PART VI

Classes and OOP

CHAPTER 25
OOP: The Big Picture

So far in this book, we've been using the term "object" generically. Really, the code written up to this point has been *object-based*—we've passed objects around our scripts, used them in expressions, called their methods, and so on. For our code to qualify as being truly *object-oriented* (OO), though, our objects will generally need to also participate in something called an *inheritance hierarchy*.

This chapter begins our exploration of the Python *class*—a device used to implement new kinds of objects in Python that support inheritance. Classes are Python's main object-oriented programming (OOP) tool, so we'll also look at OOP basics along the way in this part of the book. OOP offers a different and often more effective way of looking at programming, in which we factor code to minimize redundancy, and write new programs by *customizing* existing code instead of changing it in-place.

In Python, classes are created with a new statement: the `class` statement. As you'll see, the objects defined with classes can look a lot like the built-in types we studied earlier in the book. In fact, classes really just apply and extend the ideas we've already covered; roughly, they are packages of functions that use and process built-in object types. Classes, though, are designed to create and manage new objects, and they also support *inheritance*—a mechanism of code customization and reuse above and beyond anything we've seen so far.

One note up front: in Python, OOP is entirely optional, and you don't need to use classes just to get started. In fact, you can get plenty of work done with simpler constructs such as functions, or even simple top-level script code. Because using classes well requires some up-front planning, they tend to be of more interest to people who work in *strategic* mode (doing long-term product development) than to people who work in *tactical* mode (where time is in very short supply).

Still, as you'll see in this part of the book, classes turn out to be one of the most useful tools Python provides. When used well, classes can actually cut development time radically. They're also employed in popular Python tools like the tkinter GUI API, so most Python programmers will usually find at least a working knowledge of class basics helpful.

Why Use Classes?

Remember when I told you that programs "do things with stuff"? In simple terms, classes are just a way to define new sorts of stuff, reflecting real objects in a program's domain. For instance, suppose we decide to implement that hypothetical pizza-making robot we used as an example in Chapter 16. If we implement it using classes, we can model more of its real-world structure and relationships. Two aspects of OOP prove useful here:

Inheritance
: Pizza-making robots are kinds of robots, so they possess the usual robot-y properties. In OOP terms, we say they "inherit" properties from the general category of all robots. These common properties need to be implemented only once for the general case and can be reused by all types of robots we may build in the future.

Composition
: Pizza-making robots are really collections of components that work together as a team. For instance, for our robot to be successful, it might need arms to roll dough, motors to maneuver to the oven, and so on. In OOP parlance, our robot is an example of composition; it contains other objects that it activates to do its bidding. Each component might be coded as a class, which defines its own behavior and relationships.

General OOP ideas like inheritance and composition apply to any application that can be decomposed into a set of objects. For example, in typical GUI systems, interfaces are written as collections of widgets—buttons, labels, and so on—which are all drawn when their container is drawn (*composition*). Moreover, we may be able to write our own custom widgets—buttons with unique fonts, labels with new color schemes, and the like—which are specialized versions of more general interface devices (*inheritance*).

From a more concrete programming perspective, classes are Python program units, just like functions and modules: they are another compartment for packaging logic and data. In fact, classes also define new namespaces, much like modules. But, compared to other program units we've already seen, classes have three critical distinctions that make them more useful when it comes to building new objects:

Multiple instances
: Classes are essentially factories for generating one or more objects. Every time we call a class, we generate a new object with a distinct namespace. Each object generated from a class has access to the class's attributes *and* gets a namespace of its own for data that varies per object.

Customization via inheritance
: Classes also support the OOP notion of inheritance; we can extend a class by redefining its attributes outside the class itself. More generally, classes can build up namespace hierarchies, which define names to be used by objects created from classes in the hierarchy.

Operator overloading
: By providing special protocol methods, classes can define objects that respond to the sorts of operations we saw at work on built-in types. For instance, objects made with classes can be sliced, concatenated, indexed, and so on. Python provides hooks that classes can use to intercept and implement any built-in type operation.

OOP from 30,000 Feet

Before we see what this all means in terms of code, I'd like to say a few words about the general ideas behind OOP. If you've never done anything object-oriented in your life before now, some of the terminology in this chapter may seem a bit perplexing on the first pass. Moreover, the motivation for these terms may be elusive until you've had a chance to study the ways that programmers apply them in larger systems. OOP is as much an experience as a technology.

Attribute Inheritance Search

The good news is that OOP is much simpler to understand and use in Python than in other languages, such as C++ or Java. As a dynamically typed scripting language, Python removes much of the syntactic clutter and complexity that clouds OOP in other tools. In fact, most of the OOP story in Python boils down to this expression:

```
object.attribute
```

We've been using this expression throughout the book to access module attributes, call methods of objects, and so on. When we say this to an object that is derived from a `class` statement, however, the expression kicks off a *search* in Python—it searches a tree of linked objects, looking for the first appearance of `attribute` that it can find. When classes are involved, the preceding Python expression effectively translates to the following in natural language:

> Find the first occurrence of `attribute` by looking in `object`, then in all classes above it, from bottom to top and left to right.

In other words, attribute fetches are simply tree searches. The term *inheritance* is applied because objects lower in a tree inherit attributes attached to objects higher in that tree. As the search proceeds from the bottom up, in a sense, the objects linked into a tree are the union of all the attributes defined in all their tree parents, all the way up the tree.

In Python, this is all very literal: we really do build up trees of linked objects with code, and Python really does climb this tree at runtime searching for attributes every time we use the `object.attribute` expression. To make this more concrete, Figure 25-1 sketches an example of one of these trees.

In this figure, there is a tree of five objects labeled with variables, all of which have attached attributes, ready to be searched. More specifically, this tree links together three

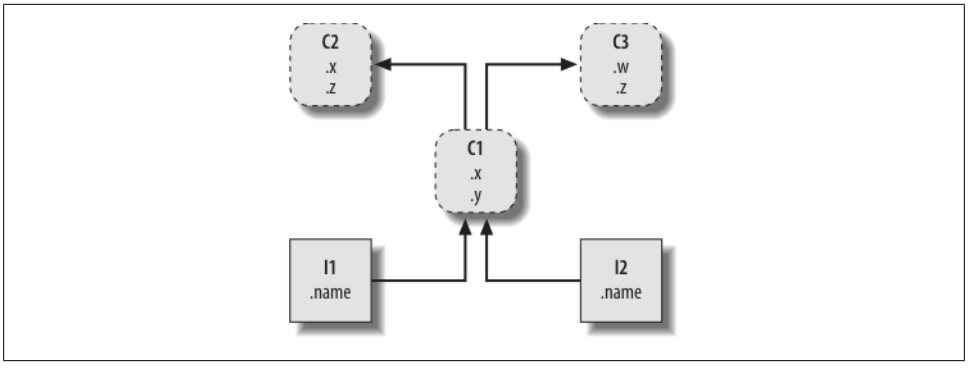

Figure 25-1. A class tree, with two instances at the bottom (I1 and I2), a class above them (C1), and two superclasses at the top (C2 and C3). All of these objects are namespaces (packages of variables), and the inheritance search is simply a search of the tree from bottom to top looking for the lowest occurrence of an attribute name. Code implies the shape of such trees.

class objects (the ovals `C1`, `C2`, and `C3`) and two *instance objects* (the rectangles `I1` and `I2`) into an inheritance search tree. Notice that in the Python object model, classes and the instances you generate from them are two distinct object types:

Classes
: Serve as instance factories. Their attributes provide behavior—data and functions—that is inherited by all the instances generated from them (e.g., a function to compute an employee's salary from pay and hours).

Instances
: Represent the concrete items in a program's domain. Their attributes record data that varies per specific object (e.g., an employee's Social Security number).

In terms of search trees, an instance inherits attributes from its class, and a class inherits attributes from all classes above it in the tree.

In Figure 25-1, we can further categorize the ovals by their relative positions in the tree. We usually call classes higher in the tree (like `C2` and `C3`) *superclasses*; classes lower in the tree (like `C1`) are known as *subclasses*.* These terms refer to relative tree positions and roles. Superclasses provide behavior shared by all their subclasses, but because the search proceeds from the bottom up, subclasses may override behavior defined in their superclasses by redefining superclass names lower in the tree.

As these last few words are really the crux of the matter of software customization in OOP, let's expand on this concept. Suppose we build up the tree in Figure 25-1, and then say this:

```
I2.w
```

* In other literature, you may also occasionally see the terms *base classes* and *derived classes* used to describe superclasses and subclasses, respectively.

Right away, this code invokes inheritance. Because this is an *`object.attribute`* expression, it triggers a search of the tree in Figure 25-1—Python will search for the attribute `w` by looking in `I2` and above. Specifically, it will search the linked objects in this order:

```
I2, C1, C2, C3
```

and stop at the first attached `w` it finds (or raise an error if `w` isn't found at all). In this case, `w` won't be found until `C3` is searched because it appears only in that object. In other words, `I2.w` resolves to `C3.w` by virtue of the automatic search. In OOP terminology, `I2` "inherits" the attribute `w` from `C3`.

Ultimately, the two instances inherit four attributes from their classes: `w`, `x`, `y`, and `z`. Other attribute references will wind up following different paths in the tree. For example:

- `I1.x` and `I2.x` both find `x` in `C1` and stop because `C1` is lower than `C2`.
- `I1.y` and `I2.y` both find `y` in `C1` because that's the only place `y` appears.
- `I1.z` and `I2.z` both find `z` in `C2` because `C2` is further to the left than `C3`.
- `I2.name` finds `name` in `I2` without climbing the tree at all.

Trace these searches through the tree in Figure 25-1 to get a feel for how inheritance searches work in Python.

The first item in the preceding list is perhaps the most important to notice—because `C1` redefines the attribute `x` lower in the tree, it effectively *replaces* the version above it in `C2`. As you'll see in a moment, such redefinitions are at the heart of software customization in OOP—by redefining and replacing the attribute, `C1` effectively customizes what it inherits from its superclasses.

Classes and Instances

Although they are technically two separate object types in the Python model, the classes and instances we put in these trees are almost identical—each type's main purpose is to serve as another kind of *namespace*—a package of variables, and a place where we can attach attributes. If classes and instances therefore sound like modules, they should; however, the objects in class trees also have automatically searched links to other namespace objects, and classes correspond to statements, not entire files.

The primary difference between classes and instances is that classes are a kind of *factory* for generating instances. For example, in a realistic application, we might have an `Employee` class that defines what it means to be an employee; from that class, we generate actual `Employee` instances. This is another difference between classes and modules: we only ever have one instance of a given module in memory (that's why we have to reload a module to get its new code), but with classes, we can make as many instances as we need.

Operationally, classes will usually have functions attached to them (e.g., `computeSalary`), and the instances will have more basic data items used by the class' functions (e.g., `hoursWorked`). In fact, the object-oriented model is not that different from the classic data-processing model of *programs* plus *records*; in OOP, instances are like records with "data," and classes are the "programs" for processing those records. In OOP, though, we also have the notion of an inheritance hierarchy, which supports software customization better than earlier models.

Class Method Calls

In the prior section, we saw how the attribute reference `I2.w` in our example class tree was translated to `C3.w` by the inheritance search procedure in Python. Perhaps just as important to understand as the inheritance of attributes, though, is what happens when we try to call methods (i.e., functions attached to classes as attributes).

If this `I2.w` reference is a *function* call, what it really means is "call the `C3.w` function to process `I2`." That is, Python will automatically map the call `I2.w()` into the call `C3.w(I2)`, passing in the instance as the first argument to the inherited function.

In fact, whenever we call a function attached to a class in this fashion, an instance of the class is always implied. This implied subject or context is part of the reason we refer to this as an *object-oriented* model—there is always a subject object when an operation is run. In a more realistic example, we might invoke a method called `giveRaise` attached as an attribute to an `Employee` class; such a call has no meaning unless qualified with the employee to whom the raise should be given.

As we'll see later, Python passes in the implied instance to a special first argument in the method, called `self` by convention. As we'll also learn, methods can be called through either an instance (e.g., `bob.giveRaise()`) or a class (e.g., `Employee.giveRaise(bob)`), and both forms serve purposes in our scripts. To see how methods receive their subjects, though, we need to move on to some code.

Coding Class Trees

Although we are speaking in the abstract here, there is tangible code behind all these ideas. We construct trees, and their objects with `class` statements and class calls, which we'll meet in more detail later. In short:

- Each `class` statement generates a new class object.
- Each time a class is called, it generates a new instance object.
- Instances are automatically linked to the classes from which they are created.
- Classes are linked to their superclasses by listing them in parentheses in a `class` header line; the left-to-right order there gives the order in the tree.

To build the tree in Figure 25-1, for example, we would run Python code of this form (I've omitted the guts of the `class` statements here):

```
class C2: ...                       # Make class objects (ovals)
class C3: ...
class C1(C2, C3): ...               # Linked to superclasses

I1 = C1()                           # Make instance objects (rectangles)
I2 = C1()                           # Linked to their classes
```

Here, we build the three class objects by running three `class` statements, and make the two instance objects by calling the class `C1` twice, as though it were a function. The instances remember the class they were made from, and the class `C1` remembers its listed superclasses.

Technically, this example is using something called *multiple inheritance*, which simply means that a class has more than one superclass above it in the class tree. In Python, if there is more than one superclass listed in parentheses in a `class` statement (like `C1`'s here), their left-to-right order gives the order in which those superclasses will be searched for attributes.

Because of the way inheritance searches proceed, the object to which you attach an attribute turns out to be crucial—it determines the name's scope. Attributes attached to instances pertain only to those single instances, but attributes attached to classes are shared by all their subclasses and instances. Later, we'll study the code that hangs attributes on these objects in depth. As we'll find:

- Attributes are usually attached to classes by assignments made within `class` statements, and not nested inside function `def` statements.
- Attributes are usually attached to instances by assignments to a special argument passed to functions inside classes, called `self`.

For example, classes provide behavior for their instances with functions created by coding `def` statements inside `class` statements. Because such nested `def`s assign names within the class, they wind up attaching attributes to the class object that will be inherited by all instances and subclasses:

```
class C1(C2, C3):                   # Make and link class C1
    def setname(self, who):         # Assign name: C1.setname
        self.name = who             # Self is either I1 or I2

I1 = C1()                           # Make two instances
I2 = C1()
I1.setname('bob')                   # Sets I1.name to 'bob'
I2.setname('mel')                   # Sets I2.name to 'mel'
print(I1.name)                      # Prints 'bob'
```

There's nothing syntactically unique about `def` in this context. Operationally, when a `def` appears inside a `class` like this, it is usually known as a *method*, and it automatically receives a special first argument—called `self` by convention—that provides a handle back to the instance to be processed.†

Because classes are factories for multiple instances, their methods usually go through this automatically passed-in `self` argument whenever they need to fetch or set attributes of the particular instance being processed by a method call. In the preceding code, `self` is used to store a name in one of two instances.

Like simple variables, attributes of classes and instances are not declared ahead of time, but spring into existence the first time they are assigned values. When a method assigns to a `self` attribute, it creates or changes an attribute in an instance at the bottom of the class tree (i.e., one of the rectangles) because `self` automatically refers to the instance being processed.

In fact, because all the objects in class trees are just namespace objects, we can fetch or set any of their attributes by going through the appropriate names. Saying `C1.setname` is as valid as saying `I1.setname`, as long as the names `C1` and `I1` are in your code's scopes.

As currently coded, our `C1` class doesn't attach a `name` attribute to an instance until the `setname` method is called. In fact, referencing `I1.name` before calling `I1.setname` would produce an undefined name error. If a class wants to guarantee that an attribute like `name` is always set in its instances, it more typically will fill out the attribute at construction time, like this:

```
class C1(C2, C3):
    def __init__(self, who):      # Set name when constructed
        self.name = who           # Self is either I1 or I2

I1 = C1('bob')                    # Sets I1.name to 'bob'
I2 = C1('mel')                    # Sets I2.name to 'mel'
print(I1.name)                    # Prints 'bob'
```

If it's coded and inherited, Python automatically calls a method named `__init__` each time an instance is generated from a class. The new instance is passed in to the `self` argument of `__init__` as usual, and any values listed in parentheses in the class call go to arguments two and beyond. The effect here is to initialize instances when they are made, without requiring extra method calls.

The `__init__` method is known as the *constructor* because of when it is run. It's the most commonly used representative of a larger class of methods called *operator overloading methods*, which we'll discuss in more detail in the chapters that follow. Such methods are inherited in class trees as usual and have double underscores at the start and end of their names to make them distinct. Python runs them automatically when instances that support them appear in the corresponding operations, and they are

† If you've ever used C++ or Java, you'll recognize that Python's `self` is the same as the `this` pointer, but `self` is always explicit in Python to make attribute accesses more obvious.

mostly an alternative to using simple method calls. They're also optional: if omitted, the operations are not supported.

For example, to implement set intersection, a class might either provide a method named `intersect`, or overload the `&` expression operator to dispatch to the required logic by coding a method named `__and__`. Because the operator scheme makes instances look and feel more like built-in types, it allows some classes to provide a consistent and natural interface, and be compatible with code that expects a built-in type.

OOP Is About Code Reuse

And that, along with a few syntax details, is most of the OOP story in Python. Of course, there's a bit more to it than just inheritance. For example, operator overloading is much more general than I've described so far—classes may also provide their own implementations of operations such as indexing, fetching attributes, printing, and more. By and large, though, OOP is about looking up attributes in trees.

So why would we be interested in building and searching trees of objects? Although it takes some experience to see how, when used well, classes support code *reuse* in ways that other Python program components cannot. With classes, we code by customizing existing software, instead of either changing existing code in-place or starting from scratch for each new project.

At a fundamental level, classes are really just packages of functions and other names, much like modules. However, the automatic attribute inheritance search that we get with classes supports customization of software above and beyond what we can do with modules and functions. Moreover, classes provide a natural structure for code that localizes logic and names, and so aids in debugging.

For instance, because methods are simply functions with a special first argument, we can mimic some of their behavior by manually passing objects to be processed to simple functions. The participation of methods in class inheritance, though, allows us to naturally customize existing software by coding subclasses with new method definitions, rather than changing existing code in-place. There is really no such concept with modules and functions.

As an example, suppose you're assigned the task of implementing an employee database application. As a Python OOP programmer, you might begin by coding a general superclass that defines default behavior common to all the kinds of employees in your organization:

```
class Employee:                        # General superclass
    def computeSalary(self): ...       # Common or default behavior
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...
```

Once you've coded this general behavior, you can specialize it for each specific kind of employee to reflect how the various types differ from the norm. That is, you can code subclasses that customize just the bits of behavior that differ per employee type; the rest of the employee types' behavior will be inherited from the more general class. For example, if engineers have a unique salary computation rule (i.e., not hours times rate), you can replace just that one method in a subclass:

```
class Engineer(Employee):              # Specialized subclass
     def computeSalary(self): ...      # Something custom here
```

Because the `computeSalary` version here appears lower in the class tree, it will replace (override) the general version in `Employee`. You then create instances of the kinds of employee classes that the real employees belong to, to get the correct behavior:

```
bob = Employee()                       # Default behavior
mel = Engineer()                       # Custom salary calculator
```

Notice that you can make instances of any class in a tree, not just the ones at the bottom—the class you make an instance from determines the level at which the attribute search will begin. Ultimately, these two instance objects might wind up embedded in a larger container object (e.g., a list, or an instance of another class) that represents a department or company using the composition idea mentioned at the start of this chapter.

When you later ask for these employees' salaries, they will be computed according to the classes from which the objects were made, due to the principles of the inheritance search:‡

```
company = [bob, mel]                   # A composite object
for emp in company:
    print(emp.computeSalary())         # Run this object's version
```

This is yet another instance of the idea of *polymorphism* introduced in Chapter 4 and revisited in Chapter 16. Recall that polymorphism means that the meaning of an operation depends on the object being operated on. Here, the method `computeSalary` is located by inheritance search in each object before it is called. In other applications, polymorphism might also be used to hide (i.e., *encapsulate*) interface differences. For example, a program that processes data streams might be coded to expect objects with input and output methods, without caring what those methods actually do:

```
def processor(reader, converter, writer):
    while 1:
        data = reader.read()
        if not data: break
```

‡ Note that the `company` list in this example could be stored in a file with Python object pickling, introduced in Chapter 9 when we met files, to yield a persistent employee database. Python also comes with a module named `shelve`, which would allow you to store the pickled representation of the class instances in an access-by-key filesystem; the third-party open source ZODB system does the same but has better support for production-quality object-oriented databases.

```
        data = converter(data)
        writer.write(data)
```

By passing in instances of subclasses that specialize the required `read` and `write` method interfaces for various data sources, we can reuse the `processor` function for any data source we need to use, both now and in the future:

```
class Reader:
    def read(self): ...               # Default behavior and tools
    def other(self): ...
class FileReader(Reader):
    def read(self): ...               # Read from a local file
class SocketReader(Reader):
    def read(self): ...               # Read from a network socket
...
processor(FileReader(...),   Converter,  FileWriter(...))
processor(SocketReader(...), Converter,  TapeWriter(...))
processor(FtpReader(...),    Converter,  XmlWriter(...))
```

Moreover, because the internal implementations of those `read` and `write` methods have been factored into single locations, they can be changed without impacting code such as this that uses them. In fact, the `processor` function might itself be a class to allow the conversion logic of `converter` to be filled in by inheritance, and to allow readers and writers to be embedded by composition (we'll see how this works later in this part of the book).

Once you get used to programming this way (by software customization), you'll find that when it's time to write a new program, much of your work may already be done—your task largely becomes one of mixing together existing superclasses that already implement the behavior required by your program. For example, someone else might have written the `Employee`, `Reader`, and `Writer` classes in this example for use in a completely different program. If so, you get all of that person's code "for free."

In fact, in many application domains, you can fetch or purchase collections of superclasses, known as *frameworks*, that implement common programming tasks as classes, ready to be mixed into your applications. These frameworks might provide database interfaces, testing protocols, GUI toolkits, and so on. With frameworks, you often simply code a subclass that fills in an expected method or two; the framework classes higher in the tree do most of the work for you. Programming in such an OOP world is just a matter of combining and specializing already debugged code by writing subclasses of your own.

Of course, it takes a while to learn how to leverage classes to achieve such OOP utopia. In practice, object-oriented work also entails substantial design work to fully realize the code reuse benefits of classes—to this end, programmers have begun cataloging common OOP structures, known as *design patterns*, to help with design issues. The actual code you write to do OOP in Python, though, is so simple that it will not in itself pose an additional obstacle to your OOP quest. To see why, you'll have to move on to Chapter 26.

Chapter Summary

We took an abstract look at classes and OOP in this chapter, taking in the big picture before we dive into syntax details. As we've seen, OOP is mostly about looking up attributes in trees of linked objects; we call this lookup an inheritance search. Objects at the bottom of the tree inherit attributes from objects higher up in the tree—a feature that enables us to program by customizing code, rather than changing it, or starting from scratch. When used well, this model of programming can cut development time radically.

The next chapter will begin to fill in the coding details behind the picture painted here. As we get deeper into Python classes, though, keep in mind that the OOP model in Python is very simple; as I've already stated, it's really just about looking up attributes in object trees. Before we move on, here's a quick quiz to review what we've covered here.

Test Your Knowledge: Quiz

1. What is the main point of OOP in Python?
2. Where does an inheritance search look for an attribute?
3. What is the difference between a class object and an instance object?
4. Why is the first argument in a class method function special?
5. What is the `__init__` method used for?
6. How do you create a class instance?
7. How do you create a class?
8. How do you specify a class's superclasses?

Test Your Knowledge: Answers

1. OOP is about code reuse—you factor code to minimize redundancy and program by customizing what already exists instead of changing code in-place or starting from scratch.
2. An inheritance search looks for an attribute first in the instance object, then in the class the instance was created from, then in all higher superclasses, progressing from the bottom to the top of the object tree, and from left to right (by default). The search stops at the first place the attribute is found. Because the lowest version of a name found along the way wins, class hierarchies naturally support customization by extension.

3. Both class and instance objects are namespaces (packages of variables that appear as attributes). The main difference between them is that classes are a kind of factory for creating multiple instances. Classes also support operator overloading methods, which instances inherit, and treat any functions nested within them as special methods for processing instances.
4. The first argument in a class method function is special because it always receives the instance object that is the implied subject of the method call. It's usually called `self` by convention. Because method functions always have this implied subject object context by default, we say they are "object-oriented"—i.e., designed to process or change objects.
5. If the `__init__` method is coded or inherited in a class, Python calls it automatically each time an instance of that class is created. It's known as the constructor method; it is passed the new instance implicitly, as well as any arguments passed explicitly to the class name. It's also the most commonly used operator overloading method. If no `__init__` method is present, instances simply begin life as empty namespaces.
6. You create a class instance by calling the class name as though it were a function; any arguments passed into the class name show up as arguments two and beyond in the `__init__` constructor method. The new instance remembers the class it was created from for inheritance purposes.
7. You create a class by running a `class` statement; like function definitions, these statements normally run when the enclosing module file is imported (more on this in the next chapter).
8. You specify a class's superclasses by listing them in parentheses in the `class` statement, after the new class's name. The left-to-right order in which the classes are listed in the parentheses gives the left-to-right inheritance search order in the class tree.

CHAPTER 26
Class Coding Basics

Now that we've talked about OOP in the abstract, it's time to see how this translates to actual code. This chapter begins to fill in the syntax details behind the class model in Python.

If you've never been exposed to OOP in the past, classes can seem somewhat complicated if taken in a single dose. To make class coding easier to absorb, we'll begin our detailed exploration of OOP by taking a first look at some basic classes in action in this chapter. We'll expand on the details introduced here in later chapters of this part of the book, but in their basic form, Python classes are easy to understand.

In fact, classes have just three primary distinctions. At a base level, they are mostly just namespaces, much like the modules we studied in Part V. Unlike modules, though, classes also have support for generating multiple objects, for namespace inheritance, and for operator overloading. Let's begin our `class` statement tour by exploring each of these three distinctions in turn.

Classes Generate Multiple Instance Objects

To understand how the multiple objects idea works, you have to first understand that there are two kinds of objects in Python's OOP model: *class* objects and *instance* objects. Class objects provide default behavior and serve as factories for instance objects. Instance objects are the real objects your programs process—each is a namespace in its own right, but inherits (i.e., has automatic access to) names in the class from which it was created. Class objects come from statements, and instances come from calls; each time you call a class, you get a new instance of that class.

This object-generation concept is very different from any of the other program constructs we've seen so far in this book. In effect, classes are essentially *factories* for generating multiple instances. By contrast, only one copy of each module is ever imported into a single program (in fact, one reason that we have to call `imp.reload` is to update the single module object so that changes are reflected once they've been made).

The following is a quick summary of the bare essentials of Python OOP. As you'll see, Python classes are in some ways similar to both `def`s and modules, but they may be quite different from what you're used to in other languages.

Class Objects Provide Default Behavior

When we run a `class` statement, we get a class object. Here's a rundown of the main properties of Python classes:

- **The `class` statement creates a class object and assigns it a name**. Just like the function `def` statement, the Python `class` statement is an *executable* statement. When reached and run, it generates a new class object and assigns it to the name in the `class` header. Also, like `def`s, `class` statements typically run when the files they are coded in are first imported.
- **Assignments inside `class` statements make class attributes**. Just like in module files, top-level assignments within a `class` statement (not nested in a `def`) generate attributes in a class object. Technically, the `class` statement scope *morphs* into the attribute namespace of the class object, just like a module's global scope. After running a `class` statement, class attributes are accessed by name qualification: *`object.name`*.
- **Class attributes provide object state and behavior**. Attributes of a class object record state information and behavior to be shared by all instances created from the class; function `def` statements nested inside a `class` generate *methods*, which process instances.

Instance Objects Are Concrete Items

When we call a class object, we get an instance object. Here's an overview of the key points behind class instances:

- **Calling a class object like a function makes a new instance object**. Each time a class is called, it creates and returns a new instance object. Instances represent concrete items in your program's domain.
- **Each instance object inherits class attributes and gets its own namespace**. Instance objects created from classes are new namespaces; they start out empty but inherit attributes that live in the class objects from which they were generated.

- **Assignments to attributes of `self` in methods make per-instance attributes**. Inside class method functions, the first argument (called `self` by convention) references the instance object being processed; assignments to attributes of `self` create or change data in the instance, not the class.

A First Example

Let's turn to a real example to show how these ideas work in practice. To begin, let's define a class named `FirstClass` by running a Python `class` statement interactively:

```
>>> class FirstClass:                  # Define a class object
...     def setdata(self, value):      # Define class methods
...         self.data = value          # self is the instance
...     def display(self):
...         print(self.data)           # self.data: per instance
...
```

We're working interactively here, but typically, such a statement would be run when the module file it is coded in is imported. Like functions created with `def`s, this class won't even exist until Python reaches and runs this statement.

Like all compound statements, the `class` starts with a header line that lists the class name, followed by a body of one or more nested and (usually) indented statements. Here, the nested statements are `def`s; they define functions that implement the behavior the class means to export.

As we learned in Part IV, `def` is really an assignment. Here, it assigns function objects to the names `setdata` and `display` in the `class` statement's scope, and so generates attributes attached to the class: `FirstClass.setdata` and `FirstClass.display`. In fact, any name assigned at the top level of the class's nested block becomes an attribute of the class.

Functions inside a class are usually called *methods*. They're coded with normal `def`s, and they support everything we've learned about functions already (they can have defaults, return values, and so on). But in a method function, the first argument automatically receives an implied instance object when called—the subject of the call. We need to create a couple of instances to see how this works:

```
>>> x = FirstClass()                   # Make two instances
>>> y = FirstClass()                   # Each is a new namespace
```

By *calling* the class this way (notice the parentheses), we generate instance objects, which are just namespaces that have access to their classes' attributes. Properly speaking, at this point, we have three objects: two instances and a class. Really, we have three linked namespaces, as sketched in Figure 26-1. In OOP terms, we say that `x` "is a" `FirstClass`, as is `y`.

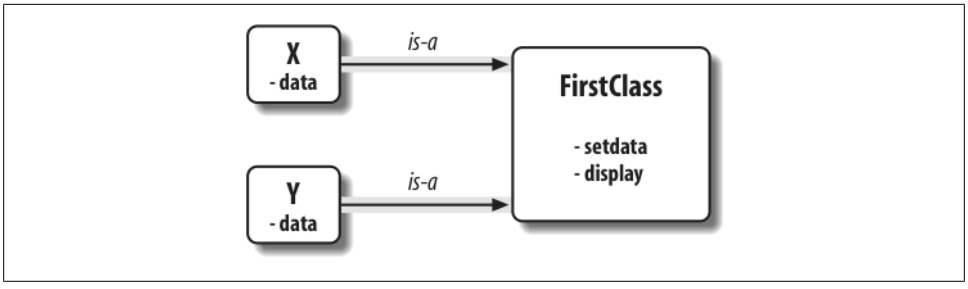

Figure 26-1. Classes and instances are linked namespace objects in a class tree that is searched by inheritance. Here, the "data" attribute is found in instances, but "setdata" and "display" are in the class above them.

The two instances start out empty but have links back to the class from which they were generated. If we qualify an instance with the name of an attribute that lives in the class object, Python fetches the name from the class by inheritance search (unless it also lives in the instance):

```
>>> x.setdata("King Arthur")      # Call methods: self is x
>>> y.setdata(3.14159)            # Runs: FirstClass.setdata(y, 3.14159)
```

Neither `x` nor `y` has a `setdata` attribute of its own, so to find it, Python follows the link from instance to class. And that's about all there is to inheritance in Python: it happens at attribute qualification time, and it just involves looking up names in linked objects (e.g., by following the is-a links in Figure 26-1).

In the `setdata` function inside `FirstClass`, the value passed in is assigned to `self.data`. Within a method, `self`—the name given to the leftmost argument by convention—automatically refers to the instance being processed (`x` or `y`), so the assignments store values in the instances' namespaces, not the class's (that's how the `data` names in Figure 26-1 are created).

Because classes can generate multiple instances, methods must go through the `self` argument to get to the instance to be processed. When we call the class's `display` method to print `self.data`, we see that it's different in each instance; on the other hand, the name `display` itself is the same in `x` and `y`, as it comes (is inherited) from the class:

```
>>> x.display()                   # self.data differs in each instance
King Arthur
>>> y.display()
3.14159
```

Notice that we stored different object types in the `data` member in each instance (a string, and a floating point). As with everything else in Python, there are no declarations for instance attributes (sometimes called *members*); they spring into existence the first time they are assigned values, just like simple variables. In fact, if we were to call `display` on one of our instances before calling `setdata`, we would trigger an undefined name error—the attribute named `data` doesn't even exist in memory until it is assigned within the `setdata` method.

As another way to appreciate how dynamic this model is, consider that we can change instance attributes in the class itself, by assigning to `self` in methods, or outside the class, by assigning to an explicit instance object:

```
>>> x.data = "New value"          # Can get/set attributes
>>> x.display()                   # Outside the class too
New value
```

Although less common, we could even generate a brand new attribute in the instance's namespace by assigning to its name outside the class's method functions:

```
>>> x.anothername = "spam"        # Can set new attributes here too!
```

This would attach a new attribute called `anothername`, which may or may not be used by any of the class's methods, to the instance object `x`. Classes usually create all of the instance's attributes by assignment to the `self` argument, but they don't have to; programs can fetch, change, or create attributes on any objects to which they have references.

Classes Are Customized by Inheritance

Besides serving as factories for generating multiple instance objects, classes also allow us to make changes by introducing new components (called *subclasses*), instead of changing existing components in-place. Instance objects generated from a class inherit the class's attributes. Python also allows classes to inherit from other classes, opening the door to coding *hierarchies* of classes that specialize behavior—by redefining attributes in subclasses that appear lower in the hierarchy, we override the more general definitions of those attributes higher in the tree. In effect, the further down the hierarchy we go, the more specific the software becomes. Here, too, there is no parallel with modules: their attributes live in a single, flat namespace that is not as amenable to customization.

In Python, instances inherit from classes, and classes inherit from superclasses. Here are the key ideas behind the machinery of attribute inheritance:

- **Superclasses are listed in parentheses in a `class` header**. To inherit attributes from another class, just list the class in parentheses in a `class` statement's header. The class that inherits is usually called a *subclass*, and the class that is inherited from is its *superclass*.
- **Classes inherit attributes from their superclasses**. Just as instances inherit the attribute names defined in their classes, classes inherit all the attribute names defined in their superclasses; Python finds them automatically when they're accessed, if they don't exist in the subclasses.
- **Instances inherit attributes from all accessible classes**. Each instance gets names from the class it's generated from, as well as all of that class's superclasses. When looking for a name, Python checks the instance, then its class, then all superclasses.

- **Each *`object.attribute`* reference invokes a new, independent search**. Python performs an independent search of the class tree for each attribute fetch expression. This includes references to instances and classes made outside `class` statements (e.g., `X.attr`), as well as references to attributes of the `self` instance argument in class method functions. Each `self.attr` expression in a method invokes a new search for `attr` in `self` and above.
- **Logic changes are made by subclassing, not by changing superclasses**. By redefining superclass names in subclasses lower in the hierarchy (class tree), subclasses replace and thus customize inherited behavior.

The net effect, and the main purpose of all this searching, is that classes support factoring and customization of code better than any other language tool we've seen so far. On the one hand, they allow us to minimize code redundancy (and so reduce maintenance costs) by factoring operations into a single, shared implementation; on the other, they allow us to program by customizing what already exists, rather than changing it in-place or starting from scratch.

A Second Example

To illustrate the role of inheritance, this next example builds on the previous one. First, we'll define a new class, `SecondClass`, that inherits all of `FirstClass`'s names and provides one of its own:

```
>>> class SecondClass(FirstClass):                  # Inherits setdata
...     def display(self):                          # Changes display
...         print('Current value = "%s"' % self.data)
...
```

`SecondClass` defines the `display` method to print with a different format. By defining an attribute with the same name as an attribute in `FirstClass`, `SecondClass` effectively replaces the `display` attribute in its superclass.

Recall that inheritance searches proceed upward from instances, to subclasses, to superclasses, stopping at the first appearance of the attribute name that it finds. In this case, since the `display` name in `SecondClass` will be found before the one in `FirstClass`, we say that `SecondClass` *overrides* `FirstClass`'s `display`. Sometimes we call this act of replacing attributes by redefining them lower in the tree *overloading*.

The net effect here is that `SecondClass` specializes `FirstClass` by changing the behavior of the `display` method. On the other hand, `SecondClass` (and any instances created from it) still inherits the `setdata` method in `FirstClass` verbatim. Let's make an instance to demonstrate:

```
>>> z = SecondClass()
>>> z.setdata(42)           # Finds setdata in FirstClass
>>> z.display()             # Finds overridden method in SecondClass
Current value = "42"
```

As before, we make a `SecondClass` instance object by calling it. The `setdata` call still runs the version in `FirstClass`, but this time the `display` attribute comes from `SecondClass` and prints a custom message. Figure 26-2 sketches the namespaces involved.

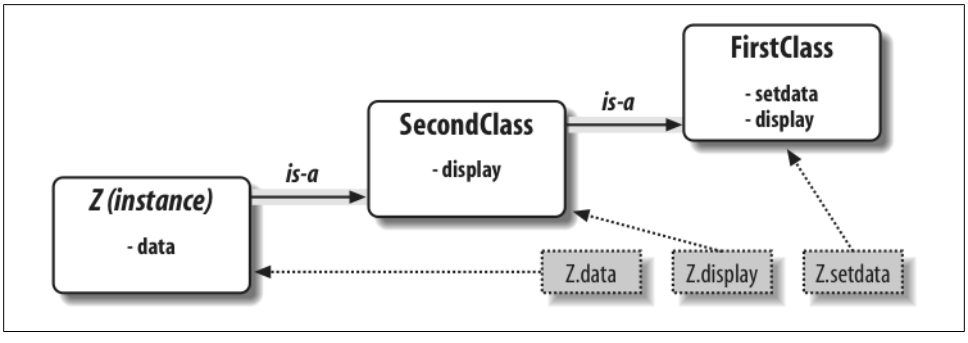

Figure 26-2. Specialization by overriding inherited names by redefining them in extensions lower in the class tree. Here, SecondClass redefines and so customizes the "display" method for its instances.

Now, here's a very important thing to notice about OOP: the specialization introduced in `SecondClass` is completely *external* to `FirstClass`. That is, it doesn't affect existing or future `FirstClass` objects, like the `x` from the prior example:

```
>>> x.display()             # x is still a FirstClass instance (old message)
New value
```

Rather than *changing* `FirstClass`, we *customized* it. Naturally, this is an artificial example, but as a rule, because inheritance allows us to make changes like this in external components (i.e., in subclasses), classes often support extension and reuse better than functions or modules can.

Classes Are Attributes in Modules

Before we move on, remember that there's nothing magic about a class name. It's just a variable assigned to an object when the `class` statement runs, and the object can be referenced with any normal expression. For instance, if our `FirstClass` was coded in a module file instead of being typed interactively, we could import it and use its name normally in a `class` header line:

```
from modulename import FirstClass          # Copy name into my scope
class SecondClass(FirstClass):             # Use class name directly
    def display(self): ...
```

Or, equivalently:

```
import modulename                           # Access the whole module
class SecondClass(modulename.FirstClass):  # Qualify to reference
    def display(self): ...
```

Like everything else, class names always live within a module, so they must follow all the rules we studied in Part V. For example, more than one class can be coded in a single module file—like other statements in a module, `class` statements are run during imports to define names, and these names become distinct module attributes. More generally, each module may arbitrarily mix any number of variables, functions, and classes, and all names in a module behave the same way. The file *food.py* demonstrates:

```
# food.py
var = 1                                 # food.var
def func():                             # food.func
    ...
class spam:                             # food.spam
    ...
class ham:                              # food.ham
    ...
class eggs:                             # food.eggs
    ...
```

This holds true even if the module and class happen to have the same name. For example, given the following file, *person.py*:

```
class person:
   ...
```

we need to go through the module to fetch the class as usual:

```
import person                           # Import module
x = person.person()                     # Class within module
```

Although this path may look redundant, it's required: `person.person` refers to the `person` class inside the `person` module. Saying just `person` gets the module, not the class, unless the `from` statement is used:

```
from person import person               # Get class from module
x = person()                            # Use class name
```

As with any other variable, we can never see a class in a file without first importing and somehow fetching it from its enclosing file. If this seems confusing, don't use the same name for a module and a class within it. In fact, common convention in Python dictates that class names should begin with an uppercase letter, to help make them more distinct:

```
import person                           # Lowercase for modules
x = person.Person()                     # Uppercase for classes
```

Also, keep in mind that although classes and modules are both namespaces for attaching attributes, they correspond to very different source code structures: a module reflects an entire *file*, but a class is a *statement* within a file. We'll say more about such distinctions later in this part of the book.

Classes Can Intercept Python Operators

Let's move on to the third major difference between classes and modules: operator overloading. In simple terms, *operator overloading* lets objects coded with classes intercept and respond to operations that work on built-in types: addition, slicing, printing, qualification, and so on. It's mostly just an automatic dispatch mechanism—expressions and other built-in operations route control to implementations in classes. Here, too, there is nothing similar in modules: modules can implement function calls, but not the behavior of expressions.

Although we could implement all class behavior as method functions, operator overloading lets objects be more tightly integrated with Python's object model. Moreover, because operator overloading makes our own objects act like built-ins, it tends to foster object interfaces that are more consistent and easier to learn, and it allows class-based objects to be processed by code written to expect a built-in type's interface. Here is a quick rundown of the main ideas behind overloading operators:

- **Methods named with double underscores (`__X__`) are special hooks**. Python operator overloading is implemented by providing specially named methods to intercept operations. The Python language defines a fixed and unchangeable mapping from each of these operations to a specially named method.
- **Such methods are called automatically when instances appear in built-in operations**. For instance, if an instance object inherits an `__add__` method, that method is called whenever the object appears in a `+` expression. The method's return value becomes the result of the corresponding expression.
- **Classes may override most built-in type operations**. There are dozens of special operator overloading method names for intercepting and implementing nearly every operation available for built-in types. This includes expressions, but also basic operations like printing and object creation.
- **There are no defaults for operator overloading methods, and none are required**. If a class does not define or inherit an operator overloading method, it just means that the corresponding operation is not supported for the class's instances. If there is no `__add__`, for example, `+` expressions raise exceptions.
- **Operators allow classes to integrate with Python's object model**. By overloading type operations, user-defined objects implemented with classes can act just like built-ins, and so provide consistency as well as compatibility with expected interfaces.

Operator overloading is an optional feature; it's used primarily by people developing tools for other Python programmers, not by application developers. And, candidly, you probably shouldn't try to use it just because it seems "cool." Unless a class needs to mimic built-in type interfaces, it should usually stick to simpler named methods. Why would an employee database application support expressions like * and +, for example? Named methods like `giveRaise` and `promote` would usually make more sense.

Because of this, we won't go into details on every operator overloading method available in Python in this book. Still, there is one operator overloading method you are likely to see in almost every realistic Python class: the `__init__` method, which is known as the *constructor* method and is used to initialize objects' state. You should pay special attention to this method, because `__init__`, along with the `self` argument, turns out to be a key requirement to understanding most OOP code in Python.

A Third Example

On to another example. This time, we'll define a subclass of `SecondClass` that implements three specially named attributes that Python will call automatically:

- `__init__` is run when a new instance object is created (`self` is the new `ThirdClass` object).*
- `__add__` is run when a `ThirdClass` instance appears in a `+` expression.
- `__str__` is run when an object is printed (technically, when it's converted to its print string by the `str` built-in function or its Python internals equivalent).

Our new subclass also defines a normally named method named `mul`, which changes the instance object in-place. Here's the new subclass:

```
>>> class ThirdClass(SecondClass):                  # Inherit from SecondClass
...     def __init__(self, value):                  # On "ThirdClass(value)"
...         self.data = value
...     def __add__(self, other):                   # On "self + other"
...         return ThirdClass(self.data + other)
...     def __str__(self):                          # On "print(self)", "str()"
...         return '[ThirdClass: %s]' % self.data
...     def mul(self, other):                       # In-place change: named
...         self.data *= other
...
>>> a = ThirdClass('abc')           # __init__ called
>>> a.display()                     # Inherited method called
Current value = "abc"
>>> print(a)                        # __str__: returns display string
[ThirdClass: abc]

>>> b = a + 'xyz'                   # __add__: makes a new instance
>>> b.display()                     # b has all ThirdClass methods
Current value = "abcxyz"
>>> print(b)                        # __str__: returns display string
[ThirdClass: abcxyz]

>>> a.mul(3)                        # mul: changes instance in-place
>>> print(a)
[ThirdClass: abcabcabc]
```

* Not to be confused with the *__init__.py* files in module packages! See Chapter 23 for more details.

`ThirdClass` "is a" `SecondClass`, so its instances inherit the customized `display` method from `SecondClass`. This time, though, `ThirdClass` creation calls pass an argument (e.g., "abc"). This argument is passed to the `value` argument in the `__init__` constructor and assigned to `self.data` there. The net effect is that `ThirdClass` arranges to set the `data` attribute automatically at construction time, instead of requiring `setdata` calls after the fact.

Further, `ThirdClass` objects can now show up in `+` expressions and `print` calls. For `+`, Python passes the instance object on the left to the `self` argument in `__add__` and the value on the right to `other`, as illustrated in Figure 26-3; whatever `__add__` returns becomes the result of the `+` expression. For `print`, Python passes the object being printed to `self` in `__str__`; whatever string this method returns is taken to be the print string for the object. With `__str__` we can use a normal `print` to display objects of this class, instead of calling the special `display` method.

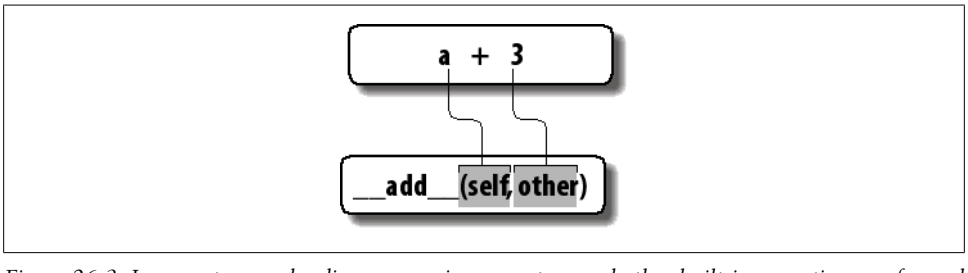

Figure 26-3. In operator overloading, expression operators and other built-in operations performed on class instances are mapped back to specially named methods in the class. These special methods are optional and may be inherited as usual. Here, a + expression triggers the __add__ method.

Specially named methods such as `__init__`, `__add__`, and `__str__` are inherited by subclasses and instances, just like any other names assigned in a `class`. If they're not coded in a class, Python looks for such names in all its superclasses, as usual. Operator overloading method names are also not built-in or reserved words; they are just attributes that Python looks for when objects appear in various contexts. Python usually calls them automatically, but they may occasionally be called by your code as well; the `__init__` method, for example, is often called manually to trigger superclass constructors (more on this later).

Notice that the `__add__` method makes and returns a *new* instance object of its class, by calling `ThirdClass` with the result value. By contrast, `mul` *changes* the current instance object in-place, by reassigning the `self` attribute. We could overload the * expression to do the latter, but this would be too different from the behavior of * for built-in types such as numbers and strings, for which it always makes new objects. Common practice dictates that overloaded operators should work the same way that built-in operator implementations do. Because operator overloading is really just an expression-to-method dispatch mechanism, though, you can interpret operators any way you like in your own class objects.

Why Use Operator Overloading?

As a class designer, you can choose to use operator overloading or not. Your choice simply depends on how much you want your object to look and feel like built-in types. As mentioned earlier, if you omit an operator overloading method and do not inherit it from a superclass, the corresponding operation will not be supported for your instances; if it's attempted, an exception will be thrown (or a standard default will be used).

Frankly, many operator overloading methods tend to be used only when implementing objects that are mathematical in nature; a vector or matrix class may overload the addition operator, for example, but an employee class likely would not. For simpler classes, you might not use overloading at all, and would rely instead on explicit method calls to implement your objects' behavior.

On the other hand, you might decide to use operator overloading if you need to pass a user-defined object to a function that was coded to expect the operators available on a built-in type like a list or a dictionary. Implementing the same operator set in your class will ensure that your objects support the same expected object interface and so are compatible with the function. Although we won't cover every operator overloading method in this book, we'll see some additional operator overloading techniques in action in Chapter 29.

One overloading method we will explore here is the `__init__` constructor method, which seems to show up in almost every realistic class. Because it allows classes to fill out the attributes in their newly created instances immediately, the constructor is useful for almost every kind of class you might code. In fact, even though instance attributes are not declared in Python, you can usually find out which attributes an instance will have by inspecting its class's `__init__` method.

The World's Simplest Python Class

We've begun studying `class` statement syntax in detail in this chapter, but I'd again like to remind you that the basic inheritance model that classes produce is very simple—all it really involves is searching for attributes in trees of linked objects. In fact, we can create a class with nothing in it at all. The following statement makes a class with no attributes attached (an empty namespace object):

```
>>> class rec: pass                # Empty namespace object
```

We need the no-operation `pass` statement (discussed in Chapter 13) here because we don't have any methods to code. After we make the class by running this statement interactively, we can start attaching attributes to the class by assigning names to it completely outside of the original `class` statement:

```
>>> rec.name = 'Bob'               # Just objects with attributes
>>> rec.age  = 40
```

And, after we've created these attributes by assignment, we can fetch them with the usual syntax. When used this way, a class is roughly similar to a "struct" in C, or a "record" in Pascal. It's basically an object with field names attached to it (we can do similar work with dictionary keys, but it requires extra characters):

```
>>> print(rec.name)                # Like a C struct or a record
Bob
```

Notice that this works even though there are no instances of the class yet; classes are objects in their own right, even without instances. In fact, they are just self-contained namespaces, so as long as we have a reference to a class, we can set or change its attributes anytime we wish. Watch what happens when we do create two instances, though:

```
>>> x = rec()                      # Instances inherit class names
>>> y = rec()
```

These instances begin their lives as completely empty namespace objects. Because they remember the class from which they were made, though, they will obtain the attributes we attached to the class by inheritance:

```
>>> x.name, y.name                 # name is stored on the class only
('Bob', 'Bob')
```

Really, these instances have no attributes of their own; they simply fetch the `name` attribute from the class object where it is stored. If we do assign an attribute to an instance, though, it creates (or changes) the attribute in that object, and no other—attribute references kick off inheritance searches, but attribute assignments affect only the objects in which the assignments are made. Here, `x` gets its own `name`, but `y` still inherits the `name` attached to the class above it:

```
>>> x.name = 'Sue'                 # But assignment changes x only
>>> rec.name, x.name, y.name
('Bob', 'Sue', 'Bob')
```

In fact, as we'll explore in more detail in Chapter 28, the attributes of a namespace object are usually implemented as dictionaries, and class inheritance trees are (generally speaking) just dictionaries with links to other dictionaries. If you know where to look, you can see this explicitly.

For example, the `__dict__` attribute is the namespace dictionary for most class-based objects (some classes may also define attributes in `__slots__`, an advanced and seldom-used feature that we'll study in Chapters 30 and 31). The following was run in Python 3.0; the order of names and set of `__X__` internal names present can vary from release to release, but the names we assigned are present in all:

```
>>> rec.__dict__.keys()
['__module__', 'name', 'age', '__dict__', '__weakref__', '__doc__']

>>> list(x.__dict__.keys())
['name']
```

```
>>> list(y.__dict__.keys())          # list() not required in Python 2.6
[]
```

Here, the class's namespace dictionary shows the `name` and `age` attributes we assigned to it, `x` has its own `name`, and `y` is still empty. Each instance has a link to its class for inheritance, though—it's called `__class__`, if you want to inspect it:

```
>>> x.__class__
<class '__main__.rec'>
```

Classes also have a `__bases__` attribute, which is a tuple of their superclasses:

```
>>> rec.__bases__                    # () empty tuple in Python 2.6
(<class 'object'>,)
```

These two attributes are how class trees are literally represented in memory by Python.

The main point to take away from this look under the hood is that Python's class model is extremely dynamic. Classes and instances are just namespace objects, with attributes created on the fly by assignment. Those assignments usually happen within the `class` statements you code, but they can occur anywhere you have a reference to one of the objects in the tree.

Even methods, normally created by a `def` nested in a `class`, can be created completely independently of any class object. The following, for example, defines a simple function outside of any class that takes one argument:

```
>>> def upperName(self):
...     return self.name.upper()     # Still needs a self
```

There is nothing about a class here yet—it's a simple function, and it can be called as such at this point, provided we pass in an object with a `name` attribute (the name `self` does not make this special in any way):

```
>>> upperName(x)                     # Call as a simple function
'SUE'
```

If we assign this simple function to an attribute of our class, though, it becomes a method, callable through any instance (as well as through the class name itself, as long as we pass in an instance manually):†

```
>>> rec.method = upperName

>>> x.method()                       # Run  method to process x
'SUE'

>>> y.method()                       # Same, but pass y to self
'BOB'
```

† In fact, this is one of the reasons the `self` argument must always be explicit in Python methods—because methods can be created as simple functions independent of a class, they need to make the implied instance argument explicit. They can be called as either functions or methods, and Python can neither guess nor assume that a simple function might eventually become a class method. The main reason for the explicit `self` argument, though, is to make the meanings of names more obvious: names not referenced through `self` are simple variables, while names referenced through `self` are obviously instance attributes.

```
>>> rec.method(x)                    # Can call through instance or class
'SUE'
```

Normally, classes are filled out by `class` statements, and instance attributes are created by assignments to `self` attributes in method functions. The point again, though, is that they don't have to be; OOP in Python really is mostly about looking up attributes in linked namespace objects.

Classes Versus Dictionaries

Although the simple classes of the prior section are meant to illustrate class model basics, the techniques they employ can also be used for real work. For example, Chapter 8 showed how to use dictionaries to record properties of entities in our programs. It turns out that classes can serve this role, too—they package information like dictionaries, but can also bundle processing logic in the form of methods. For reference, here is the example for dictionary-based records we used earlier in the book:

```
>>> rec = {}
>>> rec['name'] = 'mel'              # Dictionary-based record
>>> rec['age']  = 45
>>> rec['job']  = 'trainer/writer'
>>>
>>> print(rec['name'])
mel
```

This code emulates tools like records in other languages. As we just saw, though, there are also multiple ways to do the same with classes. Perhaps the simplest is this—trading keys for attributes:

```
>>> class rec: pass
...
>>> rec.name = 'mel'                 # Class-based record
>>> rec.age  = 45
>>> rec.job  = 'trainer/writer'
>>>
>>> print(rec.age)
40
```

This code has substantially less syntax than the dictionary equivalent. It uses an empty `class` statement to generate an empty namespace object. Once we make the empty class, we fill it out by assigning class attributes over time, as before.

This works, but a new `class` statement will be required for each distinct record we will need. Perhaps more typically, we can instead generate *instances* of an empty class to represent each distinct entity:

```
>>> class rec: pass
...
>>> pers1 = rec()                    # Instance-based records
>>> pers1.name = 'mel'
>>> pers1.job  = 'trainer'
```

```
>>> pers1.age   = 40
>>>
>>> pers2 = rec()
>>> pers2.name = 'vls'
>>> pers2.job  = 'developer'
>>>
>>> pers1.name, pers2.name
('mel', 'vls')
```

Here, we make two records from the same class. Instances start out life empty, just like classes. We then fill in the records by assigning to attributes. This time, though, there are two separate objects, and hence two separate `name` attributes. In fact, instances of the same class don't even have to have the same set of attribute names; in this example, one has a unique `age` name. Instances really are distinct namespaces, so each has a distinct attribute dictionary. Although they are normally filled out consistently by class methods, they are more flexible than you might expect.

Finally, we might instead code a more full-blown class to implement the record and its processing:

```
>>> class Person:
...     def __init__(self, name, job):       # Class = Data + Logic
...         self.name = name
...         self.job  = job
...     def info(self):
...         return (self.name, self.job)
...
>>> rec1 = Person('mel', 'trainer')
>>> rec2 = Person('vls', 'developer')
>>>
>>> rec1.job, rec2.info()
('trainer', ('vls', 'developer'))
```

This scheme also makes multiple instances, but the class is not empty this time: we've added *logic* (methods) to initialize instances at construction time and collect attributes into a tuple. The constructor imposes some consistency on instances here by always setting the `name` and `job` attributes. Together, the class's methods and instance attributes create a *package*, which combines both data and logic.

We could further extend this code by adding logic to compute salaries, parse names, and so on. Ultimately, we might link the class into a larger hierarchy to inherit an existing set of methods via the automatic attribute search of classes, or perhaps even store instances of the class in a file with Python object pickling to make them persistent. In fact, we will—in the next chapter, we'll expand on this analogy between classes and records with a more realistic running example that demonstrates class basics in action.

In the end, although types like dictionaries are flexible, classes allow us to add behavior to objects in ways that built-in types and simple functions do not directly support. Although we can store functions in dictionaries, too, using them to process implied instances is nowhere near as natural as it is in classes.

Chapter Summary

This chapter introduced the basics of coding classes in Python. We studied the syntax of the `class` statement, and we saw how to use it to build up a class inheritance tree. We also studied how Python automatically fills in the first argument in method functions, how attributes are attached to objects in a class tree by simple assignment, and how specially named operator overloading methods intercept and implement built-in operations for our instances (e.g., expressions and printing).

Now that we've learned all about the mechanics of coding classes in Python, the next chapter turns to a larger and more realistic example that ties together much of what we've learned about OOP so far. After that, we'll continue our look at class coding, taking a second pass over the model to fill in some of the details that were omitted here to keep things simple. First, though, let's work through a quiz to review the basics we've covered so far.

Test Your Knowledge: Quiz

1. How are classes related to modules?
2. How are instances and classes created?
3. Where and how are class attributes created?
4. Where and how are instance attributes created?
5. What does `self` mean in a Python class?
6. How is operator overloading coded in a Python class?
7. When might you want to support operator overloading in your classes?
8. Which operator overloading method is most commonly used?
9. What are the two key concepts required to understand Python OOP code?

Test Your Knowledge: Answers

1. Classes are always nested inside a module; they are attributes of a module object. Classes and modules are both namespaces, but classes correspond to statements (not entire files) and support the OOP notions of multiple instances, inheritance, and operator overloading. In a sense, a module is like a single-instance class, without inheritance, which corresponds to an entire file of code.
2. Classes are made by running `class` statements; instances are created by calling a class as though it were a function.

3. Class attributes are created by assigning attributes to a class object. They are normally generated by top-level assignments nested in a `class` statement—each name assigned in the `class` statement block becomes an attribute of the class object (technically, the `class` statement scope morphs into the class object's attribute namespace). Class attributes can also be created, though, by assigning attributes to the class anywhere a reference to the class object exists—i.e., even outside the `class` statement.
4. Instance attributes are created by assigning attributes to an instance object. They are normally created within class method functions inside the `class` statement by assigning attributes to the `self` argument (which is always the implied instance). Again, though, they may be created by assignment anywhere a reference to the instance appears, even outside the `class` statement. Normally, all instance attributes are initialized in the `__init__` constructor method; that way, later method calls can assume the attributes already exist.
5. `self` is the name commonly given to the first (leftmost) argument in a class method function; Python automatically fills it in with the instance object that is the implied subject of the method call. This argument need not be called `self` (though this is a very strong convention); its position is what is significant. (Ex-C++ or Java programmers might prefer to call it `this` because in those languages that name reflects the same idea; in Python, though, this argument must always be explicit.)
6. Operator overloading is coded in a Python class with specially named methods; they all begin and end with double underscores to make them unique. These are not built-in or reserved names; Python just runs them automatically when an instance appears in the corresponding operation. Python itself defines the mappings from operations to special method names.
7. Operator overloading is useful to implement objects that resemble built-in types (e.g., sequences or numeric objects such as matrixes), and to mimic the built-in type interface expected by a piece of code. Mimicking built-in type interfaces enables you to pass in class instances that also have state information—i.e., attributes that remember data between operation calls. You shouldn't use operator overloading when a simple named method will suffice, though.
8. The `__init__` constructor method is the most commonly used; almost every class uses this method to set initial values for instance attributes and perform other startup tasks.
9. The special `self` argument in method functions and the `__init__` constructor method are the two cornerstones of OOP code in Python.

CHAPTER 27
A More Realistic Example

We'll dig into more class syntax details in the next chapter. Before we do, though, I'd like to show you a more realistic example of classes in action that's more practical than what we've seen so far. In this chapter, we're going to build a set of classes that do something more concrete—recording and processing information about people. As you'll see, what we call *instances* and *classes* in Python programming can often serve the same roles as *records* and *programs* in more traditional terms.

Specifically, in this chapter we're going to code two classes:

- `Person`—a class that creates and processes information about people
- `Manager`—a customization of Person that modifies inherited behavior

Along the way, we'll make instances of both classes and test out their functionality. When we're done, I'll show you a nice example use case for classes—we'll store our instances in a *shelve* object-oriented database, to make them permanent. That way, you can use this code as a template for fleshing out a full-blown personal database written entirely in Python.

Besides actual utility, though, our aim here is also *educational*: this chapter provides a tutorial on object-oriented programming in Python. Often, people grasp the last chapter's class syntax on paper, but have trouble seeing how to get started when confronted with having to code a new class from scratch. Toward this end, we'll take it one step at a time here, to help you learn the basics; we'll build up the classes gradually, so you can see how their features come together in complete programs.

In the end, our classes will still be relatively small in terms of code, but they will demonstrate *all* of the main ideas in Python's OOP model. Despite its syntax details, Python's class system really is largely just a matter of searching for an attribute in a tree of objects, along with a special first argument for functions.

Step 1: Making Instances

OK, so much for the design phase—let's move on to implementation. Our first task is to start coding the main class, `Person`. In your favorite text editor, open a new file for the code we'll be writing. It's a fairly strong convention in Python to begin module names with a lowercase letter and class names with an uppercase letter; like the name of `self` arguments in methods, this is not required by the language, but it's so common that deviating might be confusing to people who later read your code. To conform, we'll call our new module file *person.py* and our class within it `Person`, like this:

```
# File person.py (start)

class Person:
```

All our work will be done in this file until later in this chapter. We can code any number of functions and classes in a single module file in Python, and this one's *person.py* name might not make much sense if we add unrelated components to it later. For now, we'll assume everything in it will be `Person`-related. It probably should be anyhow—as we've learned, modules tend to work best when they have a single, *cohesive* purpose.

Coding Constructors

Now, the first thing we want to do with our `Person` class is record basic information about people—to fill out record fields, if you will. Of course, these are known as instance object *attributes* in Python-speak, and they generally are created by assignment to `self` attributes in class method functions. The normal way to give instance attributes their first values is to assign them to `self` in the `__init__` *constructor method*, which contains code run automatically by Python each time an instance is created. Let's add one to our class:

```
# Add record field initialization

class Person:
    def __init__(self, name, job, pay):       # Constructor takes 3 arguments
        self.name = name                      # Fill out fields when created
        self.job  = job                       # self is the new instance object
        self.pay  = pay
```

This is a very common coding pattern: we pass in the data to be attached to an instance as arguments to the constructor method and assign them to `self` to retain them permanently. In OO terms, `self` is the newly created instance object, and `name`, `job`, and `pay` become *state information*—descriptive data saved on an object for later use. Although other techniques (such as enclosing scope references) can save details, too, instance attributes make this very explicit and easy to understand.

Notice that the argument names appear *twice* here. This code might seem a bit redundant at first, but it's not. The `job` argument, for example, is a local variable in the scope of the `__init__` function, but `self.job` is an attribute of the instance that's the implied

subject of the method call. They are two different variables, which happen to have the same name. By assigning the `job` local to the `self.job` attribute with `self.job=job`, we save the passed-in `job` on the instance for later use. As usual in Python, where a name is assigned (or what object it is assigned to) determines what it means.

Speaking of arguments, there's really nothing magical about `__init__`, apart from the fact that it's called automatically when an instance is made and has a special first argument. Despite its weird name, it's a normal function and supports all the features of functions we've already covered. We can, for example, provide *defaults* for some of its arguments, so they need not be provided in cases where their values aren't available or useful.

To demonstrate, let's make the `job` argument optional—it will default to `None`, meaning the person being created is not (currently) employed. If `job` defaults to `None`, we'll probably want to default `pay` to `0`, too, for consistency (unless some of the people you know manage to get paid without having jobs!). In fact, we have to specify a default for `pay` because according to Python's syntax rules, any arguments in a function's header after the first default must all have defaults, too:

```
# Add defaults for constructor arguments

class Person:
    def __init__(self, name, job=None, pay=0):          # Normal function args
        self.name = name
        self.job  = job
        self.pay  = pay
```

What this code means is that we'll need to pass in a name when making `Person`s, but `job` and `pay` are now optional; they'll default to `None` and `0` if omitted. The `self` argument, as usual, is filled in by Python automatically to refer to the instance object—assigning values to attributes of `self` attaches them to the new instance.

Testing As You Go

This class doesn't do much yet—it essentially just fills out the fields of a new record—but it's a real working class. At this point we could add more code to it for more features, but we won't do that yet. As you've probably begun to appreciate already, programming in Python is really a matter of *incremental prototyping*—you write some code, test it, write more code, test again, and so on. Because Python provides both an interactive session and nearly immediate turnaround after code changes, it's more natural to test as you go than to write a huge amount of code to test all at once.

Before adding more features, then, let's test what we've got so far by making a few instances of our class and displaying their attributes as created by the constructor. We could do this interactively, but as you've also probably surmised by now, interactive testing has its limits—it gets tedious to have to reimport modules and retype test cases each time you start a new testing session. More commonly, Python programmers use

the interactive prompt for simple one-off tests but do more substantial testing by writing code at the bottom of the file that contains the objects to be tested, like this:

```
# Add incremental self-test code

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

bob = Person('Bob Smith')                          # Test the class
sue = Person('Sue Jones', job='dev', pay=100000)   # Runs __init__ automatically
print(bob.name, bob.pay)                           # Fetch attached attributes
print(sue.name, sue.pay)                           # sue's and bob's attrs differ
```

Notice here that the `bob` object accepts the defaults for `job` and `pay`, but `sue` provides values explicitly. Also note how we use *keyword arguments* when making `sue`; we could pass by position instead, but the keywords may help remind us later what the data is (and they allow us to pass the arguments in any left-to-right order we like). Again, despite its unusual name, `__init__` is a normal function, supporting everything you already know about functions—including both defaults and pass-by-name keyword arguments.

When this file runs as a script, the test code at the bottom makes two instances of our class and prints two attributes of each (`name` and `pay`):

```
C:\misc> person.py
Bob Smith 0
Sue Jones 100000
```

You can also type this file's test code at Python's interactive prompt (assuming you import the `Person` class there first), but coding canned tests inside the module file like this makes it much easier to rerun them in the future.

Although this is fairly simple code, it's already demonstrating something important. Notice that `bob`'s `name` is not `sue`'s, and `sue`'s `pay` is not `bob`'s. Each is an independent record of information. Technically, `bob` and `sue` are both *namespace objects*—like all class instances, they each have their own independent copy of the state information created by the class. Because each instance of a class has its own set of `self` attributes, classes are a natural for recording information for multiple objects this way; just like built-in types, classes serve as a sort of *object factory*. Other Python program structures, such as functions and modules, have no such concept.

Using Code Two Ways

As is, the test code at the bottom of the file works, but there's a big catch—its top-level `print` statements run both when the file is run as a script and when it is imported as a module. This means if we ever decide to import the class in this file in order to use it somewhere else (and we will later in this chapter), we'll see the output of its test code

every time the file is imported. That's not very good software citizenship, though: client programs probably don't care about our internal tests and won't want to see our output mixed in with their own.

Although we could split the test code off into a separate file, it's often more convenient to code tests in the same file as the items to be tested. It would be better to arrange to run the test statements at the bottom *only* when the file is run for testing, not when the file is imported. That's exactly what the module `__name__` check is designed for, as you learned in the preceding part of this book. Here's what this addition looks like:

```
# Allow this file to be imported as well as run/tested

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

if __name__ == '__main__':                  # When run for testing only
    # self-test code
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
```

Now, we get exactly the behavior we're after—running the file as a top-level script tests it because its `__name__` is `__main__`, but importing it as a library of classes later does not:

```
C:\misc> person.py
Bob Smith 0
Sue Jones 100000

c:\misc> python
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) ...
>>> import person
>>>
```

When imported, the file now defines the class, but does not use it. When run directly, this file creates two instances of our class as before, and prints two attributes of each; again, because each instance is an independent namespace object, the values of their attributes differ.

Version Portability Note

I'm running all the code in this chapter under Python 3.0, and using the 3.0 `print` function call syntax. If you run under 2.6 the code will work as-is, but you'll notice parentheses around some output lines because the extra parentheses in `print`s turn multiple items into a tuple:

```
c:\misc> c:\python26\python person.py
('Bob Smith', 0)
('Sue Jones', 100000)
```

If this difference is the sort of detail that might keep you awake at nights, simply remove the parentheses to use 2.6 `print` statements. You can also avoid the extra parentheses portably by using formatting to yield a single object to print. Either of the following works in both 2.6 and 3.0, though the method form is newer:

```
print('{0} {1}'.format(bob.name, bob.pay))     # New format method
print('%s %s' % (bob.name, bob.pay))           # Format expression
```

Step 2: Adding Behavior Methods

Everything looks good so far—at this point, our class is essentially a record *factory*; it creates and fills out fields of records (attributes of instances, in more Pythonic terms). Even as limited as it is, though, we can still run some operations on its objects. Although classes add an extra layer of structure, they ultimately do most of their work by embedding and processing basic *core data types* like lists and strings. In other words, if you already know how to use Python's simple core types, you already know much of the Python class story; classes are really just a minor structural extension.

For example, the `name` field of our objects is a simple string, so we can extract last names from our objects by splitting on spaces and indexing. These are all core data type operations, which work whether their subjects are embedded in class instances or not:

```
>>> name = 'Bob Smith'       # Simple string, outside class
>>> name.split()             # Extract last name
['Bob', 'Smith']
>>> name.split()[-1]         # Or [1], if always just two parts
'Smith'
```

Similarly, we can give an object a pay raise by updating its `pay` field—that is, by changing its state information in-place with an assignment. This task also involves basic operations that work on Python's core objects, regardless of whether they are standalone or embedded in a class structure:

```
>>> pay = 100000             # Simple variable, outside class
>>> pay *= 1.10              # Give a 10% raise
>>> print(pay)               # Or: pay = pay * 1.10, if you like to type
110000.0                     # Or: pay = pay + (pay * .10), if you _really_ do!
```

To apply these operations to the `Person` objects created by our script, simply do to `bob.name` and `sue.pay` what we just did to `name` and `pay`. The operations are the same, but the subject objects are attached to attributes in our class structure:

```
# Process embedded built-in types: strings, mutability

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

if __name__ == '__main__':
```

```
bob = Person('Bob Smith')
sue = Person('Sue Jones', job='dev', pay=100000)
print(bob.name, bob.pay)
print(sue.name, sue.pay)
print(bob.name.split()[-1])            # Extract object's last name
sue.pay *= 1.10                        # Give this object a raise
print(sue.pay)
```

We've added the last two lines here; when they're run, we extract `bob`'s last name by using basic string and list operations and give `sue` a pay raise by modifying her `pay` attribute in-place with basic number operations. In a sense, `sue` is also a *mutable* object—her state changes in-place just like a list after an `append` call:

```
Bob Smith 0
Sue Jones 100000
Smith
110000.0
```

The preceding code works as planned, but if you show it to a veteran software developer he'll probably tell you that its general approach is not a great idea in practice. Hardcoding operations like these *outside* of the class can lead to maintenance problems in the future.

For example, what if you've hardcoded the last-name-extraction formula at many different places in your program? If you ever need to change the way it works (to support a new name structure, for instance), you'll need to hunt down and update *every* occurrence. Similarly, if the pay-raise code ever changes (e.g., to require approval or database updates), you may have multiple copies to modify. Just finding all the appearances of such code may be problematic in larger programs—they may be scattered across many files, split into individual steps, and so on.

Coding Methods

What we really want to do here is employ a software design concept known as *encapsulation*. The idea with encapsulation is to wrap up operation logic behind interfaces, such that each operation is coded only once in our program. That way, if our needs change in the future, there is just one copy to update. Moreover, we're free to change the single copy's internals almost arbitrarily, without breaking the code that uses it.

In Python terms, we want to code operations on objects in class *methods*, instead of littering them throughout our program. In fact, this is one of the things that classes are very good at—*factoring* code to remove redundancy and thus optimize maintainability. As an added bonus, turning operations into methods enables them to be applied to any instance of the class, not just those that they've been hardcoded to process.

This is all simpler in code than it may sound in theory. The following achieves encapsulation by moving the two operations from code outside the class into class methods. While we're at it, let's change our self-test code at the bottom to use the new methods we're creating, instead of hardcoding operations:

```
# Add methods to encapsulate operations for maintainability

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):                          # Behavior methods
        return self.name.split()[-1]             # self is implied subject
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))     # Must change here only

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.lastName(), sue.lastName())        # Use the new methods
    sue.giveRaise(.10)                           # instead of hardcoding
    print(sue.pay)
```

As we've learned, *methods* are simply normal functions that are attached to classes and designed to process instances of those classes. The instance is the subject of the method call and is passed to the method's `self` argument automatically.

The transformation to the methods in this version is straightforward. The new `lastName` method, for example, simply does to `self` what the previous version hardcoded for `bob`, because `self` is the implied subject when the method is called. `lastName` also returns the result, because this operation is a called function now; it computes a value for its caller to use, even if it is just to be printed. Similarly, the new `giveRaise` method just does to `self` what we did to `sue` before.

When run now, our file's output is similar to before—we've mostly just *refactored* the code to allow for easier changes in the future, not altered its behavior:

```
Bob Smith 0
Sue Jones 100000
Smith Jones
110000
```

A few coding details are worth pointing out here. First, notice that `sue`'s pay is now still an *integer* after a pay raise—we convert the math result back to an integer by calling the `int` built-in within the method. Changing the value to either `int` or `float` is probably not a significant concern for most purposes (integer and floating-point objects have the same interfaces and can be mixed within expressions), but we may need to address rounding issues in a real system (money probably matters to `Persons`!).

As we learned in Chapter 5, we might handle this by using the `round(N, 2)` built-in to round and retain cents, using the `decimal` type to fix precision, or storing monetary values as full floating-point numbers and displaying them with a `%.2f` or `{0:.2f}` formatting string to show cents. For this example, we'll simply truncate any cents with

`int`. (For another idea, also see the `money` function in the *formats.py* module of Chapter 24; you can import this tool to show pay with commas, cents, and dollar signs.)

Second, notice that we're also printing `sue`'s last name this time—because the last-name logic has been encapsulated in a method, we get to use it on *any instance* of the class. As we've seen, Python tells a method which instance to process by automatically passing it in to the first argument, usually called `self`. Specifically:

- In the first call, `bob.lastName()`, `bob` is the implied subject passed to `self`.
- In the second call, `sue.lastName()`, `sue` goes to `self` instead.

Trace through these calls to see how the instance winds up in `self`. The net effect is that the method fetches the name of the implied subject each time. The same happens for `giveRaise`. We could, for example, give `bob` a raise by calling `giveRaise` for both instances this way, too; but unfortunately, `bob`'s zero pay will prevent him from getting a raise as the program is currently coded (something we may want to address in a future 2.0 release of our software).

Finally, notice that the `giveRaise` method assumes that `percent` is passed in as a floating-point number between zero and one. That may be too radical an assumption in the real world (a 1000% raise would probably be a bug for most of us!); we'll let it pass for this prototype, but we might want to test or at least document this in a future iteration of this code. Stay tuned for a rehash of this idea in a later chapter in this book, where we'll code something called *function decorators* and explore Python's `assert` statement—alternatives that can do the validity test for us automatically during development.

Step 3: Operator Overloading

At this point, we have a fairly full-featured class that generates and initializes instances, along with two new bits of behavior for processing instances (in the form of methods). So far, so good.

As it stands, though, testing is still a bit less convenient than it needs to be—to trace our objects, we have to manually fetch and print *individual attributes* (e.g., `bob.name`, `sue.pay`). It would be nice if displaying an instance all at once actually gave us some useful information. Unfortunately, the default display format for an instance object isn't very good—it displays the object's class name, and its address in memory (which is essentially useless in Python, except as a unique identifier).

To see this, change the last line in the script to `print(sue)` so it displays the object as a whole. Here's what you'll get (the output says that `sue` is an "object" in 3.0 and an "instance" in 2.6):

```
Bob Smith 0
Sue Jones 100000
Smith Jones
<__main__.Person object at 0x02614430>
```

Providing Print Displays

Fortunately, it's easy to do better by employing *operator overloading*—coding methods in a class that intercept and process built-in operations when run on the class's instances. Specifically, we can make use of what is probably the second most commonly used operator overloading method in Python, after `__init__`: the `__str__` method introduced in the preceding chapter. `__str__` is run automatically every time an instance is converted to its print string. Because that's what printing an object does, the net transitive effect is that printing an object displays whatever is returned by the object's `__str__` method, if it either defines one itself or inherits one from a superclass (double-underscored names are inherited just like any other).

Technically speaking, the `__init__` constructor method we've already coded is operator overloading too—it is run automatically at construction time to initialize a newly created instance. Constructors are so common, though, that they almost seem like a special case. More focused methods like `__str__` allow us to tap into specific operations and provide *specialized behavior* when our objects are used in those contexts.

Let's put this into code. The following extends our class to give a custom display that lists attributes when our class's instances are displayed as a whole, instead of relying on the less useful default display:

```
# Add __str__ overload method for printing objects

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):                                  # Added method
        return '[Person: %s, %s]' % (self.name, self.pay)      # String to print

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
```

Notice that we're doing string `%` formatting to build the display string in `__str__` here; at the bottom, classes use built-in type objects and operations like these to get their work done. Again, everything you've already learned about both built-in types and functions applies to class-based code. Classes largely just add an additional layer of *structure* that packages functions and data together and supports extensions.

We've also changed our self-test code to print objects directly, instead of printing individual attributes. When run, the output is more coherent and meaningful now; the "[...]" lines are returned by our new `__str__`, run automatically by print operations:

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
```

Here's a subtle point: as we'll learn in the next chapter, a related overloading method, `__repr__`, provides an as-code low-level display of an object when present. Sometimes classes provide both a `__str__` for user-friendly displays and a `__repr__` with extra details for developers to view. Because printing runs `__str__` and the interactive prompt echoes results with `__repr__`, this can provide both target audiences with an appropriate display. Since we're not interested in displaying an as-code format, `__str__` is sufficient for our class.

Step 4: Customizing Behavior by Subclassing

At this point, our class captures much of the OOP machinery in Python: it makes instances, provides behavior in methods, and even does a bit of operator overloading now to intercept print operations in `__str__`. It effectively packages our data and logic together into a single, self-contained *software component*, making it easy to locate code and straightforward to change it in the future. By allowing us to encapsulate behavior, it also allows us to factor that code to avoid redundancy and its associated maintenance headaches.

The only major OOP concept it does not yet capture is *customization by inheritance*. In some sense, we're already doing inheritance, because instances inherit methods from their classes. To demonstrate the real power of OOP, though, we need to define a superclass/subclass relationship that allows us to extend our software and replace bits of inherited behavior. That's the main idea behind OOP, after all; by fostering a coding model based upon customization of work already done, it can dramatically cut development time.

Coding Subclasses

As a next step, then, let's put OOP's methodology to use and customize our `Person` class by extending our software hierarchy. For the purpose of this tutorial, we'll define a subclass of `Person` called `Manager` that replaces the inherited `giveRaise` method with a more specialized version. Our new class begins as follows:

```
class Manager(Person):                    # Define a subclass of Person
```

This code means that we're defining a new class named `Manager`, which inherits from and may add customizations to the superclass `Person`. In plain terms, a `Manager` is almost

like a Person (admittedly, a very long journey for a very small joke...), but Manager has a custom way to give raises.

For the sake of argument, let's assume that when a Manager gets a raise, it receives the passed-in percentage as usual, but also gets an extra bonus that defaults to 10%. For instance, if a Manager's raise is specified as 10%, it will really get 20%. (Any relation to Persons living or dead is, of course, strictly coincidental.) Our new method begins as follows; because this redefinition of giveRaise will be closer in the class tree to Manager instances than the original version in Person, it effectively replaces, and thereby customizes, the operation. Recall that according to the inheritance search rules, the *lowest* version of the name wins:

```
class Manager(Person):                          # Inherit Person attrs
    def giveRaise(self, percent, bonus=.10):    # Redefine to customize
```

Augmenting Methods: The Bad Way

Now, there are two ways we might code this Manager customization: a good way and a bad way. Let's start with the *bad way*, since it might be a bit easier to understand. The bad way is to cut and paste the code of giveRaise in Person and modify it for Manager, like this:

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        self.pay = int(self.pay * (1 + percent + bonus))    # Bad: cut-and-paste
```

This works as advertised—when we later call the giveRaise method of a Manager instance, it will run this custom version, which tacks on the extra bonus. So what's wrong with something that runs correctly?

The problem here is a very general one: any time you copy code with cut and paste, you essentially *double* your maintenance effort in the future. Think about it: because we copied the original version, if we ever have to change the way raises are given (and we probably will), we'll have to change the code in *two* places, not one. Although this is a small and artificial example, it's also representative of a universal issue—any time you're tempted to program by copying code this way, you probably want to look for a better approach.

Augmenting Methods: The Good Way

What we really want to do here is somehow *augment* the original giveRaise, instead of replacing it altogether. The *good way* to do that in Python is by calling to the original version directly, with augmented arguments, like this:

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)             # Good: augment original
```

This code leverages the fact that a class method can always be called either through an *instance* (the usual way, where Python sends the instance to the `self` argument automatically) or through the *class* (the less common scheme, where you must pass the instance manually). In more symbolic terms, recall that a normal method call of this form:

```
instance.method(args...)
```

is automatically translated by Python into this equivalent form:

```
class.method(instance, args...)
```

where the class containing the method to be run is determined by the inheritance search rule applied to the method's name. You can code *either* form in your script, but there is a slight asymmetry between the two—you must remember to pass along the instance manually if you call through the class directly. The method always needs a subject instance one way or another, and Python provides it automatically only for calls made through an instance. For calls through the class name, you need to send an instance to `self` yourself; for code inside a method like `giveRaise`, `self` already *is* the subject of the call, and hence the instance to pass along.

Calling through the class directly effectively subverts inheritance and kicks the call higher up the class tree to run a specific version. In our case, we can use this technique to invoke the default `giveRaise` in `Person`, even though it's been redefined at the `Manager` level. In some sense, we *must* call through `Person` this way, because a `self.giveRaise()` inside `Manager`'s `giveRaise` code would loop—since `self` already is a `Manager`, `self.giveRaise()` would resolve again to `Manager.giveRaise`, and so on and so forth until available memory is exhausted.

This "good" version may seem like a small difference in code, but it can make a huge difference for future *code maintenance*—because the `giveRaise` logic lives in just one place now (`Person`'s method), we have only one version to change in the future as needs evolve. And really, this form captures our intent more directly anyhow—we want to perform the standard `giveRaise` operation, but simply tack on an extra bonus. Here's our entire module file with this step applied:

```
# Add customization of one behavior in a subclass

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):
```

```
    def giveRaise(self, percent, bonus=.10):          # Redefine at this level
        Person.giveRaise(self, percent + bonus)       # Call Person's version

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 'mgr', 50000)          # Make a Manager: __init__
    tom.giveRaise(.10)                                # Runs custom version
    print(tom.lastName())                             # Runs inherited method
    print(tom)                                        # Runs inherited __str__
```

To test our `Manager` subclass customization, we've also added self-test code that makes a `Manager`, calls its methods, and prints it. Here's the new version's output:

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
```

Everything looks good here: `bob` and `sue` are as before, and when `tom` the `Manager` is given a 10% raise, he really gets 20% (his pay goes from $50K to $60K), because the customized `giveRaise` in `Manager` is run for him only. Also notice how printing `tom` as a whole at the end of the test code displays the nice format defined in `Person`'s `__str__`: `Manager` objects get this, `lastName`, and the `__init__` constructor method's code "for free" from `Person`, by inheritance.

Polymorphism in Action

To make this acquisition of inherited behavior even more striking, we can add the following code at the end of our file:

```
if __name__ == '__main__':
    ...
    print('--All three--')
    for object in (bob, sue, tom):                    # Process objects generically
        object.giveRaise(.10)                         # Run this object's giveRaise
        print(object)                                 # Run the common __str__
```

Here's the resulting output:

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
--All three--
```

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 121000]
[Person: Tom Jones, 72000]
```

In the added code, `object` is *either* a `Person` or a `Manager`, and Python runs the appropriate `giveRaise` automatically—our original version in `Person` for `bob` and `sue`, and our customized version in `Manager` for `tom`. Trace the method calls yourself to see how Python selects the right `giveRaise` method for each object.

This is just Python's notion of *polymorphism*, which we met earlier in the book, at work again—what `giveRaise` does depends on what you do it to. Here, it's made all the more obvious when it selects from code we've written ourselves in classes. The practical effect in this code is that `sue` gets another 10% but `tom` gets another 20%, because `giveRaise` is dispatched based upon the object's type. As we've learned, polymorphism is at the heart of Python's flexibility. Passing any of our three objects to a function that calls a `giveRaise` method, for example, would have the same effect: the appropriate version would be run automatically, depending on which type of object was passed.

On the other hand, printing runs the *same* `__str__` for all three objects, because it's coded just once in `Person`. `Manager` both specializes and applies the code we originally wrote in `Person`. Although this example is small, it's already leveraging OOP's talent for code customization and reuse; with classes, this almost seems automatic at times.

Inherit, Customize, and Extend

In fact, classes can be even more flexible than our example implies. In general, classes can *inherit*, *customize*, or *extend* existing code in superclasses. For example, although we're focused on customization here, we can also add unique methods to `Manager` that are not present in `Person`, if `Manager`s require something completely different (Python namesake reference intended). The following snippet illustrates. Here, `giveRaise` redefines a superclass method to customize it, but `someThingElse` defines something new to extend:

```
class Person:
    def lastName(self): ...
    def giveRaise(self): ...
    def __str__(self): ...

class Manager(Person):                      # Inherit
    def giveRaise(self, ...): ...           # Customize
    def someThingElse(self, ...): ...       # Extend

tom = Manager()
tom.lastName()                              # Inherited verbatim
tom.giveRaise()                             # Customized version
tom.someThingElse()                         # Extension here
print(tom)                                  # Inherited overload method
```

Extra methods like this code's `someThingElse` *extend* the existing software and are available on `Manager` objects only, not on `Person`s. For the purposes of this tutorial, however,

we'll limit our scope to customizing some of `Person`'s behavior by redefining it, not adding to it.

OOP: The Big Idea

As is, our code may be small, but it's fairly functional. And really, it already illustrates the main point behind OOP in general: in OOP, we program by *customizing* what has already been done, rather than copying or changing existing code. This isn't always an obvious win to newcomers at first glance, especially given the extra coding requirements of classes. But overall, the programming style implied by classes can cut development time radically compared to other approaches.

For instance, in our example we could theoretically have implemented a custom `giveRaise` operation without subclassing, but none of the other options yield code as optimal as ours:

- Although we could have simply coded `Manager` *from scratch* as new, independent code, we would have had to reimplement all the behaviors in `Person` that are the same for `Manager`s.
- Although we could have simply *changed* the existing `Person` class in-place for the requirements of `Manager`'s `giveRaise`, doing so would probably break the places where we still need the original `Person` behavior.
- Although we could have simply *copied* the `Person` class in its entirety, renamed the copy to `Manager`, and changed its `giveRaise`, doing so would introduce code redundancy that would double our work in the future—changes made to `Person` in the future would not be picked up automatically, but would have to be manually propagated to `Manager`'s code. As usual, the cut-and-paste approach may seem quick now, but it doubles your work in the future.

The *customizable hierarchies* we can build with classes provide a much better solution for software that will evolve over time. No other tools in Python support this development mode. Because we can tailor and extend our prior work by coding new subclasses, we can leverage what we've already done, rather than starting from scratch each time, breaking what already works, or introducing multiple copies of code that may all have to be updated in the future. When done right, OOP is a powerful programmer's ally.

Step 5: Customizing Constructors, Too

Our code works as it is, but if you study the current version closely, you may be struck by something a bit odd—it seems pointless to have to provide a `mgr` job name for `Manager` objects when we create them: this is already implied by the class itself. It would be better if we could somehow fill in this value automatically when a `Manager` is made.

The trick we need to improve on this turns out to be the *same* as the one we employed in the prior section: we want to customize the constructor logic for `Manager`s in such a

way as to provide a job name automatically. In terms of code, we want to redefine an `__init__` method in `Manager` that provides the `mgr` string for us. And like with the `giveRaise` customization, we also want to run the original `__init__` in `Person` by calling through the class name, so it still initializes our objects' state information attributes.

The following extension will do the job—we've coded the new `Manager` constructor and changed the call that creates `tom` to not pass in the `mgr` job name:

```
# Add customization of constructor in a subclass

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):
    def __init__(self, name, pay):                    # Redefine constructor
        Person.__init__(self, name, 'mgr', pay)       # Run original with 'mgr'
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 50000)                 # Job name not needed:
    tom.giveRaise(.10)                                # Implied/set by class
    print(tom.lastName())
    print(tom)
```

Again, we're using the same technique to augment the `__init__` constructor here that we used for `giveRaise` earlier—running the superclass version by calling through the class name directly and passing the `self` instance along explicitly. Although the constructor has a strange name, the effect is identical. Because we need `Person`'s construction logic to run too (to initialize instance attributes), we really have to call it this way; otherwise, instances would not have any attributes attached.

Calling superclass constructors from redefinitions this way turns out to be a very common coding pattern in Python. By itself, Python uses inheritance to look for and call only *one* `__init__` method at construction time—the *lowest* one in the class tree. If you need higher `__init__` methods to be run at construction time (and you usually do),

you must call them manually through the superclass's name. The upside to this is that you can be explicit about which argument to pass up to the superclass's constructor and can choose to not call it at all: not calling the superclass constructor allows you to replace its logic altogether, rather than augmenting it.

The output of this file's self-test code is the same as before—we haven't changed what it does, we've simply restructured to get rid of some logical redundancy:

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
```

OOP Is Simpler Than You May Think

In this complete form, despite their sizes, our classes capture nearly all the important concepts in Python's OOP machinery:

- Instance creation—filling out instance attributes
- Behavior methods—encapsulating logic in class methods
- Operator overloading—providing behavior for built-in operations like printing
- Customizing behavior—redefining methods in subclasses to specialize them
- Customizing constructors—adding initialization logic to superclass steps

Most of these concepts are based upon just three simple ideas: the inheritance search for attributes in object trees, the special `self` argument in methods, and operator overloading's automatic dispatch to methods.

Along the way, we've also made our code easy to change in the future, by harnessing the class's propensity for factoring code to reduce *redundancy*. For example, we wrapped up logic in methods and called back to superclass methods from extensions to avoid having multiple copies of the same code. Most of these steps were a natural outgrowth of the structuring power of classes.

By and large, that's all there is to OOP in Python. Classes certainly can become larger than this, and there are some more advanced class concepts, such as decorators and metaclasses, which we will meet in later chapters. In terms of the basics, though, our classes already do it all. In fact, if you've grasped the workings of the classes we've written, most OOP Python code should now be within your reach.

Other Ways to Combine Classes

Having said that, I should also tell you that although the basic mechanics of OOP are simple in Python, some of the art in larger programs lies in the way that classes are put together. We're focusing on *inheritance* in this tutorial because that's the mechanism

the Python language provides, but programmers sometimes combine classes in other ways, too. For example, a common coding pattern involves nesting objects inside each other to build up *composites*. We'll explore this pattern in more detail in Chapter 30, which is really more about design than about Python; as a quick example, though, we could use this composition idea to code our `Manager` extension by *embedding* a `Person`, instead of inheriting from it.

The following alternative does so by using the `__getattr__` operator overloading method we will meet in Chapter 29 to intercept undefined attribute fetches and delegate them to the embedded object with the `getattr` built-in. The `giveRaise` method here still achieves customization, by changing the argument passed along to the embedded object. In effect, `Manager` becomes a controller layer that passes calls *down* to the embedded object, rather than *up* to superclass methods:

```
# Embedding-based Manager alternative

class Person:
    ...same...

class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'mgr', pay)      # Embed a Person object
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)      # Intercept and delegate
    def __getattr__(self, attr):
        return getattr(self.person, attr)           # Delegate all other attrs
    def __str__(self):
        return str(self.person)                     # Must overload again (in 3.0)

if __name__ == '__main__':
    ...same...
```

In fact, this `Manager` alternative is representative of a general coding pattern usually known as *delegation*—a composite-based structure that manages a wrapped object and propagates method calls to it. This pattern works in our example, but it requires about twice as much code and is less well suited than inheritance to the kinds of direct customizations we meant to express (in fact, no reasonable Python programmer would code this example this way in practice, except perhaps those writing general tutorials). `Manager` isn't really a `Person` here, so we need extra code to manually dispatch method calls to the embedded object; operator overloading methods like `__str__` must be redefined (in 3.0, at least, as noted in the upcoming sidebar "Catching Built-in Attributes in 3.0" on page 662), and adding new `Manager` behavior is less straightforward since state information is one level removed.

Still, *object embedding*, and design patterns based upon it, can be a very good fit when embedded objects require more limited interaction with the container than direct customization implies. A controller layer like this alternative `Manager`, for example, might come in handy if we want to trace or validate calls to another object's methods (indeed, we will use a nearly identical coding pattern when we study *class decorators* later in the

book). Moreover, a hypothetical `Department` class like the following could *aggregate* other objects in order to treat them as a set. Add this to the bottom of the *person.py* file to try this on your own:

```
# Aggregate embedded objects into a composite

...
bob = Person(...)
sue = Person(...)
tom = Manager(...)

class Department:
    def __init__(self, *args):
        self.members = list(args)
    def addMember(self, person):
        self.members.append(person)
    def giveRaises(self, percent):
        for person in self.members:
            person.giveRaise(percent)
    def showAll(self):
        for person in self.members:
            print(person)

development = Department(bob, sue)          # Embed objects in a composite
development.addMember(tom)
development.giveRaises(.10)                 # Runs embedded objects' giveRaise
development.showAll()                       # Runs embedded objects' __str__s
```

Interestingly, this code uses both inheritance *and* composition—`Department` is a composite that embeds and controls other objects to aggregate, but the embedded `Person` and `Manager` objects themselves use inheritance to customize. As another example, a GUI might similarly use *inheritance* to customize the behavior or appearance of labels and buttons, but also *composition* to build up larger packages of embedded widgets, such as input forms, calculators, and text editors. The class structure to use depends on the objects you are trying to model.

Design issues like composition are explored in Chapter 30, so we'll postpone further investigations for now. But again, in terms of the basic mechanics of OOP in Python, our `Person` and `Manager` classes already tell the entire story. Having mastered the basics of OOP, though, developing general tools for applying it more easily in your scripts is often a natural next step—and the topic of the next section.

Catching Built-in Attributes in 3.0

In Python 3.0 (and 2.6 if new-style classes are used), the alternative delegation-based `Manager` class we just coded will not be able to intercept and delegate operator overloading method attributes like `__str__` without redefining them. Although we know that `__str__` is the only such name used in our specific example, this a general issue for delegation-based classes.

Recall that built-in operations like printing and indexing implicitly invoke operator overloading methods such as `__str__` and `__getitem__`. In 3.0, built-in operations like

these do not route their implicit attribute fetches through generic attribute managers: neither `__getattr__` (run for undefined attributes) nor its cousin `__getattribute__` (run for all attributes) is invoked. This is why we have to redefine `__str__` redundantly in the alternative `Manager`, in order to ensure that printing is routed to the embedded `Person` object when run in Python 3.0.

Technically, this happens because classic classes normally look up operator overloading names in instances at runtime, but new-style classes do not—they skip the instance entirely and look up such methods in classes. In 2.6 classic classes, built-ins *do* route attributes generically—printing, for example, routes `__str__` through `__getattr__`. New-style classes also inherit a default for `__str__` that would foil `__getattr__`, but `__getattribute__` doesn't intercept the name in 3.0 either.

This is a change, but isn't a show-stopper—delegation-based classes can generally redefine operator overloading methods to delegate them to wrapped objects in 3.0, either manually or via tools or superclasses. This topic is too advanced to explore further in this tutorial, though, so don't sweat the details too much here. Watch for it to be revisited in the attribute management coverage of Chapter 37, and again in the context of `Private` class decorators in Chapter 38.

Step 6: Using Introspection Tools

Let's make one final tweak before we throw our objects onto a database. As they are, our classes are complete and demonstrate most of the basics of OOP in Python. They still have two remaining issues we probably should iron out, though, before we go live with them:

- First, if you look at the display of the objects as they are right now, you'll notice that when you print `tom` the `Manager` labels him as a `Person`. That's not technically incorrect, since `Manager` is a kind of customized and specialized `Person`. Still, it would be more accurate to display objects with the most specific (that is, *lowest*) classes possible.
- Second, and perhaps more importantly, the current display format shows *only* the attributes we include in our `__str__`, and that might not account for future goals. For example, we can't yet verify that `tom`'s job name has been set to `mgr` correctly by `Manager`'s constructor, because the `__str__` we coded for `Person` does not print this field. Worse, if we ever expand or otherwise change the set of attributes assigned to our objects in `__init__`, we'll have to remember to also update `__str__` for new names to be displayed, or it will become out of sync over time.

The last point means that, yet again, we've made potential extra work for ourselves in the future by introducing *redundancy* in our code. Because any disparity in `__str__` will be reflected in the program's output, this redundancy may be more obvious than the other forms we addressed earlier; still, avoiding extra work in the future is generally *a good thing*.

Special Class Attributes

We can address both issues with Python's *introspection tools*—special attributes and functions that give us access to some of the internals of objects' implementations. These tools are somewhat advanced and generally used more by people writing tools for other programmers to use than by programmers developing applications. Even so, a basic knowledge of some of these tools is useful because they allow us to write code that processes classes in generic ways. In our code, for example, there are two hooks that can help us out, both of which were introduced near the end of the preceding chapter:

- The built-in *`instance`*`.__class__` attribute provides a link from an instance to the class from which it was created. Classes in turn have a `__name__`, just like modules, and a `__bases__` sequence that provides access to superclasses. We can use these here to print the name of the class from which an instance is made rather than one we've hardcoded.
- The built-in *`object`*`.__dict__` attribute provides a dictionary with one key/value pair for every attribute attached to a namespace object (including modules, classes, and instances). Because it is a dictionary, we can fetch its keys list, index by key, iterate over its keys, and so on, to process all attributes generically. We can use this here to print every attribute in any instance, not just those we hardcode in custom displays.

Here's what these tools look like in action at Python's interactive prompt. Notice how we load `Person` at the interactive prompt with a `from` statement here—class names live in and are imported from modules, exactly like function names and other variables:

```
>>> from person import Person
>>> bob = Person('Bob Smith')
>>> print(bob)                                  # Show bob's __str__
[Person: Bob Smith, 0]

>>> bob.__class__                               # Show bob's class and its name
<class 'person.Person'>
>>> bob.__class__.__name__
'Person'

>>> list(bob.__dict__.keys())                   # Attributes are really dict keys
['pay', 'job', 'name']                          # Use list to force list in 3.0

>>> for key in bob.__dict__:
        print(key, '=>', bob.__dict__[key])     # Index manually

pay => 0
job => None
name => Bob Smith

>>> for key in bob.__dict__:
        print(key, '=>', getattr(bob, key))     # obj.attr, but attr is a var

pay => 0
```

```
job => None
name => Bob Smith
```

As noted briefly in the prior chapter, some attributes accessible from an instance might not be stored in the `__dict__` dictionary if the instance's class defines `__slots__`, an optional and relatively obscure feature of new-style classes (and all classes in Python 3.0) that stores attributes in an array and that we'll discuss in Chapters 30 and 31. Since slots really belong to classes instead of instances, and since they are very rarely used in any event, we can safely ignore them here and focus on the normal `__dict__`.

A Generic Display Tool

We can put these interfaces to work in a superclass that displays accurate class names and formats all attributes of an instance of any class. Open a new file in your text editor to code the following—it's a new, independent module named *classtools.py* that implements just such a class. Because its `__str__` print overload uses generic introspection tools, it will work on *any instance*, regardless of its attributes set. And because this is a class, it automatically becomes a general formatting tool: thanks to inheritance, it can be mixed into *any class* that wishes to use its display format. As an added bonus, if we ever want to change how instances are displayed we need only change this class, as every class that inherits its `__str__` will automatically pick up the new format when it's next run:

```
# File classtools.py (new)
"Assorted class utilities and tools"

class AttrDisplay:
    """
    Provides an inheritable print overload method that displays
    instances with their class names and a name=value pair for
    each attribute stored on the instance itself (but not attrs
    inherited from its classes). Can be mixed into any class,
    and will work on any instance.
    """
    def gatherAttrs(self):
        attrs = []
        for key in sorted(self.__dict__):
            attrs.append('%s=%s' % (key, getattr(self, key)))
        return ', '.join(attrs)
    def __str__(self):
        return '[%s: %s]' % (self.__class__.__name__, self.gatherAttrs())


if __name__ == '__main__':
    class TopTest(AttrDisplay):
        count = 0
        def __init__(self):
            self.attr1 = TopTest.count
            self.attr2 = TopTest.count+1
            TopTest.count += 2
```

```
class SubTest(TopTest):
    pass

X, Y = TopTest(), SubTest()
print(X)                        # Show all instance attrs
print(Y)                        # Show lowest class name
```

Notice the docstrings here—as a general-purpose tool, we want to add some functional documentation for potential users to read. As we saw in Chapter 15, docstrings can be placed at the top of simple functions and modules, and also at the start of classes and their methods; the `help` function and the PyDoc tool extracts and displays these automatically (we'll look at docstrings again in Chapter 28).

When run directly, this module's self-test makes two instances and prints them; the `__str__` defined here shows the instance's class, and all its attributes names and values, in sorted attribute name order:

```
C:\misc> classtools.py
[TopTest: attr1=0, attr2=1]
[SubTest: attr1=2, attr2=3]
```

Instance Versus Class Attributes

If you study the `classtools` module's self-test code long enough, you'll notice that its class displays only *instance attributes*, attached to the `self` object at the bottom of the inheritance tree; that's what `self`'s `__dict__` contains. As an intended consequence, we don't see attributes inherited by the instance from classes above it in the tree (e.g., `count` in this file's self-test code). Inherited class attributes are attached to the class only, not copied down to instances.

If you ever do wish to include inherited attributes too, you can climb the `__class__` link to the instance's class, use the `__dict__` there to fetch class attributes, and then iterate through the class's `__bases__` attribute to climb to even higher superclasses (repeating as necessary). If you're a fan of simple code, running a built-in `dir` call on the instance instead of using `__dict__` and climbing would have much the same effect, since `dir` results include inherited names in the sorted results list:

```
>>> from person import Person
>>> bob = Person('Bob Smith')

# In Python 2.6:

>>> bob.__dict__.keys()                 # Instance attrs only
['pay', 'job', 'name']

>>> dir(bob)                            # + inherited attrs in classes
['__doc__', '__init__', '__module__', '__str__', 'giveRaise', 'job',
'lastName', 'name', 'pay']

# In Python 3.0:
```

```
>>> list(bob.__dict__.keys())          # 3.0 keys is a view, not a list
['pay', 'job', 'name']

>>> dir(bob)                           # 3.0 includes class type methods
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
...more lines omitted...
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'giveRaise', 'job', 'lastName', 'name', 'pay']
```

The output here varies between Python 2.6 and 3.0, because 3.0's `dict.keys` is not a list, and 3.0's `dir` returns extra class-type implementation attributes. Technically, `dir` returns more in 3.0 because classes are all "new style" and inherit a large set of operator overloading names from the class type. In fact, you'll probably want to filter out most of the `__X__` names in the 3.0 `dir` result, since they are internal implementation details and not something you'd normally want to display.

In the interest of space, we'll leave optional display of inherited class attributes with either tree climbs or `dir` as suggested experiments for now. For more hints on this front, though, watch for the *classtree.py* inheritance tree climber we will write in Chapter 28, and the *lister.py* attribute listers and climbers we'll code in Chapter 30.

Name Considerations in Tool Classes

One last subtlety here: because our `AttrDisplay` class in the `classtools` module is a general tool designed to be mixed into other arbitrary classes, we have to be aware of the potential for unintended *name collisions* with client classes. As is, I've assumed that client subclasses may want to use both its `__str__` and `gatherAttrs`, but the latter of these may be more than a subclass expects—if a subclass innocently defines a `gatherAttrs` name of its own, it will likely break our class, because the lower version in the subclass will be used instead of ours.

To see this for yourself, add a `gatherAttrs` to `TopTest` in the file's self-test code; unless the new method is identical, or intentionally customizes the original, our tool class will no longer work as planned:

```
class TopTest(AttrDisplay):
    ....
    def gatherAttrs(self):          # Replaces method in AttrDisplay!
        return 'Spam'
```

This isn't necessarily bad—sometimes we want other methods to be available to subclasses, either for direct calls or for customization. If we really meant to provide a `__str__` only, though, this is less than ideal.

To minimize the chances of name collisions like this, Python programmers often prefix methods not meant for external use with a *single underscore*: `_gatherAttrs` in our case. This isn't foolproof (what if another class defines `_gatherAttrs`, too?), but it's usually sufficient, and it's a common Python naming convention for methods internal to a class.

A better and less commonly used solution would be to use *two underscores* at the front of the method name only: `__gatherAttrs` for us. Python automatically expands such names to include the enclosing class's name, which makes them truly unique. This is a feature usually called *pseudoprivate class attributes*, which we'll expand on in Chapter 30. For now, we'll make both our methods available.

Our Classes' Final Form

Now, to use this generic tool in our classes, all we need to do is import it from its module, mix it in by inheritance in our top-level class, and get rid of the more specific `__str__` we coded before. The new `print` overload method will be inherited by instances of `Person`, as well as `Manager`; `Manager` gets `__str__` from `Person`, which now obtains it from the `AttrDisplay` coded in another module. Here is the final version of our *person.py* file with these changes applied:

```
# File person.py (final)

from classtools import AttrDisplay                  # Use generic display tool

class Person(AttrDisplay):
    """
    Create and process person records
    """
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):                             # Assumes last is last
        return self.name.split()[-1]
    def giveRaise(self, percent):                   # Percent must be 0..1
        self.pay = int(self.pay * (1 + percent))

class Manager(Person):
    """
    A customized Person with special requirements
    """
    def __init__(self, name, pay):
        Person.__init__(self, name, 'mgr', pay)
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 50000)
    tom.giveRaise(.10)
```

```
print(tom.lastName())
print(tom)
```

As this is the final revision, we've added a few *comments* here to document our work—docstrings for functional descriptions and # for smaller notes, per best-practice conventions. When we run this code now, we see all the attributes of our objects, not just the ones we hardcoded in the original `__str__`. And our final issue is resolved: because `AttrDisplay` takes class names off the `self` instance directly, each object is shown with the name of its closest (lowest) class—`tom` displays as a `Manager` now, not a `Person`, and we can finally verify that his job name has been correctly filled in by the `Manager` constructor:

```
C:\misc> person.py
[Person: job=None, name=Bob Smith, pay=0]
[Person: job=dev, name=Sue Jones, pay=100000]
Smith Jones
[Person: job=dev, name=Sue Jones, pay=110000]
Jones
[Manager: job=mgr, name=Tom Jones, pay=60000]
```

This is the more useful display we were after. From a larger perspective, though, our attribute display class has become a *general tool*, which we can mix into any class by inheritance to leverage the display format it defines. Further, all its clients will automatically pick up future changes in our tool. Later in the book, we'll meet even more powerful class tool concepts, such as decorators and metaclasses; along with Python's introspection tools, they allow us to write code that augments and manages classes in structured and maintainable ways.

Step 7 (Final): Storing Objects in a Database

At this point, our work is almost complete. We now have a *two-module system* that not only implements our original design goals for representing people, but also provides a general attribute display tool we can use in other programs in the future. By coding functions and classes in module files, we've ensured that they naturally support reuse. And by coding our software as classes, we've ensured that it naturally supports extension.

Although our classes work as planned, though, the objects they create are not real database records. That is, if we kill Python, our instances will disappear—they're transient objects in memory and are not stored in a more permanent medium like a file, so they won't be available in future program runs. It turns out that it's easy to make instance objects more permanent, with a Python feature called *object persistence*—making objects live on after the program that creates them exits. As a final step in this tutorial, let's make our objects permanent.

Pickles and Shelves

Object persistence is implemented by three standard library modules, available in every Python:

`pickle`
: Serializes arbitrary Python objects to and from a string of bytes

`dbm` *(named* `anydbm` *in Python 2.6)*
: Implements an access-by-key filesystem for storing strings

`shelve`
: Uses the other two modules to store Python objects on a file by key

We met these modules very briefly in Chapter 9 when we studied file basics. They provide powerful data storage options. Although we can't do them complete justice in this tutorial or book, they are simple enough that a brief introduction is enough to get you started.

The `pickle` module is a sort of super-general object formatting and deformatting tool: given a nearly arbitrary Python object in memory, it's clever enough to convert the object to a string of bytes, which it can use later to reconstruct the original object in memory. The `pickle` module can handle almost any object you can create—lists, dictionaries, nested combinations thereof, and class instances. The latter are especially useful things to pickle, because they provide both data (attributes) and behavior (methods); in fact, the combination is roughly equivalent to "records" and "programs." Because `pickle` is so general, it can replace extra code you might otherwise write to create and parse custom text file representations for your objects. By storing an object's pickle string on a file, you effectively make it permanent and persistent: simply load and unpickle it later to re-create the original object.

Although it's easy to use `pickle` by itself to store objects in simple flat files and load them from there later, the `shelve` module provides an extra layer of structure that allows you to store pickled objects by *key*. `shelve` translates an object to its pickled string with `pickle` and stores that string under a key in a `dbm` file; when later loading, `shelve` fetches the pickled string by key and re-creates the original object in memory with `pickle`. This is all quite a trick, but to your script a shelve* of pickled objects looks just like a *dictionary*—you index by key to fetch, assign to keys to store, and use dictionary tools such as `len`, `in`, and `dict.keys` to get information. Shelves automatically map dictionary operations to objects stored in a file.

In fact, to your script the only coding difference between a shelve and a normal dictionary is that you must *open* shelves initially and must *close* them after making changes. The net effect is that a shelve provides a simple database for storing and fetching native Python objects by keys, and thus makes them persistent across program runs. It does

* Yes, we use "shelve" as a noun in Python, much to the chagrin of a variety of editors I've worked with over the years, both electronic and human.

not support query tools such as SQL, and it lacks some advanced features found in enterprise-level databases (such as true transaction processing), but native Python objects stored on a shelve may be processed with the full power of the Python language once they are fetched back by key.

Storing Objects on a Shelve Database

Pickling and shelves are somewhat advanced topics, and we won't go into all their details here; you can read more about them in the standard library manuals, as well as application-focused books such as *Programming Python*. This is all simpler in Python than in English, though, so let's jump into some code.

Let's write a new script that throws objects of our classes onto a shelve. In your text editor, open a new file we'll call *makedb.py*. Since this is a new file, we'll need to import our classes in order to create a few instances to store. We used `from` to load a class at the interactive prompt earlier, but really, as with functions and other variables, there are two ways to load a class from a file (class names are variables like any other, and not at all magic in this context):

```
import person                          # Load class with import
bob = person.Person(...)               # Go through module name

from person import Person              # Load class with from
bob = Person(...)                      # Use name directly
```

We'll use `from` to load in our script, just because it's a bit less to type. Copy or retype this code to make instances of our classes in the new script, so we have something to store (this is a simple demo, so we won't worry about the test-code redundancy here). Once we have some instances, it's almost trivial to store them on a shelve. We simply import the `shelve` module, open a new shelve with an external filename, assign the objects to keys in the shelve, and close the shelve when we're done because we've made changes:

```
# File makedb.py: store Person objects on a shelve database

from person import Person, Manager             # Load our classes
bob = Person('Bob Smith')                      # Re-create objects to be stored
sue = Person('Sue Jones', job='dev', pay=100000)
tom = Manager('Tom Jones', 50000)

import shelve
db = shelve.open('persondb')                   # Filename where objects are stored
for object in (bob, sue, tom):                 # Use object's name attr as key
    db[object.name] = object                   # Store object on shelve by key
db.close()                                     # Close after making changes
```

Notice how we assign objects to the shelve using their own names as keys. This is just for convenience; in a shelve, the *key* can be any string, including one we might create to be unique using tools such as process IDs and timestamps (available in the `os` and `time` standard library modules). The only rule is that the keys must be strings and should

be unique, since we can store just one object per key (though that object can be a list or dictionary containing many objects). The *values* we store under keys, though, can be Python objects of almost any sort: built-in types like strings, lists, and dictionaries, as well as user-defined class instances, and nested combinations of all of these.

That's all there is to it—if this script has no output when run, it means it probably worked; we're not printing anything, just creating and storing objects:

```
C:\misc> makedb.py
```

Exploring Shelves Interactively

At this point, there are one or more real files in the current directory whose names all start with "persondb". The actual files created can vary per platform, and just like in the built-in `open` function, the filename in `shelve.open()` is relative to the current working directory unless it includes a directory path. Wherever they are stored, these files implement a keyed-access file that contains the pickled representation of our three Python objects. Don't delete these files—they are your database, and are what you'll need to copy or transfer when you back up or move your storage.

You can look at the shelve's files if you want to, either from Windows Explorer or the Python shell, but they are binary hash files, and most of their content makes little sense outside the context of the `shelve` module. With Python 3.0 and no extra software installed, our database is stored in three files (in 2.6, it's just one file, *persondb*, because the `bsddb` extension module is preinstalled with Python for shelves; in 3.0, `bsddb` is a third-party open source add-on):

```
# Directory listing module: verify files are present

>>> import glob
>>> glob.glob('person*')
['person.py', 'person.pyc', 'persondb.bak', 'persondb.dat', 'persondb.dir']

# Type the file: text mode for string, binary mode for bytes

>>> print(open('persondb.dir').read())
'Tom Jones', (1024, 91)
...more omitted...

>>> print(open('persondb.dat', 'rb').read())
b'\x80\x03cperson\nPerson\nq\x00)\x81q\x01}q\x02(X\x03\x00\x00\x00payq\x03K...
...more omitted...
```

This content isn't impossible to decipher, but it can vary on different platforms and doesn't exactly qualify as a user-friendly database interface! To verify our work better, we can write another script, or poke around our shelve at the interactive prompt. Because shelves are Python objects containing Python objects, we can process them with normal Python syntax and development modes. Here, the interactive prompt effectively becomes a *database client*:

```
>>> import shelve
>>> db = shelve.open('persondb')                # Reopen the shelve

>>> len(db)                                     # Three 'records' stored
3
>>> list(db.keys())                             # keys is the index
['Tom Jones', 'Sue Jones', 'Bob Smith']         # list to make a list in 3.0

>>> bob = db['Bob Smith']                       # Fetch bob by key
>>> print(bob)                                  # Runs __str__ from AttrDisplay
[Person: job=None, name=Bob Smith, pay=0]

>>> bob.lastName()                              # Runs lastName from Person
'Smith'

>>> for key in db:                              # Iterate, fetch, print
        print(key, '=>', db[key])

Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]

>>> for key in sorted(db):
        print(key, '=>', db[key])                # Iterate by sorted keys

Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
```

Notice that we don't have to import our `Person` or `Manager` classes here in order to load or use our stored objects. For example, we can call `bob`'s `lastName` method freely, and get his custom print display format automatically, even though we don't have his `Person` class in our scope here. This works because when Python pickles a class instance, it records its `self` instance attributes, along with the name of the class it was created from and the module where the class lives. When `bob` is later fetched from the shelve and unpickled, Python will automatically reimport the class and link `bob` to it.

The upshot of this scheme is that class instances automatically acquire all their class behavior when they are loaded in the future. We have to import our classes only to make new instances, not to process existing ones. Although a deliberate feature, this scheme has somewhat mixed consequences:

- The *downside* is that classes and their module's files must be importable when an instance is later loaded. More formally, pickleable classes must be coded at the top level of a module file accessible from a directory listed on the `sys.path` module search path (and shouldn't live in the most script files' module `__main__` unless they're always in that module when used). Because of this external module file requirement, some applications choose to pickle simpler objects such as dictionaries or lists, especially if they are to be transferred across the Internet.

- The *upside* is that changes in a class's source code file are automatically picked up when instances of the class are loaded again; there is often no need to update stored objects themselves, since updating their class's code changes their behavior.

Shelves also have well-known limitations (the database suggestions at the end of this chapter mention a few of these). For simple object storage, though, shelves and pickles are remarkably easy-to-use tools.

Updating Objects on a Shelve

Now for one last script: let's write a program that updates an instance (record) each time it runs, to prove the point that our objects really are *persistent* (i.e., that their current values are available every time a Python program runs). The following file, *updatedb.py*, prints the database and gives a raise to one of our stored objects each time. If you trace through what's going on here, you'll notice that we're getting a lot of utility "for free"—printing our objects automatically employs the general `__str__` overloading method, and we give raises by calling the `giveRaise` method we wrote earlier. This all "just works" for objects based on OOP's inheritance model, even when they live in a file:

```
# File updatedb.py: update Person object on database

import shelve
db = shelve.open('persondb')                # Reopen shelve with same filename

for key in sorted(db):                      # Iterate to display database objects
    print(key, '\t=>', db[key])             # Prints with custom format

sue = db['Sue Jones']                       # Index by key to fetch
sue.giveRaise(.10)                          # Update in memory using class method
db['Sue Jones'] = sue                       # Assign to key to update in shelve
db.close()                                  # Close after making changes
```

Because this script prints the database when it starts up, we have to run it a few times to see our objects change. Here it is in action, displaying all records and increasing `sue`'s pay each time it's run (it's a pretty good script for `sue`...):

```
c:\misc> updatedb.py
Bob Smith       => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones       => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones       => [Manager: job=mgr, name=Tom Jones, pay=50000]

c:\misc> updatedb.py
Bob Smith       => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones       => [Person: job=dev, name=Sue Jones, pay=110000]
Tom Jones       => [Manager: job=mgr, name=Tom Jones, pay=50000]

c:\misc> updatedb.py
Bob Smith       => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones       => [Person: job=dev, name=Sue Jones, pay=121000]
Tom Jones       => [Manager: job=mgr, name=Tom Jones, pay=50000]

c:\misc> updatedb.py
```

```
Bob Smith       => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones       => [Person: job=dev, name=Sue Jones, pay=133100]
Tom Jones       => [Manager: job=mgr, name=Tom Jones, pay=50000]
```

Again, what we see here is a product of the `shelve` and `pickle` tools we get from Python, and of the behavior we coded in our classes ourselves. And once again, we can verify our script's work at the interactive prompt (the shelve's equivalent of a database client):

```
c:\misc> python
>>> import shelve
>>> db = shelve.open('persondb')              # Reopen database
>>> rec = db['Sue Jones']                     # Fetch object by key
>>> print(rec)
[Person: job=dev, name=Sue Jones, pay=146410]
>>> rec.lastName()
'Jones'
>>> rec.pay
146410
```

For another example of object persistence in this book, see the sidebar in Chapter 30 titled "Why You Will Care: Classes and Persistence" on page 744. It stores a somewhat larger composite object in a flat file with `pickle` instead of `shelve`, but the effect is similar. For more details on both pickles and shelves, see other books or Python's manuals.

Future Directions

And that's a wrap for this tutorial. At this point, you've seen all the basics of Python's OOP machinery in action, and you've learned ways to avoid redundancy and its associated maintenance issues in your code. You've built full-featured classes that do real work. As an added bonus, you've made them real database records by storing them in a Python shelve, so their information lives on persistently.

There is much more we could explore here, of course. For example, we could extend our classes to make them more realistic, add new kinds of behavior to them, and so on. Giving a raise, for instance, should in practice verify that pay increase rates are between zero and one—an extension we'll add when we meet decorators later in this book. You might also mutate this example into a personal contacts database, by changing the state information stored on objects, as well as the class methods used to process it. We'll leave this a suggested exercise open to your imagination.

We could also expand our scope to use tools that either come with Python or are freely available in the open source world:

GUIs
: As is, we can only process our database with the interactive prompt's command-based interface, and scripts. We could also work on expanding our object database's usability by adding a graphical user interface for browsing and updating its records. GUIs can be built portably with either Python's `tkinter` (`Tkinter` in 2.6)

standard library support, or third-party toolkits such as WxPython and PyQt. `tkinter` ships with Python, lets you build simple GUIs quickly, and is ideal for learning GUI programming techniques; WxPython and PyQt tend to be more complex to use but often produce higher-grade GUIs in the end.

Websites
: Although GUIs are convenient and fast, the Web is hard to beat in terms of accessibility. We might also implement a website for browsing and updating records, instead of or in addition to GUIs and the interactive prompt. Websites can be constructed with either basic CGI scripting tools that come with Python, or full-featured third-party web frameworks such as Django, TurboGears, Pylons, web2Py, Zope, or Google's App Engine. On the Web, your data can still be stored in a shelve, pickle file, or other Python-based medium; the scripts that process it are simply run automatically on a server in response to requests from web browsers and other clients, and they produce HTML to interact with a user, either directly or by interfacing with Framework APIs.

Web services
: Although web clients can often parse information in the replies from websites (a technique colorfully known as "screen scraping"), we might go further and provide a more direct way to fetch records on the Web via a web services interface such as SOAP or XML-RPC calls—APIs supported by either Python itself or the third-party open source domain. Such APIs return data in a more direct form, rather than embedded in the HTML of a reply page.

Databases
: If our database becomes higher-volume or critical, we might eventually move it from shelves to a more full-featured storage mechanism such as the open source ZODB object-oriented database system (OODB), or a more traditional SQL-based relational database system such as MySQL, Oracle, PostgreSQL, or SQLite. Python itself comes with the in-process SQLite database system built-in, but other open source options are freely available on the Web. ZODB, for example, is similar to Python's `shelve` but addresses many of its limitations, supporting larger databases, concurrent updates, transaction processing, and automatic write-through on in-memory changes. SQL-based systems like MySQL offer enterprise-level tools for database storage and may be directly used from a within a Python script.

ORMs
: If we do migrate to a relational database system for storage, we don't have to sacrifice Python's OOP tools. Object-relational mappers (ORMs) like SQLObject and SQLAlchemy can automatically map relational tables and rows to and from Python classes and instances, such that we can process the stored data using normal Python class syntax. This approach provides an alternative to OODBs like `shelve` and ZODB and leverages the power of both relational databases and Python's class model.

While I hope this introduction whets your appetite for future exploration, all of these topics are of course far beyond the scope of this tutorial and this book at large. If you want to explore any of them on your own, see the Web, Python's standard library manuals, and application-focused books such as *Programming Python*. In the latter I pick up this example where we've stopped here, showing how to add both a GUI and a website on top of the database to allow for browsing and updating instance records. I hope to see you there eventually, but first, let's return to class fundamentals and finish up the rest of the core Python language story.

Chapter Summary

In this chapter, we explored all the fundamentals of Python classes and OOP in action, by building upon a simple but real example, step by step. We added constructors, methods, operator overloading, customization with subclasses, and introspection tools, and we met other concepts (such as composition, delegation, and polymorphism) along the way.

In the end, we took objects created by our classes and made them persistent by storing them on a shelve object database—an easy-to-use system for saving and retrieving native Python objects by key. While exploring class basics, we also encountered multiple ways to factor our code to reduce redundancy and minimize future maintenance costs. Finally, we briefly previewed ways to extend our code with application-programming tools such as GUIs and databases, covered in follow-up books.

In the next chapters of this part of the book we'll return to our study of the details behind Python's class model and investigate its application to some of the design concepts used to combine classes in larger programs. Before we move ahead, though, let's work through this chapter's quiz to review what we covered here. Since we've already done a lot of hands-on work in this chapter, we'll close with a set of mostly theory-oriented questions designed to make you trace through some of the code and ponder some of the bigger ideas behind it.

Test Your Knowledge: Quiz

1. When we fetch a `Manager` object from the shelve and print it, where does the display format logic come from?
2. When we fetch a `Person` object from a shelve without importing its module, how does the object know that it has a `giveRaise` method that we can call?
3. Why is it so important to move processing into methods, instead of hardcoding it outside the class?

4. Why is it better to customize by subclassing rather than copying the original and modifying?
5. Why is it better to call back to a superclass method to run default actions, instead of copying and modifying its code in a subclass?
6. Why is it better to use tools like `__dict__` that allow objects to be processed generically than to write more custom code for each type of class?
7. In general terms, when might you choose to use object embedding and composition instead of inheritance?
8. How might you modify the classes in this chapter to implement a personal contacts database in Python?

Test Your Knowledge: Answers

1. In the final version of our classes, `Manager` ultimately inherits its `__str__` printing method from `AttrDisplay` in the separate `classtools` module. `Manager` doesn't have one itself, so the inheritance search climbs to its `Person` superclass; because there is no `__str__` there either, the search climbs higher and finds it in `AttrDisplay`. The class names listed in parentheses in a `class` statement's header line provide the links to higher superclasses.
2. Shelves (really, the `pickle` module they use) automatically relink an instance to the class it was created from when that instance is later loaded back into memory. Python reimports the class from its module internally, creates an instance with its stored attributes, and sets the instance's `__class__` link to point to its original class. This way, loaded instances automatically obtain all their original methods (like `lastName`, `giveRaise`, and `__str__`), even if we have not imported the instance's class into our scope.
3. It's important to move processing into methods so that there is only one copy to change in the future, and so that the methods can be run on any instance. This is Python's notion of *encapsulation*—wrapping up logic behind interfaces, to better support future code maintenance. If you don't do so, you create code redundancy that can multiply your work effort as the code evolves in the future.
4. Customizing with subclasses reduces development effort. In OOP, we code by *customizing* what has already been done, rather than copying or changing existing code. This is the real "big idea" in OOP—because we can easily extend our prior work by coding new subclasses, we can leverage what we've already done. This is much better than either starting from scratch each time, or introducing multiple redundant copies of code that may all have to be updated in the future.

5. Copying and modifying code *doubles* your potential work effort in the future, regardless of the context. If a subclass needs to perform default actions coded in a superclass method, it's much better to call back to the original through the superclass's name than to copy its code. This also holds true for superclass constructors. Again, copying code creates redundancy, which is a major issue as code evolves.
6. Generic tools can avoid hardcoded solutions that must be kept in sync with the rest of the class as it evolves over time. A generic `__str__` print method, for example, need not be updated each time a new attribute is added to instances in an `__init__` constructor. In addition, a generic `print` method inherited by all classes only appears, and need only be modified, in one place—changes in the generic version are picked up by all classes that inherit from the generic class. Again, eliminating code *redundancy* cuts future development effort; that's one of the primary assets classes bring to the table.
7. Inheritance is best at coding extensions based on direct customization (like our `Manager` specialization of `Person`). Composition is well suited to scenarios where multiple objects are aggregated into a whole and directed by a controller layer class. Inheritance passes calls up to reuse, and composition passes down to delegate. Inheritance and composition are not mutually exclusive; often, the objects embedded in a controller are themselves customizations based upon inheritance.
8. The classes in this chapter could be used as boilerplate "template" code to implement a variety of types of databases. Essentially, you can repurpose them by modifying the constructors to record different attributes and providing whatever methods are appropriate for the target application. For instance, you might use attributes such as `name`, `address`, `birthday`, `phone`, `email`, and so on for a contacts database, and methods appropriate for this purpose. A method named `sendmail`, for example, might use Python's standard library `smptlib` module to send an email to one of the contacts automatically when called (see Python's manuals or application-level books for more details on such tools). The `AttrDisplay` tool we wrote here could be used verbatim to print your objects, because it is intentionally generic. Most of the shelve database code here can be used to store your objects, too, with minor changes.

CHAPTER 28

Class Coding Details

If you haven't quite gotten all of Python OOP yet, don't worry; now that we've had a quick tour, we're going to dig a bit deeper and study the concepts introduced earlier in further detail. In this and the following chapter, we'll take another look at class mechanics. Here, we're going to study classes, methods, and inheritance, formalizing and expanding on some of the coding ideas introduced in Chapter 26. Because the class is our last namespace tool, we'll summarize Python's namespace concepts here as well.

The next chapter continues this in-depth second pass over class mechanics by covering one specific aspect: operator overloading. Besides presenting the details, this chapter and the next also give us an opportunity to explore some larger classes than those we have studied so far.

The class Statement

Although the Python `class` statement may seem similar to tools in other OOP languages on the surface, on closer inspection, it is quite different from what some programmers are used to. For example, as in C++, the `class` statement is Python's main OOP tool, but unlike in C++, Python's `class` is not a declaration. Like a `def`, a `class` statement is an object builder, and an implicit assignment—when run, it generates a class object and stores a reference to it in the name used in the header. Also like a `def`, a `class` statement is true executable code—your class doesn't exist until Python reaches and runs the `class` statement that defines it (typically while importing the module it is coded in, but not before).

General Form

`class` is a compound statement, with a body of indented statements typically appearing under the header. In the header, superclasses are listed in parentheses after the class name, separated by commas. Listing more than one superclass leads to multiple inheritance (which we'll discuss more formally in Chapter 30). Here is the statement's general form:

```
class <name>(superclass,...):          # Assign to name
    data = value                       # Shared class data
    def method(self,...):              # Methods
        self.member = value            # Per-instance data
```

Within the `class` statement, any assignments generate class attributes, and specially named methods overload operators; for instance, a function called `__init__` is called at instance object construction time, if defined.

Example

As we've seen, classes are mostly just namespaces—that is, tools for defining names (i.e., attributes) that export data and logic to clients. So, how do you get from the `class` statement to a namespace?

Here's how. Just like in a module file, the statements nested in a `class` statement body create its attributes. When Python executes a `class` statement (not a call to a class), it runs all the statements in its body, from top to bottom. Assignments that happen during this process create names in the class's local scope, which become attributes in the associated class object. Because of this, classes resemble both modules and functions:

- Like functions, `class` statements are local scopes where names created by nested assignments live.
- Like names in a module, names assigned in a `class` statement become attributes in a class object.

The main distinction for classes is that their namespaces are also the basis of inheritance in Python; reference attributes that are not found in a class or instance object are fetched from other classes.

Because `class` is a compound statement, any sort of statement can be nested inside its body—`print`, `=`, `if`, `def`, and so on. All the statements inside the `class` statement run when the `class` statement itself runs (not when the class is later called to make an instance). Assigning names inside the `class` statement makes class attributes, and nested `def`s make class methods, but other assignments make attributes, too.

For example, assignments of simple nonfunction objects to class attributes produce *data attributes*, shared by all instances:

```
>>> class SharedData:
...     spam = 42          # Generates a class data attribute
...
>>> x = SharedData()       # Make two instances
>>> y = SharedData()
>>> x.spam, y.spam         # They inherit and share 'spam'
(42, 42)
```

Here, because the name `spam` is assigned at the top level of a `class` statement, it is attached to the class and so will be shared by all instances. We can change it by going through the class name, and we can refer to it through either instances or the class.*

```
>>> SharedData.spam = 99
>>> x.spam, y.spam, SharedData.spam
(99, 99, 99)
```

Such class attributes can be used to manage information that spans all the instances—a counter of the number of instances generated, for example (we'll expand on this idea by example in Chapter 31). Now, watch what happens if we assign the name `spam` through an instance instead of the class:

```
>>> x.spam = 88
>>> x.spam, y.spam, SharedData.spam
(88, 99, 99)
```

Assignments to instance attributes create or change the names in the instance, rather than in the shared class. More generally, inheritance searches occur only on attribute *references*, not on assignment: assigning to an object's attribute always changes that object, and no other.† For example, `y.spam` is looked up in the class by inheritance, but the assignment to `x.spam` attaches a name to `x` itself.

Here's a more comprehensive example of this behavior that stores the same name in two places. Suppose we run the following class:

```
class MixedNames:                          # Define class
    data = 'spam'                          # Assign class attr
    def __init__(self, value):             # Assign method name
        self.data = value                  # Assign instance attr
    def display(self):
        print(self.data, MixedNames.data)  # Instance attr, class attr
```

This class contains two `def`s, which bind class attributes to method functions. It also contains an `=` assignment statement; because this assignment assigns the name `data` inside the `class`, it lives in the class's local scope and becomes an attribute of the class object. Like all class attributes, this `data` is inherited and shared by all instances of the class that don't have `data` attributes of their own.

When we make instances of this class, the name `data` is attached to those instances by the assignment to `self.data` in the constructor method:

```
>>> x = MixedNames(1)          # Make two instance objects
>>> y = MixedNames(2)          # Each has its own data
```

* If you've used C++ you may recognize this as similar to the notion of C++'s "static" data members—members that are stored in the class, independent of instances. In Python, it's nothing special: all class attributes are just names assigned in the `class` statement, whether they happen to reference functions (C++'s "methods") or something else (C++'s "members"). In Chapter 31, we'll also meet Python static methods (akin to those in C++), which are just self-less functions that usually process class attributes.

† Unless the class has redefined the attribute assignment operation to do something unique with the `__setattr__` operator overloading method (discussed in Chapter 29).

```
>>> x.display(); y.display()    # self.data differs, MixedNames.data is the same
1 spam
2 spam
```

The net result is that `data` lives in two places: in the instance objects (created by the `self.data` assignment in `__init__`), and in the class from which they inherit names (created by the `data` assignment in the `class`). The class's `display` method prints both versions, by first qualifying the `self` instance, and then the class.

By using these techniques to store attributes in different objects, we determine their scope of visibility. When attached to classes, names are shared; in instances, names record per-instance data, not shared behavior or data. Although inheritance searches look up names for us, we can always get to an attribute anywhere in a tree by accessing the desired object directly.

In the preceding example, for instance, specifying `x.data` or `self.data` will return an instance name, which normally hides the same name in the class; however, `MixedNames.data` grabs the class name explicitly. We'll see various roles for such coding patterns later; the next section describes one of the most common.

Methods

Because you already know about functions, you also know about methods in classes. Methods are just function objects created by `def` statements nested in a `class` statement's body. From an abstract perspective, methods provide behavior for instance objects to inherit. From a programming perspective, methods work in exactly the same way as simple functions, with one crucial exception: a method's first argument always receives the instance object that is the implied subject of the method call.

In other words, Python automatically maps instance method calls to class method functions as follows. Method calls made through an instance, like this:

```
instance.method(args...)
```

are automatically translated to class method function calls of this form:

```
class.method(instance, args...)
```

where the class is determined by locating the method name using Python's inheritance search procedure. In fact, both call forms are valid in Python.

Besides the normal inheritance of method attribute names, the special first argument is the only real magic behind method calls. In a class method, the first argument is usually called `self` by convention (technically, only its position is significant, not its name). This argument provides methods with a hook back to the instance that is the subject of the call—because classes generate many instance objects, they need to use this argument to manage data that varies per instance.

C++ programmers may recognize Python's `self` argument as being similar to C++'s `this` pointer. In Python, though, `self` is always explicit in your code: methods must always go through `self` to fetch or change attributes of the instance being processed by the current method call. This explicit nature of `self` is by design—the presence of this name makes it obvious that you are using instance attribute names in your script, not names in the local or global scope.

Method Example

To clarify these concepts, let's turn to an example. Suppose we define the following class:

```
class NextClass:                        # Define class
    def printer(self, text):            # Define method
        self.message = text             # Change instance
        print(self.message)             # Access instance
```

The name `printer` references a function object; because it's assigned in the `class` statement's scope, it becomes a class object attribute and is inherited by every instance made from the class. Normally, because methods like `printer` are designed to process instances, we call them through instances:

```
>>> x = NextClass()                     # Make instance

>>> x.printer('instance call')          # Call its method
instance call

>>> x.message                           # Instance changed
'instance call'
```

When we call the method by qualifying an instance like this, `printer` is first located by inheritance, and then its `self` argument is automatically assigned the instance object (`x`); the `text` argument gets the string passed at the call (`'instance call'`). Notice that because Python automatically passes the first argument to `self` for us, we only actually have to pass in one argument. Inside `printer`, the name `self` is used to access or set per-instance data because it refers back to the instance currently being processed.

Methods may be called in one of two ways—through an instance, or through the class itself. For example, we can also call `printer` by going through the class name, provided we pass an instance to the `self` argument explicitly:

```
>>> NextClass.printer(x, 'class call')  # Direct class call
class call

>>> x.message                           # Instance changed again
'class call'
```

Calls routed through the instance and the class have the exact same effect, as long as we pass the same instance object ourselves in the class form. By default, in fact, you get an error message if you try to call a method without any instance:

```
>>> NextClass.printer('bad call')
TypeError: unbound method printer() must be called with NextClass instance...
```

Calling Superclass Constructors

Methods are normally called through instances. Calls to methods through a class, though, do show up in a variety of special roles. One common scenario involves the constructor method. The `__init__` method, like all attributes, is looked up by inheritance. This means that at construction time, Python locates and calls just one `__init__`. If subclass constructors need to guarantee that superclass construction-time logic runs, too, they generally must call the superclass's `__init__` method explicitly through the class:

```
class Super:
    def __init__(self, x):
        ...default code...

class Sub(Super):
    def __init__(self, x, y):
        Super.__init__(self, x)             # Run superclass __init__
        ...custom code...                   # Do my init actions

I = Sub(1, 2)
```

This is one of the few contexts in which your code is likely to call an operator overloading method directly. Naturally, you should only call the superclass constructor this way if you really want it to run—without the call, the subclass replaces it completely. For a more realistic illustration of this technique in action, see the `Manager` class example in the prior chapter's tutorial.‡

Other Method Call Possibilities

This pattern of calling methods through a class is the general basis of extending (instead of completely replacing) inherited method behavior. In Chapter 31, we'll also meet a new option added in Python 2.2, *static methods*, that allow you to code methods that do not expect instance objects in their first arguments. Such methods can act like simple instanceless functions, with names that are local to the classes in which they are coded, and may be used to manage class data. A related concept, the *class method*, receives a class when called instead of an instance and can be used to manage per-class data. These are advanced and optional extensions, though; normally, you must always pass an instance to a method, whether it is called through an instance or a class.

‡ On a somewhat related note, you can also code multiple `__init__` methods within the same class, but only the last definition will be used; see Chapter 30 for more details on multiple method definitions.

Inheritance

The whole point of a namespace tool like the `class` statement is to support name inheritance. This section expands on some of the mechanisms and roles of attribute inheritance in Python.

In Python, inheritance happens when an object is qualified, and it involves searching an attribute definition tree (one or more namespaces). Every time you use an expression of the form *`object.attr`* (where *`object`* is an instance or class object), Python searches the namespace tree from bottom to top, beginning with *`object`*, looking for the first *`attr`* it can find. This includes references to `self` attributes in your methods. Because lower definitions in the tree override higher ones, inheritance forms the basis of specialization.

Attribute Tree Construction

Figure 28-1 summarizes the way namespace trees are constructed and populated with names. Generally:

- Instance attributes are generated by assignments to `self` attributes in methods.
- Class attributes are created by statements (assignments) in `class` statements.
- Superclass links are made by listing classes in parentheses in a `class` statement header.

The net result is a tree of attribute namespaces that leads from an instance, to the class it was generated from, to all the superclasses listed in the `class` header. Python searches upward in this tree, from instances to superclasses, each time you use qualification to fetch an attribute name from an instance object.§

Specializing Inherited Methods

The tree-searching model of inheritance just described turns out to be a great way to specialize systems. Because inheritance finds names in subclasses before it checks superclasses, subclasses can replace default behavior by redefining their superclasses' attributes. In fact, you can build entire systems as hierarchies of classes, which are extended by adding new external subclasses rather than changing existing logic in-place.

§ This description isn't 100% complete, because we can also create instance and class attributes by assigning to objects outside `class` statements—but that's a much less common and sometimes more error-prone approach (changes aren't isolated to `class` statements). In Python, all attributes are always accessible by default. We'll talk more about attribute name privacy in Chapter 29 when we study `__setattr__`, in Chapter 30 when we meet `__X` names, and again in Chapter 38, where we'll implement it with a class decorator.

The idea of redefining inherited names leads to a variety of specialization techniques. For instance, subclasses may *replace* inherited attributes completely, *provide* attributes that a superclass expects to find, and *extend* superclass methods by calling back to the superclass from an overridden method. We've already seen replacement in action. Here's an example that shows how extension works:

```
>>> class Super:
...     def method(self):
...         print('in Super.method')
...
>>> class Sub(Super):
...     def method(self):                    # Override method
...         print('starting Sub.method')     # Add actions here
...         Super.method(self)               # Run default action
...         print('ending Sub.method')
...
```

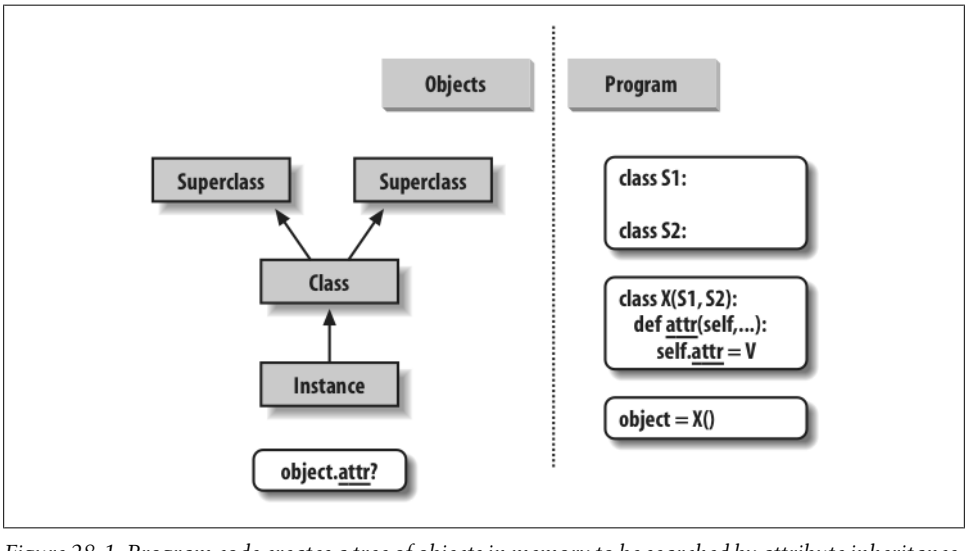

Figure 28-1. Program code creates a tree of objects in memory to be searched by attribute inheritance. Calling a class creates a new instance that remembers its class, running a class statement creates a new class, and superclasses are listed in parentheses in the class statement header. Each attribute reference triggers a new bottom-up tree search—even references to self attributes within a class's methods.

Direct superclass method calls are the crux of the matter here. The `Sub` class replaces `Super`'s `method` function with its own specialized version, but within the replacement, `Sub` calls back to the version exported by `Super` to carry out the default behavior. In other words, `Sub.method` just extends `Super.method`'s behavior, rather than replacing it completely:

```
>>> x = Super()                    # Make a Super instance
>>> x.method()                     # Runs Super.method
in Super.method

>>> x = Sub()                      # Make a Sub instance
>>> x.method()                     # Runs Sub.method, calls Super.method
starting Sub.method
in Super.method
ending Sub.method
```

This extension coding pattern is also commonly used with constructors; see the section "Methods" on page 684 for an example.

Class Interface Techniques

Extension is only one way to interface with a superclass. The file shown in this section, *specialize.py*, defines multiple classes that illustrate a variety of common techniques:

`Super`
: Defines a `method` function and a `delegate` that expects an `action` in a subclass.

`Inheritor`
: Doesn't provide any new names, so it gets everything defined in `Super`.

`Replacer`
: Overrides `Super`'s `method` with a version of its own.

`Extender`
: Customizes `Super`'s `method` by overriding and calling back to run the default.

`Provider`
: Implements the `action` method expected by `Super`'s `delegate` method.

Study each of these subclasses to get a feel for the various ways they customize their common superclass. Here's the file:

```
class Super:
    def method(self):
        print('in Super.method')            # Default behavior
    def delegate(self):
        self.action()                       # Expected to be defined

class Inheritor(Super):                     # Inherit method verbatim
    pass

class Replacer(Super):                      # Replace method completely
    def method(self):
        print('in Replacer.method')

class Extender(Super):                      # Extend method behavior
    def method(self):
        print('starting Extender.method')
        Super.method(self)
        print('ending Extender.method')
```

```
class Provider(Super):                          # Fill in a required method
    def action(self):
        print('in Provider.action')

if __name__ == '__main__':
    for klass in (Inheritor, Replacer, Extender):
        print('\n' + klass.__name__ + '...')
        klass().method()
    print('\nProvider...')
    x = Provider()
    x.delegate()
```

A few things are worth pointing out here. First, the self-test code at the end of this example creates instances of three different classes in a `for` loop. Because classes are objects, you can put them in a tuple and create instances generically (more on this idea later). Classes also have the special `__name__` attribute, like modules; it's preset to a string containing the name in the class header. Here's what happens when we run the file:

```
% python specialize.py

Inheritor...
in Super.method

Replacer...
in Replacer.method

Extender...
starting Extender.method
in Super.method
ending Extender.method

Provider...
in Provider.action
```

Abstract Superclasses

Notice how the `Provider` class in the prior example works. When we call the `delegate` method through a `Provider` instance, *two* independent inheritance searches occur:

1. On the initial `x.delegate` call, Python finds the `delegate` method in `Super` by searching the `Provider` instance and above. The instance `x` is passed into the method's `self` argument as usual.
2. Inside the `Super.delegate` method, `self.action` invokes a new, independent inheritance search of `self` and above. Because `self` references a `Provider` instance, the `action` method is located in the `Provider` subclass.

This "filling in the blanks" sort of coding structure is typical of OOP frameworks. At least in terms of the `delegate` method, the superclass in this example is what is sometimes called an *abstract superclass*—a class that expects parts of its behavior to be

provided by its subclasses. If an expected method is not defined in a subclass, Python raises an undefined name exception when the inheritance search fails.

Class coders sometimes make such subclass requirements more obvious with `assert` statements, or by raising the built-in `NotImplementedError` exception with `raise` statements (we'll study statements that may trigger exceptions in depth in the next part of this book). As a quick preview, here's the `assert` scheme in action:

```
class Super:
    def delegate(self):
        self.action()
    def action(self):
        assert False, 'action must be defined!'       # If this version is called

>>> X = Super()
>>> X.delegate()
AssertionError: action must be defined!
```

We'll meet `assert` in Chapters 32 and 33; in short, if its first expression evaluates to false, it raises an exception with the provided error message. Here, the expression is always false so as to trigger an error message if a method is not redefined, and inheritance locates the version here. Alternatively, some classes simply raise a `NotImplementedError` exception directly in such method stubs to signal the mistake:

```
class Super:
    def delegate(self):
        self.action()
    def action(self):
        raise NotImplementedError('action must be defined!')

>>> X = Super()
>>> X.delegate()
NotImplementedError: action must be defined!
```

For instances of subclasses, we still get the exception unless the subclass provides the expected method to replace the default in the superclass:

```
>>> class Sub(Super): pass
...
>>> X = Sub()
>>> X.delegate()
NotImplementedError: action must be defined!

>>> class Sub(Super):
...     def action(self): print('spam')
...
>>> X = Sub()
>>> X.delegate()
spam
```

For a somewhat more realistic example of this section's concepts in action, see the "Zoo animal hierarchy" exercise (exercise 8) at the end of Chapter 31, and its solution in "Part VI, Classes and OOP" on page 1122 in Appendix B. Such taxonomies are a

traditional way to introduce OOP, but they're a bit removed from most developers' job descriptions.

Python 2.6 and 3.0 Abstract Superclasses

As of Python 2.6 and 3.0, the prior section's abstract superclasses (a.k.a. "abstract base classes"), which require methods to be filled in by subclasses, may also be implemented with special class syntax. The way we code this varies slightly depending on the version. In Python 3.0, we use a keyword argument in a `class` header, along with special `@` decorator syntax, both of which we'll study in detail later in this book:

```
from abc import ABCMeta, abstractmethod

class Super(metaclass=ABCMeta):
    @abstractmethod
    def method(self, ...):
        pass
```

But in Python 2.6, we use a class attribute instead:

```
class Super:
    __metaclass__ = ABCMeta
    @abstractmethod
    def method(self, ...):
        pass
```

Either way, the effect is the same—we can't make an instance unless the method is defined lower in the class tree. In 3.0, for example, here is the special syntax equivalent of the prior section's example:

```
>>> from abc import ABCMeta, abstractmethod
>>>
>>> class Super(metaclass=ABCMeta):
...     def delegate(self):
...         self.action()
...     @abstractmethod
...     def action(self):
...         pass
...
>>> X = Super()
TypeError: Can't instantiate abstract class Super with abstract methods action

>>> class Sub(Super): pass
...
>>> X = Sub()
TypeError: Can't instantiate abstract class Sub with abstract methods action

>>> class Sub(Super):
...     def action(self): print('spam')
...
>>> X = Sub()
>>> X.delegate()
spam
```

Coded this way, a class with an abstract method cannot be instantiated (that is, we cannot create an instance by calling it) unless all of its abstract methods have been defined in subclasses. Although this requires more code, the advantage of this approach is that errors for missing methods are issued when we attempt to make an instance of the class, not later when we try to call a missing method. This feature may also be used to define an expected interface, automatically verified in client classes.

Unfortunately, this scheme also relies on two advanced language tools we have not met yet—*function decorators*, introduced in Chapter 31 and covered in depth in Chapter 38, as well as *metaclass declarations*, mentioned in Chapter 31 and covered in Chapter 39—so we will finesse other facets of this option here. See Python's standard manuals for more on this, as well as precoded abstract superclasses Python provides.

Namespaces: The Whole Story

Now that we've examined class and instance objects, the Python namespace story is complete. For reference, I'll quickly summarize all the rules used to resolve names here. The first things you need to remember are that qualified and unqualified names are treated differently, and that some scopes serve to initialize object namespaces:

- Unqualified names (e.g., `X`) deal with scopes.
- Qualified attribute names (e.g., `object.X`) use object namespaces.
- Some scopes initialize object namespaces (for modules and classes).

Simple Names: Global Unless Assigned

Unqualified simple names follow the LEGB lexical scoping rule outlined for functions in Chapter 17:

Assignment (`X = value`)
: Makes names local: creates or changes the name `X` in the current local scope, unless declared global.

Reference (`X`)
: Looks for the name `X` in the current local scope, then any and all enclosing functions, then the current global scope, then the built-in scope.

Attribute Names: Object Namespaces

Qualified attribute names refer to attributes of specific objects and obey the rules for modules and classes. For class and instance objects, the reference rules are augmented to include the inheritance search procedure:

Assignment (`object.X = value`)
: Creates or alters the attribute name `X` in the namespace of the *`object`* being qualified, and none other. Inheritance-tree climbing happens only on attribute reference, not on attribute assignment.

Reference (`object.X`)
: For class-based objects, searches for the attribute name `X` in *`object`*, then in all accessible classes above it, using the inheritance search procedure. For nonclass objects such as modules, fetches `X` from *`object`* directly.

The "Zen" of Python Namespaces: Assignments Classify Names

With distinct search procedures for qualified and unqualified names, and multiple lookup layers for both, it can sometimes be difficult to tell where a name will wind up going. In Python, the place where you *assign* a name is crucial—it fully determines the scope or object in which a name will reside. The file *manynames.py* illustrates how this principle translates to code and summarizes the namespace ideas we have seen throughout this book:

```
# manynames.py

X = 11                          # Global (module) name/attribute (X, or manynames.X)

def f():
    print(X)                    # Access global X (11)

def g():
    X = 22                      # Local (function) variable (X, hides module X)
    print(X)

class C:
    X = 33                      # Class attribute (C.X)
    def m(self):
        X = 44                  # Local variable in method (X)
        self.X = 55             # Instance attribute (instance.X)
```

This file assigns the same name, `X`, five times. Because this name is assigned in five different locations, though, all five `X`s in this program are completely different variables. From top to bottom, the assignments to `X` here generate: a module attribute (`11`), a local variable in a function (`22`), a class attribute (`33`), a local variable in a method (`44`), and an instance attribute (`55`). Although all five are named `X`, the fact that they are all assigned at different places in the source code or to different objects makes all of these unique variables.

You should take the time to study this example carefully because it collects ideas we've been exploring throughout the last few parts of this book. When it makes sense to you, you will have achieved a sort of Python namespace nirvana. Of course, an alternative route to nirvana is to simply run the program and see what happens. Here's the remainder of this source file, which makes an instance and prints all the `X`s that it can fetch:

```
# manynames.py, continued

if __name__ == '__main__':
    print(X)                    # 11: module (a.k.a. manynames.X outside file)
    f()                         # 11: global
    g()                         # 22: local
    print(X)                    # 11: module name unchanged

    obj = C()                   # Make instance
    print(obj.X)                # 33: class name inherited by instance

    obj.m()                     # Attach attribute name X to instance now
    print(obj.X)                # 55: instance
    print(C.X)                  # 33: class (a.k.a. obj.X if no X in instance)

    #print(C.m.X)               # FAILS: only visible in method
    #print(g.X)                 # FAILS: only visible in function
```

The outputs that are printed when the file is run are noted in the comments in the code; trace through them to see which variable named `X` is being accessed each time. Notice in particular that we can go through the class to fetch its attribute (`C.X`), but we can never fetch local variables in functions or methods from outside their `def` statements. Locals are visible only to other code within the `def`, and in fact only live in memory while a call to the function or method is executing.

Some of the names defined by this file are visible *outside the file* to other modules, but recall that we must always import before we can access names in another file—that is the main point of modules, after all:

```
# otherfile.py

import manynames

X = 66
print(X)                        # 66: the global here
print(manynames.X)              # 11: globals become attributes after imports

manynames.f()                   # 11: manynames's X, not the one here!
manynames.g()                   # 22: local in other file's function

print(manynames.C.X)            # 33: attribute of class in other module
I = manynames.C()
print(I.X)                      # 33: still from class here
I.m()
print(I.X)                      # 55: now from instance!
```

Notice here how `manynames.f()` prints the `X` in `manynames`, not the `X` assigned in this file—scopes are always determined by the position of assignments in your source code (i.e., lexically) and are never influenced by what imports what or who imports whom. Also, notice that the instance's own `X` is not created until we call `I.m()`—attributes, like all variables, spring into existence when assigned, and not before. Normally we create instance attributes by assigning them in class `__init__` constructor methods, but this isn't the only option.

Finally, as we learned in Chapter 17, it's also possible for a function to *change* names outside itself, with `global` and (in Python 3.0) `nonlocal` statements—these statements provide write access, but also modify assignment's namespace binding rules:

```
X = 11                          # Global in module

def g1():
    print(X)                    # Reference global in module

def g2():
    global X
    X = 22                      # Change global in module

def h1():
    X = 33                      # Local in function
    def nested():
        print(X)                # Reference local in enclosing scope

def h2():
    X = 33                      # Local in function
    def nested():
        nonlocal X              # Python 3.0 statement
        X = 44                  # Change local in enclosing scope
```

Of course, you generally shouldn't use the same name for every variable in your script—but as this example demonstrates, even if you do, Python's namespaces will work to keep names used in one context from accidentally clashing with those used in another.

Namespace Dictionaries

In Chapter 22, we learned that module namespaces are actually implemented as dictionaries and exposed with the built-in `__dict__` attribute. The same holds for class and instance objects: attribute qualification is really a dictionary indexing operation internally, and attribute inheritance is just a matter of searching linked dictionaries. In fact, instance and class objects are mostly just dictionaries with links inside Python. Python exposes these dictionaries, as well as the links between them, for use in advanced roles (e.g., for coding tools).

To help you understand how attributes work internally, let's work through an interactive session that traces the way namespace dictionaries grow when classes are involved. We saw a simpler version of this type of code in Chapter 26, but now that we know more about methods and superclasses, let's embellish it here. First, let's define a superclass and a subclass with methods that will store data in their instances:

```
>>> class super:
...     def hello(self):
...         self.data1 = 'spam'
...
>>> class sub(super):
...     def hola(self):
```

```
...         self.data2 = 'eggs'
...
```

When we make an instance of the subclass, the instance starts out with an empty namespace dictionary, but it has links back to the class for the inheritance search to follow. In fact, the inheritance tree is explicitly available in special attributes, which you can inspect. Instances have a `__class__` attribute that links to their class, and classes have a `__bases__` attribute that is a tuple containing links to higher superclasses (I'm running this on Python 3.0; name formats and some internal attributes vary slightly in 2.6):

```
>>> X = sub()
>>> X.__dict__                          # Instance namespace dict
{}

>>> X.__class__                         # Class of instance
<class '__main__.sub'>

>>> sub.__bases__                       # Superclasses of class
(<class '__main__.super'>,)

>>> super.__bases__                     # () empty tuple in Python 2.6
(<class 'object'>,)
```

As classes assign to `self` attributes, they populate the instance objects—that is, attributes wind up in the instances' attribute namespace dictionaries, not in the classes'. An instance object's namespace records data that can vary from instance to instance, and `self` is a hook into that namespace:

```
>>> Y = sub()

>>> X.hello()
>>> X.__dict__
{'data1': 'spam'}

>>> X.hola()
>>> X.__dict__
{'data1': 'spam', 'data2': 'eggs'}

>>> sub.__dict__.keys()
['__module__', '__doc__', 'hola']

>>> super.__dict__.keys()
['__dict__', '__module__', '__weakref__', 'hello', '__doc__']

>>> Y.__dict__
{}
```

Notice the extra underscore names in the class dictionaries; Python sets these automatically. Most are not used in typical programs, but there are tools that use some of them (e.g., `__doc__` holds the docstrings discussed in Chapter 15).

Also, observe that `Y`, a second instance made at the start of this series, still has an empty namespace dictionary at the end, even though `X`'s dictionary has been populated by assignments in methods. Again, each instance has an independent namespace dictionary, which starts out empty and can record completely different attributes than those recorded by the namespace dictionaries of other instances of the same class.

Because attributes are actually dictionary keys inside Python, there are really two ways to fetch and assign their values—by qualification, or by key indexing:

```
>>> X.data1, X.__dict__['data1']
('spam', 'spam')

>>> X.data3 = 'toast'
>>> X.__dict__
{'data1': 'spam', 'data3': 'toast', 'data2': 'eggs'}

>>> X.__dict__['data3'] = 'ham'
>>> X.data3
'ham'
```

This equivalence applies only to attributes actually attached to the instance, though. Because attribute fetch qualification also performs an inheritance search, it can access attributes that namespace dictionary indexing cannot. The inherited attribute `X.hello`, for instance, cannot be accessed by `X.__dict__['hello']`.

Finally, here is the built-in `dir` function we met in Chapters 4 and 15 at work on class and instance objects. This function works on anything with attributes: `dir(`*`object`*`)` is similar to an *`object`*`.__dict__.keys()` call. Notice, though, that `dir` sorts its list and includes some system attributes. As of Python 2.2, `dir` also collects inherited attributes automatically, and in 3.0 it includes names inherited from the `object` class that is an implied superclass of all classes:‖

```
>>> X.__dict__, Y.__dict__
({'data1': 'spam', 'data3': 'ham', 'data2': 'eggs'}, {})
>>> list(X.__dict__.keys())                              # Need list in 3.0
['data1', 'data3', 'data2']

# In Python 2.6:

>>>> dir(X)
['__doc__', '__module__', 'data1', 'data2', 'data3', 'hello', 'hola']
>>> dir(sub)
['__doc__', '__module__', 'hello', 'hola']
>>> dir(super)
['__doc__', '__module__', 'hello']
```

‖ As you can see, the contents of attribute dictionaries and `dir` call results may change over time. For example, because Python now allows built-in types to be subclassed like classes, the contents of `dir` results for built-in types have expanded to include operator overloading methods, just like our `dir` results here for user-defined classes under Python 3.0. In general, attribute names with leading and trailing double underscores are interpreter-specific. Type subclasses will be discussed further in Chapter 31.

```
# In Python 3.0:

>>> dir(X)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
...more omitted...
'data1', 'data2', 'data3', 'hello', 'hola']

>>> dir(sub)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
...more omitted...
'hello', 'hola']

>>> dir(super)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
...more omitted...
'hello'
]
```

Experiment with these special attributes on your own to get a better feel for how namespaces actually do their attribute business. Even if you will never use these in the kinds of programs you write, seeing that they are just normal dictionaries will help demystify the notion of namespaces in general.

Namespace Links

The prior section introduced the special `__class__` and `__bases__` instance and class attributes, without really explaining why you might care about them. In short, these attributes allow you to inspect inheritance hierarchies within your own code. For example, they can be used to display a class tree, as in the following example:

```
# classtree.py

"""
Climb inheritance trees using namespace links,
displaying higher superclasses with indentation
"""

def classtree(cls, indent):
    print('.' * indent + cls.__name__)     # Print class name here
    for supercls in cls.__bases__:         # Recur to all superclasses
        classtree(supercls, indent+3)      # May visit super > once

def instancetree(inst):
    print('Tree of %s' % inst)             # Show instance
    classtree(inst.__class__, 3)           # Climb to its class

def selftest():
    class A:      pass
    class B(A):   pass
    class C(A):   pass
    class D(B,C): pass
    class E:      pass
    class F(D,E): pass
```

```
        instancetree(B())
        instancetree(F())

if __name__ == '__main__': selftest()
```

The `classtree` function in this script is *recursive*—it prints a class's name using `__name__`, then climbs up to the superclasses by calling itself. This allows the function to traverse arbitrarily shaped class trees; the recursion climbs to the top, and stops at root superclasses that have empty `__bases__` attributes. When using recursion, each active level of a function gets its own copy of the local scope; here, this means that `cls` and `indent` are different at each `classtree` level.

Most of this file is self-test code. When run standalone in Python 3.0, it builds an empty class tree, makes two instances from it, and prints their class tree structures:

```
C:\misc> c:\python26\python classtree.py
Tree of <__main__.B instance at 0x02557328>
...B
......A
Tree of <__main__.F instance at 0x02557328>
...F
......D
.........B
............A
.........C
............A
......E
```

When run under Python 3.0, the tree includes the implied `object` superclasses that are automatically added above standalone classes, because all classes are "new style" in 3.0 (more on this change in Chapter 31):

```
C:\misc> c:\python30\python classtree.py
Tree of <__main__.B object at 0x02810650>
...B
......A
.........object
Tree of <__main__.F object at 0x02810650>
...F
......D
.........B
............A
..............object
.........C
............A
..............object
......E
.........object
```

Here, indentation marked by periods is used to denote class tree height. Of course, we could improve on this output format, and perhaps even sketch it in a GUI display. Even as is, though, we can import these functions anywhere we want a quick class tree display:

```
C:\misc> c:\python30\python
>>> class Emp: pass
...
>>> class Person(Emp): pass
>>> bob = Person()

>>> import classtree
>>> classtree.instancetree(bob)
Tree of <__main__.Person object at 0x028203B0>
...Person
......Emp
.........object
```

Regardless of whether you will ever code or use such tools, this example demonstrates one of the many ways that you can make use of special attributes that expose interpreter internals. You'll see another when we code the *lister.py* general-purpose class display tools in the section "Multiple Inheritance: "Mix-in" Classes" on page 756—there, we will extend this technique to also display attributes in each object in a class tree. And in the last part of this book, we'll revisit such tools in the context of Python tool building at large, to code tools that implement attribute privacy, argument validation, and more. While not for every Python programmer, access to internals enables powerful development tools.

Documentation Strings Revisited

The last section's example includes a docstring for its module, but remember that docstrings can be used for class components as well. Docstrings, which we covered in detail in Chapter 15, are string literals that show up at the top of various structures and are automatically saved by Python in the corresponding objects' `__doc__` attributes. This works for module files, function `def`s, and classes and methods.

Now that we know more about classes and methods, the following file, *docstr.py*, provides a quick but comprehensive example that summarizes the places where docstrings can show up in your code. All of these can be triple-quoted blocks:

```
"I am: docstr.__doc__"

def func(args):
    "I am: docstr.func.__doc__"
    pass

class spam:
    "I am: spam.__doc__ or docstr.spam.__doc__"
    def method(self, arg):
        "I am: spam.method.__doc__ or self.method.__doc__"
        pass
```

The main advantage of documentation strings is that they stick around at runtime. Thus, if it's been coded as a docstring, you can qualify an object with its `__doc__` attribute to fetch its documentation:

```
>>> import docstr
>>> docstr.__doc__
'I am: docstr.__doc__'

>>> docstr.func.__doc__
'I am: docstr.func.__doc__'

>>> docstr.spam.__doc__
'I am: spam.__doc__ or docstr.spam.__doc__'

>>> docstr.spam.method.__doc__
'I am: spam.method.__doc__ or self.method.__doc__'
```

A discussion of the *PyDoc* tool, which knows how to format all these strings in reports, appears in Chapter 15. Here it is running on our code under Python 2.6 (Python 3.0 shows additional attributes inherited from the implied `object` superclass in the new-style class model—run this on your own to see the 3.0 extras, and watch for more about this difference in Chapter 31):

```
>>> help(docstr)
Help on module docstr:

NAME
    docstr - I am: docstr.__doc__

FILE
    c:\misc\docstr.py

CLASSES
    spam

    class spam
     |  I am: spam.__doc__ or docstr.spam.__doc__
     |
     |  Methods defined here:
     |
     |  method(self, arg)
     |      I am: spam.method.__doc__ or self.method.__doc__

FUNCTIONS
    func(args)
        I am: docstr.func.__doc__
```

Documentation strings are available at runtime, but they are less flexible syntactically than # comments (which can appear anywhere in a program). Both forms are useful tools, and any program documentation is good (as long as it's accurate, of course!). As a best-practice rule of thumb, use docstrings for functional documentation (what your objects do) and hash-mark comments for more micro-level documentation (how arcane expressions work).

Classes Versus Modules

Let's wrap up this chapter by briefly comparing the topics of this book's last two parts: modules and classes. Because they're both about namespaces, the distinction can be confusing. In short:

- Modules
  - —Are data/logic packages
  - —Are created by writing Python files or C extensions
  - —Are used by being imported
- Classes
  - —Implement new objects
  - —Are created by `class` statements
  - —Are used by being called
  - —Always live within a module

Classes also support extra features that modules don't, such as operator overloading, multiple instance generation, and inheritance. Although both classes and modules are namespaces, you should be able to tell by now that they are very different things.

Chapter Summary

This chapter took us on a second, more in-depth tour of the OOP mechanisms of the Python language. We learned more about classes, methods, and inheritance, and we wrapped up the namespace story in Python by extending it to cover its application to classes. Along the way, we looked at some more advanced concepts, such as abstract superclasses, class data attributes, namespace dictionaries and links, and manual calls to superclass methods and constructors.

Now that we've learned all about the mechanics of coding classes in Python, Chapter 29 turns to a specific facet of those mechanics: operator overloading. After that we'll explore common design patterns, looking at some of the ways that classes are commonly used and combined to optimize code reuse. Before you read ahead, though, be sure to work though the usual chapter quiz to review what we've covered here.

Test Your Knowledge: Quiz

1. What is an abstract superclass?
2. What happens when a simple assignment statement appears at the top level of a `class` statement?

3. Why might a class need to manually call the `__init__` method in a superclass?
4. How can you augment, instead of completely replacing, an inherited method?
5. What...was the capital of Assyria?

Test Your Knowledge: Answers

1. An abstract superclass is a class that calls a method, but does not inherit or define it—it expects the method to be filled in by a subclass. This is often used as a way to generalize classes when behavior cannot be predicted until a more specific subclass is coded. OOP frameworks also use this as a way to dispatch to client-defined, customizable operations.
2. When a simple assignment statement (`X = Y`) appears at the top level of a `class` statement, it attaches a data attribute to the class (*`Class`*`.X`). Like all class attributes, this will be shared by all instances; data attributes are not callable method functions, though.
3. A class must manually call the `__init__` method in a superclass if it defines an `__init__` constructor of its own, but it also must still kick off the superclass's construction code. Python itself automatically runs just one constructor—the lowest one in the tree. Superclass constructors are called through the class name, passing in the `self` instance manually: *`Superclass`*`.__init__(self, ...)`.
4. To augment instead of completely replacing an inherited method, redefine it in a subclass, but call back to the superclass's version of the method manually from the new version of the method in the subclass. That is, pass the `self` instance to the superclass's version of the method manually: *`Superclass.method`*`(self, ...)`.
5. Ashur (or Qalat Sherqat), Calah (or Nimrud), the short-lived Dur Sharrukin (or Khorsabad), and finally Nineveh.

CHAPTER 29

Operator Overloading

This chapter continues our in-depth survey of class mechanics by focusing on operator overloading. We looked briefly at operator overloading in prior chapters; here, we'll fill in more details and look at a handful of commonly used overloading methods. Although we won't demonstrate each of the many operator overloading methods available, those we will code here are a representative sample large enough to uncover the possibilities of this Python class feature.

The Basics

Really "operator overloading" simply means *intercepting* built-in operations in class methods—Python automatically invokes your methods when instances of the class appear in built-in operations, and your method's return value becomes the result of the corresponding operation. Here's a review of the key ideas behind overloading:

- Operator overloading lets classes intercept normal Python operations.
- Classes can overload all Python expression operators.
- Classes can also overload built-in operations such as printing, function calls, attribute access, etc.
- Overloading makes class instances act more like built-in types.
- Overloading is implemented by providing specially named class methods.

In other words, when certain specially named methods are provided in a class, Python automatically calls them when instances of the class appear in their associated expressions. As we've learned, operator overloading methods are never required and generally don't have defaults; if you don't code or inherit one, it just means that your class does not support the corresponding operation. When used, though, these methods allow classes to emulate the interfaces of built-in objects, and so appear more consistent.

Constructors and Expressions: __init__ and __sub__

Consider the following simple example: its `Number` class, coded in the file *number.py*, provides a method to intercept instance construction (`__init__`), as well as one for catching subtraction expressions (`__sub__`). Special methods such as these are the hooks that let you tie into built-in operations:

```
class Number:
    def __init__(self, start):                  # On Number(start)
        self.data = start
    def __sub__(self, other):                   # On instance - other
        return Number(self.data - other)        # Result is a new instance

>>> from number import Number                   # Fetch class from module
>>> X = Number(5)                               # Number.__init__(X, 5)
>>> Y = X - 2                                   # Number.__sub__(X, 2)
>>> Y.data                                      # Y is new Number instance
3
```

As discussed previously, the `__init__` constructor method seen in this code is the most commonly used operator overloading method in Python; it's present in most classes. In this chapter, we will tour some of the other tools available in this domain and look at example code that applies them in common use cases.

Common Operator Overloading Methods

Just about everything you can do to built-in objects such as integers and lists has a corresponding specially named method for overloading in classes. Table 29-1 lists a few of the most common; there are many more. In fact, many overloading methods come in multiple versions (e.g., `__add__`, `__radd__`, and `__iadd__` for addition), which is one reason there are so many. See other Python books, or the Python language reference manual, for an exhaustive list of the special method names available.

Table 29-1. Common operator overloading methods

Method	Implements	Called for
`__init__`	Constructor	Object creation: `X = Class(args)`
`__del__`	Destructor	Object reclamation of `X`
`__add__`	Operator +	`X + Y`, `X += Y` if no `__iadd__`
`__or__`	Operator | (bitwise OR)	`X | Y`, `X |= Y` if no `__ior__`
`__repr__`, `__str__`	Printing, conversions	`print(X)`, `repr(X)`, `str(X)`
`__call__`	Function calls	`X(*args, **kargs)`
`__getattr__`	Attribute fetch	`X.undefined`
`__setattr__`	Attribute assignment	`X.any = value`
`__delattr__`	Attribute deletion	`del X.any`
`__getattribute__`	Attribute fetch	`X.any`

Method	Implements	Called for
`__getitem__`	Indexing, slicing, iteration	`X[key]`, `X[i:j]`, `for` loops and other iterations if no `__iter__`
`__setitem__`	Index and slice assignment	`X[key] = value`, `X[i:j] = sequence`
`__delitem__`	Index and slice deletion	`del X[key]`, `del X[i:j]`
`__len__`	Length	`len(X)`, truth tests if no `__bool__`
`__bool__`	Boolean tests	`bool(X)`, truth tests (named `__nonzero__` in 2.6)
`__lt__, __gt__, __le__, __ge__, __eq__, __ne__`	Comparisons	`X < Y, X > Y, X <= Y, X >= Y, X == Y, X != Y` (or else `__cmp__` in 2.6 only)
`__radd__`	Right-side operators	`Other + X`
`__iadd__`	In-place augmented operators	`X += Y` (or else `__add__`)
`__iter__, __next__`	Iteration contexts	`I=iter(X)`; `next(I)`; `for` loops, `in` if no `__contains__`, all comprehensions, `map(F,X)`, others (`__next__` is named `next` in 2.6)
`__contains__`	Membership test	`item in X` (any iterable)
`__index__`	Integer value	`hex(X)`, `bin(X)`, `oct(X)`, `O[X]`, `O[X:]` (replaces Python 2 `__oct__`, `__hex__`)
`__enter__, __exit__`	Context manager (Chapter 33)	`with obj as var:`
`__get__, __set__, __delete__`	Descriptor attributes (Chapter 37)	`X.attr, X.attr = value, del X.attr`
`__new__`	Creation (Chapter 39)	Object creation, before `__init__`

All overloading methods have names that start and end with two underscores to keep them distinct from other names you define in your classes. The mappings from special method names to expressions or operations are predefined by the Python language (and documented in the standard language manual). For example, the name `__add__` always maps to `+` expressions by Python language definition, regardless of what an `__add__` method's code actually does.

Operator overloading methods may be inherited from superclasses if not defined, just like any other methods. Operator overloading methods are also all optional—if you don't code or inherit one, that operation is simply unsupported by your class, and attempting it will raise an exception. Some built-in operations, like printing, have defaults (inherited for the implied `object` class in Python 3.0), but most built-ins fail for class instances if no corresponding operator overloading method is present.

Most overloading methods are used only in advanced programs that require objects to behave like built-ins; the `__init__` constructor tends to appear in most classes, however, so pay special attention to it. We've already met the `__init__` initialization-time constructor method, and a few of the others in Table 29-1. Let's explore some of the additional methods in the table by example.

Indexing and Slicing: __getitem__ and __setitem__

If defined in a class (or inherited by it), the `__getitem__` method is called automatically for instance-indexing operations. When an instance `X` appears in an indexing expression like `X[i]`, Python calls the `__getitem__` method inherited by the instance, passing `X` to the first argument and the index in brackets to the second argument. For example, the following class returns the square of an index value:

```
>>> class Indexer:
...     def __getitem__(self, index):
...         return index ** 2
...
>>> X = Indexer()
>>> X[2]                                # X[i] calls X.__getitem__(i)
4

>>> for i in range(5):
...     print(X[i], end=' ')            # Runs __getitem__(X, i) each time
...
0 1 4 9 16
```

Intercepting Slices

Interestingly, in addition to indexing, `__getitem__` is also called for *slice expressions*. Formally speaking, built-in types handle slicing the same way. Here, for example, is slicing at work on a built-in list, using upper and lower bounds and a stride (see Chapter 7 if you need a refresher on slicing):

```
>>> L = [5, 6, 7, 8, 9]
>>> L[2:4]                              # Slice with slice syntax
[7, 8]
>>> L[1:]
[6, 7, 8, 9]
>>> L[:-1]
[5, 6, 7, 8]
>>> L[::2]
[5, 7, 9]
```

Really, though, slicing bounds are bundled up into a *slice object* and passed to the list's implementation of indexing. In fact, you can always pass a slice object manually—slice syntax is mostly syntactic sugar for indexing with a slice object:

```
>>> L[slice(2, 4)]                      # Slice with slice objects
[7, 8]
>>> L[slice(1, None)]
[6, 7, 8, 9]
>>> L[slice(None, -1)]
[5, 6, 7, 8]
>>> L[slice(None, None, 2)]
[5, 7, 9]
```

This matters in classes with a __getitem__ method—the method will be called both for basic indexing (with an index) and for slicing (with a slice object). Our previous class won't handle slicing because its math assumes integer indexes are passed, but the following class will. When called for indexing, the argument is an integer as before:

```
>>> class Indexer:
...     data = [5, 6, 7, 8, 9]
...     def __getitem__(self, index):    # Called for index or slice
...         print('getitem:', index)
...         return self.data[index]      # Perform index or slice
...
>>> X = Indexer()
>>> X[0]                                 # Indexing sends __getitem__ an integer
getitem: 0
5
>>> X[1]
getitem: 1
6
>>> X[-1]
getitem: -1
9
```

When called for slicing, though, the method receives a slice object, which is simply passed along to the embedded list indexer in a new index expression:

```
>>> X[2:4]                               # Slicing sends __getitem__ a slice object
getitem: slice(2, 4, None)
[7, 8]
>>> X[1:]
getitem: slice(1, None, None)
[6, 7, 8, 9]
>>> X[:-1]
getitem: slice(None, -1, None)
[5, 6, 7, 8]
>>> X[::2]
getitem: slice(None, None, 2)
[5, 7, 9]
```

If used, the __setitem__ index assignment method similarly intercepts both index and slice assignments—it receives a slice object for the latter, which may be passed along in another index assignment in the same way:

```
def __setitem__(self, index, value):     # Intercept index or slice assignment
    ...
    self.data[index] = value              # Assign index or slice
```

In fact, __getitem__ may be called automatically in even more contexts than indexing and slicing, as the next section explains.

Slicing and Indexing in Python 2.6

Prior to Python 3.0, classes could also define __getslice__ and __setslice__ methods to intercept slice fetches and assignments specifically; they were passed the bounds of the slice expression and were preferred over __getitem__ and __setitem__ for slices.

These slice-specific methods have been removed in 3.0, so you should use `__getitem__` and `__setitem__` instead and allow for both indexes and slice objects as arguments. In most classes, this works without any special code, because indexing methods can manually pass along the slice object in the square brackets of another index expression (as in our example). See the section "Membership: __contains__, __iter__, and __getitem__" on page 716 for another example of slice interception at work.

Also, don't confuse the (arguably unfortunately named) `__index__` method in Python 3.0 for index interception; this method returns an integer value for an instance when needed and is used by built-ins that convert to digit strings:

```
>>> class C:
...     def __index__(self):
...         return 255
...
>>> X = C()
>>> hex(X)                  # Integer value
'0xff'
>>> bin(X)
'0b11111111'
>>> oct(X)
'0o377'
```

Although this method does not intercept instance indexing like `__getitem__`, it is also used in contexts that require an integer—including indexing:

```
>>> ('C' * 256)[255]
'C'
>>> ('C' * 256)[X]          # As index (not X[i])
'C'
>>> ('C' * 256)[X:]         # As index (not X[i:])
'C'
```

This method works the same way in Python 2.6, except that it is not called for the `hex` and `oct` built-in functions (use `__hex__` and `__oct__` in 2.6 instead to intercept these calls).

Index Iteration: __getitem__

Here's a trick that isn't always obvious to beginners, but turns out to be surprisingly useful. The `for` statement works by repeatedly indexing a sequence from zero to higher indexes, until an out-of-bounds exception is detected. Because of that, `__getitem__` also turns out to be one way to overload iteration in Python—if this method is defined, `for` loops call the class's `__getitem__` each time through, with successively higher offsets. It's a case of "buy one, get one free"—any built-in or user-defined object that responds to indexing also responds to iteration:

```
>>> class stepper:
...     def __getitem__(self, i):
...         return self.data[i]
...
```

```
>>> X = stepper()                    # X is a stepper object
>>> X.data = "Spam"
>>>
>>> X[1]                             # Indexing calls __getitem__
'p'
>>> for item in X:                   # for loops call __getitem__
...     print(item, end=' ')         # for indexes items 0..N
...
S p a m
```

In fact, it's really a case of "buy one, get a bunch free." Any class that supports `for` loops automatically supports all iteration contexts in Python, many of which we've seen in earlier chapters (iteration contexts were presented in Chapter 14). For example, the `in` membership test, list comprehensions, the `map` built-in, list and tuple assignments, and type constructors will also call `__getitem__` automatically, if it's defined:

```
>>> 'p' in X                         # All call __getitem__ too
True

>>> [c for c in X]                   # List comprehension
['S', 'p', 'a', 'm']

>>> list(map(str.upper, X))          # map calls (use list() in 3.0)
['S', 'P', 'A', 'M']

>>> (a, b, c, d) = X                 # Sequence assignments
>>> a, c, d
('S', 'a', 'm')

>>> list(X), tuple(X), ''.join(X)
(['S', 'p', 'a', 'm'], ('S', 'p', 'a', 'm'), 'Spam')

>>> X
<__main__.stepper object at 0x00A8D5D0>
```

In practice, this technique can be used to create objects that provide a sequence interface and to add logic to built-in sequence type operations; we'll revisit this idea when extending built-in types in Chapter 31.

Iterator Objects: __iter__ and __next__

Although the `__getitem__` technique of the prior section works, it's really just a fallback for iteration. Today, all iteration contexts in Python will try the `__iter__` method first, before trying `__getitem__`. That is, they prefer the iteration protocol we learned about in Chapter 14 to repeatedly indexing an object; only if the object does not support the iteration protocol is indexing attempted instead. Generally speaking, you should prefer `__iter__` too—it supports general iteration contexts better than `__getitem__` can.

Technically, iteration contexts work by calling the `iter` built-in function to try to find an `__iter__` method, which is expected to return an iterator object. If it's provided, Python then repeatedly calls this iterator object's `__next__` method to produce items

until a `StopIteration` exception is raised. If no such `__iter__` method is found, Python falls back on the `__getitem__` scheme and repeatedly indexes by offsets as before, until an `IndexError` exception is raised. A `next` built-in function is also available as a convenience for manual iterations: `next(I)` is the same as `I.__next__()`.

Version skew note: As described in Chapter 14, if you are using Python 2.6, the `I.__next__()` method just described is named `I.next()` in your Python, and the `next(I)` built-in is present for portability: it calls `I.next()` in 2.6 and `I.__next__()` in 3.0. Iteration works the same in 2.6 in all other respects.

User-Defined Iterators

In the `__iter__` scheme, classes implement user-defined iterators by simply implementing the iteration protocol introduced in Chapters 14 and 20 (refer back to those chapters for more background details on iterators). For example, the following file, *iters.py*, defines a user-defined iterator class that generates squares:

```
class Squares:
    def __init__(self, start, stop):     # Save state when created
        self.value = start - 1
        self.stop  = stop
    def __iter__(self):                  # Get iterator object on iter
        return self
    def __next__(self):                  # Return a square on each iteration
        if self.value == self.stop:      # Also called by next built-in
            raise StopIteration
        self.value += 1
        return self.value ** 2

% python
>>> from iters import Squares
>>> for i in Squares(1, 5):              # for calls iter, which calls __iter__
...     print(i, end=' ')                # Each iteration calls __next__
...
1 4 9 16 25
```

Here, the iterator object is simply the instance `self`, because the `__next__` method is part of this class. In more complex scenarios, the iterator object may be defined as a separate class and object with its own state information to support multiple active iterations over the same data (we'll see an example of this in a moment). The end of the iteration is signaled with a Python `raise` statement (more on raising exceptions in the next part of this book). Manual iterations work as for built-in types as well:

```
>>> X = Squares(1, 5)                   # Iterate manually: what loops do
>>> I = iter(X)                         # iter calls __iter__
>>> next(I)                             # next calls __next__
1
>>> next(I)
4
```

```
...more omitted...
>>> next(I)
25
>>> next(I)                          # Can catch this in try statement
StopIteration
```

An equivalent coding of this iterator with `__getitem__` might be less natural, because the `for` would then iterate through all offsets zero and higher; the offsets passed in would be only indirectly related to the range of values produced (`0..N` would need to map to `start..stop`). Because `__iter__` objects retain explicitly managed state between `next` calls, they can be more general than `__getitem__`.

On the other hand, using iterators based on `__iter__` can sometimes be more complex and less convenient than using `__getitem__`. They are really designed for iteration, not random indexing—in fact, they don't overload the indexing expression at all:

```
>>> X = Squares(1, 5)
>>> X[1]
AttributeError: Squares instance has no attribute '__getitem__'
```

The `__iter__` scheme is also the implementation for all the other iteration contexts we saw in action for `__getitem__` (membership tests, type constructors, sequence assignment, and so on). However, unlike our prior `__getitem__` example, we also need to be aware that a class's `__iter__` may be designed for a single traversal, not many. For example, the `Squares` class is a one-shot iteration; once you've iterated over an instance of that class, it's empty. You need to make a new iterator object for each new iteration:

```
>>> X = Squares(1, 5)
>>> [n for n in X]                   # Exhausts items
[1, 4, 9, 16, 25]
>>> [n for n in X]                   # Now it's empty
[]
>>> [n for n in Squares(1, 5)]       # Make a new iterator object
[1, 4, 9, 16, 25]
>>> list(Squares(1, 3))
[1, 4, 9]
```

Notice that this example would probably be simpler if it were coded with generator functions (topics or expressions introduced in Chapter 20 and related to iterators):

```
>>> def gsquares(start, stop):
...     for i in range(start, stop+1):
...         yield i ** 2
...
>>> for i in gsquares(1, 5):         # or: (x ** 2 for x in range(1, 5))
...     print(i, end=' ')
...
1 4 9 16 25
```

Unlike the class, the function automatically saves its state between iterations. Of course, for this artificial example, you could in fact skip both techniques and simply use a `for` loop, `map`, or a list comprehension to build the list all at once. The best and fastest way to accomplish a task in Python is often also the simplest:

```
>>> [x ** 2 for x in range(1, 6)]
[1, 4, 9, 16, 25]
```

However, classes may be better at modeling more complex iterations, especially when they can benefit from state information and inheritance hierarchies. The next section explores one such use case.

Multiple Iterators on One Object

Earlier, I mentioned that the iterator object may be defined as a separate class with its own state information to support multiple active iterations over the same data. Consider what happens when we step across a built-in type like a string:

```
>>> S = 'ace'
>>> for x in S:
...     for y in S:
...         print(x + y, end=' ')
...
aa ac ae ca cc ce ea ec ee
```

Here, the outer loop grabs an iterator from the string by calling `iter`, and each nested loop does the same to get an independent iterator. Because each active iterator has its own state information, each loop can maintain its own position in the string, regardless of any other active loops.

We saw related examples earlier, in Chapters 14 and 20. For instance, generator functions and expressions, as well as built-ins like `map` and `zip`, proved to be single-iterator objects; by contrast, the `range` built-in and other built-in types, like lists, support multiple active iterators with independent positions.

When we code user-defined iterators with classes, it's up to us to decide whether we will support a single active iteration or many. To achieve the multiple-iterator effect, `__iter__` simply needs to define a new stateful object for the iterator, instead of returning `self`.

The following, for example, defines an iterator class that skips every other item on iterations. Because the iterator object is created anew for each iteration, it supports multiple active loops:

```
class SkipIterator:
    def __init__(self, wrapped):
        self.wrapped = wrapped                    # Iterator state information
        self.offset  = 0
    def __next__(self):
        if self.offset >= len(self.wrapped):      # Terminate iterations
            raise StopIteration
        else:
            item = self.wrapped[self.offset]      # else return and skip
            self.offset += 2
            return item

class SkipObject:
```

```
    def __init__(self, wrapped):                    # Save item to be used
        self.wrapped = wrapped
    def __iter__(self):
        return SkipIterator(self.wrapped)           # New iterator each time

if __name__ == '__main__':
    alpha = 'abcdef'
    skipper = SkipObject(alpha)                     # Make container object
    I = iter(skipper)                               # Make an iterator on it
    print(next(I), next(I), next(I))                # Visit offsets 0, 2, 4

    for x in skipper:                # for calls __iter__ automatically
        for y in skipper:            # Nested fors call __iter__ again each time
            print(x + y, end=' ')    # Each iterator has its own state, offset
```

When run, this example works like the nested loops with built-in strings. Each active loop has its own position in the string because each obtains an independent iterator object that records its own state information:

```
% python skipper.py
a c e
aa ac ae ca cc ce ea ec ee
```

By contrast, our earlier `Squares` example supports just one active iteration, unless we call `Squares` again in nested loops to obtain new objects. Here, there is just one `SkipObject`, with multiple iterator objects created from it.

As before, we could achieve similar results with built-in tools—for example, slicing with a third bound to skip items:

```
>>> S = 'abcdef'
>>> for x in S[::2]:
...     for y in S[::2]:            # New objects on each iteration
...         print(x + y, end=' ')
...
aa ac ae ca cc ce ea ec ee
```

This isn't quite the same, though, for two reasons. First, each slice expression here will physically store the result list all at once in memory; iterators, on the other hand, produce just one value at a time, which can save substantial space for large result lists. Second, slices produce new objects, so we're not really iterating over the same object in multiple places here. To be closer to the class, we would need to make a single object to step across by slicing ahead of time:

```
>>> S = 'abcdef'
>>> S = S[::2]
>>> S
'ace'
>>> for x in S:
...     for y in S:                 # Same object, new iterators
...         print(x + y, end=' ')
...
aa ac ae ca cc ce ea ec ee
```

This is more similar to our class-based solution, but it still stores the slice result in memory all at once (there is no generator form of built-in slicing today), and it's only equivalent for this particular case of skipping every other item.

Because iterators can do anything a class can do, they are much more general than this example may imply. Regardless of whether our applications require such generality, user-defined iterators are a powerful tool—they allow us to make arbitrary objects look and feel like the other sequences and iterables we have met in this book. We could use this technique with a database object, for example, to support iterations over database fetches, with multiple cursors into the same query result.

Membership: __contains__, __iter__, and __getitem__

The iteration story is even richer than we've seen thus far. Operator overloading is often *layered*: classes may provide specific methods, or more general alternatives used as fallback options. For example:

- Comparisons in Python 2.6 use specific methods such as `__lt__` for less than if present, or else the general `__cmp__`. Python 3.0 uses only specific methods, not `__cmp__`, as discussed later in this chapter.
- Boolean tests similarly try a specific `__bool__` first (to give an explicit `True`/`False` result), and if it's absent fall back on the more general `__len__` (a nonzero length means `True`). As we'll also see later in this chapter, Python 2.6 works the same but uses the name `__nonzero__` instead of `__bool__`.

In the iterations domain, classes normally implement the `in` membership operator as an iteration, using either the `__iter__` method or the `__getitem__` method. To support more specific membership, though, classes may code a `__contains__` method—when present, this method is preferred over `__iter__`, which is preferred over `__getitem__`. The `__contains__` method should define membership as applying to keys for a *mapping* (and can use quick lookups), and as a search for *sequences*.

Consider the following class, which codes all three methods and tests membership and various iteration contexts applied to an instance. Its methods print trace messages when called:

```
class Iters:
    def __init__(self, value):
        self.data = value
    def __getitem__(self, i):                # Fallback for iteration
        print('get[%s]:' % i, end='')        # Also for index, slice
        return self.data[i]
    def __iter__(self):                      # Preferred for iteration
        print('iter=> ', end='')             # Allows only 1 active iterator
        self.ix = 0
        return self
    def __next__(self):
        print('next:', end='')
```

```
        if self.ix == len(self.data): raise StopIteration
        item = self.data[self.ix]
        self.ix += 1
        return item
    def __contains__(self, x):                # Preferred for 'in'
        print('contains: ', end='')
        return x in self.data


X = Iters([1, 2, 3, 4, 5])                    # Make instance
print(3 in X)                                 # Membership
for i in X:                                   # For loops
    print(i, end=' | ')

print()
print([i ** 2 for i in X])                    # Other iteration contexts
print( list(map(bin, X)) )

I = iter(X)                                   # Manual iteration (what other contexts do)
while True:
    try:
        print(next(I), end=' @ ')
    except StopIteration:
        break
```

When run as it is, this script's output is as follows—the specific `__contains__` intercepts membership, the general `__iter__` catches other iteration contexts such that `__next__` is called repeatedly, and `__getitem__` is never called:

```
contains: True
iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:
iter=> next:next:next:next:next:next:[1, 4, 9, 16, 25]
iter=> next:next:next:next:next:next:['0b1', '0b10', '0b11', '0b100', '0b101']
iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:
```

Watch what happens to this code's output if we comment out its `__contains__` method, though—membership is now routed to the general `__iter__` instead:

```
iter=> next:next:next:True
iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:
iter=> next:next:next:next:next:next:[1, 4, 9, 16, 25]
iter=> next:next:next:next:next:next:['0b1', '0b10', '0b11', '0b100', '0b101']
iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:
```

And finally, here is the output if both `__contains__` and `__iter__` are commented out—the indexing `__getitem__` fallback is called with successively higher indexes for membership and other iteration contexts:

```
get[0]:get[1]:get[2]:True
get[0]:1 | get[1]:2 | get[2]:3 | get[3]:4 | get[4]:5 | get[5]:
get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:[1, 4, 9, 16, 25]
get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:['0b1', '0b10', '0b11', '0b100','0b101']
get[0]:1 @ get[1]:2 @ get[2]:3 @ get[3]:4 @ get[4]:5 @ get[5]:
```

As we've seen, the `__getitem__` method is even more general: besides iterations, it also intercepts explicit indexing as well as slicing. Slice expressions trigger `__getitem__` with a slice object containing bounds, both for built-in types and user-defined classes, so slicing is automatic in our class:

```
>>> X = Iters('spam')                  # Indexing
>>> X[0]                               # __getitem__(0)
get[0]:'s'

>>> 'spam'[1:]                         # Slice syntax
'pam'
>>> 'spam'[slice(1, None)]             # Slice object
'pam'

>>> X[1:]                              # __getitem__(slice(..))
get[slice(1, None, None)]:'pam'
>>> X[:-1]
get[slice(None, -1, None)]:'spa'
```

In more realistic iteration use cases that are not sequence-oriented, though, the `__iter__` method may be easier to write since it must not manage an integer index, and `__contains__` allows for membership optimization as a special case.

Attribute Reference: `__getattr__` and `__setattr__`

The `__getattr__` method intercepts attribute qualifications. More specifically, it's called with the attribute name as a string whenever you try to qualify an instance with an *undefined* (nonexistent) attribute name. It is not called if Python can find the attribute using its inheritance tree search procedure. Because of its behavior, `__getattr__` is useful as a hook for responding to attribute requests in a generic fashion. For example:

```
>>> class empty:
...     def __getattr__(self, attrname):
...         if attrname == "age":
...             return 40
...         else:
...             raise AttributeError, attrname
...
>>> X = empty()
>>> X.age
40
>>> X.name
...error text omitted...
AttributeError: name
```

Here, the `empty` class and its instance `X` have no real attributes of their own, so the access to `X.age` gets routed to the `__getattr__` method; `self` is assigned the instance (`X`), and `attrname` is assigned the undefined attribute name string (`"age"`). The class makes `age` look like a real attribute by returning a real value as the result of the `X.age` qualification expression (`40`). In effect, `age` becomes a *dynamically computed* attribute.

For attributes that the class doesn't know how to handle, `__getattr__` raises the built-in `AttributeError` exception to tell Python that these are bona fide undefined names; asking for `X.name` triggers the error. You'll see `__getattr__` again when we see delegation and properties at work in the next two chapters, and I'll say more about exceptions in Part VII.

A related overloading method, `__setattr__`, intercepts *all* attribute assignments. If this method is defined, `self.attr = value` becomes `self.__setattr__('attr', value)`. This is a bit trickier to use because assigning to any `self` attributes within `__setattr__` calls `__setattr__` again, causing an infinite recursion loop (and eventually, a stack overflow exception!). If you want to use this method, be sure that it assigns any instance attributes by indexing the attribute dictionary, discussed in the next section. That is, use `self.__dict__['name'] = x`, not `self.name = x`:

```
>>> class accesscontrol:
...     def __setattr__(self, attr, value):
...         if attr == 'age':
...             self.__dict__[attr] = value
...         else:
...             raise AttributeError, attr + ' not allowed'
...
>>> X = accesscontrol()
>>> X.age = 40                                  # Calls __setattr__
>>> X.age
40
>>> X.name = 'mel'
...text omitted...
AttributeError: name not allowed
```

These two attribute-access overloading methods allow you to control or specialize access to attributes in your objects. They tend to play highly specialized roles, some of which we'll explore later in this book.

Other Attribute Management Tools

For future reference, also note that there are other ways to manage attribute access in Python:

- The `__getattribute__` method intercepts all attribute fetches, not just those that are undefined, but when using it you must be more cautious than with `__getattr__` to avoid loops.
- The `property` built-in function allows us to associate methods with fetch and set operations on a specific class attribute.
- *Descriptors* provide a protocol for associating `__get__` and `__set__` methods of a class with accesses to a specific class attribute.

Because these are somewhat advanced tools not of interest to every Python programmer, we'll defer a look at properties until Chapter 31 and detailed coverage of all the attribute management techniques until Chapter 37.

Emulating Privacy for Instance Attributes: Part 1

The following code generalizes the previous example, to allow each subclass to have its own list of private names that cannot be assigned to its instances:

```
class PrivateExc(Exception): pass                    # More on exceptions later

class Privacy:
    def __setattr__(self, attrname, value):         # On self.attrname = value
        if attrname in self.privates:
            raise PrivateExc(attrname, self)
        else:
            self.__dict__[attrname] = value          # self.attrname = value loops!

class Test1(Privacy):
    privates = ['age']

class Test2(Privacy):
    privates = ['name', 'pay']
    def __init__(self):
        self.__dict__['name'] = 'Tom'

x = Test1()
y = Test2()

x.name = 'Bob'
y.name = 'Sue'                                       # Fails

y.age  = 30
x.age  = 40                                          # Fails
```

In fact, this is a first-cut solution for an implementation of *attribute privacy* in Python (i.e., disallowing changes to attribute names outside a class). Although Python doesn't support private declarations per se, techniques like this can emulate much of their purpose. This is a partial solution, though; to make it more effective, it must be augmented to allow subclasses to set private attributes more naturally, too, and to use `__getattr__` and a wrapper (sometimes called a proxy) class to check for private attribute fetches.

We'll postpone a more complete solution to attribute privacy until Chapter 38, where we'll use *class decorators* to intercept and validate attributes more generally. Even though privacy can be emulated this way, though, it almost never is in practice. Python programmers are able to write large OOP frameworks and applications without private declarations—an interesting finding about access controls in general that is beyond the scope of our purposes here.

Catching attribute references and assignments is generally a useful technique; it supports *delegation*, a design technique that allows controller objects to wrap up embedded objects, add new behaviors, and route other operations back to the wrapped objects (more on delegation and wrapper classes in Chapter 30).

String Representation: __repr__ and __str__

The next example exercises the `__init__` constructor and the `__add__` overload method, both of which we've already seen, as well as defining a `__repr__` method that returns a string representation for instances. String formatting is used to convert the managed `self.data` object to a string. If defined, `__repr__` (or its sibling, `__str__`) is called automatically when class instances are printed or converted to strings. These methods allow you to define a better display format for your objects than the default instance display.

The default display of instance objects is neither useful nor pretty:

```
>>> class adder:
...     def __init__(self, value=0):
...         self.data = value                   # Initialize data
...     def __add__(self, other):
...         self.data += other                  # Add other in-place (bad!)
...
>>> x = adder()                                 # Default displays
>>> print(x)
<__main__.adder object at 0x025D66B0>
>>> x
<__main__.adder object at 0x025D66B0>
```

But coding or inheriting string representation methods allows us to customize the display:

```
>>> class addrepr(adder):                       # Inherit __init__, __add__
...     def __repr__(self):                     # Add string representation
...         return 'addrepr(%s)' % self.data    # Convert to as-code string
...
>>> x = addrepr(2)                              # Runs __init__
>>> x + 1                                       # Runs __add__
>>> x                                           # Runs __repr__
addrepr(3)
>>> print(x)                                    # Runs __repr__
addrepr(3)
>>> str(x), repr(x)                             # Runs __repr__ for both
('addrepr(3)', 'addrepr(3)')
```

So why two display methods? Mostly, to support different audiences. In full detail:

- `__str__` is tried first for the `print` operation and the `str` built-in function (the internal equivalent of which `print` runs). It generally should return a user-friendly display.
- `__repr__` is used in all other contexts: for interactive echoes, the `repr` function, and nested appearances, as well as by `print` and `str` if no `__str__` is present. It should generally return an as-code string that could be used to re-create the object, or a detailed display for developers.

In a nutshell, `__repr__` is used everywhere, except by `print` and `str` when a `__str__` is defined. Note, however, that while printing falls back on `__repr__` if no `__str__` is

defined, the inverse is not true—other contexts, such as interactive echoes, use `__repr__` only and don't try `__str__` at all:

```
>>> class addstr(adder):
...     def __str__(self):                      # __str__ but no __repr__
...         return '[Value: %s]' % self.data    # Convert to nice string
...
>>> x = addstr(3)
>>> x + 1
>>> x                                           # Default __repr__
<__main__.addstr object at 0x00B35EF0>
>>> print(x)                                    # Runs __str__
[Value: 4]
>>> str(x), repr(x)
('[Value: 4]', '<__main__.addstr object at 0x00B35EF0>')
```

Because of this, `__repr__` may be best if you want a *single* display for all contexts. By defining both methods, though, you can support different displays in different contexts—for example, an end-user display with `__str__`, and a low-level display for programmers to use during development with `__repr__`. In effect, `__str__` simply overrides `__repr__` for user-friendly display contexts:

```
>>> class addboth(adder):
...     def __str__(self):
...         return '[Value: %s]' % self.data    # User-friendly string
...     def __repr__(self):
...         return 'addboth(%s)' % self.data    # As-code string
...
>>> x = addboth(4)
>>> x + 1
>>> x                                           # Runs __repr__
addboth(5)
>>> print(x)                                    # Runs __str__
[Value: 5]
>>> str(x), repr(x)
('[Value: 5]', 'addboth(5)')
```

I should mention two usage notes here. First, keep in mind that `__str__` and `__repr__` must both return strings; other result types are not converted and raise errors, so be sure to run them through a converter if needed. Second, depending on a container's string-conversion logic, the user-friendly display of `__str__` might only apply when objects appear at the top level of a print operation; objects nested in larger objects might still print with their `__repr__` or its default. The following illustrates both of these points:

```
>>> class Printer:
...     def __init__(self, val):
...         self.val = val
...     def __str__(self):                  # Used for instance itself
...         return str(self.val)            # Convert to a string result
...
>>> objs = [Printer(2), Printer(3)]
>>> for x in objs: print(x)                 # __str__ run when instance printed
...                                         # But not when instance in a list!
```

```
2
3
>>> print(objs)
[<__main__.Printer object at 0x025D06F0>, <__main__.Printer object at ...more...
>>> objs
[<__main__.Printer object at 0x025D06F0>, <__main__.Printer object at ...more...
```

To ensure that a custom display is run in all contexts regardless of the container, code `__repr__`, not `__str__`; the former is run in all cases if the latter doesn't apply:

```
>>> class Printer:
...     def __init__(self, val):
...         self.val = val
...     def __repr__(self):                 # __repr__ used by print if no __str__
...         return str(self.val)            # __repr__ used if echoed or nested
...
>>> objs = [Printer(2), Printer(3)]
>>> for x in objs: print(x)                 # No __str__: runs __repr__
...
2
3
>>> print(objs)                             # Runs __repr__, not ___str__
[2, 3]
>>> objs
[2, 3]
```

In practice, `__str__` (or its low-level relative, `__repr__`) seems to be the second most commonly used operator overloading method in Python scripts, behind `__init__`. Any time you can print an object and see a custom display, one of these two tools is probably in use.

Right-Side and In-Place Addition: __radd__ and __iadd__

Technically, the `__add__` method that appeared in the prior example does not support the use of instance objects on the right side of the `+` operator. To implement such expressions, and hence support *commutative*-style operators, code the `__radd__` method as well. Python calls `__radd__` only when the object on the right side of the `+` is your class instance, but the object on the left is not an instance of your class. The `__add__` method for the object on the left is called instead in all other cases:

```
>>> class Commuter:
...     def __init__(self, val):
...         self.val = val
...     def __add__(self, other):
...         print('add', self.val, other)
...         return self.val + other
...     def __radd__(self, other):
...         print('radd', self.val, other)
...         return other + self.val
...
>>> x = Commuter(88)
>>> y = Commuter(99)
```

```
>>> x + 1                       # __add__: instance + noninstance
add 88 1
89
>>> 1 + y                       # __radd__: noninstance + instance
radd 99 1
100
>>> x + y                       # __add__: instance + instance, triggers __radd__
add 88 <__main__.Commuter object at 0x02630910>
radd 99 88
187
```

Notice how the order is reversed in `__radd__`: `self` is really on the right of the `+`, and `other` is on the left. Also note that `x` and `y` are instances of the same class here; when instances of different classes appear mixed in an expression, Python prefers the class of the one on the left. When we add the two instances together, Python runs `__add__`, which in turn triggers `__radd__` by simplifying the left operand.

In more realistic classes where the class type may need to be propagated in results, things can become trickier: type testing may be required to tell whether it's safe to convert and thus avoid nesting. For instance, without the `isinstance` test in the following, we could wind up with a `Commuter` whose `val` is another `Commuter` when two instances are added and `__add__` triggers `__radd__`:

```
>>> class Commuter:                 # Propagate class type in results
...     def __init__(self, val):
...         self.val = val
...     def __add__(self, other):
...         if isinstance(other, Commuter): other = other.val
...         return Commuter(self.val + other)
...     def __radd__(self, other):
...         return Commuter(other + self.val)
...     def __str__(self):
...         return '<Commuter: %s>' % self.val
...
>>> x = Commuter(88)
>>> y = Commuter(99)
>>> print(x + 10)                   # Result is another Commuter instance
<Commuter: 98>
>>> print(10 + y)
<Commuter: 109>

>>> z = x + y                       # Not nested: doesn't recur to __radd__
>>> print(z)
<Commuter: 187>
>>> print(z + 10)
<Commuter: 197>
>>> print(z + z)
<Commuter: 374>
```

In-Place Addition

To also implement `+=` in-place augmented addition, code either an `__iadd__` or an `__add__`. The latter is used if the former is absent. In fact, the prior section's `Commuter` class supports `+=` already for this reason, but `__iadd__` allows for more efficient in-place changes:

```
>>> class Number:
...     def __init__(self, val):
...         self.val = val
...     def __iadd__(self, other):              # __iadd__ explicit: x += y
...         self.val += other                   # Usually returns self
...         return self
...
>>> x = Number(5)
>>> x += 1
>>> x += 1
>>> x.val
7
>>> class Number:
...     def __init__(self, val):
...         self.val = val
...     def __add__(self, other):               # __add__ fallback: x = (x + y)
...         return Number(self.val + other)     # Propagates class type
...
>>> x = Number(5)
>>> x += 1
>>> x += 1
>>> x.val
7
```

Every binary operator has similar right-side and in-place overloading methods that work the same (e.g., `__mul__`, `__rmul__`, and `__imul__`). Right-side methods are an advanced topic and tend to be fairly rarely used in practice; you only code them when you need operators to be commutative, and then only if you need to support such operators at all. For instance, a `Vector` class may use these tools, but an `Employee` or `Button` class probably would not.

Call Expressions: __call__

The `__call__` method is called when your instance is called. No, this isn't a circular definition—if defined, Python runs a `__call__` method for function call expressions applied to your instances, passing along whatever positional or keyword arguments were sent:

```
>>> class Callee:
...     def __call__(self, *pargs, **kargs):        # Intercept instance calls
...         print('Called:', pargs, kargs)          # Accept arbitrary arguments
...
>>> C = Callee()
>>> C(1, 2, 3)                                      # C is a callable object
```

```
Called: (1, 2, 3) {}
>>> C(1, 2, 3, x=4, y=5)
Called: (1, 2, 3) {'y': 5, 'x': 4}
```

More formally, all the argument-passing modes we explored in Chapter 18 are supported by the `__call__` method—whatever is passed to the instance is passed to this method, along with the usual implied instance argument. For example, the method definitions:

```
class C:
    def __call__(self, a, b, c=5, d=6): ...          # Normals and defaults

class C:
    def __call__(self, *pargs, **kargs): ...         # Collect arbitrary arguments

class C:
    def __call__(self, *pargs, d=6, **kargs): ...    # 3.0 keyword-only argument
```

all match all the following instance calls:

```
X = C()
X(1, 2)                                  # Omit defaults
X(1, 2, 3, 4)                            # Positionals
X(a=1, b=2, d=4)                         # Keywords
X(*[1, 2], **dict(c=3, d=4))             # Unpack arbitrary arguments
X(1, *(2,), c=3, **dict(d=4))            # Mixed modes
```

The net effect is that classes and instances with a `__call__` support the exact same argument syntax and semantics as normal functions and methods.

Intercepting call expression like this allows class instances to emulate the look and feel of things like functions, but also retain state information for use during calls (we saw a similar example while exploring scopes in Chapter 17, but you should be more familiar with operator overloading here):

```
>>> class Prod:
...     def __init__(self, value):                # Accept just one argument
...         self.value = value
...     def __call__(self, other):
...         return self.value * other
...
>>> x = Prod(2)                                   # "Remembers" 2 in state
>>> x(3)                                          # 3 (passed) * 2 (state)
6
>>> x(4)
8
```

In this example, the `__call__` may seem a bit gratuitous at first glance. A simple method can provide similar utility:

```
>>> class Prod:
...     def __init__(self, value):
...         self.value = value
...     def comp(self, other):
...         return self.value * other
...
```

```
>>> x = Prod(3)
>>> x.comp(3)
9
>>> x.comp(4)
12
```

However, `__call__` can become more useful when interfacing with APIs that expect functions—it allows us to code objects that conform to an expected function call interface, but also retain state information. In fact, it's probably the third most commonly used operator overloading method, behind the `__init__` constructor and the `__str__` and `__repr__` display-format alternatives.

Function Interfaces and Callback-Based Code

As an example, the `tkinter` GUI toolkit (named `Tkinter` in Python 2.6) allows you to register functions as event handlers (a.k.a. callbacks); when events occur, `tkinter` calls the registered objects. If you want an event handler to retain state between events, you can register either a class's bound method or an instance that conforms to the expected interface with `__call__`. In this section's code, both `x.comp` from the second example and `x` from the first can pass as function-like objects this way.

I'll have more to say about `bound` methods in the next chapter, but for now, here's a hypothetical example of `__call__` applied to the GUI domain. The following class defines an object that supports a function-call interface, but also has state information that remembers the color a button should change to when it is later pressed:

```
class Callback:
    def __init__(self, color):                # Function + state information
        self.color = color
    def __call__(self):                       # Support calls with no arguments
        print('turn', self.color)
```

Now, in the context of a GUI, we can register instances of this class as event handlers for buttons, even though the GUI expects to be able to invoke event handlers as simple functions with no arguments:

```
cb1 = Callback('blue')                        # Remember blue
cb2 = Callback('green')

B1 = Button(command=cb1)                      # Register handlers
B2 = Button(command=cb2)                      # Register handlers
```

When the button is later pressed, the instance object is called as a simple function, exactly like in the following calls. Because it retains state as instance attributes, though, it remembers what to do:

```
cb1()                                         # On events: prints 'blue'
cb2()                                         # Prints 'green'
```

In fact, this is probably the best way to retain state information in the Python language—better than the techniques discussed earlier for functions (global variables,

enclosing function scope references, and default mutable arguments). With OOP, the state remembered is made explicit with attribute assignments.

Before we move on, there are two other ways that Python programmers sometimes tie information to a callback function like this. One option is to use default arguments in `lambda` functions:

```
cb3 = (lambda color='red': 'turn ' + color)  # Or: defaults
print(cb3())
```

The other is to use *bound methods* of a class. A bound method object is a kind of object that remembers the `self` instance and the referenced function. A bound method may therefore be called as a simple function without an instance later:

```
class Callback:
    def __init__(self, color):                # Class with state information
        self.color = color
    def changeColor(self):                    # A normal named method
        print('turn', self.color)

cb1 = Callback('blue')
cb2 = Callback('yellow')

B1 = Button(command=cb1.changeColor)          # Reference, but don't call
B2 = Button(command=cb2.changeColor)          # Remembers function+self
```

In this case, when this button is later pressed it's as if the GUI does this, which invokes the `changeColor` method to process the object's state information:

```
object = Callback('blue')
cb = object.changeColor                       # Registered event handler
cb()                                          # On event prints 'blue'
```

This technique is simpler, but less general than overloading calls with `__call__`; again, watch for more about bound methods in the next chapter.

You'll also see another `__call__` example in Chapter 31, where we will use it to implement something known as a *function decorator*—a callable object often used to add a layer of logic on top of an embedded function. Because `__call__` allows us to attach state information to a callable object, it's a natural implementation technique for a function that must remember and call another function.

Comparisons: `__lt__`, `__gt__`, and Others

As suggested in Table 29-1, classes can define methods to catch all six comparison operators: `<`, `>`, `<=`, `>=`, `==`, and `!=`. These methods are generally straightforward to use, but keep the following qualifications in mind:

- Unlike the `__add__`/`__radd__` pairings discussed earlier, there are no right-side variants of comparison methods. Instead, reflective methods are used when only one operand supports comparison (e.g., `__lt__` and `__gt__` are each other's reflection).
- There are no implicit relationships among the comparison operators. The truth of `==` does not imply that `!=` is false, for example, so both `__eq__` and `__ne__` should be defined to ensure that both operators behave correctly.
- In Python 2.6, a `__cmp__` method is used by all comparisons if no more specific comparison methods are defined; it returns a number that is less than, equal to, or greater than zero, to signal less than, equal, and greater than results for the comparison of its two arguments (`self` and another operand). This method often uses the `cmp(x, y)` built-in to compute its result. Both the `__cmp__` method and the `cmp` built-in function are removed in Python 3.0: use the more specific methods instead.

We don't have space for an in-depth exploration of comparison methods, but as a quick introduction, consider the following class and test code:

```
class C:
    data = 'spam'
    def __gt__(self, other):                # 3.0 and 2.6 version
        return self.data > other
    def __lt__(self, other):
        return self.data < other

X = C()
print(X > 'ham')                            # True  (runs __gt__)
print(X < 'ham')                            # False (runs __lt__)
```

When run under Python 3.0 or 2.6, the prints at the end display the expected results noted in their comments, because the class's methods intercept and implement comparison expressions.

The 2.6 `__cmp__` Method (Removed in 3.0)

In Python 2.6, the `__cmp__` method is used as a fallback if more specific methods are not defined: its integer result is used to evaluate the operator being run. The following produces the same result under 2.6, for example, but fails in 3.0 because `__cmp__` is no longer used:

```
class C:
    data = 'spam'                           # 2.6 only
    def __cmp__(self, other):               # __cmp__ not used in 3.0
        return cmp(self.data, other)        # cmp not defined in 3.0

X = C()
print(X > 'ham')                            # True  (runs __cmp__)
print(X < 'ham')                            # False (runs __cmp__)
```

Notice that this fails in 3.0 because `__cmp__` is no longer special, not because the `cmp` built-in function is no longer present. If we change the prior class to the following to try to simulate the `cmp` call, the code still works in 2.6 but fails in 3.0:

```
class C:
    data = 'spam'
    def __cmp__(self, other):
        return (self.data > other) - (self.data < other)
```

So why, you might be asking, did I just show you a comparison method that is no longer supported in 3.0? While it would be easier to erase history entirely, this book is designed to support both 2.6 and 3.0 readers. Because `__cmp__` may appear in code 2.6 readers must reuse or maintain, it's fair game in this book. Moreover, `__cmp__` was removed more abruptly than the `__getslice__` method described earlier, and so may endure longer. If you use 3.0, though, or care about running your code under 3.0 in the future, don't use `__cmp__` anymore: use the more specific comparison methods instead.

Boolean Tests: __bool__ and __len__

As mentioned earlier, classes may also define methods that give the Boolean nature of their instances—in Boolean contexts, Python first tries `__bool__` to obtain a direct Boolean value and then, if that's missing, tries `__len__` to determine a truth value from the object length. The first of these generally uses object state or other information to produce a Boolean result:

```
>>> class Truth:
...     def __bool__(self): return True
...
>>> X = Truth()
>>> if X: print('yes!')
...
yes!

>>> class Truth:
...     def __bool__(self): return False
...
>>> X = Truth()
>>> bool(X)
False
```

If this method is missing, Python falls back on length because a nonempty object is considered true (i.e., a nonzero length is taken to mean the object is true, and a zero length means it is false):

```
>>> class Truth:
...     def __len__(self): return 0
...
>>> X = Truth()
>>> if not X: print('no!')
```

```
...
no!
```

If both methods are present Python prefers `__bool__` over `__len__`, because it is more specific:

```
>>> class Truth:
...     def __bool__(self): return True          # 3.0 tries __bool__ first
...     def __len__(self): return 0              # 2.6 tries __len__ first
...
>>> X = Truth()
>>> if X: print('yes!')
...
yes!
```

If neither truth method is defined, the object is vacuously considered true (which has potential implications for metaphysically inclined readers!):

```
>>> class Truth:
...     pass
...
>>> X = Truth()
>>> bool(X)
True
```

And now that we've managed to cross over into the realm of philosophy, let's move on to look at one last overloading context: object demise.

Booleans in Python 2.6

Python 2.6 users should use `__nonzero__` instead of `__bool__` in all of the code in the section "Boolean Tests: __bool__ and __len__" on page 730. Python 3.0 renamed the 2.6 `__nonzero__` method to `__bool__`, but Boolean tests work the same otherwise (both 3.0 and 2.6 use `__len__` as a fallback).

If you don't use the 2.6 name, the very first test in this section will work the same for you anyhow, but only because `__bool__` is not recognized as a special method name in 2.6, and objects are considered true by default!

To witness this version difference live, you need to return `False`:

```
C:\misc> c:\python30\python
>>> class C:
...     def __bool__(self):
...         print('in bool')
...         return False
...
>>> X = C()
>>> bool(X)
in bool
False
>>> if X: print(99)
...
in bool
```

This works as advertised in 3.0. In 2.6, though, `__bool__` is ignored and the object is always considered true:

```
C:\misc> c:\python26\python
>>> class C:
...     def __bool__(self):
...         print('in bool')
...         return False
...
>>> X = C()
>>> bool(X)
True
>>> if X: print(99)
...
99
```

In 2.6, use `__nonzero__` for Boolean values (or return `0` from the `__len__` fallback method to designate false):

```
C:\misc> c:\python26\python
>>> class C:
...     def __nonzero__(self):
...         print('in nonzero')
...         return False
...
>>> X = C()
>>> bool(X)
in nonzero
False
>>> if X: print(99)
...
in nonzero
```

But keep in mind that `__nonzero__` works in 2.6 only; if used in 3.0 it will be silently ignored and the object will be classified as true by default—just like using `__bool__` in 2.6!

Object Destruction: `__del__`

We've seen how the `__init__` constructor is called whenever an instance is generated. Its counterpart, the destructor method `__del__`, is run automatically when an instance's space is being reclaimed (i.e., at "garbage collection" time):

```
>>> class Life:
...     def __init__(self, name='unknown'):
...         print('Hello', name)
...         self.name = name
...     def __del__(self):
...         print('Goodbye', self.name)
...
>>> brian = Life('Brian')
Hello Brian
>>> brian = 'loretta'
Goodbye Brian
```

Here, when `brian` is assigned a string, we lose the last reference to the `Life` instance and so trigger its destructor method. This works, and it may be useful for implementing some cleanup activities (such as terminating server connections). However, destructors are not as commonly used in Python as in some OOP languages, for a number of reasons.

For one thing, because Python automatically reclaims all space held by an instance when the instance is reclaimed, destructors are not necessary for space management.* For another, because you cannot always easily predict when an instance will be reclaimed, it's often better to code termination activities in an explicitly called method (or `try`/`finally` statement, described in the next part of the book); in some cases, there may be lingering references to your objects in system tables that prevent destructors from running.

In fact, `__del__` can be tricky to use for even more subtle reasons. Exceptions raised within it, for example, simply print a warning message to `sys.stderr` (the standard error stream) rather than triggering an exception event, because of the unpredictable context under which it is run by the garbage collector. In addition, cyclic (a.k.a. circular) references among objects may prevent garbage collection from happening when you expect it to; an optional cycle detector, enabled by default, can automatically collect such objects eventually, but only if they do not have `__del__` methods. Since this is relatively obscure, we'll ignore further details here; see Python's standard manuals' coverage of both `__del__` and the `gc` garbage collector module for more information.

Chapter Summary

That's as many overloading examples as we have space for here. Most of the other operator overloading methods work similarly to the ones we've explored, and all are just hooks for intercepting built-in type operations; some overloading methods, for example, have unique argument lists or return values. We'll see a few others in action later in the book:

- Chapter 33 uses the `__enter__` and `__exit__` `with` statement context manager methods.
- Chapter 37 uses the `__get__` and `__set__` class descriptor fetch/set methods.
- Chapter 39 uses the `__new__` object creation method in the context of metaclasses.

* In the current C implementation of Python, you also don't need to close file objects held by the instance in destructors because they are automatically closed when reclaimed. However, as mentioned in Chapter 9, it's better to explicitly call file close methods because auto-close-on-reclaim is a feature of the implementation, not of the language itself (this behavior can vary under Jython, for instance).

In addition, some of the methods we've studied here, such as `__call__` and `__str__`, will be employed by later examples in this book. For complete coverage, though, I'll defer to other documentation sources—see Python's standard language manual or reference books for details on additional overloading methods.

In the next chapter, we leave the realm of class mechanics behind to explore common design patterns—the ways that classes are commonly used and combined to optimize code reuse. Before you read on, though, take a moment to work though the chapter quiz below to review the concepts we've covered.

Test Your Knowledge: Quiz

1. What two operator overloading methods can you use to support iteration in your classes?
2. What two operator overloading methods handle printing, and in what contexts?
3. How can you intercept slice operations in a class?
4. How can you catch in-place addition in a class?
5. When should you provide operator overloading?

Test Your Knowledge: Answers

1. Classes can support iteration by defining (or inheriting) `__getitem__` or `__iter__`. In all iteration contexts, Python tries to use `__iter__` (which returns an object that supports the iteration protocol with a `__next__` method) first: if no `__iter__` is found by inheritance search, Python falls back on the `__getitem__` indexing method (which is called repeatedly, with successively higher indexes).
2. The `__str__` and `__repr__` methods implement object print displays. The former is called by the `print` and `str` built-in functions; the latter is called by `print` and `str` if there is no `__str__`, and always by the `repr` built-in, interactive echoes, and nested appearances. That is, `__repr__` is used everywhere, except by `print` and `str` when a `__str__` is defined. A `__str__` is usually used for user-friendly displays; `__repr__` gives extra details or the object's as-code form.
3. Slicing is caught by the `__getitem__` indexing method: it is called with a slice object, instead of a simple index. In Python 2.6, `__getslice__` (defunct in 3.0) may be used as well.
4. In-place addition tries `__iadd__` first, and `__add__` with an assignment second. The same pattern holds true for all binary operators. The `__radd__` method is also available for right-side addition.

5. When a class naturally matches, or needs to emulate, a built-in type's interfaces. For example, collections might imitate sequence or mapping interfaces. You generally shouldn't implement expression operators if they don't naturally map to your objects, though—use normally named methods instead.

CHAPTER 30
Designing with Classes

So far in this part of the book, we've concentrated on using Python's OOP tool, the class. But OOP is also about *design issues*—i.e., how to use classes to model useful objects. This chapter will touch on a few core OOP ideas and present some additional examples that are more realistic than those shown so far.

Along the way, we'll code some common OOP design patterns in Python, such as inheritance, composition, delegation, and factories. We'll also investigate some design-focused class concepts, such as pseudoprivate attributes, multiple inheritance, and bound methods. Many of the design terms mentioned here require more explanation than I can provide in this book; if this material sparks your curiosity, I suggest exploring a text on OOP design or design patterns as a next step.

Python and OOP

Let's begin with a review—Python's implementation of OOP can be summarized by three ideas:

Inheritance
: Inheritance is based on attribute lookup in Python (in `X.name` expressions).

Polymorphism
: In `X.method`, the meaning of `method` depends on the type (class) of `X`.

Encapsulation
: Methods and operators implement behavior; data hiding is a convention by default.

By now, you should have a good feel for what inheritance is all about in Python. We've also talked about Python's polymorphism a few times already; it flows from Python's lack of type declarations. Because attributes are always resolved at runtime, objects that implement the same interfaces are interchangeable; clients don't need to know what sorts of objects are implementing the methods they call.

Encapsulation means packaging in Python—that is, hiding implementation details behind an object's interface. It does not mean enforced privacy, though that can be implemented with code, as we'll see in Chapter 38. Encapsulation allows the implementation of an object's interface to be changed without impacting the users of that object.

Overloading by Call Signatures (or Not)

Some OOP languages also define polymorphism to mean overloading functions based on the type signatures of their arguments. But because there are no type declarations in Python, this concept doesn't really apply; polymorphism in Python is based on object *interfaces*, not types.

You can try to overload methods by their argument lists, like this:

```
class C:
    def meth(self, x):
        ...
    def meth(self, x, y, z):
        ...
```

This code will run, but because the `def` simply assigns an object to a name in the class's scope, the last definition of the method function is the only one that will be retained (it's just as if you say `X = 1` and then `X = 2`; `X` will be `2`).

Type-based selections can always be coded using the type-testing ideas we met in Chapters 4 and 9, or the argument list tools introduced in Chapter 18:

```
class C:
    def meth(self, *args):
        if len(args) == 1:
            ...
        elif type(arg[0]) == int:
            ...
```

You normally shouldn't do this, though—as described in Chapter 16, you should write your code to expect an object interface, not a specific data type. That way, it will be useful for a broader category of types and applications, both now and in the future:

```
class C:
    def meth(self, x):
        x.operation()                    # Assume x does the right thing
```

It's also generally considered better to use distinct method names for distinct operations, rather than relying on call signatures (no matter what language you code in).

Although Python's object model is straightforward, much of the art in OOP is in the way we combine classes to achieve a program's goals. The next section begins a tour of some of the ways larger programs use classes to their advantage.

OOP and Inheritance: "Is-a" Relationships

We've explored the mechanics of inheritance in depth already, but I'd like to show you an example of how it can be used to model real-world relationships. From a programmer's point of view, inheritance is kicked off by attribute qualifications, which trigger searches for names in instances, their classes, and then any superclasses. From a designer's point of view, inheritance is a way to specify set membership: a class defines a set of properties that may be inherited and customized by more specific sets (i.e., subclasses).

To illustrate, let's put that pizza-making robot we talked about at the start of this part of the book to work. Suppose we've decided to explore alternative career paths and open a pizza restaurant. One of the first things we'll need to do is hire employees to serve customers, prepare the food, and so on. Being engineers at heart, we've decided to build a robot to make the pizzas; but being politically and cybernetically correct, we've also decided to make our robot a full-fledged employee with a salary.

Our pizza shop team can be defined by the four classes in the example file, *employees.py*. The most general class, `Employee`, provides common behavior such as bumping up salaries (`giveRaise`) and printing (`__repr__`). There are two kinds of employees, and so two subclasses of `Employee`: `Chef` and `Server`. Both override the inherited `work` method to print more specific messages. Finally, our pizza robot is modeled by an even more specific class: `PizzaRobot` is a kind of `Chef`, which is a kind of `Employee`. In OOP terms, we call these relationships "is-a" links: a robot is a chef, which is a(n) employee. Here's the *employees.py* file:

```
class Employee:
    def __init__(self, name, salary=0):
        self.name   = name
        self.salary = salary
    def giveRaise(self, percent):
        self.salary = self.salary + (self.salary * percent)
    def work(self):
        print(self.name, "does stuff")
    def __repr__(self):
        return "<Employee: name=%s, salary=%s>" % (self.name, self.salary)

class Chef(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 50000)
    def work(self):
        print(self.name, "makes food")

class Server(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 40000)
    def work(self):
        print(self.name, "interfaces with customer")

class PizzaRobot(Chef):
```

```
    def __init__(self, name):
        Chef.__init__(self, name)
    def work(self):
        print(self.name, "makes pizza")

if __name__ == "__main__":
    bob = PizzaRobot('bob')         # Make a robot named bob
    print(bob)                      # Run inherited __repr__
    bob.work()                      # Run type-specific action
    bob.giveRaise(0.20)             # Give bob a 20% raise
    print(bob); print()

    for klass in Employee, Chef, Server, PizzaRobot:
        obj = klass(klass.__name__)
        obj.work()
```

When we run the self-test code included in this module, we create a pizza-making robot named `bob`, which inherits names from three classes: `PizzaRobot`, `Chef`, and `Employee`. For instance, printing `bob` runs the `Employee.__repr__` method, and giving `bob` a raise invokes `Employee.giveRaise` because that's where the inheritance search finds that method:

```
C:\python\examples> python employees.py
<Employee: name=bob, salary=50000>
bob makes pizza
<Employee: name=bob, salary=60000.0>

Employee does stuff
Chef makes food
Server interfaces with customer
PizzaRobot makes pizza
```

In a class hierarchy like this, you can usually make instances of any of the classes, not just the ones at the bottom. For instance, the `for` loop in this module's self-test code creates instances of all four classes; each responds differently when asked to work because the `work` method is different in each. Really, these classes just simulate real-world objects; `work` prints a message for the time being, but it could be expanded to do real work later.

OOP and Composition: "Has-a" Relationships

The notion of composition was introduced in Chapter 25. From a programmer's point of view, composition involves embedding other objects in a container object, and activating them to implement container methods. To a designer, composition is another way to represent relationships in a problem domain. But, rather than set membership, composition has to do with components—parts of a whole.

Composition also reflects the relationships between parts, called a "has-a" relationships. Some OOP design texts refer to composition as *aggregation* (or distinguish between the two terms by using aggregation to describe a weaker dependency between

container and contained); in this text, a "composition" simply refers to a collection of embedded objects. The composite class generally provides an interface all its own and implements it by directing the embedded objects.

Now that we've implemented our employees, let's put them in the pizza shop and let them get busy. Our pizza shop is a composite object: it has an oven, and it has employees like servers and chefs. When a customer enters and places an order, the components of the shop spring into action—the server takes the order, the chef makes the pizza, and so on. The following example (the file *pizzashop.py*) simulates all the objects and relationships in this scenario:

```
from employees import PizzaRobot, Server

class Customer:
    def __init__(self, name):
        self.name = name
    def order(self, server):
        print(self.name, "orders from", server)
    def pay(self, server):
        print(self.name, "pays for item to", server)

class Oven:
    def bake(self):
        print("oven bakes")

class PizzaShop:
    def __init__(self):
        self.server = Server('Pat')          # Embed other objects
        self.chef   = PizzaRobot('Bob')      # A robot named bob
        self.oven   = Oven()

    def order(self, name):
        customer = Customer(name)            # Activate other objects
        customer.order(self.server)          # Customer orders from server
        self.chef.work()
        self.oven.bake()
        customer.pay(self.server)

if __name__ == "__main__":
    scene = PizzaShop()                      # Make the composite
    scene.order('Homer')                     # Simulate Homer's order
    print('...')
    scene.order('Shaggy')                    # Simulate Shaggy's order
```

The `PizzaShop` class is a container and controller; its constructor makes and embeds instances of the employee classes we wrote in the last section, as well as an `Oven` class defined here. When this module's self-test code calls the `PizzaShop order` method, the embedded objects are asked to carry out their actions in turn. Notice that we make a new `Customer` object for each order, and we pass on the embedded `Server` object to `Customer` methods; customers come and go, but the server is part of the pizza shop composite. Also notice that employees are still involved in an inheritance relationship; composition and inheritance are complementary tools.

When we run this module, our pizza shop handles two orders—one from Homer, and then one from Shaggy:

```
C:\python\examples> python pizzashop.py
Homer orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Homer pays for item to <Employee: name=Pat, salary=40000>
...
Shaggy orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Shaggy pays for item to <Employee: name=Pat, salary=40000>
```

Again, this is mostly just a toy simulation, but the objects and interactions are representative of composites at work. As a rule of thumb, classes can represent just about any objects and relationships you can express in a sentence; just replace *nouns* with classes, and *verbs* with methods, and you'll have a first cut at a design.

Stream Processors Revisited

For a more realistic composition example, recall the generic data stream processor function we partially coded in the introduction to OOP in Chapter 25:

```
def processor(reader, converter, writer):
    while 1:
        data = reader.read()
        if not data: break
        data = converter(data)
        writer.write(data)
```

Rather than using a simple function here, we might code this as a class that uses composition to do its work to provide more structure and support inheritance. The following file, *streams.py*, demonstrates one way to code the class:

```
class Processor:
    def __init__(self, reader, writer):
        self.reader = reader
        self.writer = writer
    def process(self):
        while 1:
            data = self.reader.readline()
            if not data: break
            data = self.converter(data)
            self.writer.write(data)
    def converter(self, data):
        assert False, 'converter must be defined'       # Or raise exception
```

This class defines a `converter` method that it expects subclasses to fill in; it's an example of the *abstract superclass* model we outlined in Chapter 28 (more on `assert` in Part VII). Coded this way, `reader` and `writer` objects are embedded within the class instance (*composition*), and we supply the conversion logic in a subclass rather than passing in a converter function (*inheritance*). The file *converters.py* shows how:

```
from streams import Processor

class Uppercase(Processor):
    def converter(self, data):
        return data.upper()

if __name__ == '__main__':
    import sys
    obj = Uppercase(open('spam.txt'), sys.stdout)
    obj.process()
```

Here, the `Uppercase` class inherits the stream-processing loop logic (and anything else that may be coded in its superclasses). It needs to define only what is unique about it—the data conversion logic. When this file is run, it makes and runs an instance that reads from the file *spam.txt* and writes the uppercase equivalent of that file to the `stdout` stream:

```
C:\lp4e> type spam.txt
spam
Spam
SPAM!

C:\lp4e> python converters.py
SPAM
SPAM
SPAM!
```

To process different sorts of streams, pass in different sorts of objects to the class construction call. Here, we use an output file instead of a stream:

```
C:\lp4e> python
>>> import converters
>>> prog = converters.Uppercase(open('spam.txt'), open('spamup.txt', 'w'))
>>> prog.process()

C:\lp4e> type spamup.txt
SPAM
SPAM
SPAM!
```

But, as suggested earlier, we could also pass in arbitrary objects wrapped up in classes that define the required input and output method interfaces. Here's a simple example that passes in a writer class that wraps up the text inside HTML tags:

```
C:\lp4e> python
>>> from converters import Uppercase
>>>
>>> class HTMLize:
...     def write(self, line):
...         print('<PRE>%s</PRE>' % line.rstrip())
...
>>> Uppercase(open('spam.txt'), HTMLize()).process()
<PRE>SPAM</PRE>
<PRE>SPAM</PRE>
<PRE>SPAM!</PRE>
```

If you trace through this example's control flow, you'll see that we get both uppercase conversion (by inheritance) and HTML formatting (by composition), even though the core processing logic in the original `Processor` superclass knows nothing about either step. The processing code only cares that writers have a `write` method and that a method named `convert` is defined; it doesn't care what those methods do when they are called. Such polymorphism and encapsulation of logic is behind much of the power of classes.

As is, the `Processor` superclass only provides a file-scanning loop. In more realistic work, we might extend it to support additional programming tools for its subclasses, and, in the process, turn it into a full-blown framework. Coding such a tool once in a superclass enables you to reuse it in all of your programs. Even in this simple example, because so much is packaged and inherited with classes, all we had to code was the HTML formatting step; the rest was free.

For another example of composition at work, see exercise 9 at the end of Chapter 31 and its solution in Appendix B; it's similar to the pizza shop example. We've focused on inheritance in this book because that is the main tool that the Python language itself provides for OOP. But, in practice, composition is used as much as inheritance as a way to structure classes, especially in larger systems. As we've seen, inheritance and composition are often complementary (and sometimes alternative) techniques. Because composition is a design issue outside the scope of the Python language and this book, though, I'll defer to other resources for more on this topic.

Why You Will Care: Classes and Persistence

I've mentioned Python's `pickle` and `shelve` object persistence support a few times in this part of the book because it works especially well with class instances. In fact, these tools are often compelling enough to motivate the use of classes in general—by picking or shelving a class instance, we get data storage that contains both data and logic combined.

For example, besides allowing us to simulate real-world interactions, the pizza shop classes developed in this chapter could also be used as the basis of a persistent restaurant database. Instances of classes can be stored away on disk in a single step using Python's `pickle` or `shelve` modules. We used shelves to store instances of classes in the OOP tutorial in Chapter 27, but the object pickling interface is remarkably easy to use as well:

```
import pickle
object = someClass()
file   = open(filename, 'wb')      # Create external file
pickle.dump(object, file)          # Save object in file

import pickle
file   = open(filename, 'rb')
object = pickle.load(file)         # Fetch it back later
```

Pickling converts in-memory objects to serialized byte streams (really, strings), which may be stored in files, sent across a network, and so on; unpickling converts back from byte streams to identical in-memory objects. Shelves are similar, but they automatically pickle objects to an access-by-key database, which exports a dictionary-like interface:

```
import shelve
object = someClass()
dbase  = shelve.open('filename')
dbase['key'] = object             # Save under key

import shelve
dbase  = shelve.open('filename')
object = dbase['key']             # Fetch it back later
```

In our pizza shop example, using classes to model employees means we can get a simple database of employees and shops with little extra work—pickling such instance objects to a file makes them persistent across Python program executions:

```
>>> from pizzashop import PizzaShop
>>> shop = PizzaShop()
>>> shop.server, shop.chef
(<Employee: name=Pat, salary=40000>, <Employee: name=Bob, salary=50000>)
>>> import pickle
>>> pickle.dump(shop, open('shopfile.dat', 'wb'))
```

This stores an entire composite `shop` object in a file all at once. To bring it back later in another session or program, a single step suffices as well. In fact, objects restored this way retain both state and behavior:

```
>>> import pickle
>>> obj = pickle.load(open('shopfile.dat', 'rb'))
>>> obj.server, obj.chef
(<Employee: name=Pat, salary=40000>, <Employee: name=Bob, salary=50000>)
>>> obj.order('Sue')
Sue orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Sue pays for item to <Employee: name=Pat, salary=40000>
```

See the standard library manual and later examples for more on pickles and shelves.

OOP and Delegation: "Wrapper" Objects

Beside inheritance and composition, object-oriented programmers often also talk about something called *delegation*, which usually implies controller objects that embed other objects to which they pass off operation requests. The controllers can take care of administrative activities, such as keeping track of accesses and so on. In Python, delegation is often implemented with the `__getattr__` method hook; because it intercepts accesses to nonexistent attributes, a *wrapper* class (sometimes called a *proxy* class) can use `__getattr__` to route arbitrary accesses to a wrapped object. The wrapper class retains the interface of the wrapped object and may add additional operations of its own.

Consider the file *trace.py*, for instance:

```
class wrapper:
    def __init__(self, object):
        self.wrapped = object                    # Save object
    def __getattr__(self, attrname):
        print('Trace:', attrname)                # Trace fetch
        return getattr(self.wrapped, attrname)   # Delegate fetch
```

Recall from Chapter 29 that `__getattr__` gets the attribute name as a string. This code makes use of the `getattr` built-in function to fetch an attribute from the wrapped object by name string—`getattr(X,N)` is like `X.N`, except that `N` is an expression that evaluates to a string at runtime, not a variable. In fact, `getattr(X,N)` is similar to `X.__dict__[N]`, but the former also performs an inheritance search, like `X.N`, while the latter does not (see "Namespace Dictionaries" on page 696 for more on the `__dict__` attribute).

You can use the approach of this module's wrapper class to manage access to any object with attributes—lists, dictionaries, and even classes and instances. Here, the `wrapper` class simply prints a trace message on each attribute access and delegates the attribute request to the embedded `wrapped` object:

```
>>> from trace import wrapper
>>> x = wrapper([1,2,3])                  # Wrap a list
>>> x.append(4)                           # Delegate to list method
Trace: append
>>> x.wrapped                             # Print my member
[1, 2, 3, 4]

>>> x = wrapper({"a": 1, "b": 2})         # Wrap a dictionary
>>> x.keys()                              # Delegate to dictionary method
Trace: keys
['a', 'b']
```

The net effect is to augment the entire interface of the `wrapped` object, with additional code in the `wrapper` class. We can use this to log our method calls, route method calls to extra or custom logic, and so on.

We'll revive the notions of wrapped objects and delegated operations as one way to extend built-in types in Chapter 31. If you are interested in the delegation design pattern, also watch for the discussions in Chapters 31 and 38 of *function decorators*, a strongly related concept designed to augment a specific function or method call rather than the entire interface of an object, and *class decorators*, which serve as a way to automatically add such delegation-based wrappers to all instances of a class.

Version skew note: In Python 2.6, operator overloading methods run by built-in operations are routed through generic attribute interception methods like `__getattr__`. Printing a wrapped object directly, for example, calls this method for `__repr__` or `__str__`, which then passes the call on to the wrapped object. In Python 3.0, this no longer happens: printing does not trigger `__getattr__`, and a default display is used instead. In 3.0, new-style classes look up operator overloading methods in classes and skip the normal instance lookup entirely. We'll return to this issue in Chapter 37, in the context of managed attributes; for now, keep in mind that you may need to redefine operator overloading methods in wrapper classes (either by hand, by tools, or by superclasses) if you want them to be intercepted in 3.0.

Pseudoprivate Class Attributes

Besides larger structuring goals, class designs often must address name usage too. In Part V, we learned that every name assigned at the top level of a module file is exported. By default, the same holds for classes—data hiding is a convention, and clients may fetch or change any class or instance attribute they like. In fact, attributes are all "public" and "virtual," in C++ terms; they're all accessible everywhere and are looked up dynamically at runtime.*

That said, Python today does support the notion of name "mangling" (i.e., expansion) to localize some names in classes. Mangled names are sometimes misleadingly called "private attributes," but really this is just a way to *localize* a name to the class that created it—name mangling does not prevent access by code outside the class. This feature is mostly intended to avoid namespace collisions in instances, not to restrict access to names in general; mangled names are therefore better called "pseudoprivate" than "private."

Pseudoprivate names are an advanced and entirely optional feature, and you probably won't find them very useful until you start writing general tools or larger class hierarchies for use in multiprogrammer projects. In fact, they are not always used even when they probably should be—more commonly, Python programmers code internal names with a single underscore (e.g., `_X`), which is just an informal convention to let you know that a name shouldn't be changed (it means nothing to Python itself).

Because you may see this feature in other people's code, though, you need to be somewhat aware of it, even if you don't use it yourself.

* This tends to scare people with a C++ background unnecessarily. In Python, it's even possible to change or completely delete a class method at runtime. On the other hand, almost nobody ever does this in practical programs. As a scripting language, Python is more about enabling than restricting. Also, recall from our discussion of operator overloading in Chapter 29 that `__getattr__` and `__setattr__` can be used to emulate privacy, but are generally not used for this purpose in practice. More on this when we code a more realistic privacy decorator Chapter 38.

Name Mangling Overview

Here's how name mangling works: names inside a `class` statement that start with two underscores but don't end with two underscores are automatically expanded to include the name of the enclosing class. For instance, a name like `__X` within a class named `Spam` is changed to `_Spam__X` automatically: the original name is prefixed with a single underscore and the enclosing class's name. Because the modified name contains the name of the enclosing class, it's somewhat unique; it won't clash with similar names created by other classes in a hierarchy.

Name mangling happens only in `class` statements, and only for names that begin with two leading underscores. However, it happens for *every* name preceded with double underscores—both class attributes (like method names) and instance attribute names assigned to `self` attributes. For example, in a class named `Spam`, a method named `__meth` is mangled to `_Spam__meth`, and an instance attribute reference `self.__X` is transformed to `self._Spam__X`. Because more than one class may add attributes to an instance, this mangling helps avoid clashes—but we need to move on to an example to see how.

Why Use Pseudoprivate Attributes?

One of the main problems that the pseudoprivate attribute feature is meant to alleviate has to do with the way instance attributes are stored. In Python, all instance attributes wind up in the single instance object at the bottom of the class tree. This is different from the C++ model, where each class gets its own space for data members it defines.

Within a class method in Python, whenever a method assigns to a `self` attribute (e.g., `self.`*`attr`* `=` *`value`*), it changes or creates an attribute in the instance (inheritance searches happen only on reference, not on assignment). Because this is true even if multiple classes in a hierarchy assign to the same attribute, collisions are possible.

For example, suppose that when a programmer codes a class, she assumes that she owns the attribute name `X` in the instance. In this class's methods, the name is set, and later fetched:

```
class C1:
    def meth1(self): self.X = 88          # I assume X is mine
    def meth2(self): print(self.X)
```

Suppose further that another programmer, working in isolation, makes the same assumption in a class that he codes:

```
class C2:
    def metha(self): self.X = 99          # Me too
    def methb(self): print(self.X)
```

Both of these classes work by themselves. The problem arises if the two classes are ever mixed together in the same class tree:

```
class C3(C1, C2): ...
I = C3()                                # Only 1 X in I!
```

Now, the value that each class gets back when it says `self.X` will depend on which class assigned it last. Because all assignments to `self.X` refer to the same single instance, there is only one `X` attribute—`I.X`—no matter how many classes use that attribute name.

To guarantee that an attribute belongs to the class that uses it, prefix the name with double underscores everywhere it is used in the class, as in this file, *private.py*:

```
class C1:
    def meth1(self): self.__X = 88      # Now X is mine
    def meth2(self): print(self.__X)    # Becomes _C1__X in I
class C2:
    def metha(self): self.__X = 99      # Me too
    def methb(self): print(self.__X)    # Becomes _C2__X in I

class C3(C1, C2): pass
I = C3()                                # Two X names in I

I.meth1(); I.metha()
print(I.__dict__)
I.meth2(); I.methb()
```

When thus prefixed, the `X` attributes will be expanded to include the names of their classes before being added to the instance. If you run a `dir` call on `I` or inspect its namespace dictionary after the attributes have been assigned, you'll see the expanded names, `_C1__X` and `_C2__X`, but not `X`. Because the expansion makes the names unique within the instance, the class coders can safely assume that they truly own any names that they prefix with two underscores:

```
% python private.py
{'_C2__X': 99, '_C1__X': 88}
88
99
```

This trick can avoid potential name collisions in the instance, but note that it does not amount to true privacy. If you know the name of the enclosing class, you can still access either of these attributes anywhere you have a reference to the instance by using the fully expanded name (e.g., `I._C1__X = 77`). On the other hand, this feature makes it less likely that you will *accidentally* step on a class's names.

Pseudoprivate attributes are also useful in larger frameworks or tools, both to avoid introducing new method names that might accidentally hide definitions elsewhere in the class tree and to reduce the chance of internal methods being replaced by names defined lower in the tree. If a method is intended for use only within a class that may be mixed into other classes, the double underscore prefix ensures that the method won't interfere with other names in the tree, especially in multiple-inheritance scenarios:

```
class Super:
    def method(self): ...                   # A real application method

class Tool:
```

```
    def __method(self): ...                 # Becomes _Tool__method
    def other(self): self.__method()        # Use my internal method

class Sub1(Tool, Super): ...
    def actions(self): self.method()        # Runs Super.method as expected

class Sub2(Tool):
    def __init__(self): self.method = 99    # Doesn't break Tool.__method
```

We met multiple inheritance briefly in Chapter 25 and will explore it in more detail later in this chapter. Recall that superclasses are searched according to their left-to-right order in `class` header lines. Here, this means `Sub1` prefers `Tool` attributes to those in `Super`. Although in this example we could force Python to pick the application class's methods first by switching the order of the superclasses listed in the `Sub1` class header, pseudoprivate attributes resolve the issue altogether. Pseudoprivate names also prevent subclasses from accidentally redefining the internal method's names, as in `Sub2`.

Again, I should note that this feature tends to be of use primarily for larger, multiprogrammer projects, and then only for selected names. Don't be tempted to clutter your code unnecessarily; only use this feature for names that truly need to be controlled by a single class. For simpler programs, it's probably overkill.

For more examples that make use of the `__X` naming feature, see the *lister.py* mix-in classes introduced later in this chapter, in the section on multiple inheritance, as well as the discussion of `Private` class decorators in Chapter 38. If you care about privacy in general, you might want to review the emulation of private instance attributes sketched in the section "Attribute Reference: __getattr__ and __setattr__" on page 718 in Chapter 29, and watch for the `Private` class decorator in Chapter 38 that we will base upon this special method. Although it's possible to emulate true access controls in Python classes, this is rarely done in practice, even for large systems.

Methods Are Objects: Bound or Unbound

Methods in general, and bound methods in particular, simplify the implementation of many design goals in Python. We met bound methods briefly while studying `__call__` in Chapter 29. The full story, which we'll flesh out here, turns out to be more general and flexible than you might expect.

In Chapter 19, we learned how functions can be processed as normal objects. Methods are a kind of object too, and can be used generically in much the same way as other objects—they can be assigned, passed to functions, stored in data structures, and so on. Because class methods can be accessed from an instance or a class, though, they actually come in two flavors in Python:

Unbound class method objects: no `self`
: Accessing a function attribute of a class by qualifying the class returns an unbound method object. To call the method, you must provide an instance object explicitly as the first argument. In Python 3.0, an unbound method is the same as a simple function and can be called though the class's name; in 2.6 it's a distinct type and cannot be called without providing an instance.

Bound instance method objects: `self` + *function pairs*
: Accessing a function attribute of a class by qualifying an instance returns a bound method object. Python automatically packages the instance with the function in the bound method object, so you don't need to pass an instance to call the method.

Both kinds of methods are full-fledged objects; they can be transferred around a program at will, just like strings and numbers. Both also require an instance in their first argument when run (i.e., a value for `self`). This is why we had to pass in an instance explicitly when calling superclass methods from subclass methods in the previous chapter; technically, such calls produce unbound method objects.

When calling a bound method object, Python provides an instance for you automatically—the instance used to create the bound method object. This means that bound method objects are usually interchangeable with simple function objects, and makes them especially useful for interfaces originally written for functions (see the sidebar "Why You Will Care: Bound Methods and Callbacks" on page 756 for a realistic example).

To illustrate, suppose we define the following class:

```
class Spam:
    def doit(self, message):
        print(message)
```

Now, in normal operation, we make an instance and call its method in a single step to print the passed-in argument:

```
object1 = Spam()
object1.doit('hello world')
```

Really, though, a *bound* method object is generated along the way, just before the method call's parentheses. In fact, we can fetch a bound method without actually calling it. An *`object.name`* qualification is an object expression. In the following, it returns a bound method object that packages the instance (`object1`) with the method function (`Spam.doit`). We can assign this bound method pair to another name and then call it as though it were a simple function:

```
object1 = Spam()
x = object1.doit          # Bound method object: instance+function
x('hello world')          # Same effect as object1.doit('...')
```

On the other hand, if we qualify the class to get to `doit`, we get back an *unbound* method object, which is simply a reference to the function object. To call this type of method, we must pass in an instance as the leftmost argument:

```
object1 = Spam()
t = Spam.doit            # Unbound method object (a function in 3.0: see ahead)
t(object1, 'howdy')      # Pass in instance (if the method expects one in 3.0)
```

By extension, the same rules apply within a class's method if we reference `self` attributes that refer to functions in the class. A `self.method` expression is a bound method object because `self` is an instance object:

```
class Eggs:
    def m1(self, n):
        print(n)
    def m2(self):
        x = self.m1      # Another bound method object
        x(42)            # Looks like a simple function

Eggs().m2()              # Prints 42
```

Most of the time, you call methods immediately after fetching them with attribute qualification, so you don't always notice the method objects generated along the way. But if you start writing code that calls objects generically, you need to be careful to treat unbound methods specially—they normally require an explicit instance object to be passed in.†

Unbound Methods are Functions in 3.0

In Python 3.0, the language has dropped the notion of *unbound methods*. What we describe as an unbound method here is treated as a simple function in 3.0. For most purposes, this makes no difference to your code; either way, an instance will be passed to a method's first argument when it's called through an instance.

Programs that do explicit type testing might be impacted, though—if you print the type of an instance-less class method, it displays "unbound method" in 2.6, and "function" in 3.0.

Moreover, in 3.0 it is OK to call a method without an instance, as long as the method does not expect one and you call it only through the class and never through an instance. That is, Python 3.0 will pass along an instance to methods only for through-instance calls. When calling through a class, you must pass an instance manually only if the method expects one:

```
C:\misc> c:\python30\python
>>> class Selfless:
```

† See the discussion of static and class methods in Chapter 31 for an optional exception to this rule. Like bound methods, static methods can masquerade as basic functions because they do not expect instances when called. Python supports three kinds of class methods—instance, static, and class—and 3.0 allows simple functions in classes, too.

```
...     def __init__(self, data):
...         self.data = data
...     def selfless(arg1, arg2):               # A simple function in 3.0
...         return arg1 + arg2
...     def normal(self, arg1, arg2):           # Instance expected when called
...         return self.data + arg1 + arg2
...
>>> X = Selfless(2)
>>> X.normal(3, 4)                  # Instance passed to self automatically
9
>>> Selfless.normal(X, 3, 4)        # self expected by method: pass manually
9
>>> Selfless.selfless(3, 4)         # No instance: works in 3.0, fails in 2.6!
7
```

The last test in this fails in 2.6, because unbound methods require an instance to be passed by default; it works in 3.0 because such methods are treated as simple functions not requiring an instance. Although this removes some potential error trapping in 3.0 (what if a programmer accidentally forgets to pass an instance?), it allows class methods to be used as simple functions as long as they are not passed and do not expect a "self" instance argument.

The following two calls still fail in both 3.0 and 2.6, though—the first (calling through an instance) automatically passes an instance to a method that does not expect one, while the second (calling through a class) does not pass an instance to a method that does expect one:

```
>>> X.selfless(3, 4)
TypeError: selfless() takes exactly 2 positional arguments (3 given)

>>> Selfless.normal(3, 4)
TypeError: normal() takes exactly 3 positional arguments (2 given)
```

Because of this change, the `staticmethod` decorator described in the next chapter is not needed in 3.0 for methods without a `self` argument that are called only through the class name, and never through an instance—such methods are run as simple functions, without receiving an instance argument. In 2.6, such calls are errors unless an instance is passed manually (more on static methods in the next chapter).

It's important to be aware of the differences in behavior in 3.0, but bound methods are generally more important from a practical perspective anyway. Because they pair together the instance and function in a single object, they can be treated as callables generically. The next section demonstrates what this means in code.

For a more visual illustration of unbound method treatment in Python 3.0 and 2.6, see also the *lister.py* example in the multiple inheritance section later in this chapter. Its classes print the value of methods fetched from both instances and classes, in both versions of Python.

Bound Methods and Other Callable Objects

As mentioned earlier, bound methods can be processed as generic objects, just like simple functions—they can be passed around a program arbitrarily. Moreover, because bound methods combine both a function and an instance in a single package, they can be treated like any other callable object and require no special syntax when invoked. The following, for example, stores four bound method objects in a list and calls them later with normal call expressions:

```
>>> class Number:
...     def __init__(self, base):
...         self.base = base
...     def double(self):
...         return self.base * 2
...     def triple(self):
...         return self.base * 3
...
>>> x = Number(2)                                  # Class instance objects
>>> y = Number(3)                                  # State + methods
>>> z = Number(4)
>>> x.double()                                     # Normal immediate calls
4

>>> acts = [x.double, y.double, y.triple, z.double]   # List of bound methods
>>> for act in acts:                               # Calls are deferred
...     print(act())                               # Call as though functions
...
4
6
9
8
```

Like simple functions, bound method objects have introspection information of their own, including attributes that give access to the instance object and method function they pair. Calling the bound method simply dispatches the pair:

```
>>> bound = x.double
>>> bound.__self__, bound.__func__
(<__main__.Number object at 0x0278F610>, <function double at 0x027A4ED0>)
>>> bound.__self__.base
2
>>> bound()                                # Calls bound.__func__(bound.__self__, ...)
4
```

In fact, bound methods are just one of a handful of callable object types in Python. As the following demonstrates, simple functions coded with a `def` or `lambda`, instances that inherit a `__call__`, and bound instance methods can all be treated and called the same way:

```
>>> def square(arg):
...     return arg ** 2                         # Simple functions (def or lambda)
...
>>> class Sum:
...     def __init__(self, val):                # Callable instances
```

```
...         self.val = val
...     def __call__(self, arg):
...         return self.val + arg
...
>>> class Product:
...     def __init__(self, val):                  # Bound methods
...         self.val = val
...     def method(self, arg):
...         return self.val * arg
...
>>> sobject = Sum(2)
>>> pobject = Product(3)
>>> actions = [square, sobject, pobject.method]  # Function, instance, method

>>> for act in actions:                          # All 3 called same way
...     print(act(5))                            # Call any 1-arg callable
...
25
7
15
>>> actions[-1](5)                               # Index, comprehensions, maps
15
>>> [act(5) for act in actions]
[25, 7, 15]
>>> list(map(lambda act: act(5), actions))
[25, 7, 15]
```

Technically speaking, classes belong in the callable objects category too, but we normally call them to generate instances rather than to do actual work, as shown here:

```
>>> class Negate:
...     def __init__(self, val):                  # Classes are callables too
...         self.val = -val                       # But called for object, not work
...     def __repr__(self):                       # Instance print format
...         return str(self.val)
...
>>> actions = [square, sobject, pobject.method, Negate]      # Call a class too
>>> for act in actions:
...     print(act(5))
...
25
7
15
-5
>>> [act(5) for act in actions]                   # Runs __repr__ not __str__!
[25, 7, 15, -5]

>>> table = {act(5): act for act in actions}      # 2.6/3.0 dict comprehension
>>> for (key, value) in table.items():
...     print('{0:2} => {1}'.format(key, value))  # 2.6/3.0 str.format
...
-5 => <class '__main__.Negate'>
25 => <function square at 0x025D4978>
15 => <bound method Product.method of <__main__.Product object at 0x025D0F90>>
 7 => <__main__.Sum object at 0x025D0F70>
```

As you can see, bound methods, and Python's callable objects model in general, are some of the many ways that Python's design makes for an incredibly flexible language.

You should now understand the method object model. For other examples of bound methods at work, see the upcoming sidebar "Why You Will Care: Bound Methods and Callbacks" as well as the prior chapter's discussion of callback handlers in the section on the method `__call__`.

Why You Will Care: Bound Methods and Callbacks

Because bound methods automatically pair an instance with a class method function, you can use them anywhere a simple function is expected. One of the most common places you'll see this idea put to work is in code that registers methods as event callback handlers in the `tkinter` GUI interface (named `Tkinter` in Python 2.6). Here's the simple case:

```
def handler():
    ...use globals for state...
...
widget = Button(text='spam', command=handler)
```

To register a handler for button click events, we usually pass a callable object that takes no arguments to the `command` keyword argument. Function names (and `lambda`s) work here, and so do class methods, as long as they are bound methods:

```
class MyWidget:
    def handler(self):
        ...use self.attr for state...
    def makewidgets(self):
        b = Button(text='spam', command=self.handler)
```

Here, the event handler is `self.handler`—a bound method object that remembers both `self` and `MyGui.handler`. Because `self` will refer to the original instance when `handler` is later invoked on events, the method will have access to instance attributes that can retain state between events. With simple functions, state normally must be retained in global variables or enclosing function scopes instead. See also the discussion of `__call__` operator overloading in Chapter 29 for another way to make classes compatible with function-based APIs.

Multiple Inheritance: "Mix-in" Classes

Many class-based designs call for combining disparate sets of methods. In a `class` statement, more than one superclass can be listed in parentheses in the header line. When you do this, you use something called *multiple inheritance*—the class and its instances inherit names from *all* the listed superclasses.

When searching for an attribute, Python's inheritance search traverses all superclasses in the class header from left to right until a match is found. Technically, because any of the superclasses may have superclasses of its own, this search can be a bit more complex for larger class tress:

- In classic classes (the default until Python 3.0), the attribute search proceeds depth-first all the way to the top of the inheritance tree, and then from left to right.
- In new-style classes (and all classes in 3.0), the attribute search proceeds across by tree levels, in a more breadth-first fashion (see the new-style class discussion in the next chapter).

In either model, though, when a class has multiple superclasses, they are searched from left to right according to the order listed in the `class` statement header lines.

In general, multiple inheritance is good for modeling objects that belong to more than one set. For instance, a person may be an engineer, a writer, a musician, and so on, and inherit properties from all such sets. With multiple inheritance, objects obtain the union of the behavior in all their superclasses.

Perhaps the most common way multiple inheritance is used is to "mix in" general-purpose methods from superclasses. Such superclasses are usually called *mix-in classes*—they provide methods you add to application classes by inheritance. In a sense, mix-in classes are similar to modules: they provide packages of methods for use in their client subclasses. Unlike simple functions in modules, though, methods in mix-ins also have access to the `self` instance, for using state information and other methods. The next section demonstrates one common use case for such tools.

Coding Mix-in Display Classes

As we've seen, Python's default way to print a class instance object isn't incredibly useful:

```
>>> class Spam:
...     def __init__(self):                    # No __repr__ or __str__
...         self.data1 = "food"
...
>>> X = Spam()
>>> print(X)                                   # Default: class, address
<__main__.Spam object at 0x00864818>           # Displays "instance" in Python 2.6
```

As you saw in Chapter 29 when studying operator overloading, you can provide a `__str__` or `__repr__` method to implement a custom string representation of your own. But, rather than coding one of these in each and every class you wish to print, why not code it once in a general-purpose tool class and inherit it in all your classes?

That's what mix-ins are for. Defining a display method in a mix-in superclass once enables us to reuse it anywhere we want to see a custom display format. We've already seen tools that do related work:

- Chapter 27's `AttrDisplay` class formatted instance attributes in a generic `__str__` method, but it did not climb class trees and was used in single-inheritance mode only.
- Chapter 28's *classtree.py* module defined functions for climbing and sketching class trees, but it did not display object attributes along the way and was not architected as an inheritable class.

Here, we're going to revisit these examples' techniques and expand upon them to code a set of three mix-in classes that serve as generic display tools for listing instance attributes, inherited attributes, and attributes on all objects in a class tree. We'll also use our tools in multiple-inheritance mode and deploy coding techniques that make classes better suited to use as generic tools.

Listing instance attributes with __dict__

Let's get started with the simple case—listing attributes attached to an instance. The following class, coded in the file *lister.py*, defines a mix-in called `ListInstance` that overloads the `__str__` method for all classes that include it in their header lines. Because this is coded as a class, `ListInstance` is a generic tool whose formatting logic can be used for instances of any subclass:

```
# File lister.py

class ListInstance:
    """
    Mix-in class that provides a formatted print() or str() of
    instances via inheritance of __str__, coded here; displays
    instance attrs only; self is the instance of lowest class;
    uses __X names to avoid clashing with client's attrs
    """
    def __str__(self):
        return '<Instance of %s, address %s:\n%s>' % (
                            self.__class__.__name__,         # My class's name
                            id(self),                        # My address
                            self.__attrnames())              # name=value list
    def __attrnames(self):
        result = ''
        for attr in sorted(self.__dict__):                   # Instance attr dict
            result += '\tname %s=%s\n' % (attr, self.__dict__ [attr])
        retubrn result
```

`ListInstance` uses some previously explored tricks to extract the instance's class name and attributes:

- Each instance has a built-in `__class__` attribute that references the class from which it was created, and each class has a `__name__` attribute that references the name in the header, so the expression `self.__class__.__name__` fetches the name of an instance's class.

- This class does most of its work by simply scanning the instance's attribute dictionary (remember, it's exported in `__dict__`) to build up a string showing the names and values of all instance attributes. The dictionary's keys are sorted to finesse any ordering differences across Python releases.

In these respects, `ListInstance` is similar to Chapter 27's attribute display; in fact, it's largely just a variation on a theme. Our class here uses two additional techniques, though:

- It displays the instance's memory address by calling the `id` built-function, which returns any object's address (by definition, a unique object identifier, which will be useful in later mutations of this code).
- It uses the *pseudoprivate* naming pattern for its worker method: `__attrnames`. As we learned earlier in his chapter, Python automatically localizes any such name to its enclosing class by expanding the attribute name to include the class name (in this case, it becomes `_ListInstance__attrnames`). This holds true for both class attributes (like methods) and instance attributes attached to `self`. This behavior is useful in a general tool like this, as it ensures that its names don't clash with any names used in its client subclasses.

Because `ListInstance` defines a `__str__` operator overloading method, instances derived from this class display their attributes automatically when printed, giving a bit more information than a simple address. Here is the class in action, in single-inheritance mode (this code works the same in both Python 3.0 and 2.6):

```
>>> from lister import ListInstance
>>> class Spam(ListInstance):                    # Inherit a __str__ method
...     def __init__(self):
...         self.data1 = 'food'
...
>>> x = Spam()
>>> print(x)                                     # print() and str() run __str__
<Instance of Spam, address 40240880:
        name data1=food
>
```

You can also fetch the listing output as a string without printing it with `str`, and interactive echoes still use the default format:

```
>>> str(x)
'<Instance of Spam, address 40240880:\n\tname data1=food\n>'
>>> x                                            # The __repr__ still is a default
<__main__.Spam object at 0x026606F0>
```

The `ListInstance` class is useful for any classes you write—even classes that already have one or more superclasses. This is where *multiple inheritance* comes in handy: by adding `ListInstance` to the list of superclasses in a class header (i.e., mixing it in), you get its `__str__` "for free" while still inheriting from the existing superclass(es). The file *testmixin.py* demonstrates:

```
# File testmixin.py

from lister import *                    # Get lister tool classes

class Super:
    def __init__(self):                 # Superclass __init__
        self.data1 = 'spam'             # Create instance attrs
    def ham(self):
        pass

class Sub(Super, ListInstance):         # Mix in ham and a __str__
    def __init__(self):                 # listers have access to self
        Super.__init__(self)
        self.data2 = 'eggs'             # More instance attrs
        self.data3 = 42
    def spam(self):                     # Define another method here
        pass

if __name__ == '__main__':
    X = Sub()
    print(X)                            # Run mixed-in __str__
```

Here, `Sub` inherits names from both `Super` and `ListInstance`; it's a composite of its own names and names in both its superclasses. When you make a `Sub` instance and print it, you automatically get the custom representation mixed in from `ListInstance` (in this case, this script's output is the same under both Python 3.0 and 2.6, except for object addresses):

```
C:\misc> C:\python30\python testmixin.py
<Instance of Sub, address 40962576:
        name data1=spam
        name data2=eggs
        name data3=42
>
```

`ListInstance` works in any class it's mixed into because `self` refers to an instance of the subclass that pulls this class in, whatever that may be. In a sense, mix-in classes are the class equivalent of modules—packages of methods useful in a variety of clients. For example, here is `Lister` working again in single-inheritance mode on a different class's instances, with `import` and attributes set outside the class:

```
>>> import lister
>>> class C(lister.ListInstance): pass
...
>>> x = C()
>>> x.a = 1; x.b = 2; x.c = 3
>>> print(x)
<Instance of C, address 40961776:
        name a=1
        name b=2
        name c=3
>
```

Besides the utility they provide, mix-ins optimize code maintenance, like all classes do. For example, if you later decide to extend `ListInstance`'s `__str__` to also print all the class attributes that an instance inherits, you're safe; because it's an inherited method, changing `__str__` automatically updates the display of each subclass that imports the class and mixes it in. Since it's now officially "later," let's move on to the next section to see what such an extension might look like.

Listing inherited attributes with dir

As it is, our `Lister` mix-in displays instance attributes only (i.e., names attached to the instance object itself). It's trivial to extend the class to display all the attributes accessible from an instance, though—both its own and those it inherits from its classes. The trick is to use the `dir` built-in function instead of scanning the instance's `__dict__` dictionary; the latter holds instance attributes only, but the former also collects all inherited attributes in Python 2.2 and later.

The following mutation codes this scheme; I've renamed it to facilitate simple testing, but if this were to *replace* the original version, all existing clients would pick up the new display automatically:

```
# File lister.py, continued

class ListInherited:
    """
    Use dir() to collect both instance attrs and names
    inherited from its classes; Python 3.0 shows more
    names than 2.6 because of the implied object superclass
    in the new-style class model; getattr() fetches inherited
    names not in self.__dict__; use __str__, not __repr__,
    or else this loops when printing bound methods!
    """
    def __str__(self):
        return '<Instance of %s, address %s:\n%s>' % (
                           self.__class__.__name__,         # My class's name
                           id(self),                        # My address
                           self.__attrnames())              # name=value list
    def __attrnames(self):
        result = ''
        for attr in dir(self):                              # Instance dir()
            if attr[:2] == '__' and attr[-2:] == '__':      # Skip internals
                result += '\tname %s=<>\n' % attr
            else:
                result += '\tname %s=%s\n' % (attr, getattr(self, attr))
        return result
```

Notice that this code skips `__X__` names' values; most of these are internal names that we don't generally care about in a generic listing like this. This version also must use the `getattr` built-in function to fetch attributes by name string instead of using instance attribute dictionary indexing—`getattr` employs the inheritance search protocol, and some of the names we're listing here are not stored on the instance itself.

To test the new version, change the *testmixin.py* file to use this new class instead:

```
class Sub(Super, ListInherited):                    # Mix in a __str__
```

This file's output varies per release. In Python 2.6, we get the following; notice the name mangling at work in the lister's method name (I shortened its full value display to fit on this page):

```
C:\misc> c:\python26\python testmixin.py
<Instance of Sub, address 40073136:
        name _ListInherited__attrnames=<bound method Sub.__attrnames of <...more...>>
        name __doc__=<>
        name __init__=<>
        name __module__=<>
        name __str__=<>
        name data1=spam
        name data2=eggs
        name data3=42
        name ham=<bound method Sub.ham of <__main__.Sub instance at 0x026377B0>>
        name spam=<bound method Sub.spam of <__main__.Sub instance at 0x026377B0>>
>
```

In Python 3.0, more attributes are displayed because all classes are "new-style" and inherit names from the implied `object` superclass (more on this in Chapter 31). Because so many names are inherited from the default superclass, I've omitted many here; run this on your own for the full listing:

```
C:\misc> c:\python30\python testmixin.py
<Instance of Sub, address 40831792:
        name _ListInherited__attrnames=<bound method Sub.__attrnames of <...more...>>
        name __class__=<>
        name __delattr__=<>
        name __dict__=<>
        name __doc__=<>
        name __eq__=<>
        ...more names omitted...
        name __repr__=<>
        name __setattr__=<>
        name __sizeof__=<>
        name __str__=<>
        name __subclasshook__=<>
        name __weakref__=<>
        name data1=spam
        name data2=eggs
        name data3=42
        name ham=<bound method Sub.ham of <__main__.Sub object at 0x026F0B30>>
        name spam=<bound method Sub.spam of <__main__.Sub object at 0x026F0B30>>
>
```

One caution here—now that we're displaying inherited methods too, we have to use `__str__` instead of `__repr__` to overload printing. With `__repr__`, this code will *loop*—displaying the value of a method triggers the `__repr__` of the method's class, in order to display the class. That is, if the lister's `__repr__` tries to display a method, displaying the method's class will trigger the lister's `__repr__` again. Subtle, but true! Change

__str__ to __repr__ here to see this for yourself. If you must use __repr__ in such a context, you can avoid the loops by using isinstance to compare the type of attribute values against types.MethodType in the standard library, to know which items to skip.

Listing attributes per object in class trees

Let's code one last extension. As it is, our lister doesn't tell us which class an inherited name comes from. As we saw in the *classtree.py* example near the end of Chapter 28, though, it's straightforward to climb class inheritance trees in code. The following mix-in class makes use of this same technique to display attributes grouped by the classes they live in—it sketches the full class tree, displaying attributes attached to each object along the way. It does so by traversing the inheritance tree from an instance's __class__ to its class, and then from the class's __bases__ to all superclasses recursively, scanning object __dicts__s along the way:

```
# File lister.py, continued

class ListTree:
    """
    Mix-in that returns an __str__ trace of the entire class
    tree and all its objects' attrs at and above self;
    run by print(), str() returns constructed string;
    uses __X attr names to avoid impacting clients;
    uses generator expr to recurse to superclasses;
    uses str.format() to make substitutions clearer
    """
    def __str__(self):
        self.__visited = {}
        return '<Instance of {0}, address {1}:\n{2}{3}>'.format(
                            self.__class__.__name__,
                            id(self),
                            self.__attrnames(self, 0),
                            self.__listclass(self.__class__, 4))

    def __listclass(self, aClass, indent):
        dots = '.' * indent
        if aClass in self.__visited:
            return '\n{0}<Class {1}:, address {2}: (see above)>\n'.format(
                            dots,
                            aClass.__name__,
                            id(aClass))
        else:
            self.__visited[aClass] = True
            genabove = (self.__listclass(c, indent+4) for c in aClass.__bases__)
            return '\n{0}<Class {1}, address {2}:\n{3}{4}{5}>\n'.format(
                            dots,
                            aClass.__name__,
                            id(aClass),
                            self.__attrnames(aClass, indent),
                            ''.join(genabove),
                            dots)

    def __attrnames(self, obj, indent):
```

```
        spaces = ' ' * (indent + 4)
        result = ''
        for attr in sorted(obj.__dict__):
            if attr.startswith('__') and attr.endswith('__'):
                result += spaces + '{0}=<>\n'.format(attr)
            else:
                result += spaces + '{0}={1}\n'.format(attr, getattr(obj, attr))
        return result
```

Note the use of a *generator expression* to direct the recursive calls for superclasses; it's activated by the nested string `join` method. Also see how this version uses the Python 3.0 and 2.6 string `format` method instead of `%` formatting expressions, to make substitutions clearer; when many substitutions are applied like this, explicit argument numbers may make the code easier to decipher. In short, in this version we exchange the first of the following lines for the second:

```
        return '<Instance of %s, address %s:\n%s%s>' % (...)          # Expression
        return '<Instance of {0}, address {1}:\n{2}{3}>'.format(...)  # Method
```

Now, change *testmixin.py* to inherit from this new class again to test:

```
class Sub(Super, ListTree):          # Mix in a __str__
```

The file's tree-sketcher output in Python 2.6 is then as follows:

```
C:\misc> c:\python26\python testmixin.py
<Instance of Sub, address 40728496:
    _ListTree__visited={}
    data1=spam
    data2=eggs
    data3=42

....<Class Sub, address 40701168:
        __doc__=<>
        __init__=<>
        __module__=<>
        spam=<unbound method Sub.spam>

........<Class Super, address 40701120:
            __doc__=<>
            __init__=<>
            __module__=<>
            ham=<unbound method Super.ham>
........>

........<Class ListTree, address 40700688:
            _ListTree__attrnames=<unbound method ListTree.__attrnames>
            _ListTree__listclass=<unbound method ListTree.__listclass>
            __doc__=<>
            __module__=<>
            __str__=<>
........>
....>
>
```

Notice in this output how methods are *unbound* now under 2.6, because we fetch them from classes directly, instead of from instances. Also observe how the lister's `__visited` table has its name mangled in the instance's attribute dictionary; unless we're very unlucky, this won't clash with other data there.

Under Python 3.0, we get extra attributes and superclasses again. Notice that unbound methods are simple *functions* in 3.0, as described in an earlier note in this chapter (and that again, I've deleted most built-in attributes in `object` to save space here; run this on your own for the complete listing):

```
C:\misc> c:\python30\python testmixin.py
<Instance of Sub, address 40635216:
    _ListTree__visited={}
    data1=spam
    data2=eggs
    data3=42

....<Class Sub, address 40914752:
        __doc__=<>
        __init__=<>
        __module__=<>
        spam=<function spam at 0x026D53D8>

........<Class Super, address 40829952:
            __dict__=<>
            __doc__=<>
            __init__=<>
            __module__=<>
            __weakref__=<>
            ham=<function ham at 0x026D5228>

............<Class object, address 505114624:
                __class__=<>
                __delattr__=<>
                __doc__=<>
                __eq__=<>
                ...more omitted...
                __repr__=<>
                __setattr__=<>
                __sizeof__=<>
                __str__=<>
                __subclasshook__=<>
............>
........>

........<Class ListTree, address 40829496:
            _ListTree__attrnames=<function __attrnames at 0x026D5660>
            _ListTree__listclass=<function __listclass at 0x026D56A8>
            __dict__=<>
            __doc__=<>
            __module__=<>
            __str__=<>
            __weakref__=<>
```

```
.....<Class object:, address 505114624: (see above)>
.....>
....>
>
```

This version avoids listing the same class object twice by keeping a table of classes *visited* so far (this is why an object's `id` is included—to serve as a key for a previously displayed item). Like the transitive module reloader of Chapter 24, a dictionary works to avoid repeats and cycles here because class objects may be dictionary keys; a set would provide similar functionality.

This version also takes care to avoid large internal objects by skipping `__X__` names again. If you comment out the test for these names, their values will display normally. Here's an excerpt from the output in 2.6 with this temporary change made (it's much larger in its entirety, and it gets even worse in 3.0, which is why these names are probably better skipped!):

```
C:\misc> c:\python26\python testmixin.py
...more omitted...

.....<Class ListTree, address 40700688:
           _ListTree__attrnames=<unbound method ListTree.__attrnames>
           _ListTree__listclass=<unbound method ListTree.__listclass>
           __doc__=
    Mix-in that returns the __str__ trace of the entire class
    tree and all its objects' attrs at and above self;
    run by print, str returns constructed string;
    uses __X attr names to avoid impacting clients;
    uses generator expr to recurse to superclasses;
    uses str.format() to make substitutions clearer

           __module__=lister
           __str__=<unbound method ListTree.__str__>
.....>
```

For more fun, try mixing this class into something more substantial, like the `Button` class of Python's `tkinter` GUI toolkit module. In general, you'll want to name `ListTree` first (leftmost) in a `class` header, so its `__str__` is picked up; `Button` has one, too, and the leftmost superclass is searched first in multiple inheritance. The output of the following is fairly massive (18K characters), so run this code on your own to see the full listing (and if you're using Python 2.6, recall that you should use `Tkinter` for the module name instead of `tkinter`):

```
>>> from lister import ListTree
>>> from tkinter import Button                  # Both classes have a __str__
>>> class MyButton(ListTree, Button): pass      # ListTree first: use its __str__
...
>>> B = MyButton(text='spam')
>>> open('savetree.txt', 'w').write(str(B))     # Save to a file for later viewing
18247
>>> print(B)                                    # Print the display here
<Instance of MyButton, address 44355632:
    _ListTree__visited={}
```

```
    _name=44355632
    _tclCommands=[]
    ...much more omitted...
>
```

Of course, there's much more we could do here (sketching the tree in a GUI might be a natural next step), but we'll leave further work as a suggested exercise. We'll also extend this code in the exercises at the end of this part of the book, to list superclass names in parentheses at the start of instance and class displays.

The main point here is that OOP is all about code reuse, and mix-in classes are a powerful example. Like almost everything else in programming, multiple inheritance can be a useful device when applied well. In practice, though, it is an advanced feature and can become complicated if used carelessly or excessively. We'll revisit this topic as a gotcha at the end of the next chapter. In that chapter, we'll also meet the new-style class model, which modifies the search order for one special multiple inheritance case.

Supporting slots: Because they scan instance dictionaries, the `ListInstance` and `ListTree` classes presented here don't directly support attributes stored in *slots*—a newer and relatively rarely used option we'll meet in the next chapter, where instance attributes are declared in a `__slots__` class attribute. For example, if in *textmixin.py* we assign `__slots__=['data1']` in `Super` and `__slots__=['data3']` in `Sub`, only the `data2` attribute is displayed in the instance by these two lister classes; `ListTree` also displays `data1` and `data3`, but as attributes of the `Super` and `Sub` class objects and with a special format for their values (technically, they are class-level descriptors).

To better support slot attributes in these classes, change the `__dict__` scanning loops to also iterate through `__slots__` lists using code the next chapter will present, and use the `getattr` built-in function to fetch values instead of `__dict__` indexing (`ListTree` already does). Since instances inherit only the lowest class's `__slots__`, you may also need to come up with a policy when `__slots__` lists appear in multiple superclasses (`ListTree` already displays them as class attributes). `ListInherited` is immune to all this, because `dir` results combine both `__dict__` names and all classes' `__slots__` names.

Alternatively, as a policy we could simply let our code handle slot-based attributes as it currently does, rather than complicating it for a rare, advanced feature. Slots and normal instance attributes are different kinds of names. We'll investigate slots further in the next chapter; I omitted addressing them in these examples to avoid a forward dependency (not counting this note, of course!)—not exactly a valid design goal, but reasonable for a book.

Classes Are Objects: Generic Object Factories

Sometimes, class-based designs require objects to be created in response to conditions that can't be predicted when a program is written. The factory design pattern allows such a deferred approach. Due in large part to Python's flexibility, factories can take multiple forms, some of which don't seem special at all.

Because classes are objects, it's easy to pass them around a program, store them in data structures, and so on. You can also pass classes to functions that generate arbitrary kinds of objects; such functions are sometimes called *factories* in OOP design circles. Factories are a major undertaking in a strongly typed language such as C++ but are almost trivial to implement in Python. The call syntax we met in Chapter 18 can call any class with any number of constructor arguments in one step to generate any sort of instance:‡

```
def factory(aClass, *args):            # Varargs tuple
    return aClass(*args)               # Call aClass (or apply in 2.6 only)

class Spam:
    def doit(self, message):
        print(message)

class Person:
    def __init__(self, name, job):
        self.name = name
        self.job  = job

object1 = factory(Spam)                        # Make a Spam object
object2 = factory(Person, "Guido", "guru")     # Make a Person object
```

In this code, we define an object generator function called `factory`. It expects to be passed a class object (any class will do) along with one or more arguments for the class's constructor. The function uses special "varargs" call syntax to call the function and return an instance.

The rest of the example simply defines two classes and generates instances of both by passing them to the `factory` function. And that's the only factory function you'll ever need to write in Python; it works for any class and any constructor arguments.

One possible improvement worth noting is that to support keyword arguments in constructor calls, the factory can collect them with a `**args` argument and pass them along in the class call, too:

```
def factory(aClass, *args, **kwargs):        # +kwargs dict
    return aClass(*args, **kwargs)           # Call aClass
```

‡ Actually, this syntax can invoke any callable object, including functions, classes, and methods. Hence, the `factory` function here can also run any callable object, not just a class (despite the argument name). Also, as we learned in Chapter 18, Python 2.6 has an alternative to `aClass(*args)`: the `apply(aClass, args)` built-in call, which has been removed in Python 3.0 because of its redundancy and limitations.

By now, you should know that everything is an "object" in Python, including things like classes, which are just compiler input in languages like C++. However, as mentioned at the start of this part of the book, only objects *derived* from classes are OOP objects in Python.

Why Factories?

So what good is the `factory` function (besides providing an excuse to illustrate class objects in this book)? Unfortunately, it's difficult to show applications of this design pattern without listing much more code than we have space for here. In general, though, such a factory might allow code to be insulated from the details of dynamically configured object construction.

For instance, recall the `processor` example presented in the abstract in Chapter 25, and then again as a composition example in this chapter. It accepts reader and writer objects for processing arbitrary data streams. The original version of this example manually passed in instances of specialized classes like `FileWriter` and `SocketReader` to customize the data streams being processed; later, we passed in hardcoded file, stream, and formatter objects. In a more dynamic scenario, external devices such as configuration files or GUIs might be used to configure the streams.

In such a dynamic world, we might not be able to hardcode the creation of stream interface objects in our scripts, but might instead create them at runtime according to the contents of a configuration file.

For example, the file might simply give the string name of a stream class to be imported from a module, plus an optional constructor call argument. Factory-style functions or code might come in handy here because they would allow us to fetch and pass in classes that are not hardcoded in our program ahead of time. Indeed, those classes might not even have existed at all when we wrote our code:

```
classname = ...parse from config file...
classarg  = ...parse from config file...

import streamtypes                              # Customizable code
aclass = getattr(streamtypes, classname)        # Fetch from module
reader = factory(aclass, classarg)              # Or aclass(classarg)
processor(reader, ...)
```

Here, the `getattr` built-in is again used to fetch a module attribute given a string name (it's like saying ***`obj.attr`***, but ***`attr`*** is a string). Because this code snippet assumes a single constructor argument, it doesn't strictly need `factory` or `apply`—we could make an instance with just `aclass(classarg)`. They may prove more useful in the presence of unknown argument lists, however, and the general factory coding pattern can improve the code's flexibility.

Other Design-Related Topics

In this chapter, we've seen inheritance, composition, delegation, multiple inheritance, bound methods, and factories—all common patterns used to combine classes in Python programs. We've really only scratched the surface here in the design patterns domain, though. Elsewhere in this book you'll find coverage of other design-related topics, such as:

- *Abstract superclasses* (Chapter 28)
- *Decorators* (Chapters 31 and 38)
- *Type subclasses* (Chapter 31)
- *Static and class methods* (Chapter 31)
- *Managed attributes* (Chapter 37)
- *Metaclasses* (Chapters 31 and 39)

For more details on design patterns, though, we'll delegate to other resources on OOP at large. Although patterns are important in OOP work, and are often more natural in Python than other languages, they are not specific to Python itself.

Chapter Summary

In this chapter, we sampled common ways to use and combine classes to optimize their reusability and factoring benefits—what are usually considered design issues that are often independent of any particular programming language (though Python can make them easier to implement). We studied *delegation* (wrapping objects in proxy classes), *composition* (controlling embedded objects), and *inheritance* (acquiring behavior from other classes), as well as some more esoteric concepts such as pseudoprivate attributes, multiple inheritance, bound methods, and factories.

The next chapter ends our look at classes and OOP by surveying more advanced class-related topics; some of its material may be of more interest to tool writers than application programmers, but it still merits a review by most people who will do OOP in Python. First, though, another quick chapter quiz.

Test Your Knowledge: Quiz

1. What is multiple inheritance?
2. What is delegation?
3. What is composition?

4. What are bound methods?
5. What are pseudoprivate attributes used for?

Test Your Knowledge: Answers

1. Multiple inheritance occurs when a class inherits from more than one superclass; it's useful for mixing together multiple packages of class-based code. The left-to-right order in `class` statement headers determines the order of attribute searches.
2. Delegation involves wrapping an object in a proxy class, which adds extra behavior and passes other operations to the wrapped object. The proxy retains the interface of the wrapped object.
3. Composition is a technique whereby a controller class embeds and directs a number of objects, and provides an interface all its own; it's a way to build up larger structures with classes.
4. Bound methods combine an instance and a method function; you can call them without passing in an instance object explicitly because the original instance is still available.
5. Pseudoprivate attributes (whose names begin with two leading underscores: *__X*) are used to localize names to the enclosing class. This includes both class attributes like methods defined inside the class, and `self` instance attributes assigned inside the class. Such names are expanded to include the class name, which makes them unique.