Chapter 8

Objects and Classes

The style of programming we have seen so far is called *procedural programming*. This was the first programming paradigm, developed in the 1950's and 1960's alongside the growth of hardware technology. In this chapter we look at *object-oriented programming*, a more modern style of coding that was developed in the 1970's and 1980's. The motivation for the development of object-oriented programming was the repeated failure of programmers to eliminate bugs from large programs written in the procedural style. Object-oriented programming is not more powerful than procedural programming any program that can be written in one style can be written in the other. However, in many situations it is easier to write correct programs in the object-oriented style. In Chapter 8 we will look at programs that use graphics, animation and simulations, and you will see that all of these lend themselves naturally to the object-oriented style.

8.1 Concepts

There is some terminology that you must understand to get started on writing object-oriented programs. The terminology is not difficult, but it is uses the building blocks we have already seen in a new way. Here are the basic pieces.

A *class* is a structure that holds both data and functions for manipulating that data. We have seen some classes before without calling them classes. For example, Python has a class called List. A list holds sequences of whatever data you give it, and it has functions, or *methods*, such as append( ) for manipulating that data. The class itself is a pattern. It says what kinds of data and what kinds of functionality instances of the class will have. You can construct specific instances of a class, which are called *objects*. For example, we might have a class Person. This would describe the data and functionality needed to represent people. Each object of this class would represent one individual person.

Like all data, the data in an object is stored in variables. There are two types of variables in classes: *instance variables* and*class variables*. Each object of a class has its own copy of each of the instance variables, so each object can have different data stored in the instance variables. For example, in our Person class there might be an instance variable to hold a persons name. Each object of the class will represent a different person, and so each will have its own name. The class variables are shared by all objects of the class; there is only one copy of the class variables, and it is visible to all objects of the class. Our Person class might have a class variable to hold the population size. The number of people in existence is the same regardless of which person we ask for this number. We refer to the instance variables of the class through objects of the class using a dot-notation: <object_name>.<variable_name>. For example, if x is an object of class Person we would refer to the name instance variable as x.name. We refer to the class variables through the class itself. Person.population might be the class variable that represent the size of the Person population.

Objects have their own functions for manipulating their data. We call the functions contained in an object its *methods*. If class Person has a GetOlder( ) method that adds one to the age of a person, and if x is an object of this class, we would tell object x to get older with the code

```
x.GetOlder( )
```

You might think of x.GetOlder() as a command to object x to increase its age variable.

As we said, each object of a class has its own copies of the instance variables and methods of the class. The class definition needs some way to refer to an individual object's data and methods. The word self is used for this. This word only appears in class definitions; it always refers to *the current object*. Every method of every class has self as its first argument. For example, the GetOlder( ) method has header

```
def GetOlder( self ):
```

When we call this method on a specific object, as in x.GetOlder( ), the object x is passed as the argument for self. We might have a method SetAge( ) that would set the persons age to a specific value. This would have header

```
def SetAge( self, myAge ):
```

and would be called as in x.SetAge(19). All methods have self as their first argument; you never explicitly pass a value for self, but the system substitutes the object whose method you are invoking for this argument.

The word self is used for similar reasons in references to instance variables within the methods of a class. self.age is the age variable for whatever object is being referenced. For example, here is the complete code for the SetAge( ) and GetOlder( ) methods:

```
def SetAge( self, myAge ):
        self.age = myAge

def GetOlder( self ):
        self.age = self.age + 1
```

If we call x.SetAge(19), then the object stored in variable x is passed for the self argument in method SetAge( ). Then self.age, which is the value of the age instance variable of object x, is set to 19. This notation probably seems cumbersome at first, but with a little practice it will begin to seem natural: x.foobar() runs method foobar() on object x.

Every class has a specific method called a *constructor* that is run when a new object of that class is created. The purpose of a constructor is to give initial values to the instance variables of the class. If the constructor has arguments, values for those arguments must be provided when a new object is constructed. Constructors in Python all have the name __init__( ). Of course, since constructors are methods they all have self as their first argument. For example, we might want the constructor for the Person class to give a name to the new person. The constructor that does this is

```
def __init__(self, myName ):
        self.name = myName
        self.age = 0
```

Remember that the constructor should initialize all of the instance variables of the class, even those that are not mentioned in the constructor's arguments. If we fail to initialize the variable then it doesn't exist. For example, if the constructor for class Person omits the line

```
self.age = 0
```

then new persons will not have an age variable. If other methods, such as a Print( ) method, refer to the age of the person and the age variable doesn't exist, our program will crash when these methods are called.

We construct objects of a class by using the class name as a function. The function that is actually called is the __init__( ) method of this class, so Python

expects us to provide all of the arguments (except self) of this method. For example, the __init__() method above for class Person needs a string for the name, so we could construct a new person with

```
x = Person( "bob" )
```

Here, finally, is the complete code for a simple program that uses classes. This defines the class Person that we have been discussing. The main() function of the program makes use of this class definition to construct several persons, set the age variables, and print out their information. Note that the main() function is simple and intuitive. This is typically the case with object-oriented programs the hard work in such programs is implementing the classes; once that is accomplished they are usually easy to use. Note also that classes can be used in more than one program. One of the goals of object-oriented programming is to make reusable classes that can be written carefully and then used in a wide range of applications. A programmer doesn't need to know the details of how a class is implemented in order to use it. All the programmer needs to know is what data the class holds and what methods it offers. The List and Dictionary classes in Python are illustrations of this. We have used these classes to write many programs without any knowledge of how the classes are implemented. These classes are very powerful. It is the object-oriented paradigm that allows us to separate the implementation of the classes from their use.

```
class Person:
    def __init__(self, myName):
        self.name = myName
        self.age = 0

    def SetAge(self, myAge):
        self.age = myAge

    def GetOlder(self):
        self.age = self.age + 1

    def Print(self):
        print("%s is %d years old."%(self.name,self.age))


def main():
    x = Person( "bob")
    y = Person( "suzie")
    z = Person( "joe")

    x.SetAge(57)
    x.GetOlder()
    x.GetOlder()
    x.Print()

    y.Print()

main()
```

Program 8.1.1: A complete object-oriented program