` class had the `area()`, `circumference()`, and `edge_distance_from_origin()` methods. All these methods return a single `float` value, so from the point of view of a user of these classes they could just as well be data attributes, but read-only, of course. In the `ShapeAlt.py` file alternative implementations of `Point` and `Circle` are provided, and all the methods mentioned here are provided as properties. This allows us to write code like this:

```
circle = Shape.Circle(5, 28, 45)  # assumes: import ShapeAlt as Shape
circle.radius                     # returns: 5
circle.edge_distance_from_origin  # returns: 48.0
```

Here are the implementations of the getter methods for the `ShapeAlt.Circle` class's `area` and `edge_ distance_from_origin` properties:

```
@property
def area(self):
    return math.pi * (self.radius ** 2)

@property
def edge_distance_from_origin(self):
    return abs(self.distance_from_origin - self.radius)
```

If we provide only getters as we have done here, the properties are read-only. The code for the `area` property is the same as for the previous `area()` method. The `edge_distance_from_origin`'s code is slightly different from before because it now accesses the base class's `distance_from_origin` property instead of calling a `distance_from_origin()` method. The most notable difference to both is the `property` *decorator*. A decorator is a function that takes a function or method as its argument and returns a "decorated" version, that is, a version of the function or method that is modified in some way. A decorator is indicated by preceding its name with an at symbol (`@`). For now, just treat decorators as syntax—in Chapter 8 we will see how to create custom decorators.

The `property()` decorator function is built-in and takes up to four arguments: a getter function, a setter function, a deleter function, and a docstring. The effect of using `@property` is the same as calling the `property()` function with just one argument, the getter function. We could have created the `area` property like this:

```
def area(self):
    return math.pi * (self.radius ** 2)
area = property(area)
```

We rarely use this syntax, since using a decorator is shorter and clearer.

In the previous subsection we noted that no validation is performed on the `Circle`'s `radius` attribute. We can provide validation by making `radius` into a property. This does not require any changes to the `Circle.__init__()` method, and any code that accesses the `Circle.radius` attribute will continue to work unchanged—only now the radius will be validated whenever it is set.

Python programmers normally use properties rather than the explicit getters and setters (e.g., `getRadius()` and `setRadius()`) that are so commonly used in other object-oriented languages. This is because it is so easy to change a data attribute into a property without affecting the use of the class.

To turn an attribute into a readable/writable property we must create a private attribute where the data is actually held and supply getter and setter methods. Here is the radius's getter, setter, and docstring in full:

```
@property
def radius(self):
    """The circle's radius

    >>> circle = Circle(-2)
    Traceback (most recent call last):
    ...
    AssertionError: radius must be nonzero and non-negative
    >>> circle = Circle(4)
    >>> circle.radius = -1
    Traceback (most recent call last):
    ...
    AssertionError: radius must be nonzero and non-negative
    >>> circle.radius = 6
    """
    return self.__radius

@radius.setter
def radius(self, radius):
    assert radius > 0, "radius must be nonzero and non-negative"
    self.__radius = radius
```

We use an `assert` to ensure a nonzero and non-negative radius and store the radius's value in the private attribute `self.__radius`. Notice that the getter and setter (and deleter if we needed one) all have the same name—it is the decorators that distinguish them, and the decorators rename them appropriately so that no name conflicts occur.

The decorator for the setter may look strange at first sight. Every property that is created has a `getter`, `setter`, and `deleter` attribute, so once the radius property is created using `@property`, the `radius.getter`, `radius.setter`, and `radius.deleter` attributes become available. The `radius.getter` is set to the

getter method by the `@property` decorator. The other two are set up by Python so that they do nothing (so the attribute cannot be written to or deleted), unless they are used as decorators, in which case they in effect replace themselves with the method they are used to decorate.

The `Circle`'s initializer, `Circle.__init__()`, includes the statement `self.radius = radius`; this will call the `radius` property's setter, so if an invalid radius is given when a `Circle` is created an `AssertionError` exception will be raised. Similarly, if an attempt is made to set an existing `Circle`'s `radius` to an invalid value, again the setter will be called and an exception raised. The docstring includes doctests to test that the exception is correctly raised in these cases. (Testing is covered more fully in Chapter 9.)

Both the `Point` and `Circle` types are custom data types that have sufficient functionality to be useful. Most of the data types that we are likely to create are like this, but occasionally it is necessary to create a custom data type that is complete in every respect. We will see examples of this in the next subsection.

Creating Complete Fully Integrated Data Types

When creating a complete data type two possibilities are open to us. One is to create the data type from scratch. Although the data type will inherit `object` (as all Python classes do), every data attribute and method that the data type requires (apart from `__new__()`) must be provided. The other possibility is to inherit from an existing data type that is similar to the one we want to create. In this case the work usually involves reimplementing those methods we want to behave differently and "unimplementing" those methods we don't want at all.

In the following subsubsection we will implement a `FuzzyBool` data type from scratch, and in the subsubsection after that we will implement the same type but will use inheritance to reduce the work we must do. The built-in `bool` type is two-valued (`True` and `False`), but in some areas of AI (Artificial Intelligence), fuzzy Booleans are used, which have values corresponding to "true" and "false", and also to intermediates between them. In our implementations we will use floating-point values with 0.0 denoting `False` and 1.0 denoting `True`. In this system, 0.5 means 50 percent true, and 0.25 means 25 percent true, and so on. Here are some usage examples (they work the same with either implementation):

```
a = FuzzyBool.FuzzyBool(.875)
b = FuzzyBool.FuzzyBool(.25)
a >= b                                  # returns: True
bool(a), bool(b)                        # returns: (True, False)
~a                                      # returns: FuzzyBool(0.125)
```

```
a & b                                   # returns: FuzzyBool(0.25)
b |= FuzzyBool.FuzzyBool(.5)            # b is now: FuzzyBool(0.5)
"a={0:.1%} b={1:.0%}".format(a, b)      # returns: 'a=87.5% b=50%'
```

We want the `FuzzyBool` type to support the complete set of comparison operators (`<`, `<=`, `==`, `!=`, `>=`, `>`), and the three basic logical operations, not (`~`), and (`&`), and or (`|`). In addition to the logical operations we want to provide a couple of other logical methods, `conjunction()` and `disjunction()`, that take as many `FuzzyBools` as we like and return the appropriate resultant `FuzzyBool`. And to complete the data type we want to provide conversions to types `bool`, `int`, `float`, and `str`, and have an `eval()`-able representational form. The final requirements are that `FuzzyBool` supports `str.format()` format specifications, that `FuzzyBools` can be used as dictionary keys or as members of sets, and that `FuzzyBools` are immutable—but with the provision of augmented assignment operators (`&=` and `|=`) to ensure that they are convenient to use.

Table 6.1 (242 ◀) lists the comparison special methods, Table 6.2 (➤ 250) lists the fundamental special methods, and Table 6.3 (➤ 253) lists the numeric special methods—these include the bitwise operators (`~`, `&`, and `|`) which `FuzzyBools` use for their logical operators, and also arithmetic operators such as `+` and `-` which `FuzzyBool` does not implement because they are inappropriate.

Creating Data Types from Scratch

To create the `FuzzyBool` type from scratch means that we must provide an attribute to hold the `FuzzyBool`'s value and all the methods that we require. Here are the `class` line and the initializer, taken from `FuzzyBool.py`:

```
class FuzzyBool:

    def __init__(self, value=0.0):
        self.__value = value if 0.0 <= value <= 1.0 else 0.0
```

`ShapeAlt.Circle.radius` property 247 ◀

We have made the value attribute private because we want `FuzzyBool` to behave like immutables, so allowing access to the attribute would be wrong. Also, if an out-of-range value is given we force it to take a fail-safe value of 0.0 (false). In the previous subsection's `ShapeAlt.Circle` class we used a stricter policy, raising an exception if an invalid radius value was used when creating a new `Circle` object. The `FuzzyBool`'s inheritance tree is shown in Figure 6.4.

The simplest logical operator is logical NOT, for which we have coopted the bitwise inversion operator (`~`):

```
    def __invert__(self):
        return FuzzyBool(1.0 - self.__value)
```

Table 6.2 *Fundamental Special Methods*

Special Method	Usage	Description
`__bool__(self)`	`bool(x)`	If provided, returns a truth value for `x`; useful for `if x: ...`
`__format__(self, format_spec)`	`"{0}".format(x)`	Provides `str.format()` support for custom classes
`__hash__(self)`	`hash(x)`	If provided, `x` can be used as a dictionary key or held in a `set`
`__init__(self, args)`	`x = X(args)`	Called when an object is initialized
`__new__(cls, args)`	`x = X(args)`	Called when an object is created
`__repr__(self)`	`repr(x)`	Returns a string representation of `x`; where possible `eval(repr(x)) == x`
`__repr__(self)`	`ascii(x)`	Returns a string representation of `x` using only ASCII characters
`__str__(self)`	`str(x)`	Returns a human-comprehensible string representation of `x`

Reimplementing `__new__()` ➤ 256

`ascii()` 68 ◀

`str.format()` 83 ◀

The `__del__()` Special Method

The `__del__(self)` special method is called when an object is destroyed—at least in theory. In practice, `__del__()` may never be called, even at program termination. Furthermore, when we write `del x`, all that happens is that the object reference `x` is deleted and the count of how many object references refer to the object that was referred to by `x` is decreased by 1. Only when this count reaches 0 is `__del__()` likely to be called, but Python offers no guarantee that it will ever be called. In view of this, `__del__()` is very rarely reimplemented—none of the examples in this book reimplements it—and it should not be used to free up resources, so it is not suitable to be used for closing files, disconnecting network connections, or disconnecting database connections.

Python provides two separate mechanisms for ensuring that resources are properly released. One is to use a `try ... finally` block as we have seen before and will see again in Chapter 7, and the other is to use a context object in conjunction with a `with` statement—this is covered in Chapter 8.

The bitwise logical AND operator (`&`) is provided by the `__and__()` special method, and the in-place version (`&=`) is provided by `__iand__()`:

```
def __and__(self, other):
    return FuzzyBool(min(self.__value, other.__value))
```

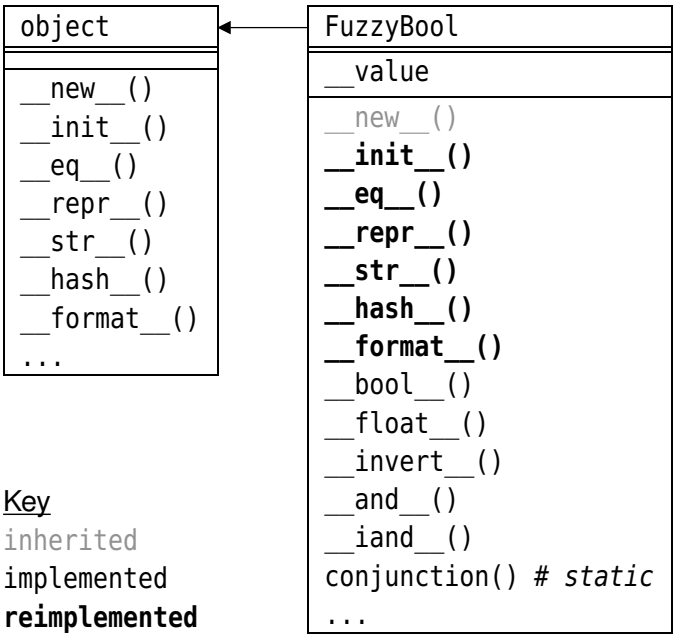

Figure 6.4 *The FuzzyBool class's inheritance hierarchy*

```
def __iand__(self, other):
    self.__value = min(self.__value, other.__value)
    return self
```

The bitwise AND operator returns a new `FuzzyBool` based on this one and the other one, whereas the augmented assignment (in-place) version updates the private value. Strictly speaking, this is not immutable behavior, but it does match the behavior of some other Python immutables, such as `int`, where, for example, using `+=` looks like the left-hand operand is being changed but in fact it is re-bound to refer to a new `int` object that holds the result of the addition. In this case no rebinding is needed because we really do change the `FuzzyBool` itself. And we return `self` to support the chaining of operations.

We could also implement `__rand__()`. This method is called when `self` and `other` are of different types and the `__and__()` method is not implemented for that particular pair of types. This isn't needed for the `FuzzyBool` class. Most of the special methods for binary operators have both "i" (in-place) and "r" (reflect, that is, swap operands) versions.

We have not shown the implementation for `__or__()` which provides the bitwise `|` operator, or for `__ior__()` which provides the in-place `|=` operator, since both are the same as the equivalent AND methods except that we take the maximum value instead of the minimum value of `self` and `other`.

```
def __repr__(self):
    return ("{0}({1})".format(self.__class__.__name__,
                              self.__value))
```

We have created an `eval()`-able representational form. For example, given `f = FuzzyBool.FuzzyBool(.75)`; `repr(f)` will produce the string `'FuzzyBool(0.75)'`.

All objects have some special attributes automatically supplied by Python, one of which is called `__class__`, a reference to the object's class. All classes have a private `__name__` attribute, again provided automatically. We have used these attributes to provide the class name used for the representation form. This means that if the `FuzzyBool` class is subclassed just to add extra methods, the inherited `__repr__()` method will work correctly without needing to be reimplemented, since it will pick up the subclass's class name.

```
def __str__(self):
    return str(self.__value)
```

For the string form we just return the floating-point value formatted as a string. We don't have to use `super()` to avoid infinite recursion because we call `str()` on the `self.__value` attribute, not on the instance itself.

```
def __bool__(self):
    return self.__value > 0.5

def __int__(self):
    return round(self.__value)

def __float__(self):
    return self.__value
```

The `__bool__()` special method converts the instance to a Boolean, so it must always return either `True` or `False`. The `__int__()` special method provides integer conversion. We have used the built-in `round()` function because `int()` simply truncates (so would return 0 for any `FuzzyBool` value except 1.0). Floating-point conversion is easy because the value is already a floating-point number.

```
def __lt__(self, other):
    return self.__value < other.__value

def __eq__(self, other):
    return self.__value == other.__value
```

To provide the complete set of comparisons (<, <=, ==, !=, >=, >) it is necessary to implement at least three of them, <, <=, and ==, since Python can infer > from <, != from ==, and >= from <=. We have shown only two representative methods here since all of them are very similar.*

Complete comparisons ➤ 379

```
def __hash__(self):
    return hash(id(self))
```

* In fact, we implemented only the `__lt__()` and `__eq__()` methods quoted here—the other comparison methods were automatically generated; we will see how in Chapter 8.

Table 6.3 *Numeric and Bitwise Special Methods*

Special Method	Usage	Special Method	Usage
`__abs__(self)`	`abs(x)`	`__complex__(self)`	`complex(x)`
`__float__(self)`	`float(x)`	`__int__(self)`	`int(x)`
`__index__(self)`	`bin(x) oct(x) hex(x)`	`__round__(self, digits)`	`round(x, digits)`
`__pos__(self)`	`+x`	`__neg__(self)`	`-x`
`__add__(self, other)`	`x + y`	`__sub__(self, other)`	`x - y`
`__iadd__(self, other)`	`x += y`	`__isub__(self, other)`	`x -= y`
`__radd__(self, other)`	`y + x`	`__rsub__(self, other)`	`y - x`
`__mul__(self, other)`	`x * y`	`__mod__(self, other)`	`x % y`
`__imul__(self, other)`	`x *= y`	`__imod__(self, other)`	`x %= y`
`__rmul__(self, other)`	`y * x`	`__rmod__(self, other)`	`y % x`
`__floordiv__(self, other)`	`x // y`	`__truediv__(self, other)`	`x / y`
`__ifloordiv__(self, other)`	`x //= y`	`__itruediv__(self, other)`	`x /= y`
`__rfloordiv__(self, other)`	`y // x`	`__rtruediv__(self, other)`	`y / x`
`__divmod__(self, other)`	`divmod(x, y)`	`__rdivmod__(self, other)`	`divmod(y, x)`
`__pow__(self, other)`	`x ** y`	`__and__(self, other)`	`x & y`
`__ipow__(self, other)`	`x **= y`	`__iand__(self, other)`	`x &= y`
`__rpow__(self, other)`	`y ** x`	`__rand__(self, other)`	`y & x`
`__xor__(self, other)`	`x ^ y`	`__or__(self, other)`	`x | y`
`__ixor__(self, other)`	`x ^= y`	`__ior__(self, other)`	`x |= y`
`__rxor__(self, other)`	`y ^ x`	`__ror__(self, other)`	`y | x`
`__lshift__(self, other)`	`x << y`	`__rshift__(self, other)`	`x >> y`
`__ilshift__(self, other)`	`x <<= y`	`__irshift__(self, other)`	`x >>= y`
`__rlshift__(self, other)`	`y << x`	`__rrshift__(self, other)`	`y >> x`
		`__invert__(self)`	`~x`

By default, instances of custom classes support operator == (which always returns `False`), and are hashable (so can be dictionary keys and can be added to sets). But if we reimplement the `__eq__()` special method to provide proper equality testing, instances are no longer hashable. This can be fixed by providing a `__hash__()` special method as we have done here.

Python provides hash functions for strings, numbers, frozen sets, and other classes. Here we have simply used the built-in `hash()` function (which can operate on any type which has a `__hash__()` special method), and given it the object's unique ID from which to calculate the hash. (We can't use the private `self.__value` since that can change as a result of augmented assignment, whereas an object's hash value must never change.)

The built-in `id()` function returns a unique integer for the object it is given as its argument. This integer is usually the object's address in memory, but all that we can assume is that no two objects have the same ID. Behind the scenes the `is` operator uses the `id()` function to determine whether two object references refer to the same object.

```
    def __format__(self, format_spec):
        return format(self.__value, format_spec)
```

The built-in `format()` function is only really needed in class definitions. It takes a single object and an optional format specification and returns a string with the object suitably formatted.

FuzzyBool usage examples 248 ➤

When an object is used in a format string the object's `__format__()` method is called with the object and the format specification as arguments. The method returns the instance suitably formatted as we saw earlier.

All the built-in classes already have suitable `__format__()` methods; here we make use of the `float.__format__()` method by passing the floating-point value and the format string we have been given. We could have achieved exactly the same thing like this:

```
def __format__(self, format_spec):
    return self.__value.__format__(format_spec)
```

Using the `format()` function requires a tiny bit less typing and is clearer to read. Nothing forces us to use the `format()` function at all, so we could invent our own format specification language and interpret it inside the `__format__()` method, as long as we return a string.

```
    @staticmethod
    def conjunction(*fuzzies):
        return FuzzyBool(min([float(x) for x in fuzzies]))
```

The built-in `staticmethod()` function is designed to be used as a decorator as we have done here. Static methods are simply methods that do *not* get `self` or any other first argument specially passed by Python.

The `&` operator can be chained, so given `FuzzyBool`'s `f`, `g`, and `h`, we can get the conjunction of all of them by writing `f & g & h`. This works fine for small numbers of `FuzzyBools`, but if we have a dozen or more it starts to become rather inefficient since each `&` represents a function call. With the method given here we can achieve the same thing using a single function call of `FuzzyBool.FuzzyBool.conjunction(f, g, h)`. This can be written more concisely using a `FuzzyBool` instance, but since static methods don't get `self`, if we call one using an instance and we want to process that instance we must pass it ourselves—for example, `f.conjunction(f, g, h)`.

We have not shown the corresponding `disjunction()` method since it differs only in its name and that it uses `max()` rather than `min()`.

Some Python programmers consider the use of static methods to be un-Pythonic, and use them only if they are converting code from another language (such as C++ or Java), or if they have a method that does not use `self`. In Python, rather than using static methods it is usually better to create a module function instead, as we will see in the next subsubsection, or a class method, as we will see in the last section.

In a similar vein, creating a variable inside a class definition but outside any method creates a static (class) variable. For constants it is usually more convenient to use private module globals, but class variables can often be useful for sharing data among all of a class's instances.

We have now completed the implementation of the `FuzzyBool` class "from scratch". We have had to reimplement 15 methods (17 if we had done the minimum of all four comparison operators), and have implemented two static methods. In the following subsubsection we will show an alternative implementation, this time based on the inheritance of `float`. It involves the reimplementations of just eight methods and the implementation of two module functions—and the "unimplementation" of 32 methods.

In most object-oriented languages inheritance is used to create new classes that have all the methods and attributes of the classes they inherit, as well as the additional methods and attributes that we want the new class to have. Python fully supports this, allowing us to add new methods, or to reimplement inherited methods so as to modify their behavior. But in addition, Python allows us to effectively unimplement methods, that is, to make the new class behave as though it does not have some of the methods that it inherits. Doing this might not appeal to object-oriented purists since it breaks polymorphism, but in Python at least, it can occasionally be a useful technique.

Creating Data Types from Other Data Types

The `FuzzyBool` implementation in this subsubsection is in the file `FuzzyBoolAlt.py`. One immediate difference from the previous version is that instead of providing static methods for `conjunction()` and `disjunction()`, we have provided them as module functions. For example:

```
def conjunction(*fuzzies):
    return FuzzyBool(min(fuzzies))
```

The code for this is much simpler than before because `FuzzyBoolAlt.FuzzyBool` objects are `float` subclasses, and so can be used directly in place of a `float` without needing any conversion. (The inheritance tree is shown in Figure 6.5.) Accessing the function is also cleaner than before. Instead of having to specify both the module and the class (or using an instance), having done `import FuzzyBoolAlt` we can just write `FuzzyBoolAlt.conjunction()`.

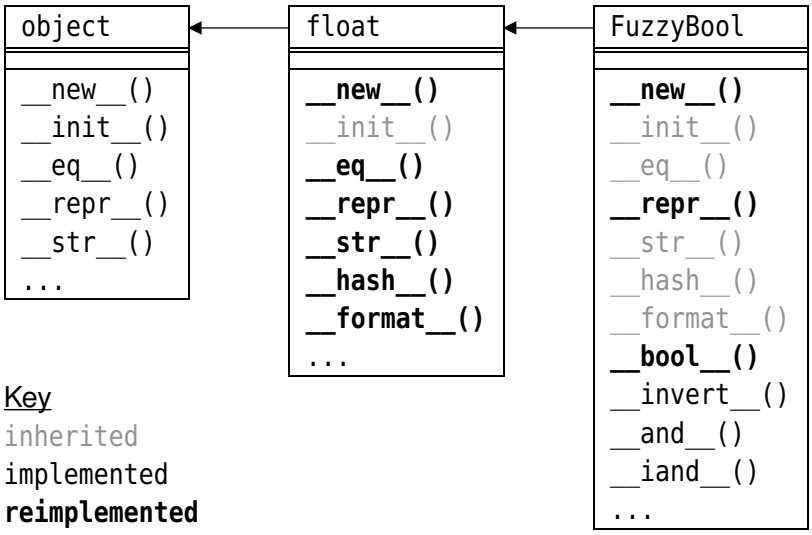

Figure 6.5 *The alternative FuzzyBool class's inheritance hierarchy*

Here is the `FuzzyBool`'s `class` line and its `__new__()` method:

```
class FuzzyBool(float):

    def __new__(cls, value=0.0):
        return super().__new__(cls,
                value if 0.0 <= value <= 1.0 else 0.0)
```

When we create a new class it is usually mutable and relies on `object.__new__()` to create the raw uninitialized object. But in the case of immutable classes we need to do the creation and initialization in one step since once an immutable object has been created it cannot be changed.

The `__new__()` method is called before any object has been created (since object creation is what `__new__()` does), so it cannot have a `self` object passed to it since one doesn't yet exist. In fact, `__new__()` is a *class method*—these are similar to normal methods except that they are called on the class rather than on an instance and Python supplies as their first argument the class they are called on. The variable name `cls` for class is just a convention, in the same way that `self` is the conventional name for the object itself.

So when we write `f = FuzzyBool(0.7)`, under the hood Python calls `FuzzyBool.__new__(FuzzyBool, 0.7)` to create a new object—say, *fuzzy*—and then calls *fuzzy*`.__init__()` to do any further initialization, and finally returns an object reference to the *fuzzy* object—it is this object reference that `f` is set to. Most of `__new__()`'s work is passed on to the base class implementation, `object.__new__()`; all we do is make sure that the value is in range.

Class methods are set up by using the built-in `classmethod()` function used as a decorator. But as a convenience we don't have to bother writing `@classmethod` before `def __new__()` because Python already knows that this method is always a class method. We do need to use the decorator if we want to create other class methods, though, as we will see in the chapter's final section.

Now that we have seen a class method we can clarify the different kinds of methods that Python provides. Class methods have their first argument added by Python and it is the method's class; normal methods have their first argument added by Python and it is the instance the method was called on; and static methods have no first argument added. And all the kinds of methods get any arguments we pass to them (as their second and subsequent arguments in the case of class and normal methods, and as their first and subsequent arguments for static methods).

```
def __invert__(self):
    return FuzzyBool(1.0 - float(self))
```

This method is used to provide support for the bitwise NOT operator (~) just the same as before. Notice that instead of accessing a private attribute that holds the `FuzzyBool`'s value we use `self` directly. This is thanks to inheriting `float` which means that a `FuzzyBool` can be used wherever a `float` is expected—providing none of the `FuzzyBool`'s "unimplemented" methods are used, of course.

```
def __and__(self, other):
    return FuzzyBool(min(self, other))

def __iand__(self, other):
    return FuzzyBool(min(self, other))
```

The logic for these is also the same as before (although the code is subtly different), and just like the `__invert__()` method we can use both `self` and `other`

directly as though they were `floats`. We have omitted the OR versions since they differ only in their names (`__or__()` and `__ior__()`) and that they use `max()` rather than `min()`.

```
def __repr__(self):
    return ("{0}({1})".format(self.__class__.__name__,
                              super().__repr__()))
```

We must reimplement the `__repr__()` method since the base class version `float.__repr__()` just returns the number as a string, whereas we need the class name to make the representation `eval()`-able. For the `str.format()`'s second argument we cannot just pass `self` since that will result in an infinite recursion of calls to this `__repr__()` method, so instead we call the base class implementation.

We don't have to reimplement the `__str__()` method because the base class version, `float.__str__()`, is sufficient and will be used in the absence of a `FuzzyBool.__str__()` reimplementation.

```
def __bool__(self):
    return self > 0.5

def __int__(self):
    return round(self)
```

When a `float` is used in a Boolean context it is `False` if its value is 0.0 and `True` otherwise. This is not the appropriate behavior for `FuzzyBools`, so we have had to reimplement this method. Similarly, using `int(self)` would simply truncate, turning everything but 1.0 into 0, so here we use `round()` to produce 0 for values up to 0.5 and 1 for values up to and including the maximum of 1.0.

We have not reimplemented the `__hash__()` method, the `__format__()` method, or any of the methods that are used to provide the comparison operators, since all those provided by the `float` base class work correctly for `FuzzyBools`.

The methods we have reimplemented provide a complete implementation of the `FuzzyBool` class—and have required far less code than the implementation presented in the previous subsubsection. However, this new `FuzzyBool` class has inherited more than 30 methods which don't make sense for `FuzzyBools`. For example, none of the basic numeric and bitwise shift operators (+, -, *, /, <<, >>, etc.) can sensibly be applied to `FuzzyBools`. Here is how we could begin to "unimplement" addition:

```
def __add__(self, other):
    raise NotImplementedError()
```

We would also have to write the same code for the `__iadd__()` and `__radd__()` methods to completely prevent addition. (Note that `NotImplementedError` is a standard exception and is different from the built-in `NotImplemented` object.) An

alternative to raising a `NotImplementedError` exception, especially if we want to more closely mimic the behavior of Python's built-in classes, is to raise a `TypeError`. Here is how we can make `FuzzyBool.__add__()` behave just like built-in classes that are faced with an invalid operation:

```
def __add__(self, other):
    raise TypeError("unsupported operand type(s) for +: "
                    "'{0}' and '{1}'".format(
                self.__class__.__name__, other.__class__.__name__))
```

For unary operations, we want to unimplement in a way that mimics the behavior of built-in types, the code is slightly easier:

```
def __neg__(self):
    raise TypeError("bad operand type for unary -: '{0}'".format(
                    self.__class__.__name__))
```

For comparison operators, there is a much simpler idiom. For example, to unimplement ==, we would write:

```
def __eq__(self, other):
    return NotImplemented
```

If a method implementing a comparison operator (<, <=, ==, !=, >=, >), returns the built-in `NotImplemented` object and an attempt is made to use the method, Python will first try the reverse comparison by swapping the operands (in case the `other` object has a suitable comparison method since the `self` object does not), and if that doesn't work Python raises a `TypeError` exception with a message that explains that the operation is not supported for operands of the types used. But for all noncomparison methods that we don't want, we must raise either a `NotImplementedError` or a `TypeError` exception as we did for the `__add__()` and `__neg__()` methods shown earlier.

It would be tedious to unimplement every method we don't want as we have done here, although it does work and has the virtue of being easy to understand. Here we will look at a more advanced technique for unimplementing methods—it is used in the `FuzzyBoolAlt` module—but it is probably best to skip to the next section (➤ 261) and return here only if the need arises in practice.

Here is the code for unimplementing the two unary operations we don't want:

```
for name, operator in (("__neg__", "-"),
                       ("__index__", "index()")):
    message = ("bad operand type for unary {0}: '{{self}}'"
               .format(operator))
    exec("def {0}(self): raise TypeError(\"{1}\".format("
         "self=self.__class__.__name__))".format(name, message))
```

The built-in `exec()` function dynamically executes the code passed to it from the object it is given. In this case we have given it a string, but it is also possible to pass some other kinds of objects. By default, the code is executed in the context of the enclosing scope, in this case within the definition of the `FuzzyBool` class, so the `def` statements that are executed create `FuzzyBool` methods which is what we want. The code is executed just once, when the `FuzzyBoolAlt` module is imported. Here is the code that is generated for the first tuple `("__neg__", "-")`:

Dynamic programming ➤ 349

```
def __neg__(self):
    raise TypeError("bad operand type for unary -: '{self}'"
                    .format(self=self.__class__.__name__))
```

We have made the exception and error message match those that Python uses for its own types. The code for handling binary methods and *n*-ary functions (such as `pow()`) follows a similar pattern but with a different error message. For completeness, here is the code we have used:

```
for name, operator in (("__xor__", "^"), ("__ixor__", "^="),
        ("__add__", "+"), ("__iadd__", "+="), ("__radd__", "+"),
        ("__sub__", "-"), ("__isub__", "-="), ("__rsub__", "-"),
        ("__mul__", "*"), ("__imul__", "*="), ("__rmul__", "*"),
        ("__pow__", "**"), ("__ipow__", "**="),
        ("__rpow__", "**"), ("__floordiv__", "//"),
        ("__ifloordiv__", "//="), ("__rfloordiv__", "//"),
        ("__truediv__", "/"), ("__itruediv__", "/="),
        ("__rtruediv__", "/"), ("__divmod__", "divmod()"),
        ("__rdivmod__", "divmod()"), ("__mod__", "%"),
        ("__imod__", "%="), ("__rmod__", "%"),
        ("__lshift__", "<<"), ("__ilshift__", "<<="),
        ("__rlshift__", "<<"), ("__rshift__", ">>"),
        ("__irshift__", ">>="), ("__rrshift__", ">>")):
    message = ("unsupported operand type(s) for {0}: "
               "'{{self}}'{{join}} {{args}}".format(operator))
    exec("def {0}(self, *args):\n"
         "    types = [\"'\" + arg.__class__.__name__ + \"'\" "
         "for arg in args]\n"
         "    raise TypeError(\"{1}\".format("
         "self=self.__class__.__name__, "
         "join=(\" and\" if len(args) == 1 else \",\"),"
         "args=\", \".join(types)))".format(name, message))
```

This code is slightly more complicated than before because for binary operators we must output messages where the two types are listed as *type1 and type2*, but for three or more types we must list them as *type1, type2, type3* to mimic

the built-in behavior. Here is the code that is generated for the first tuple `("__xor__", "^")`:

```
    def __xor__(self, *args):
        types = ["'" + arg.__class__.__name__ + "'" for arg in args]
        raise TypeError("unsupported operand type(s) for ^: "
                        "'{self}'{join} {args}".format(
                        self=self.__class__.__name__,
                        join=(" and" if len(args) == 1 else ","),
                        args=", ".join(types)))
```

The two `for` … `in` loop blocks we have used here can be simply cut and pasted, and then we can add or remove unary operators and methods from the first one and binary or *n*-ary operators and methods from the second one to unimplement whatever methods are not required.

With this last piece of code in place, if we had two `FuzzyBools`, `f` and `g`, and tried to add them using `f + g`, we would get a `TypeError` exception with the message "unsupported operand type(s) for +: 'FuzzyBool' and 'FuzzyBool'", which is exactly the behavior we want.

Creating classes the way we did for the first `FuzzyBool` implementation is much more common and is sufficient for almost every purpose. However, if we need to create an immutable class, the way to do it is to reimplement `object.__new__()` having inherited one of Python's immutable types such as `float`, `int`, `str`, or `tuple`, and then implement all the other methods we need. The disadvantage of doing this is that we may need to unimplement some methods—this breaks polymorphism, so in most cases using aggregation as we did in the first `FuzzyBool` implementation is a much better approach.

Custom Collection Classes

In this section's subsections we will look at custom classes that are responsible for large amounts of data. The first class we will review, `Image`, is one that holds image data. This class is typical of many data-holding custom classes in that it not only provides in-memory access to its data, but also has methods for saving and loading the data to and from disk. The second and third classes we will study, `SortedList` and `SortedDict`, are designed to fill a rare and surprising gap in Python's standard library for intrinsically sorted collection data types.

Creating Classes That Aggregate Collections

A simple way of representing a 2D color image is as a two-dimensional array with each array element being a color. So to represent a 100×100 image we must store 10 000 colors. For the `Image` class (in file `Image.py`), we will take a

potentially more efficient approach. An `Image` stores a single background color, plus the colors of those points in the image that differ from the background color. This is done by using a dictionary as a kind of sparse array, with each key being an (x, y) coordinate and the corresponding value being the color of that point. If we had a 100×100 image and half its points are the background color, we would need to store only 5 000 + 1 colors, a considerable saving in memory.

The `Image.py` module follows what should now be a familiar pattern: It starts with a shebang line, then copyright information in comments, then a module docstring with some doctests, and then the imports, in this case of the `os` and `pickle` modules. We will briefly cover the use of the `pickle` module when we cover saving and loading images. After the imports we create some custom exception classes:

Pickles ➤ 292

```
class ImageError(Exception): pass
class CoordinateError(ImageError): pass
```

We have shown only the first two exception classes; the others (`LoadError`, `SaveError`, `ExportError`, and `NoFilenameError`) are all created the same way and all inherit from `ImageError`. Users of the `Image` class can choose to test for any of the specific exceptions, or just for the base class `ImageError` exception.

The rest of the module consists of the `Image` class and at the end the standard three lines for running the module's doctests. Before looking at the class and its methods, let's look at how it can be used:

```
border_color = "#FF0000"    # red
square_color = "#0000FF"    # blue
width, height = 240, 60
midx, midy = width // 2, height // 2
image = Image.Image(width, height, "square_eye.img")
for x in range(width):
    for y in range(height):
        if x < 5 or x >= width - 5 or y < 5 or y >= height - 5:
            image[x, y] = border_color
        elif midx - 20 < x < midx + 20 and midy - 20 < y < midy + 20:
            image[x, y] = square_color
image.save()
image.export("square_eye.xpm")
```

Notice that we can use the item access operator (`[]`) for setting colors in the image. Brackets can also be used for getting or deleting (effectively setting to the background color) the color at a particular (x, y) coordinate. The coordinates are passed as a single tuple object (thanks to the comma operator), the same as if we wrote `image[(x, y)]`. Achieving this kind of seamless syntax integration is easy in Python—we just have to implement the appropriate special methods,

which in the case of the item access operator are `__getitem__()`, `__setitem__()`, and `__delitem__()`.

The `Image` class uses HTML-style hexadecimal strings to represent colors. The background color must be set when the image is created; otherwise, it defaults to white. The `Image` class saves and loads images in its own custom format, but it can also export in the `.xpm` format which is understood by many image processing applications. The `.xpm` image produced by the code snippet is shown in Figure 6.6.

Figure 6.6 *The square_eye.xpm image*

We will now review the `Image` class's methods, starting with the `class` line and the initializer:

```
class Image:

    def __init__(self, width, height, filename="",
                 background="#FFFFFF"):
        self.filename = filename
        self.__background = background
        self.__data = {}
        self.__width = width
        self.__height = height
        self.__colors = {self.__background}
```

When an `Image` is created, the user (i.e., the class's user) must provide a width and height, but the filename and background color are optional since defaults are provided. The `self.__data` dictionary's keys are (x, y) coordinates and its values are color strings. The `self.__colors` set is initialized with the background color; it is used to keep track of the unique colors used by the image.

All the data attributes are private except for the filename, so we must provide a means by which users of the class can access them. This is easily done using properties.*

```
    @property
    def background(self):
        return self.__background
```

*In Chapter 8 we will see a completely different approach to providing attribute access, using special methods such as `__getattr__()` and `__setattr__()`, that is useful in some circumstances.

```
    @property
    def width(self):
        return self.__width

    @property
    def height(self):
        return self.__height

    @property
    def colors(self):
        return set(self.__colors)
```

When returning a data attribute from an object we need to be aware of whether the attribute is of an immutable or mutable type. It is always safe to return immutable attributes since they can't be changed, but for mutable attributes we must consider some trade-offs. Returning a reference to a mutable attribute is very fast and efficient because no copying takes place—but it also means that the caller now has access to the object's internal state and might change it in a way that invalidates the object. One policy to consider is to always return a copy of mutable data attributes, unless profiling shows a significant negative effect on performance. (In this case, an alternative to keeping the set of unique colors would be to return `set(self.__data.values()) | {self.__background}` whenever the set of colors was needed; we will return to this theme shortly.)

Copying collections 146 ◀

```
    def __getitem__(self, coordinate):
        assert len(coordinate) == 2, "coordinate should be a 2-tuple"
        if (not (0 <= coordinate[0] < self.width) or
            not (0 <= coordinate[1] < self.height)):
            raise CoordinateError(str(coordinate))
        return self.__data.get(tuple(coordinate), self.__background)
```

This method returns the color for a given coordinate using the item access operator (`[]`). The special methods for the item access operators and some other collection-relevant special methods are listed in Table 6.4.

We have chosen to apply two policies for item access. The first policy is that a precondition for using an item access method is that the coordinate it is passed is a sequence of length 2 (usually a 2-tuple), and we use an assertion to ensure this. The second policy is that any coordinate values are accepted, but if either is out of range, we raise a custom exception.

We have used the `dict.get()` method with a default value of the background color to retrieve the color for the given coordinate. This ensures that if the color has never been set for the coordinate the background color is correctly returned instead of a `KeyError` exception being raised.

Table 6.4 *Collection Special Methods*

Special Method	Usage	Description
`__contains__(self, x)`	`x in y`	Returns `True` if `x` is in sequence `y` or if `x` is a key in mapping `y`
`__delitem__(self, k)`	`del y[k]`	Deletes the `k`-th item of sequence `y` or the item with key `k` in mapping `y`
`__getitem__(self, k)`	`y[k]`	Returns the `k`-th item of sequence `y` or the value for key `k` in mapping `y`
`__iter__(self)`	`for x in y: pass`	Returns an iterator for sequence `y`'s items or mapping `y`'s keys
`__len__(self)`	`len(y)`	Returns the number of items in `y`
`__reversed__(self)`	`reversed(y)`	Returns a backward iterator for sequence `y`'s items or mapping `y`'s keys
`__setitem__(self, k, v)`	`y[k] = v`	Sets the `k`-th item of sequence `y` or the value for key `k` in mapping `y`, to `v`

```
    def __setitem__(self, coordinate, color):
        assert len(coordinate) == 2, "coordinate should be a 2-tuple"
        if (not (0 <= coordinate[0] < self.width) or
            not (0 <= coordinate[1] < self.height)):
            raise CoordinateError(str(coordinate))
        if color == self.__background:
            self.__data.pop(tuple(coordinate), None)
        else:
            self.__data[tuple(coordinate)] = color
            self.__colors.add(color)
```

If the user sets a coordinate's value to the background color we can simply delete the corresponding dictionary item since any coordinate not in the dictionary is assumed to have the background color. We must use `dict.pop()` and give a dummy second argument rather than use `del` because doing so avoids a `KeyError` being raised if the key (coordinate) is not in the dictionary.

If the color is different from the background color, we set it for the given coordinate and add it to the set of the unique colors used by the image.

```
    def __delitem__(self, coordinate):
        assert len(coordinate) == 2, "coordinate should be a 2-tuple"
        if (not (0 <= coordinate[0] < self.width) or
            not (0 <= coordinate[1] < self.height)):
            raise CoordinateError(str(coordinate))
        self.__data.pop(tuple(coordinate), None)
```

If a coordinate's color is deleted the effect is to make that coordinate's color the background color. Again we use `dict.pop()` to remove the item since it will work correctly whether or not an item with the given coordinate is in the dictionary.

Both `__setitem__()` and `__delitem__()` have the potential to make the set of colors contain more colors than the image actually uses. For example, if a unique nonbackground color is deleted at a certain pixel, the color remains in the color set even though it is no longer used. Similarly, if a pixel has a unique nonbackground color and is set to the background color, the unique color is no longer used, but remains in the color set. This means that, at worst, the color set could contain more colors than are actually used by the image (but never less).

We have chosen to accept the trade-off of potentially having more colors in the color set than are actually used for the sake of better performance, that is, to make setting and deleting a color as fast as possible—especially since storing a few more colors isn't usually a problem. Of course, if we wanted to ensure that the set of colors was in sync we could either create an additional method that could be called whenever we wanted, or accept the overhead and do the computation automatically when it was needed. In either case, the code is very simple (and is used when a new image is loaded):

```
self.__colors = (set(self.__data.values()) |
                 {self.__background})
```

This simply overwrites the set of colors with the set of colors actually used in the image unioned with the background color.

We have not provided a `__len__()` implementation since it does not make sense for a two-dimensional object. Also, we cannot provide a representational form since an `Image` cannot be created fully formed just by calling `Image()`, so we do not provide `__repr__()` (or `__str__()`) implementations either. If a user calls `repr()` or `str()` on an `Image` object, the `object.__repr__()` base class implementation will return a suitable string, for example, `'<Image.Image object at 0x9c794ac>'`. This is a standard format used for non-`eval()`-able objects. The hexadecimal number is the object's ID—this is unique (normally it is the object's address in memory), but transient.

We want users of the `Image` class to be able to save and load their image data, so we have provided two methods, `save()` and `load()`, to carry out these tasks.

We have chosen to save the data by *pickling* it. In Python-speak pickling is a way of serializing (converting into a sequence of bytes, or into a string) a Python object. What is so powerful about pickling is that the pickled object can be a collection data type, such as a list or a dictionary, and even if the pickled object has other objects inside it (including other collections, which

may include other collections, etc.), the whole lot will be pickled—and without duplicating objects that occur more than once.

A pickle can be read back directly into a Python variable—we don't have to do any parsing or other interpretation ourselves. So using pickles is ideal for saving and loading ad hoc collections of data, especially for small programs and for programs created for personal use. However, pickles have no security mechanisms (no encryption, no digital signature), so loading a pickle that comes from an untrusted source could be dangerous. In view of this, for programs that are not purely for personal use, it is best to create a custom file format that is specific to the program. In Chapter 7 we show how to read and write custom binary, text, and XML file formats.

```
def save(self, filename=None):
    if filename is not None:
        self.filename = filename
    if not self.filename:
        raise NoFilenameError()

    fh = None
    try:
        data = [self.width, self.height, self.__background,
                self.__data]
        fh = open(self.filename, "wb")
        pickle.dump(data, fh, pickle.HIGHEST_PROTOCOL)
    except (EnvironmentError, pickle.PicklingError) as err:
        raise SaveError(str(err))
    finally:
        if fh is not None:
            fh.close()
```

The first part of the function is concerned purely with the filename. If the `Image` object was created with no filename and no filename has been set since, then the `save()` method must be given an explicit filename (in which case it behaves as "save as" and sets the internally used filename). If no filename is specified the current filename is used, and if there is no current filename and none is given an exception is raised.

We create a list (`data`) to hold the objects we want to save, including the `self.__data` dictionary of coordinate–color items, but excluding the set of unique colors since that data can be reconstructed. Then we open the file to write in binary mode and call the `pickle.dump()` function to write the data object to the file. And that's it!

The pickle module can serialize data using various formats (called *protocols* in the documentation), with the one to use specified by the third argument to `pickle.dump()`. Protocol 0 is ASCII and is useful for debugging. We have used protocol 3 (`pickle.HIGHEST_PROTOCOL`), a compact binary format which is why

we had to open the file in binary mode. When reading pickles no protocol is specified—the `pickle.load()` function is smart enough to work out the protocol for itself.

```
def load(self, filename=None):
    if filename is not None:
        self.filename = filename
    if not self.filename:
        raise NoFilenameError()

    fh = None
    try:
        fh = open(self.filename, "rb")
        data = pickle.load(fh)
        (self.__width, self.__height, self.__background,
         self.__data) = data
        self.__colors = (set(self.__data.values()) |
                         {self.__background})
    except (EnvironmentError, pickle.UnpicklingError) as err:
        raise LoadError(str(err))
    finally:
        if fh is not None:
            fh.close()
```

This function starts off the same as the `save()` function to get the filename of the file to load. The file must be opened in read binary mode, and the data is read using the single statement, `data = pickle.load(fh)`. The `data` object is an exact reconstruction of the one we saved, so in this case it is a list with the width and height integers, the background color string, and the dictionary of coordinate–color items. We use tuple unpacking to assign each of the `data` list's items to the appropriate variable, so any previously held image data is (correctly) lost.

The set of unique colors is reconstructed by making a set of all the colors in the coordinate–color dictionary and then adding the background color.

```
def export(self, filename):
    if filename.lower().endswith(".xpm"):
        self.__export_xpm(filename)
    else:
        raise ExportError("unsupported export format: " +
                          os.path.splitext(filename)[1])
```

We have provided one generic export method that uses the file extension to determine which private method to call—or raises an exception for file formats that cannot be exported. In this case we only support saving to `.xpm` files (and then only for images with fewer than 8 930 colors). We haven't quoted the

`__export_xpm()` method because it isn't really relevant to this chapter's theme, but it is in the book's source code, of course.

We have now completed our coverage of the custom `Image` class. This class is typical of those used to hold program-specific data, providing access to the data items it contains, the ability to save and load all its data to and from disk, and with only the essential methods it needs provided. In the next two subsections we will see how to create two generic custom collection types that offer complete APIs.

Creating Collection Classes Using Aggregation

In this subsection we will develop a complete custom collection data type, `SortedList`, that holds a list of items in sorted order. The items are sorted using their less than operator (<), provided by the `__lt__()` special method, or by using a key function if one is given. The class tries to match the API of the built-in `list` class to make it as easy to learn and use as possible, but some methods cannot sensibly be provided—for example, using the concatenation operator (+) could result in items being out of order, so we do not implement it.

As always when creating custom classes, we are faced with the choices of inheriting a class that is similar to the one we want to make, or creating a class from scratch and aggregating instances of any other classes we need inside it, or doing a mixture of both. For this subsection's `SortedList` we use aggregation (and implicitly inherit `object`, of course), and for the following subsection's `SortedDict` we will use both aggregation and inheritance (inheriting `dict`).

In Chapter 8 we will see that classes can make promises about the API they offer. For example, a `list` provides the `MutableSequence` API which means that it supports the `in` operator, the `iter()` and `len()` built-in functions, and the item access operator (`[]`) for getting, setting, and deleting items, and an `insert()` method. The `SortedList` class implemented here does not support item setting and does not have an `insert()` method, so it does not provide a `MutableSequence` API. If we were to create `SortedList` by inheriting `list`, the resultant class would claim to be a mutable sequence but would not have the complete API. In view of this the `SortedList` does not inherit `list` and so makes no promises about its API. On the other hand, the next subsection's `SortedDict` class supports the complete `MutableMapping` API that the `dict` class provides, so we can make it a `dict` subclass.

Here are some basic examples of using a `SortedList`:

```
letters = SortedList.SortedList(("H", "c", "B", "G", "e"), str.lower)
# str(letters) == "['B', 'c', 'e', 'G', 'H']"
letters.add("G")
letters.add("f")
letters.add("A")
```

```
# str(letters) == "['A', 'B', 'c', 'e', 'f', 'G', 'G', 'H']"
letters[2] # returns: 'c'
```

A `SortedList` object *aggregates* (is composed of) two private attributes; a function, `self.__key()` (held as object reference `self.__key`), and a list, `self.__list`.

Lambda functions
182 ◀

The key function is passed as the second argument (or using the `key` keyword argument if no initial sequence is given). If no key function is specified the following private module function is used:

```
_identity = lambda x: x
```

This is the identity function: It simply returns its argument unchanged, so when it is used as a `SortedList`'s key function it means that the sort key for each object in the list is the object itself.

The `SortedList` type does not allow the item access operator (`[]`) to change an item (so it does not implement the `__setitem__()` special method), nor does it provide the `append()` or `extend()` method since these might invalidate the ordering. The only way to add items is to pass a sequence when the `SortedList` is created or to add them later using the `SortedList.add()` method. On the other hand, we can safely use the item access operator for getting or deleting the item at a given index position since neither operation affects the ordering, so both the `__getitem__()` and `__delitem__()` special methods are implemented.

We will now review the class method by method, starting as usual with the `class` line and the initializer:

```
class SortedList:

    def __init__(self, sequence=None, key=None):
        self.__key = key or _identity
        assert hasattr(self.__key, "__call__")
        if sequence is None:
            self.__list = []
        elif (isinstance(sequence, SortedList) and
              sequence.key == self.__key):
            self.__list = sequence.__list[:]
        else:
            self.__list = sorted(list(sequence), key=self.__key)
```

Since a function's name is an object reference (to its function), we can hold functions in variables just like any other object reference. Here the private `self.__key` variable holds a reference to the key function that was passed in, or to the identity function. The method's first statement relies on the fact that the `or` operator returns its first operand if it is `True` in a Boolean context (which a not-`None` key function is), or its second operand otherwise. A slightly longer but

more obvious alternative would have been `self.__key = key if key is not None else _identity`.

Once we have the key function, we use an assert to ensure that it is callable. The built-in `hasattr()` function returns `True` if the object passed as its first argument has the attribute whose name is passed as its second argument. There are corresponding `setattr()` and `delattr()` functions—these functions are covered in Chapter 8. All callable objects, for example, functions and methods, have a `__call__` attribute.

To make the creation of `SortedLists` as similar as possible to the creation of `lists` we have an optional `sequence` argument that corresponds to the single optional argument that `list()` accepts. The `SortedList` class aggregates a `list` collection in the private variable `self.__list` and keeps the items in the aggregated list in sorted order using the given key function.

The `elif` clause uses type testing to see whether the given sequence is a `SortedList` and if that is the case whether it has the same key function as this sorted list. If these conditions are met we simply shallow-copy the sequence's list without needing to sort it. If most key functions are created on the fly using `lambda`, even though two may have the same code they will not compare as equal, so the efficiency gain may not be realized in practice.

```
@property
def key(self):
    return self.__key
```

Once a sorted list is created its key function is fixed, so we keep it as a private variable to prevent users from changing it. But some users may want to get a reference to the key function (as we will see in the next subsection), and so we have made it accessible by providing the read-only `key` property.

```
def add(self, value):
    index = self.__bisect_left(value)
    if index == len(self.__list):
        self.__list.append(value)
    else:
        self.__list.insert(index, value)
```

When this method is called the given value must be inserted into the private `self.__list` in the correct position to preserve the list's order. The private `SortedList.__bisect_left()` method returns the required index position as we will see in a moment. If the new value is larger than any other value in the list it must go at the end, so the index position will be equal to the list's length (list index positions go from `0` to `len(L) - 1`)—if this is the case we append the new value. Otherwise, we insert the new value at the given index position—which will be at index position 0 if the new value is smaller than any other value in the list.

```
    def __bisect_left(self, value):
        key = self.__key(value)
        left, right = 0, len(self.__list)
        while left < right:
            middle = (left + right) // 2
            if self.__key(self.__list[middle]) < key:
                left = middle + 1
            else:
                right = middle
        return left
```

This private method calculates the index position where the given value belongs in the list, that is, the index position where the value is (if it is in the list), or where it should go (if it isn't in the list). It computes the comparison key for the given value using the sorted list's key function, and compares the comparison key with the computed comparison keys of the items that the method examines. The algorithm used is *binary search* (also called *binary chop*), which has excellent performance even on very large lists—for example, at most, 21 comparisons are required to find a value's position in a list of 1 000 000 items.* Compare this with a plain unsorted `list` which uses linear search and needs an average of 500 000 comparisons, and at worst 1 000 000 comparisons, to find a value in a list of 1 000 000 items.

```
    def remove(self, value):
        index = self.__bisect_left(value)
        if index < len(self.__list) and self.__list[index] == value:
            del self.__list[index]
        else:
            raise ValueError("{0}.remove(x): x not in list".format(
                             self.__class__.__name__))
```

This method is used to remove the first occurrence of the given value. It uses the `SortedList.__bisect_left()` method to find the index position where the value belongs and then tests to see whether that index position is within the list and that the item at that position is the same as the given value. If the conditions are met the item is removed; otherwise, a `ValueError` exception is raised (which is what `list.remove()` does in the same circumstances).

```
    def remove_every(self, value):
        count = 0
        index = self.__bisect_left(value)
        while (index < len(self.__list) and
               self.__list[index] == value):
```

*Python's `bisect` module provides the `bisect.bisect_left()` function and some others, but at the time of this writing none of the `bisect` module's functions can work with a key function.

```
            del self.__list[index]
            count += 1
        return count
```

This method is similar to the `SortedList.remove()` method, and is an extension of the list API. It starts off by finding the index position where the first occurrence of the value belongs in the list, and then loops as long as the index position is within the list and the item at the index position is the same as the given value. The code is slightly subtle since at each iteration the matching item is deleted, and as a consequence, after each deletion the item at the index position is the item that followed the deleted item.

```
    def count(self, value):
        count = 0
        index = self.__bisect_left(value)
        while (index < len(self.__list) and
               self.__list[index] == value):
            index += 1
            count += 1
        return count
```

This method returns the number of times the given value occurs in the list (which could be 0). It uses a very similar algorithm to `SortedList.remove_every()`, only here we must increment the index position in each iteration.

```
    def index(self, value):
        index = self.__bisect_left(value)
        if index < len(self.__list) and self.__list[index] == value:
            return index
        raise ValueError("{0}.index(x): x not in list".format(
                         self.__class__.__name__))
```

Since a `SortedList` is ordered we can use a fast binary search to find (or not find) the value in the list.

```
    def __delitem__(self, index):
        del self.__list[index]
```

The `__delitem__()` special method provides support for the `del L[n]` syntax, where `L` is a sorted list and `n` is an integer index position. We don't test for an out-of-range index since if one is given the `self.__list[index]` call will raise an `IndexError` exception, which is the behavior we want.

```
    def __getitem__(self, index):
        return self.__list[index]
```

This method provides support for the `x = L[n]` syntax, where `L` is a sorted list and `n` is an integer index position.

```
    def __setitem__(self, index, value):
        raise TypeError("use add() to insert a value and rely on "
                        "the list to put it in the right place")
```

We don't want the user to change an item at a given index position (so `L[n] = x` is disallowed); otherwise, the sorted list's order might be invalidated. The `TypeError` exception is the one used to signify that an operation is not supported by a particular data type.

```
    def __iter__(self):
        return iter(self.__list)
```

This method is easy to implement since we can just return an iterator to the private list using the built-in `iter()` function. This method is used to support the `for` *`value`* `in` *`iterable`* syntax.

Note that if a sequence is required it is this method that is used. So to convert a `SortedList`, *`L`*, to a plain list we can call `list(`*`L`*`)`, and behind the scenes Python will call `SortedList.__iter__(`*`L`*`)` to provide the sequence that the `list()` function requires.

```
    def __reversed__(self):
        return reversed(self.__list)
```

This provides support for the built-in `reversed()` function so that we can write, for example, `for` *`value`* `in reversed(`*`iterable`*`)`.

```
    def __contains__(self, value):
        index = self.__bisect_left(value)
        return (index < len(self.__list) and
                self.__list[index] == value)
```

The `__contains__()` method provides support for the `in` operator. Once again we are able to use a fast binary search rather than the slow linear search used by a plain `list`.

```
    def clear(self):
        self.__list = []

    def pop(self, index=-1):
        return self.__list.pop(index)

    def __len__(self):
        return len(self.__list)

    def __str__(self):
        return str(self.__list)
```

The `SortedList.clear()` method discards the existing list and replaces it with a new empty list. The `SortedList.pop()` method removes and returns the item at the given index position, or raises an `IndexError` exception if the index is out of range. For the `pop()`, `__len__()`, and `__str__()` methods, we simply pass on the work to the aggregated `self.__list` object.

We do not reimplement the `__repr__()` special method, so the base class `object.__repr__()` will be called when the user writes `repr(L)` and `L` is a `SortedList`. This will produce a string such as `'<SortedList.SortedList object at 0x97e7cec>'`, although the hexadecimal ID will vary, of course. We cannot provide a sensible `__repr__()` implementation because we would need to give the key function and we cannot represent a function object reference as an `eval()`-able string.

We have not implemented the `insert()`, `reverse()`, or `sort()` method because none of them is appropriate. If any of them are called an `AttributeError` exception will be raised.

If we copy a sorted list using the `L[:]` idiom we will get a plain `list` object, rather than a `SortedList`. The easiest way to get a copy is to import the `copy` module and use the `copy.copy()` function—this is smart enough to copy a sorted list (and instances of most other custom classes) without any help. However, we have decided to provide an explicit `copy()` method:

```
def copy(self):
    return SortedList(self, self.__key)
```

By passing `self` as the first argument we ensure that `self.__list` is simply shallow-copied rather than being copied and re-sorted. (This is thanks to the `__init__()` method's type-testing `elif` clause.) The theoretical performance advantage of copying this way is not available to the `copy.copy()` function, but we can easily make it available by adding this line:

```
__copy__ = copy
```

When `copy.copy()` is called it tries to use the object's `__copy__()` special method, falling back to its own code if one isn't provided. With this line in place `copy.copy()` will now use the `SortedList.copy()` method for sorted lists. (It is also possible to provide a `__deepcopy__()` special method, but this is slightly more involved—the `copy` module's online documentation has the details.)

We have now completed the implementation of the `SortedList` class. In the next subsection we will make use of a `SortedList` to provide a sorted list of keys for the `SortedDict` class.

Creating Collection Classes Using Inheritance

The SortedDict class shown in this subsection attempts to mimic a dict as closely as possible. The major difference is that a SortedDict's keys are always ordered based on a specified key function or on the identity function. SortedDict provides the same API as dict (except for having a non-eval()-able repr()), plus two extra methods that make sense only for an ordered collection.* (Note that Python 3.1 introduced the collections.OrderedDict class—this class is different from SortedDict since it is insertion-ordered rather than key-ordered.)

collections.OrderedDict 136 ◀

Here are a few examples of use to give a flavor of how SortedDict works:

```
d = SortedDict.SortedDict(dict(s=1, A=2, y=6), str.lower)
d["z"] = 4
d["T"] = 5
del d["y"]
d["n"] = 3
d["A"] = 17
str(d) # returns: "{'A': 17, 'n': 3, 's': 1, 'T': 5, 'z': 4}"
```

The SortedDict implementation uses both aggregation and inheritance. The sorted list of keys is aggregated as an instance variable, whereas the SortedDict class itself inherits the dict class. We will start our code review by looking at the class line and the initializer, and then we will look at all of the other methods in turn.

```
class SortedDict(dict):

    def __init__(self, dictionary=None, key=None, **kwargs):
        dictionary = dictionary or {}
        super().__init__(dictionary)
        if kwargs:
            super().update(kwargs)
        self.__keys = SortedList.SortedList(super().keys(), key)
```

The dict base class is specified in the class line. The initializer tries to mimic the dict() function, but adds a second argument for the key function. The super().__init__() call is used to initialize the SortedDict using the base class dict.__init__() method. Similarly, if keyword arguments have been used, we use the base class dict.update() method to add them to the dictionary. (Note that only one occurrence of any keyword argument is accepted, so none of the keys in the kwargs keyword arguments can be "dictionary" or "key".)

*The SortedDict class presented here is different from the one in *Rapid GUI Programming with Python and Qt* by this author, ISBN 0132354187, and from the one in the Python Package Index.

We keep a copy of all the dictionary's keys in a sorted list stored in the `self.__keys` variable. We pass the dictionary's keys to initialize the sorted list using the base class's `dict.keys()` method—we must not use `SortedDict.keys()` because that relies on the `self.__keys` variable which will exist only *after* the `SortedList` of keys has been created.

```
def update(self, dictionary=None, **kwargs):
    if dictionary is None:
        pass
    elif isinstance(dictionary, dict):
        super().update(dictionary)
    else:
        for key, value in dictionary.items():
            super().__setitem__(key, value)
    if kwargs:
        super().update(kwargs)
    self.__keys = SortedList.SortedList(super().keys(),
                                        self.__keys.key)
```

This method is used to update one dictionary's items with another dictionary's items, or with keyword arguments, or both. Items which exist only in the other dictionary are added to this one, and for items whose keys appear in both dictionaries, the other dictionary's value replaces the original value. We have had to extend the behavior slightly in that we keep the original dictionary's key function, even if the other dictionary is a `SortedDict`.

The updating is done in two phases. First we update the dictionary's items. If the given dictionary is a `dict` subclass (which includes `SortedDict`, of course), we use the base class `dict.update()` to perform the update—using the base class version is essential to avoid calling `SortedDict.update()` recursively and going into an infinite loop. If the dictionary is not a `dict` we iterate over the dictionary's items and set each key–value pair individually. (If the dictionary object is not a `dict` and does not have an `items()` method an `AttributeError` exception will quite rightly be raised.) If keyword arguments have been used we again call the base class `update()` method to incorporate them.

A consequence of the updating is that the `self.__keys` list becomes out of date, so we replace it with a new `SortedList` with the dictionary's keys (again obtained from the base class, since the `SortedDict.keys()` method relies on the `self.__keys` list which we are in the process of updating), and with the original sorted list's key function.

```
@classmethod
def fromkeys(cls, iterable, value=None, key=None):
    return cls({k: value for k in iterable}, key)
```

The `dict` API includes the `dict.fromkeys()` class method. This method is used to create a new dictionary based on an iterable. Each element in the iterable becomes a key, and each key's value is either `None` or the specified value.

Because this is a class method the first argument is provided automatically by Python and is the class. For a `dict` the class will be `dict`, and for a `SortedDict` it is `SortedDict`. The return value is a dictionary of the given class. For example:

```
class MyDict(SortedDict.SortedDict): pass
d = MyDict.fromkeys("VEINS", 3)
str(d)    # returns: "{'E': 3, 'I': 3, 'N': 3, 'S': 3, 'V': 3}"
d.__class__.__name__    # returns: 'MyDict'
```

So when inherited class methods are called, their `cls` variable is set to the correct class, just like when normal methods are called and their `self` variable is set to the current object. This is different from and better than using a static method because a static method is tied to a particular class and does not know whether it is being executed in the context of its original class or that of a subclass.

```
def __setitem__(self, key, value):
    if key not in self:
        self.__keys.add(key)
    return super().__setitem__(key, value)
```

This method implements the `d[key] = value` syntax. If the key isn't in the dictionary we add it to the list of keys, relying on the `SortedList` to put it in the right place. Then we call the base class method, and return its result to the caller to support chaining, for example, `x = d[key] = value`.

Notice that in the `if` statement we check to see whether the key already exists in the `SortedDict` by using `not in self`. Because `SortedDict` inherits `dict`, a `SortedDict` can be used wherever a `dict` is expected, and in this case `self` is a `SortedDict`. When we reimplement `dict` methods in `SortedDict`, if we need to call the base class implementation to get it to do some of the work for us, we must be careful to call the method using `super()`, as we do in this method's last statement; doing so prevents the reimplementation of the method from calling itself and going into infinite recursion.

We do not reimplement the `__getitem__()` method since the base class version works fine and has no effect on the ordering of the keys.

```
def __delitem__(self, key):
    try:
        self.__keys.remove(key)
    except ValueError:
        raise KeyError(key)
    return super().__delitem__(key)
```

Generator Functions

A *generator function* or *generator method* is one which contains a `yield` expression. When a generator function is called it returns an iterator. Values are extracted from the iterator one at a time by calling its `__next__()` method. At each call to `__next__()` the generator function's `yield` expression's value (`None` if none is specified) is returned. If the generator function finishes or executes a `return` a `StopIteration` exception is raised.

In practice we rarely call `__next__()` or catch a `StopIteration`. Instead, we just use a generator like any other iterable. Here are two almost equivalent functions. The one on the left returns a list and the one on the right returns a generator.

```
# Build and return a list
def letter_range(a, z):
    result = []
    while ord(a) < ord(z):
        result.append(a)
        a = chr(ord(a) + 1)
    return result
```

```
# Return each value on demand
def letter_range(a, z):
    while ord(a) < ord(z):
        yield a
        a = chr(ord(a) + 1)
```

We can iterate over the result produced by either function using a `for` loop, for example, `for letter in letter_range("m", "v"):`. But if we want a list of the resultant letters, although calling `letter_range("m", "v")` is sufficient for the left-hand function, for the right-hand generator function we must use `list(letter_range("m", "v"))`.

Generator functions and methods (and generator expressions) are covered more fully in Chapter 8.

This method provides the `del d[key]` syntax. If the key is not present the `SortedList.remove()` call will raise a `ValueError` exception. If this occurs we catch the exception and raise a `KeyError` exception instead so as to match the `dict` class's API. Otherwise, we return the result of calling the base class implementation to delete the item with the given key from the dictionary itself.

```
def setdefault(self, key, value=None):
    if key not in self:
        self.__keys.add(key)
    return super().setdefault(key, value)
```

This method returns the value for the given key if the key is in the dictionary; otherwise, it creates a new item with the given key and value and returns the value. For the `SortedDict` we must make sure that the key is added to the keys list if the key is not already in the dictionary.

```
    def pop(self, key, *args):
        if key not in self:
            if len(args) == 0:
                raise KeyError(key)
            return args[0]
        self.__keys.remove(key)
        return super().pop(key, args)
```

If the given key is in the dictionary this method returns the corresponding value and removes the key–value item from the dictionary. The key must also be removed from the keys list.

The implementation is quite subtle because the `pop()` method must support two different behaviors to match `dict.pop()`. The first is `d.pop(k)`; here the value for key `k` is returned, or if there is no key `k`, a `KeyError` is raised. The second is `d.pop(k, value)`; here the value for key `k` is returned, or if there is no key `k`, `value` (which could be `None`) is returned. In all cases, if key `k` exists, the corresponding item is removed.

```
    def popitem(self):
        item = super().popitem()
        self.__keys.remove(item[0])
        return item
```

The `dict.popitem()` method removes and returns a random key–value item from the dictionary. We must call the base class version first since we don't know in advance which item will be removed. We remove the item's key from the keys list, and then return the item.

```
    def clear(self):
        super().clear()
        self.__keys.clear()
```

Here we clear all the dictionary's items and all the keys list's items.

```
    def values(self):
        for key in self.__keys:
            yield self[key]

    def items(self):
        for key in self.__keys:
            yield (key, self[key])

    def __iter__(self):
        return iter(self.__keys)

    keys = __iter__
```

Dictionaries have four methods that return iterators: `dict.values()` for the dictionary's values, `dict.items()` for the dictionary's key–value items, `dict.keys()` for the keys, and the `__iter__()` special method that provides support for the `iter(d)` syntax, and operates on the keys. (Actually, the base class versions of these methods return dictionary views, but for most purposes the behavior of the iterators implemented here is the same.)

Since the `__iter__()` method and the `keys()` method have identical behavior, instead of implementing `keys()`, we simply create an object reference called `keys` and set it to refer to the `__iter__()` method. With this in place, users of `SortedDict` can call `d.keys()` or `iter(d)` to get an iterator over a dictionary's keys, just the same as they can call `d.values()` to get an iterator over the dictionary's values.

The `values()` and `items()` methods are generator methods—see the sidebar "Generator Functions" (279 ◀) for a brief explanation of generator methods. In both cases they iterate over the sorted keys list, so they always return iterators that iterate in key order (with the key order depending on the key function given to the initializer). For the `items()` and `values()` methods, the values are looked up using the `d[k]` syntax (which uses `dict.__getitem__()` under the hood), since we can treat `self` as a `dict`.

Generators ➤ 341

```
def __repr__(self):
    return object.__repr__(self)

def __str__(self):
    return ("{" + ", ".join(["{0!r}: {1!r}".format(k, v)
                             for k, v in self.items()]) + "}")
```

We cannot provide an `eval()`-able representation of a `SortedDict` because we can't produce an `eval()`-able representation of the key function. So for the `__repr__()` reimplementation we bypass `dict.__repr__()`, and instead call the ultimate base class version, `object.__repr__()`. This produces a string of the kind used for non-`eval()`-able representations, for example, `'<SortedDict.SortedDict object at 0xb71fff5c>'`.

We have implemented the `SortedDict.__str__()` method ourselves because we want the output to show the items in key sorted order. The method could have been written like this instead:

```
items = []
for key, value in self.items():
    items.append("{0!r}: {1!r}".format(key, value))
return "{" + ", ".join(items) + "}"
```

Using a list comprehension is shorter and avoids the need for the temporary `items` variable.

The base class methods `dict.get()`, `dict.__getitem__()` (for the `v = d[k]` syntax), `dict.__len__()` (for `len(d)`), and `dict.__contains__()` (for `x in d`) all work fine as they are and don't affect the key ordering, so we have not needed to reimplement them.

The last `dict` method that we must reimplement is `copy()`.

```
def copy(self):
    d = SortedDict()
    super(SortedDict, d).update(self)
    d.__keys = self.__keys.copy()
    return d
```

The easiest reimplementation is simply `def copy(self): return SortedDict(self)`. We've chosen a slightly more complicated solution that avoids re-sorting the already sorted keys. We create an empty sorted dictionary, then update it with the items in the original sorted dictionary using the base class `dict.update()` to avoid the `SortedDict.update()` reimplementation, and replace the dictionary's `self.__keys SortedList` with a shallow copy of the original one.

When `super()` is called with no arguments it works on the base class and the `self` object. But we can make it work on any class and any object by passing in a class and an object explicitly. Using this syntax, the `super()` call works on the immediate *base* class of the class it is given, so in this case the code has the same effect as (and could be written as) `dict.update(d, self)`.

In view of the fact that Python's sort algorithm is very fast, and is particularly well optimized for partially sorted lists, the efficiency gain is likely to be little or nothing except for huge dictionaries. However, the implementation shows that at least in principle, a custom `copy()` method can be more efficient than using the `copy_of_x = ClassOfX(x)` idiom that Python's built-in types support. And just as we did for `SortedList`, we have set `__copy__ = copy` so that the `copy.copy()` function uses our custom copy method rather than its own code.

```
def value_at(self, index):
    return self[self.__keys[index]]

def set_value_at(self, index, value):
    self[self.__keys[index]] = value
```

These two methods represent an extension to the `dict` API. Since, unlike a plain `dict`, a `SortedDict` is ordered, it follows that the concept of key index positions is applicable. For example, the first item in the dictionary is at index position 0, and the last at position `len(d) - 1`. Both of these methods operate on the dictionary item whose key is at the `index`-th position in the sorted keys list. Thanks to inheritance, we can look up values in the `SortedDict` using the item access op-

erator (`[]`) applied directly to `self`, since `self` is a `dict`. If an out-of-range index is given the methods raise an `IndexError` exception.

We have now completed the implementation of the `SortedDict` class. It is not often that we need to create complete generic collection classes like this, but when we do, Python's special methods allow us to fully integrate our class so that its users can treat it like any of the built-in or standard library classes.

Summary

This chapter covered all the fundamentals of Python's support for object-oriented programming. We began by showing some of the disadvantages of a purely procedural approach and how these could be avoided by using object orientation. We then described some of the most common terminology used in object-oriented programming, including many "duplicate" terms such as *base class* and *super class*.

We saw how to create simple classes with data attributes and custom methods. We also saw how to inherit classes and how to add additional data attributes and additional methods, and how methods can be "unimplemented". Unimplementing is needed when we inherit a class but want to restrict the methods that our subclass provides, but it should be used with care since it breaks the expectation that a subclass can be used wherever one of its base classes can be used, that is, it breaks polymorphism.

Custom classes can be seamlessly integrated so that they support the same syntaxes as Python's built-in and library classes. This is achieved by implementing special methods. We saw how to implement special methods to support comparisons, how to provide representational and string forms, and how to provide conversions to other types such as `int` and `float` when it makes sense to do so. We also saw how to implement the `__hash__()` method to make a custom class's instances usable as dictionary keys or as members of a set.

Data attributes by themselves provide no mechanism for ensuring that they are set to valid values. We saw how easy it is to replace data attributes with properties—this allows us to create read-only properties, and for writable properties makes it easy to provide validation.

Most of the classes we create are "incomplete" since we tend to provide only the methods that we actually need. This works fine in Python, but in addition it is possible to create complete custom classes that provide every relevant method. We saw how to do this for single valued classes, both by using aggregation and more compactly by using inheritance. We also saw how to do this for multivalued (collection) classes. Custom collection classes can provide the same facilities as the built-in collection classes, including support for `in`, `len()`, `iter()`, `reversed()`, and the item access operator (`[]`).

We learned that object creation and initialization are separate operations and that Python allows us to control both, although in almost every case we only need to customize initialization. We also learned that although it is always safe to return an object's immutable data attributes, we should normally only ever return copies of an object's mutable data attributes to avoid the object's internal state leaking out and being accidentally invalidated.

Python provides normal methods, static methods, class methods, and module functions. We saw that most methods are normal methods, with class methods being occasionally useful. Static methods are rarely used, since class methods or module functions are almost always better alternatives.

The built-in `repr()` method calls an object's `__repr__()` special method. Where possible, `eval(repr(x)) == x`, and we saw how to support this. When an `eval()`-able representation string cannot be produced we use the base class `object.__repr__()` method to produce a non-`eval()`-able representation in a standard format.

Type testing using the built-in `isinstance()` function can provide some efficiency benefits, although object-oriented purists would almost certainly avoid its use. Accessing base class methods is achieved by calling the built-in `super()` function, and is essential to avoid infinite recursion when we need to call a base class method inside a subclass's reimplementation of that method.

Generator functions and methods do lazy evaluation, returning (via the `yield` expression) each value one at a time on request and raising a `StopIteration` when (and if) they run out of values. Generators can be used wherever an iterator is expected, and for finite generators, all their values can be extracted into a tuple or list by passing the iterator returned by the generator to `tuple()` or `list()`.

The object-oriented approach almost invariably simplifies code compared with a purely procedural approach. With custom classes we can guarantee that only valid operations are available (since we implement only appropriate methods), and that no operation can put an object into an invalid state (e.g., by using properties to apply validation). Once we start using object orientation our style of programming is likely to change from being about global data structures and the global functions that are applied to the data, to creating classes and implementing the methods that are applicable to them. Object orientation makes it possible to package up data and those methods that make sense for the data. This helps us avoid mixing up all our data and functions together, and makes it easier to produce maintainable programs since functionality is kept separated out into individual classes.

Exercises

The first two exercises involve modifying classes we covered in this chapter, and the last two exercises involve creating new classes from scratch.

1. Modify the `Point` class (from `Shape.py` or `ShapeAlt.py`), to support the following operations, where `p`, `q`, and `r` are `Points` and `n` is a number:

```
p = q + r   # Point.__add__()
p += q      # Point.__iadd__()
p = q - r   # Point.__sub__()
p -= q      # Point.__isub__()
p = q * n   # Point.__mul__()
p *= n      # Point.__imul__()
p = q / n   # Point.__truediv__()
p /= n      # Point.__itruediv__()
p = q // n  # Point.__floordiv__()
p //= n     # Point.__ifloordiv__()
```

   The in-place methods are all four lines long, including the `def` line, and the other methods are each just two lines long, including the `def` line, and of course they are all very similar and quite simple. With a minimal description and a doctest for each it adds up to around one hundred thirty new lines. A model solution is provided in `Shape_ans.py`; the same code is also in `ShapeAlt_ans.py`.

2. Modify the `Image.py` class to provide a `resize(width, height)` method. If the new width or height is smaller than the current value, any colors outside the new boundaries must be deleted. If either width or height is `None` then use the existing width or height. At the end, make sure you regenerate the `self.__colors` set. Return a Boolean to indicate whether a change was made or not. The method can be implemented in fewer than 20 lines (fewer than 35 including a docstring with a simple doctest). A solution is provided in `Image_ans.py`.

3. Implement a `Transaction` class that takes an amount, a date, a currency (default "USD"—U.S. dollars), a USD conversion rate (default 1), and a description (default `None`). All of the data attributes must be private. Provide the following read-only properties: `amount`, `date`, `currency`, `usd_conversion_rate`, `description`, and `usd` (calculated from `amount * usd_conversion_rate`). This class can be implemented in about sixty lines including some simple doctests. A model solution for this exercise (and the next one) is in file `Account.py`.

4. Implement an `Account` class that holds an account number, an account name, and a list of `Transactions`. The number should be a read-only prop-

erty; the name should be a read-write property with an assertion to ensure that the name is at least four characters long. The class should support the built-in `len()` function (returning the number of transactions), and should provide two calculated read-only properties: `balance` which should return the account's balance in USD and `all_usd` which should return `True` if all the transactions are in USD and `False` otherwise. Three other methods should be provided: `apply()` to apply (add) a transaction, `save()`, and `load()`. The `save()` and `load()` methods should use a binary pickle with the filename being the account number with extension `.acc`; they should save and load the account number, the name, and all the transactions. This class can be implemented in about ninety lines with some simple doctests that include saving and loading—use code such as `name = os.path.join(tempfile.gettempdir(), account_name)` to provide a suitable temporary filename, and make sure you delete the temporary file after the tests have finished. A model solution is in file `Account.py`.