CHAPTER 7

More Abstraction

In the previous chapters, you looked at Python's main built-in object types (numbers, strings, lists, tuples, and dictionaries); you peeked at the wealth of built-in functions and standard libraries; and you even created your own functions. Now, only one thing seems to be missing—making your own objects. And that's what you do in this chapter.

You may wonder how useful this is. It might be cool to make your own kinds of objects, but what would you use them for? With all the dictionaries and sequences and numbers and strings available, can't you just use them and make the functions do the job? Certainly, but making your own objects (and especially types or *classes* of objects) is a central concept in Python—so central, in fact, that Python is called an *object-oriented* language (along with Smalltalk, C++, Java, and many others). In this chapter, you learn how to make objects. You learn about polymorphism and encapsulation, methods and attributes, superclasses, and inheritance— you learn a lot. So let's get started.

Note If you're already familiar with the concepts of object-oriented programming, you probably know about *constructors*. Constructors will not be dealt with in this chapter; for a full discussion, see Chapter 9.

The Magic of Objects

In object-oriented programming, the term *object* loosely means a collection of data (attributes) with a set of methods for accessing and manipulating those data. There are several reasons for using objects instead of sticking with global variables and functions. Some of the most important benefits of objects include the following:

- **Polymorphism**: You can use the same operations on objects of different classes, and they will work as if "by magic."
- **Encapsulation**: You hide unimportant details of how objects work from the outside world.
- **Inheritance**: You can create specialized classes of objects from general ones.

In many presentations of object-oriented programming, the order of these concepts is different. Encapsulation and inheritance are presented first, and then they are used to model real-world objects. That's all fine and dandy, but in my opinion, the most interesting feature of

object-oriented programming is polymorphism. It is also the feature that confuses most people (in my experience). Therefore I'll start with polymorphism, and try to show that this concept alone should be enough to make you like object-oriented programming.

Polymorphism

The term *polymorphism* is derived from a Greek word meaning "having multiple forms." Basically, that means that even if you don't know what kind of object a variable refers to, you may still be able to perform operations on it that will work differently depending on the type (or class) of the object. For example, assume that you are creating an online payment system for a commercial web site that sells food. Your program receives a "shopping cart" of goods from another part of the system (or other similar systems that may be designed in the future)—all you need to worry about is summing up the total and billing some credit card.

Your first thought may be to specify exactly how the goods must be represented when your program receives them. For example, you may want to receive them as tuples, like this:

```
('SPAM', 2.50)
```

If all you need is a descriptive tag and a price, this is fine. But it's not very flexible. Let's say that some clever person starts an auctioning service as part of the web site—where the price of an item is gradually reduced until someone buys it. It would be nice if the user could put the object in her shopping cart, proceed to the checkout (your part of the system), and just wait until the price was right before clicking the Pay button.

But that wouldn't work with the simple tuple scheme. For that to work, the object would need to check its current price (through some network magic) each time your code asked for the price—it couldn't be frozen like in a tuple. You can solve that by making a function:

```
# Don't do it like this...
def getPrice(object):
    if isinstance(object, tuple):
        return object[1]
    else:
        return magic_network_method(object)
```

■ **Note** The type/class checking and use of `isinstance` here is meant to illustrate a point—namely that type checking isn't generally a satisfactory solution. Avoid type checking if you possibly can. The function `isinstance` is described in the section "Investigating Inheritance," later in this chapter.

In the preceding code, I use the function `isinstance` to find out whether the object is a tuple. If it is, its second element is returned; otherwise, some "magic" network method is called.

Assuming that the network stuff already exists, you've solved the problem—for now. But this still isn't very flexible. What if some clever programmer decides that she'll represent the price as a string with a hex value, stored in a dictionary under the key `'price'`? No problem—you just update your function:

```
# Don't do it like this...
def getPrice(object):
    if isinstance(object, tuple):
        return object[1]
    elif isinstance(object, dict):
        return int(object['price'])
    else:
        return magic_network_method(object)
```

Now, surely you must have covered every possibility? But let's say someone decides to add a new type of dictionary with the price stored under a different key. What do you do now? You could certainly update `getPrice` again, but for how long could you continue doing that? Every time someone wanted to implement some priced object differently, you would need to reimplement your module. But what if you already sold your module and moved on to other, cooler projects—what would the client do then? Clearly, this is an inflexible and impractical way of coding the different behaviors.

So what do you do instead? You let the objects handle the operation themselves. It sounds really obvious, but think about how much easier things will get. Every new object type can retrieve or calculate its own price and return it to you—all you have to do is ask for it. And this is where polymorphism (and, to some extent, encapsulation) enters the scene.

Polymorphism and Methods

You receive an object and have no idea of how it is implemented—it may have any one of many "shapes." All you know is that you can ask for its price, and that's enough for you. The way you do that should be familiar:

```
>>> object.getPrice()
2.5
```

Functions that are bound to object attributes like this are called *methods*. You've already encountered them in the form of string, list, and dictionary methods. There, too, you saw some polymorphism:

```
>>> 'abc'.count('a')
1
>>> [1, 2, 'a'].count('a')
1
```

If you had a variable `x`, you wouldn't need to know whether it was a string or a list to call the `count` method—it would work regardless (as long as you supplied a single character as the argument).

Let's do an experiment. The standard library `random` contains a function called `choice` that selects a random element from a sequence. Let's use that to give your variable a value:

```
>>> from random import choice
>>> x = choice(['Hello, world!', [1, 2, 'e', 'e', 4]])
```

After performing this, `x` can either contain the string `'Hello, world!'` *or* the list `[1, 2, 'e', 'e', 4]`—you don't know, and you don't have to worry about it. All you care about is how many times you find `'e'` in `x`, and you can find that out regardless of whether `x` is a list or a string. By calling the `count` method as before, you find out just that:

```
>>> x.count('e')
2
```

In this case, it seems that the list won out. But the point is that you didn't need to check. Your only requirement was that `x` has a method called `count` that takes a single character as an argument and returned an integer. If someone else had made his own class of objects that had this method, it wouldn't matter to you—you could use his objects just as well as the strings and lists.

Polymorphism Comes in Many Forms

Polymorphism is at work every time you can "do something" to an object without having to know exactly what kind of object it is. This doesn't apply only to methods—we've already used polymorphism a lot in the form of built-in operators and functions. Consider the following:

```
>>> 1+2
3
>>> 'Fish'+'license'
'Fishlicense'
```

Here, the plus operator (`+`) works fine for both numbers (integers in this case) and strings (as well as other types of sequences). To illustrate the point, let's say you wanted to make a function called `add` that added two things together. You could simply define it like this (equivalent to, but less efficient than, the `add` function from the `operator` module):

```
def add(x, y):
    return x+y
```

This would also work with many kinds of arguments:

```
>>> add(1, 2)
3
>>> add('Fish', 'license')
'Fishlicense'
```

This might seem silly, but the point is that the arguments can be *anything that supports addition*.[1] If you want to write a function that prints a message about the length of an object, all that's required is that it *has* a length (that the `len` function will work on it):

```
def length_message(x):
    print "The length of", repr(x), "is", len(x)
```

As you can see, the function also uses `repr`, but `repr` is one of the grand masters of polymorphism—it works with anything. Let's see how:

```
>>> length_message('Fnord')
The length of 'Fnord' is 5
>>> length_message([1, 2, 3])
The length of [1, 2, 3] is 3
```

Many functions and operators are polymorphic—probably most of yours will be, too, even if you don't intend them to be. Just by using polymorphic functions and operators, the polymorphism "rubs off." In fact, virtually the only thing you can do to destroy this polymorphism is to do explicit type checking with functions such as `type`, `` ` ``, and `issubclass`. If you can, you *really* should avoid destroying polymorphism this way. What matters should be that an object acts the way you want, not whether it is of the right type (or class).

■ **Note** The form of polymorphism discussed here, which is so central to the Python way of programming, is sometimes called "duck typing." The term derives from the phrase, "If it quacks like a duck ..." For more information, see `http://en.wikipedia.org/wiki/Duck_typing`.

Encapsulation

Encapsulation is the principle of hiding unnecessary details from the rest of the world. This may sound like polymorphism—there, too, you use an object without knowing its inner details. The two concepts are similar because they are both *principles of abstraction*. They both help you deal with the components of your program without caring about unnecessary detail, just as functions do.

But encapsulation isn't the same as polymorphism. Polymorphism enables you to call the methods of an object without knowing its class (type of object). Encapsulation enables you to use the object without worrying about how it's constructed. Does it still sound similar? Let's construct an example *with* polymorphism, but *without* encapsulation. Assume that you have a class called `OpenObject` (you learn how to create classes later in this chapter):

```
>>> o = OpenObject() # This is how we create objects...
>>> o.setName('Sir Lancelot')
>>> o.getName()
'Sir Lancelot'
```

1. Note that these objects need to support addition with each other. So calling `add(1, 'license')` would not work.

You create an object (by calling the class as if it were a function) and bind the variable `o` to it. You can then use the methods `setName` and `getName` (assuming that they are methods that are supported by the class `OpenObject`). Everything seems to be working perfectly. However, let's assume that `o` stores its name in the global variable `globalName`:

```
>>> globalName
'Sir Lancelot'
```

This means that you need to worry about the contents of `globalName` when you use instances (objects) of the class `OpenObject`. In fact, you must make sure that no one changes it:

```
>>> globalName = 'Sir Gumby'
>>> o.getName()
'Sir Gumby'
```

Things get even more problematic if you try to create more than one `OpenObject` because they will all be messing with the same variable:

```
>>> o1 = OpenObject()
>>> o2 = OpenObject()
>>> o1.setName('Robin Hood')
>>> o2.getName()
'Robin Hood'
```

As you can see, setting the name of one automatically sets the name of the other—not exactly what you want.

Basically, you want to treat objects as abstract. When you call a method, you don't want to worry about anything else, such as not disturbing global variables. So how can you "encapsulate" the name within the object? No problem. You make it an *attribute*.

Attributes are variables that are a part of the object, just like methods; actually, methods are almost like attributes bound to functions. (You'll see an important difference between methods and functions in the section "Attributes, Functions, and Methods," later in this chapter.) If you rewrite the class to use an attribute instead of a global variable, and you rename it `ClosedObject`, it works like this:

```
>>> c = ClosedObject()
>>> c.setName('Sir Lancelot')
>>> c.getName()
'Sir Lancelot'
```

So far, so good. But for all you know, this could still be stored in a global variable. Let's make another object:

```
>>> r = ClosedObject()
>>> r.setName('Sir Robin')
r.getName()
'Sir Robin'
```

Here, you can see that the new object has its name set properly, which is probably what you expected. But what has happened to the first object now?

```
>>> c.getName()
'Sir Lancelot'
```

The name is still there! This is because the object has its own *state*. The state of an object is described by its attributes (like its name, for example). The methods of an object may change these attributes. So it's like lumping together a bunch of functions (the methods) and giving them access to some variables (the attributes) where they can keep values stored between function calls.

You'll see even more details on Python's encapsulation mechanisms in the section "Privacy Revisited," later in the chapter.

Inheritance

Inheritance is another way of dealing with laziness (in the positive sense). Programmers want to avoid typing the same code more than once. We avoided that earlier by making functions, but now I will address a more subtle problem. What if you have a class already, and you want to make one that is very similar? Perhaps one that adds only a few methods? When making this new class, you don't want to need to copy all the code from the old one over to the new one.

For example, you may already have a class called `Shape`, which knows how to draw itself on the screen. Now you want to make a class called `Rectangle`, which *also* knows how to draw itself on the screen, but which can, in addition, calculate its own area. You wouldn't want to do all the work of making a new `draw` method when `Shape` has one that works just fine. So what do you do? You let `Rectangle` *inherit* the methods from `Shape`. You can do this in such a way that when `draw` is called on a `Rectangle` object, the method from the `Shape` class is called automatically (see the section "Specifying a Superclass," later in this chapter).

Classes and Types

By now, you're getting a feeling for what classes are—or you *may* be getting impatient for me to tell you how to make the darn things. Before jumping into the technicalities, let's have a look at what a class is, and how it is different from (or similar to) a type.

What *Is* a Class, Exactly?

I've been throwing around the word *class* a lot, using it more or less synonymously with words such as *kind* or *type*. In many ways that's exactly what a class is—a kind of object. All objects *belong* to a class and are said to be *instances* of that class.

So, for example, if you look outside your window and see a bird, that bird is an instance of the class "birds." This is a very general (abstract) class that has several *sub*classes; your bird might belong to the subclass "larks." You can think of the class "birds" as the set of all birds, while the class "larks" is just a subset of that. When the objects belonging to one class form a subset of the objects belonging to another class, the first is called a *subclass* of the second. Thus, "larks" is a subclass of "birds." Conversely, "birds" is a *superclass* of "larks."

■ **Note** In everyday speech, we denote classes of objects with plural nouns such as "birds" and "larks." In Python, it is customary to use singular, capitalized nouns such as `Bird` and `Lark`.

When stated like this, subclasses and superclasses are easy to understand. But in object-oriented programming, the subclass relation has important implications because a class is defined by what methods it supports. All the instances of a class have these methods, so all the instances of all *subclasses* must *also* have them. Defining subclasses is then only a matter of defining *more* methods (or, perhaps, overriding some of the existing ones).

For example, `Bird` might supply the method `fly`, while `Penguin` (a subclass of `Bird`) might add the method `eatFish`. When making a `Penguin` class, you would probably also want to *override* a method of the superclass, namely the `fly` method. In a `Penguin` instance, this method should either do nothing, or possibly raise an exception (see Chapter 8), given that penguins can't fly.

■ **Note** In older versions of Python, there was a sharp distinction between types and classes. Built-in objects had types; your custom objects had classes. You could create classes, but not types. In recent versions of Python, things are starting to change. The division between basic types and classes is blurring. You can now make subclasses (or subtypes) of the built-in types, and the types are behaving more like classes. Chances are you won't notice this change much until you become more familiar with the language. If you're interested, you can find more information on the topic in Chapter 9.

Making Your Own Classes

Finally, you get to make your own classes! Here is a simple example:

```
__metaclass__ = type # Make sure we get new style classes

class Person:

    def setName(self, name):
        self.name = name

    def getName(self):
        return self.name

    def greet(self):
        print "Hello, world! I'm %s." % self.name
```

■ **Note** There is a difference between so-called *old-style* and *new-style* classes. There is really no reason to use the old-style classes anymore, except that they're what you get by default in Python versions prior to 3.0. To get new-style classes, you should place the assignment `__metaclass__ = type` at the beginning of your script or module. (I may not explicitly include this statement in every example.) There are also other solutions, such as subclassing a new-style class (for example, `object`). You learn more about subclassing in a minute. In Python 3.0, there is no need to worry about this, as old-style classes don't exist there. You find more information about this in Chapter 9.

This example contains three method definitions, which are like function definitions except that they are written inside a `class` statement. `Person` is, of course, the name of the class. The `class` statement creates its own namespace where the functions are defined. (See the section "The Class Namespace" later in this chapter.) All this seems fine, but you may wonder what this `self` parameter is. It refers to the object itself. And what object is that? Let's make a couple of instances and see:

```
>>> foo = Person()
>>> bar = Person()
>>> foo.setName('Luke Skywalker')
>>> bar.setName('Anakin Skywalker')
>>> foo.greet()
Hello, world! I'm Luke Skywalker.
>>> bar.greet()
Hello, world! I'm Anakin Skywalker.
```

Okay, so this example may be a bit obvious, but perhaps it clarifies what `self` is. When I call `setName` and `greet` on `foo`, `foo` itself is automatically passed as the first parameter in each case—the parameter that I have so fittingly called `self`. You may, in fact, call it whatever you like, but because it is always the object itself, it is almost always called `self`, by convention.

It should be obvious why `self` is useful, and even necessary here. Without it, none of the methods would have access to the object itself—the object whose attributes they are supposed to manipulate. As before, the attributes are also accessible from the outside:

```
>>> foo.name
'Luke Skywalker'
>>> bar.name = 'Yoda'
>>> bar.greet()
Hello, world! I'm Yoda.
```

■ **Tip** Another way of viewing this is that `foo.greet()` is simply a convenient way of writing `Person.greet(foo)`, if you know that `foo` is an instance of `Person`.

Attributes, Functions, and Methods

The `self` parameter (mentioned in the previous section) is, in fact, what distinguishes methods from functions. Methods (or, more technically, *bound* methods) have their first parameter bound to the instance they belong to, so you don't have to supply it. While you can certainly bind an attribute to a plain function, it won't have that special `self` parameter:

```
>>> class Class:
        def method(self):
            print 'I have a self!'

>>> def function():
        print "I don't..."

>>> instance = Class()
>>> instance.method()
I have a self!
>>> instance.method = function
>>> instance.method()
I don't...
```

Note that the `self` parameter is not dependent on calling the method the way I've done until now, as `instance.method`. You're free to use another variable that refers to the same method:

```
>>> class Bird:
        song = 'Squaawk!'
        def sing(self):
            print self.song

>>> bird = Bird()
>>> bird.sing()
Squaawk!
>>> birdsong = bird.sing
>>> birdsong()
Squaawk!
```

Even though the last method call looks exactly like a function call, the variable `birdsong` refers to the bound method `bird.sing`, which means that it still has access to the `self` parameter (that is, it is still bound to the same instance of the class).

Privacy Revisited

By default, you can access the attributes of an object from the "outside." Let's revisit the example from the earlier discussion on encapsulation:

```
>>> c.name
'Sir Lancelot'
>>> c.name = 'Sir Gumby'
```

```
>>> c.getName()
'Sir Gumby'
```

Some programmers are okay with this, but some (like the creators of Smalltalk, a language where attributes of an object are accessible only to the methods of the same object) feel that it breaks with the principle of encapsulation. They believe that the state of the object should be *completely hidden* (inaccessible) to the outside world. You might wonder why they take such an extreme stand. Isn't it enough that each object manages its own attributes? Why should you hide them from the world? After all, if you just used the `name` attribute directly on `ClosedObject` (the class of `c` in this case), you wouldn't need to make the `setName` and `getName` methods.

The point is that other programmers may not know (and perhaps shouldn't know) what's going on inside your object. For example, `ClosedObject` may send an email message to some administrator every time an object changes its name. This could be part of the `setName` method. But what happens when you set `c.name` directly? Nothing happens—no email message is sent. To avoid this sort of thing, you have *private* attributes. These are attributes that are not accessible outside the object; they are accessible only through *accessor* methods, such as `getName` and `setName`.

■ **Note** In Chapter 9, you learn about *properties*, a powerful alternative to accessors.

Python doesn't support privacy directly, but relies on the programmer to know when it is safe to modify an attribute from the outside. After all, you should know how to use an object before using that object. It *is*, however, possible to achieve something like private attributes with a little trickery.

To make a method or attribute private (inaccessible from the outside), simply start its name with two underscores:

```
class Secretive:

    def __inaccessible(self):
        print "Bet you can't see me..."

    def accessible(self):
        print "The secret message is:"
        self.__inaccessible()
```

Now `__inaccessible` is inaccessible to the outside world, while it can still be used inside the class (for example, from `accessible`):

```
>>> s = Secretive()
>>> s.__inaccessible()
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in ?
    s.__inaccessible()
AttributeError: Secretive instance has no attribute '__inaccessible'
```

```
>>> s.accessible()
The secret message is:
Bet you can't see me...
```

Although the double underscores are a bit strange, this seems like a standard private method, as found in other languages. What's not so standard is what actually happens. Inside a class definition, all names beginning with a double underscore are "translated" by adding a single underscore and the class name to the beginning:

```
>>> Secretive._Secretive__inaccessible
<unbound method Secretive.__inaccessible>
```

If you know how this works behind the scenes, it is still possible to access private methods outside the class, even though you're not supposed to:

```
>>> s._Secretive__inaccessible()
Bet you can't see me...
```

So, in short, you can't be sure that others won't access the methods and attributes of your objects, but this sort of name-mangling is a pretty strong signal that they *shouldn't*.

If you don't want the name-mangling effect, but you still want to send a signal for other objects to stay away, you can use a *single* initial underscore. This is mostly just a convention, but has some practical effects. For example, names with an initial underscore aren't imported with starred imports (`from module import *`).[2]

The Class Namespace

The following two statements are (more or less) equivalent:

```
def foo(x): return x*x
foo = lambda x: x*x
```

Both create a function that returns the square of its argument, and both bind the variable `foo` to that function. The name `foo` may be defined in the global (module) scope, or it may be local to some function or method. The same thing happens when you define a class: all the code in the `class` statement is executed in a special namespace—the *class namespace*. This namespace is accessible later by all members of the class. Not all Python programmers know that class definitions are simply code sections that are executed, but it can be useful information. For example, you aren't restricted to `def` statements inside the class definition block:

```
>>> class C:
        print 'Class C being defined...'

Class C being defined...
>>>
```

2. Some languages support several *degrees* of privacy for its member variables (attributes). Java, for example, has four different levels. Python doesn't really have equivalent privacy support, although single and double initial underscores do to some extent give you two levels of privacy.

Okay, that was a bit silly. But consider the following:

```
class MemberCounter:
    members = 0
    def init(self):
        MemberCounter.members += 1

>>> m1 = MemberCounter()
>>> m1.init()
>>> MemberCounter.members
1
>>> m2 = MemberCounter()
>>> m2.init()
>>> MemberCounter.members
2
```

In the preceding code, a variable is defined in the class scope, which can be accessed by all the members (instances), in this case to count the number of class members. Note the use of `init` to initialize all the instances: I'll automate that (that is, turn it into a proper constructor) in Chapter 9.

This class scope variable is accessible from every instance as well, just as methods are:

```
>>> m1.members
2
>>> m2.members
2
```

What happens when you rebind the `members` attribute in an instance?

```
>>> m1.members = 'Two'
>>> m1.members
'Two'
>>> m2.members
2
```

The new `members` value has been written into an attribute in `m1`, shadowing the class-wide variable. This mirrors the behavior of local and global variables in functions, as discussed in the sidebar "The Problem of Shadowing" in Chapter 6.

Specifying a Superclass

As I discussed earlier in the chapter, subclasses expand on the definitions in their superclasses. You indicate the superclass in a `class` statement by writing it in parentheses after the class name:

```
class Filter:
    def init(self):
        self.blocked = []
    def filter(self, sequence):
        return [x for x in sequence if x not in self.blocked]
```

```
class SPAMFilter(Filter): # SPAMFilter is a subclass of Filter
    def init(self): # Overrides init method from Filter superclass
        self.blocked = ['SPAM']
```

`Filter` is a general class for filtering sequences. Actually it doesn't filter out anything:

```
>>> f = Filter()
>>> f.init()
>>> f.filter([1, 2, 3])
[1, 2, 3]
```

The usefulness of the `Filter` class is that it can be used as a base class (superclass) for other classes, such as `SPAMFilter`, which filters out `'SPAM'` from sequences:

```
>>> s = SPAMFilter()
>>> s.init()
>>> s.filter(['SPAM', 'SPAM', 'SPAM', 'SPAM', 'eggs', 'bacon', 'SPAM'])
['eggs', 'bacon']
```

Note two important points in the definition of `SPAMFilter`:

- I override the definition of `init` from `Filter` by simply providing a new definition.
- The definition of the `filter` method carries over (is inherited) from `Filter`, so you don't need to write the definition again.

The second point demonstrates why inheritance is useful: I can now make a number of different filter classes, all subclassing `Filter`, and for each one I can simply use the `filter` method I have already implemented. Talk about useful laziness . . .

Investigating Inheritance

If you want to find out whether a class is a subclass of another, you can use the built-in method `issubclass`:

```
>>> issubclass(SPAMFilter, Filter)
True
>>> issubclass(Filter, SPAMFilter)
False
```

If you have a class and want to know its base classes, you can access its special attribute `__bases__`:

```
>>> SPAMFilter.__bases__
(<class __main__.Filter at 0x171e40>,)
>>> Filter.__bases__
()
```

In a similar manner, you can check whether an object is an instance of a class by using `isinstance`:

```
>>> s = SPAMFilter()
>>> isinstance(s, SPAMFilter)
True
>>> isinstance(s, Filter)
True
>>> isinstance(s, str)
False
```

■ **Note** Using `isinstance` is usually not good practice. Relying on polymorphism is almost always better.

As you can see, `s` is a (direct) member of the class `SPAMFilter`, but it is also an indirect member of `Filter` because `SPAMFilter` is a subclass of `Filter`. Another way of putting it is that all `SPAMFilters` are `Filters`. As you can see in the preceding example, `isinstance` also works with types, such as the string type (`str`).

If you just want to find out which class an object belongs to, you can use the `__class__` attribute:

```
>>> s.__class__
<class __main__.SPAMFilter at 0x1707c0>
```

■ **Note** If you have a new-style class, either by setting `__metaclass__ = type` or subclassing `object`, you could also use `type(s)` to find the class of your instance.

Multiple Superclasses

I'm sure you noticed a small detail in the previous section that may have seemed odd: the plural form in `__bases__`. I said you could use it to find the base classes of a class, which implies that it may have more than one. This is, in fact, the case. To show how it works, let's create a few classes:

```
class Calculator:
    def calculate(self, expression):
        self.value = eval(expression)

class Talker:
    def talk(self):
        print 'Hi, my value is', self.value

class TalkingCalculator(Calculator, Talker):
    pass
```

The subclass (`TalkingCalculator`) does nothing by itself; it inherits all its behavior from its superclasses. The point is that it inherits both `calculate` from `Calculator` and `talk` from `Talker`, making it a talking calculator:

```
>>> tc = TalkingCalculator()
>>> tc.calculate('1+2*3')
>>> tc.talk()
Hi, my value is 7
```

This is called *multiple inheritance*, and can be a very powerful tool. However, unless you know you need multiple inheritance, you may want to stay away from it, as it can, in some cases, lead to unforeseen complications.

If you are using multiple inheritance, there is one thing you should look out for: if a method is implemented differently by two or more of the superclasses (that is, you have two different methods with the same name), you must be careful about the order of these superclasses (in the class statement). The methods in the earlier classes *override* the methods in the later ones. So if the `Calculator` class in the preceding example had a method called `talk`, it would override (and make inaccessible) the `talk` method of the `Talker`. Reversing their order, like this:

```
class TalkingCalculator(Talker, Calculator): pass
```

would make the `talk` method of the `Talker` accessible. If the superclasses share a common superclass, the order in which the superclasses are visited while looking for a given attribute or method is called the *method resolution order* (MRO), and follows a rather complicated algorithm. Luckily, it works very well, so you probably don't need to worry about it.

Interfaces and Introspection

The "interface" concept is related to polymorphism. When you handle a polymorphic object, you only care about its interface (or "protocol")—the methods and attributes known to the world. In Python, you don't explicitly specify which methods an object needs to have to be acceptable as a parameter. For example, you don't write interfaces explicitly (as you do in Java); you just assume that an object can do what you ask it to do. If it can't, the program will fail.

Usually, you simply require that objects conform to a certain interface (in other words, implement certain methods), but if you want to, you can be quite flexible in your demands. Instead of just calling the methods and hoping for the best, you can check whether the required methods are present, and if not, perhaps do something else:

```
>>> hasattr(tc, 'talk')
True
>>> hasattr(tc, 'fnord')
False
```

In the preceding code, you find that `tc` (a `TalkingCalculator`, as described earlier in this chapter) has the attribute `talk` (which refers to a method), but not the attribute `fnord`. If you wanted to, you could even check whether the `talk` attribute was callable:

```
>>> callable(getattr(tc, 'talk', None))
True
```

```
>>> callable(getattr(tc, 'fnord', None))
False
```

■ **Note** The function `callable` is no longer available in Python 3.0. Instead of `callable(x)`, you can use `hasattr(x, '__call__')`.

Note that instead of using `hasattr` in an `if` statement and accessing the attribute directly, I'm using `getattr`, which allows me to supply a default value (in this case `None`) that will be used if the attribute is not present. I then use `callable` on the returned object.

■ **Note** The inverse of `getattr` is `setattr`, which can be used to set the attributes of an object:

```
>>> setattr(tc, 'name', 'Mr. Gumby')
>>> tc.name
'Mr. Gumby'
```

If you want to see all the values stored in an object, you can examine its `__dict__` attribute. And if you *really* want to find out what an object is made of, you should take a look at the `inspect` module. It is meant for fairly advanced users who want to make object browsers (programs that enable you to browse Python objects in a graphical manner) and other similar programs that require such functionality. For more information on exploring objects and modules, see the section "Exploring Modules" in Chapter 10.

Some Thoughts on Object-Oriented Design

Many books have been written about object-oriented program design, and although that's not the focus of this book, I'll give you some pointers:

- Gather what belongs together. If a function manipulates a global variable, the two of them might be better off in a class, as an attribute and a method.
- Don't let objects become too intimate. Methods should mainly be concerned with the attributes of their own instance. Let other instances manage their own state.
- Go easy on the inheritance, *especially* multiple inheritance. Inheritance is useful at times, but can make things unnecessarily complex in some cases. And multiple inheritance can be very difficult to get right and even harder to debug.
- Keep it simple. Keep your methods small. As a rule of thumb, it should be possible to read (and understand) most of your methods in, say, 30 seconds. For the rest, try to keep them shorter than one page or screen.

When determining which classes you need and which methods they should have, you may try something like this:

1. Write down a description of your problem (what should the program do?). Underline all the nouns, verbs, and adjectives.
2. Go through the nouns, looking for potential classes.
3. Go through the verbs, looking for potential methods.
4. Go through the adjectives, looking for potential attributes.
5. Allocate methods and attributes to your classes.

Now you have a first sketch of an *object-oriented model*. You may also want to think about what responsibilities and relationships (such as inheritance or cooperation) the classes and objects will have. To refine your model, you can do the following:

1. Write down (or dream up) a set of *use cases*—scenarios of how your program may be used. Try to cover all the functionality.
2. Think through every use case step by step, making sure that everything you need is covered by your model. If something is missing, add it. If something isn't quite right, change it. Continue until you are satisfied.

When you have a model you think will work, you can start hacking away. Chances are you'll need to revise your model or revise parts of your program. Luckily, that's easy in Python, so don't worry about it. Just dive in. (If you would like some more guidance in the ways of object-oriented programming, check out the list of suggested books in Chapter 19.)

A Quick Summary

This chapter has given you more than just information about the Python language; it has introduced you to several concepts that may have been completely foreign to you. Here's a summary:

Objects: An object consists of attributes and methods. An attribute is merely a variable that is part of an object, and a method is more or less a function that is stored in an attribute. One difference between (bound) methods and other functions is that methods always receive the object they are part of as their first argument, usually called `self`.

Classes: A class represents a set (or kind) of objects, and every object (instance) has a class. The class's main task is to define the methods its instances will have.

Polymorphism: Polymorphism is the characteristic of being able to treat objects of different types and classes alike—you don't need to know which class an object belongs to in order to call one of its methods.

Encapsulation: Objects may hide (or encapsulate) their internal state. In some languages, this means that their state (their attributes) is available only through their methods. In

Python, all attributes are publicly available, but programmers should still be careful about accessing an object's state directly, since they might unwittingly make the state inconsistent in some way.

Inheritance: One class may be the subclass of one or more other classes. The subclass then inherits all the methods of the superclasses. You can use more than one superclass, and this feature can be used to compose orthogonal (independent and unrelated) pieces of functionality. A common way of implementing this is using a core superclass along with one or more *mix-in* superclasses.

Interfaces and introspection: In general, you don't want to prod an object too deeply. You rely on polymorphism, and call the methods you need. However, if you want to find out what methods or attributes an object has, there are functions that will do the job for you.

Object-oriented design: There are many opinions about how (or whether!) to do object-oriented design. No matter where you stand on the issue, it's important to understand your problem thoroughly, and to create a design that is easy to understand.

New Functions in This Chapter

Function	Description
`callable(object)`	Determines if the object is callable (such as a function or a method)
`getattr(object, name[, default])`	Gets the value of an attribute, optionally providing a default
`hasattr(object, name)`	Determines if the object has the given attribute
`isinstance(object, class)`	Determines if the object is an instance of the class
`issubclass(A, B)`	Determines if `A` is a subclass of `B`
`random.choice(sequence)`	Chooses a random element from a nonempty sequence
`setattr(object, name, value)`	Sets the given attribute of the object to `value`
`type(object)`	Returns the type of the object

What Now?

You've learned a lot about creating your own objects and how useful that can be. Before diving headlong into the magic of Python's special methods (Chapter 9), let's take a breather with a little chapter about exception handling.