CHAPTER 19

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, used them in expressions, called their methods, and so on. To qualify as being truly *object-oriented* (OO), though, objects generally need to also participate in something called an inheritance hierarchy.

This chapter begins the exploration of the Python *class*—a device used to implement new kinds of objects in Python. 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. In Python, classes are created with a new statement: the `class`. As we'll see, the objects defined with classes can look a lot like the built-in types we saw earlier in the book. They will also support inheritance—a mechanism of code customization and reuse, above and beyond anything we've seen so far.

One note up front: 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. But classes turn out to be one of the most useful tools Python provides, and we will show you why here. 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 we told you that programs do things with stuff? In simple terms, classes are just a way to define new sorts of stuff, which reflect real objects in your program's domain. In simple terms, classes are just a way to define new kinds of objects in your program's domain. For instance, suppose we've decided to implement that hypothetical pizza-making robot we used as an example in Chapter 12. If

we implement it using classes, we can model more of its real-world structure and relationships:

Inheritance
: Pizza-making robots are a kind of robot, and so 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 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 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 a Python program unit, just like functions and modules. They are another compartment for packaging logic and data. In fact, classes also define a new namespace 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 roughly 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; they are extended by redefining their 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 work on built-in types. For instance, objects made

with classes can be sliced, concatenated, indexed, and so on. Python provides hooks classes can use to intercept and implement any built-in type operation.

OOP from 30,000 Feet

Before we show what this all means in terms of code, we'd like to say a few words about the general ideas behind OOP here. If you've never done anything object-oriented in your life before now, some of the words we'll be using in this chapter may seem a bit perplexing on the first pass. Moreover, the motivation for using such words 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 all along in the book so far, 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, the expression kicks off a *search* in Python—it searches a tree of linked objects, for the first appearance of the *`attribute`* that it can find. In fact, when classes are involved, the Python expression above translates to the following in natural language:

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

In other words, attribute fetches are simply tree searches. We call this search procedure *inheritance*, because objects lower in a tree inherit attributes attached to objects higher in a tree, just because the attribute search proceeds from bottom to top in the tree. In a sense, the automatic search performed by inheritance means that 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 say *`object.attribute`*. To make this more concrete, Figure 19-1 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 *class objects* (the ovals: `C1`, `C2`, `C3`), and two *instance objects* (the rectangles: `I1`,

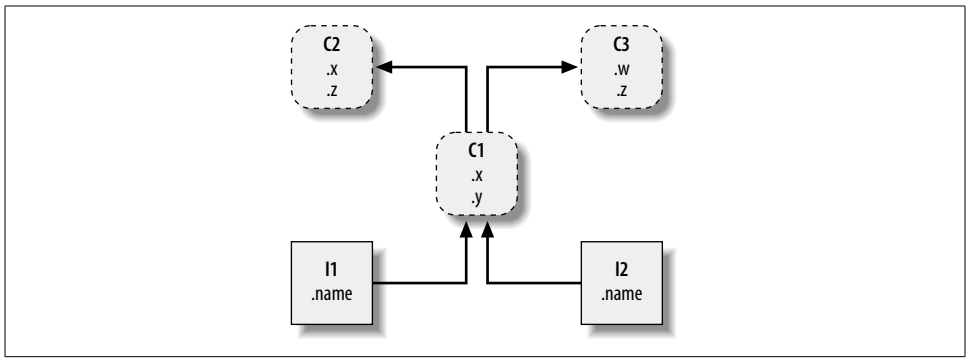

Figure 19-1. A class tree

I2), into an inheritance search tree. 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 employee 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 19-1, we can further categorize the ovals by their relative position 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 both to relative tree positions and roles. By virtue of inheritance search, superclasses provide behavior shared by all their subclasses. Because the search proceeds bottom-up, subclasses may override behavior defined in their superclasses by redefining superclass names lower in the tree.

Since these last few words are really the crux of the matter of software customization in OOP, let's expand on this concept. Suppose we've built up the tree in Figure 19-1, and then say this:

```
I2.w
```

Right away, we're doing inheritance here. Because this is one of those `object.attribute` expressions, it triggers a search of the tree in Figure 19-1. Python will

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

search for 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 it can't be found at all). For this expression, `w` won't be found until it searches `C3` as a last step, because it only appears in that object. In other words, `I2.w` resolves to `C3.w`, by virtue of the automatic search. In OOP terminology, `I2` "inherits" attribute `w` from `C3`. 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 more to the left than `C3`.
- `I2.name` finds `name` in `I2`, without climbing the tree at all.

Ultimately, the two instances inherit four attributes from their classes: `w`, `x`, `y`, and `z`. Trace these searches through the tree in Figure 19-1 to get a feel for how inheritance does its search in Python. The first in the list above is perhaps the most important to notice—because `C1` redefines attribute `x` lower in the tree, it effectively *replaces* the version above in `C2`. As you'll see in a moment, such redefinitions are at the heart of software customization in OOP.

All of the class and instance objects we put in these trees are just packages of names known as *namespaces*—places where we can attach attributes. If that sounds like modules, it should; the only major difference here is that objects in class trees also have automatically-searched links to other namespace objects.

Coding Class Trees

Although we are speaking in the abstract here, there is tangible code behind all these ideas. We construct such trees and their objects with `class` statements and class calls, which we'll meet in more detail. But 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 class they are created from.
- Classes are linked to their superclasses, by listing them in parenthesis in a `class` header line; the left-to-right order there gives the order in the tree.

To build the tree in Figure 19-1, for example, we would run Python code of this form (we've omitted the guts of the `class` statements here):

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

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

Here, we build the three class objects by running three `class` statements, and make the two instance objects by calling a class twice as though it were a function. The instances remember the class they were made from, and 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, the left-to-right order of superclasses listed in parenthesis in a `class` statement (like `C1`'s here) gives the order in which superclasses are searched, if there is more than one.

Because of the way inheritance searches, the object you attach an attribute to turns out to be crucial—it determines the name's scope. Attributes attached to instances only pertain to a single instance, 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.
- 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 automatically receives a special first argument—called `self` by convention—which 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

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

attributes of the particular instance being processed by a method call. In the code above, `self` is used to store a name on 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 a value. When methods assign to `self` attributes, they create or change an attribute in an instance at the bottom of the class tree (i.e., 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 names `C1` and `I1` are in your code's scopes.

If a class wants to guarantee that an attribute like `name` is always set in its instances, it more typically would 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'
```

If coded and inherited, a method named `__init__` is called automatically by Python 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 parenthesis in the class call go to arguments two and beyond. The effect here is to initialize instances when made, without requiring extra method calls.

The `__init__` method is known as a *constructor*, because of when it is run. It's the most commonly used representative of a larger class of methods called *operator overloading* methods. 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. They're run by Python automatically when objects appear in expressions, and are mostly an alternative to using simple method calls. They're also optional: if omitted, the operation is 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. There's a bit more to OOP in Python than inheritance; for example, operator overloading is much more general than described so far—classes may also provide the implementation of indexing, attribute fetches, 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 a module. However, the automatic attribute inheritance search that we get from classes, supports customization of software above and beyond 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 could 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 exiting code in-place. There is really no such concept with modules and functions.

Here's 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 that differs from the norm. You code subclasses that customize just the bits of behavior that differ per employee type; the rest of employee behavior will be inherited from the more general class. For example, if engineers have a unique salary computation rule (maybe it's not hours times rate), replace just that one method in a subclass:

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

Because the computeSalary version here is lower in the class tree, it will replace (override) the general version in Employee. Create instances of the kind of employee class that a real employee belongs to, to get the correct behavior. Notice that we 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 attribute search will begin:

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

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 we later ask for salaries, they will be computed according to the class the object was made from, due to inheritance search—yet another instance of the *polymorphism* idea for functions introduced in Chapter 12:*

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

Polymorphism means that the meaning of an operation depends on the object being operated on. Here, method computeSalary is located by inheritance 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
        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, the processor function can be reused 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(...))
```

* Note that the company list in this example could be stored on a file with Python object pickling, introduced later in this book, to yield a persistent employee database.

Moreover, the internal implementation of those `read` and `write` methods 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 embed readers and writers by composition (we'll see how later in this part of the book).

Once you get used to programming by *software customization* this way, 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 mixing together existing superclasses that already implement the behavior required by your program. For example, both the `Employee` and reader and writer classes in these examples may have already been written by someone else, for use in a completely different program. If so, you'll get all their code "for free."

In fact, in many application domains, you can fetch or purchase collections of superclasses, known as *frameworks*, which 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 awhile to learn how to leverage classes to achieve such OOP utopia. In practice, object-oriented work also entails substantial design 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 is so simple that it will not, by itself, pose an additional obstacle to your OOP quests. To see why, you'll have to move on to Chapter 20.