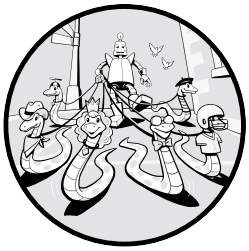
2

MODELING PHYSICAL OBJECTS WITH OBJECT-ORIENTED PROGRAMMING



In this chapter I'll introduce the general concepts behind object-oriented programming. I'll show a simple example program written using procedural programming, introduce classes as the basis of writing OOP code, and explain how the elements of a class work together. I'll then rewrite the first procedural example as a class in the object-oriented style, and show how you create an object from a class.

In the remainder of the chapter, I'll go through some increasingly complex classes that represent physical objects to demonstrate how OOP fixes the problems of procedural programming we ran into in Chapter 1. This should give you give you give a solid understanding of the underlying object-oriented concepts and how they can improve your programming skills.

Building Software Models of Physical Objects

To describe a physical object in our everyday world, we often reference its attributes. When talking about a desk, you might describe its color, dimensions, weight, material, and so on. Some objects have attributes that apply only to them and not others. A car could be described by its number of doors, but a shirt could not. A box could be sealed or open, empty or full, but those characteristics would not apply to a block of wood. Additionally, some objects are capable of performing actions. A car can go forward, back up, and turn left or right.

To model a real-world object in code, we need to decide what data will represent that object's attributes, and what operations it can perform. These two concepts are often referred to as an object's *state* and *behavior*, respectively: the state is the data that the object remembers and the behaviors are the actions that the object can do.

State and Behavior: Light Switch Example

Listing 2-1 is a software model of a standard two-position light switch written in procedural Python. This is a trivial example, but it will demonstrate state and behavior.

File: LightSwitch_Procedural.py

```
# Procedural light switch

❶ def turnOn():
      global switchIsOn
      # turn the light on
      switchIsOn = True

❷ def turnOff():
      global switchIsOn
      # turn the light off
      switchIsOn = False

# Main code
❸ switchIsOn = False  # a global Boolean variable

# Test code
print(switchIsOn)
turnOn()
print(switchIsOn)
turnOff()
print(switchIsOn)
turnOn()
print(switchIsOn)
```

Listing 2-1: Model of a light switch written with procedural code

The switch can only be in one of two positions: on or off. To model the state, we only need a single Boolean variable. We name this variable `switchIsOn` ❸ and say that `True` means on, and `False` indicates off. When the switch comes from the factory it is in the off position, so we initially set `switchIsOn` to `False`.

Next, we look at the behavior. This switch can only perform two actions: "turn on" and "turn off." We therefore built two functions, `turnOn()` ❶ and `turnOff()` ❷, which set the value of the single Boolean variable to `True` and `False`, respectively.

I've added some test code at the end to turn the switch on and off a few times. The output is exactly what we would expect:

```
False
True
False
True
```

This is an extremely simplistic example, but starting with small functions like these makes the transition to an OOP approach easier. As I explained in Chapter 1, because we've used a global variable to represent the state (in this case, the variable `switchIsOn`) this code will only work for a single light switch, but one of the main goals of writing functions is to make reusable code. I'll therefore rebuild the light switch code using object-oriented programming, but I need to work through a bit of the underlying theory first.

Introduction to Classes and Objects

The first step to understanding what an object is and how it works is to understand the relationship between a class and an object. I'll give formal definitions later, but for now, you can think of a class as a template or a blueprint that defines what an object will look like when one is created. We create objects from a class.

As an analogy, imagine if we started an on-demand cake baking business. Being "on-demand," we only create a cake when an order for one comes in. We specialize in Bundt cakes, and have spent a lot of time developing the cake pan in Figure 2-1 to make sure our cakes are not only tasty, but also beautiful and consistent.

The pan defines what a Bundt cake will look like when we create one, but it certainly is not a cake. The pan represents our class. When an order comes in, we create a Bundt cake from our pan. The cake is an object made using the cake pan.

Using the pan, we can create any number of cakes. Our cakes could have different attributes, like different flavors, different types of frosting, and optional extras like chocolate chips, but all the cakes come from the same cake pan.

Figure 2-1: A cake pan as a metaphor for a class

Figure 2-2: A cake as a metaphor for an object made from the cake pan class

Table 2-1 provides some other real-world examples to help clarify the relationship between a class and an object.

Table 2-1: Examples of real-world classes and objects

Class	Object made from the class
Blueprint for a house	House
Sandwich listed on a menu	Sandwich in your hand
Die used to manufacture a 25-cent coin	A single quarter
Manuscript of a book written by an author	Physical or electronic copy of the book

Classes, Objects, and Instantiation

Let's see how this works in code.

class Code that defines what an object will remember (its data or state) and the things that it will be able to do (its functions or behavior).

To get a feel for what a class looks like, here is the code of a light switch written as a class:

```
# OO_LightSwitch

class LightSwitch():
    def __init__(self):
        self.switchIsOn = False

    def turnOn(self):
        # turn the switch on
        self.switchIsOn = True

    def turnOff(self):
        # turn the switch off
        self.switchIsOn = False
```

We'll go through the details in just a bit, but the things to notice are that this code defines a single variable, `self.switchIsOn`, which is initialized in one function, and contains two other functions for the behaviors: `turnOn()` and `turnoff()`.

If you write the code of a class and try to run it, nothing happens, in the same way as when you run a Python program that consists of only functions and no function calls. You have to explicitly tell Python to make an object from the class.

To create a `LightSwitch` object from our `LightSwitch` class, we typically use a line like this:

```
oLightSwitch = LightSwitch()
```

This says: find the `LightSwitch` class, create a `LightSwitch` object from that class, and assign the resulting object to the variable `oLightSwitch`.

NOTE *As a naming convention in this book, I will generally use the prefix of a lowercase "o" to denote a variable that represents an object. This is not required, but it's a way to remind myself that the variable represents an object.*

Another word that you'll come across in OOP is *instance*. The words instance and object are essentially interchangeable; however, to be precise, we would say that a `LightSwitch` object is an instance of the `LightSwitch` class.

instantiation The process of creating an object from a class.

In the previous assignment statement, we went through the instantiation process to create a `LightSwitch` object from the `LightSwitch` class. We can also use this as a verb; we *instantiate* a `LightSwitch` object from the `LightSwitch` class.

Writing a Class in Python

Let's discuss the different parts of a class, and the details of instantiating and using an object. Listing 2-2 shows the general form of a class in Python.

```
class <ClassName>():

    def __init__(self, <optional param1>, ..., <optional paramN>):
        #  any initialization code here

    # Any number of functions that access the data
    # Each has the form:

    def <functionName1>(self, <optional param1>, ..., <optional paramN>):
        #  body of function

    #  ... more functions

    def <functionNameN>(self, <optional param1>, ..., <optional paramN>):
        #  body of function
```

Listing 2-2: The typical form of a class in Python

You begin a class definition with a `class` statement specifying the name you want to give the class. The convention for class names is to use camel case, with the first letter uppercase (for example, `LightSwitch`). Following the name you can optionally add a set of parentheses, but the statement must end with a colon to indicate that you're about to begin the body of the class. (I'll explain what can go inside the parentheses in Chapter 10, when we discuss inheritance.)

Within the body of the class, you can define any number of functions. All the functions are considered part of the class, and the code that defines

them must be indented. Each function represents some behavior that an object created from the class can perform. All functions must have at least one parameter, which by convention is named `self` (I'll explain what this name means in Chapter 3). OOP functions are given a special name: *method*.

method A function defined inside a class. A method always has at least one parameter, which is usually named `self`.

The first method in every class should have the special name `__init__`. Whenever you create an object from a class, this method will run automatically. Therefore, this method is the logical place to put any initialization code that you want to run whenever you instantiate an object from a class. The name `__init__` is reserved by Python for this very task, and must be written exactly this way, with two underscores before and after the word *init* (which must be lowercase). In reality, the `__init__()` method is not strictly required. However, it's generally considered good practice to include it and use it for initialization.

NOTE *When you instantiate an object from a class, Python takes care of constructing the object (allocating memory) for you. The special `__init__()` method is called the "initializer" method, where you give variables initial values. (Most other OOP languages require a method named `new()`, which is often referred to as a* constructor.*)*

Scope and Instance Variables

In procedural programming, there are two principal levels of scope: variables created in the main code have *global* scope and are available anywhere in a program, while variables created inside a function have *local* scope and only live as long as the function runs. When the function exits, all local variables (variables with local scope) literally go away.

Object-oriented programming and classes introduce a third level of scope, typically called *object scope*, though sometimes referred to as *class scope* or more rarely as *instance scope*. They all mean the same thing: the scope consists of all the code inside the class definition.

Methods can have both local variables and *instance variables*. In a method, any variable whose name does not start with `self.` is a local variable and will go away when that method exits, meaning other methods within the class can no longer use that variable. *Instance variables* have object scope, which means they are available to *all* methods defined in a class. Instance variables and object scope are the keys to understanding how objects remember data.

instance variable In a method, any variable whose name begins, by convention, with the prefix `self.` (for example, `self.x`). Instance variables have object scope.

Just like local and global variables, instance variables are created when they are first given a value, and do not need any special declaration. The `__init__()` method is the logical place to initialize instance variables. Here we have an example of a class where the `__init__()` method initializes an instance variable `self.count` (read as "self dot count") to `0` and another method, `increment()`, simply adds one to `self.count`:

```
class MyClass():
    def __init__(self):
        self.count = 0  #  create self.count and set it to 0
    def increment(self):
        self.count = self.count + 1  # increment the variable
```

When you instantiate an object from the `MyClass` class, the `__init__()` method runs and sets the value of the instance variable `self.count` to zero. If you then call the `increment()` method, the value of `self.count` goes from zero to one. If you call `increment()` again the value goes from one to two, and on and on.

Each object created from a class gets its own set of instance variables, independent of any other objects instantiated from that class. In the case of the `LightSwitch` class there is only one instance variable, `self.switchIsOn`, so every `LightSwitch` object will have its own `self.switchIsOn`. Therefore, you can have multiple `LightSwitch` objects, each with its own independent value of `True` or `False` for its `self.switchIsOn` variable.

Differences Between Functions and Methods

To recap, there are three key differences between a function and a method:

1. All methods of a class must be indented under the class statement.
2. All methods have a special first parameter that (by convention) is named `self`.
3. Methods in a class can use instance variables, written in the form `self.` *`<variableName>`*.

Now that you know what methods are, I'll show you how to create an object from a class and how to use the different methods that are available in a class.

Creating an Object from a Class

As I said earlier, a class simply defines what an object will look like. To use a class, you have to tell Python to make an object from the class. The typical way to do this is to use an assignment statement like this:

```
<object> = <ClassName>(<optional arguments>)
```

This single line of code invokes a sequence of steps that ends with Python handing you back a new instance of the class, which you typically store into a variable. That variable then refers to the resulting object.

THE INSTANTIATION PROCESS

Figure 2-3 shows the steps involved in instantiating a `LightSwitch` object from the `LightSwitch` class, going from the assignment statement into Python, then to the code of the class, then back out through Python again, and finally back to the assignment statement.

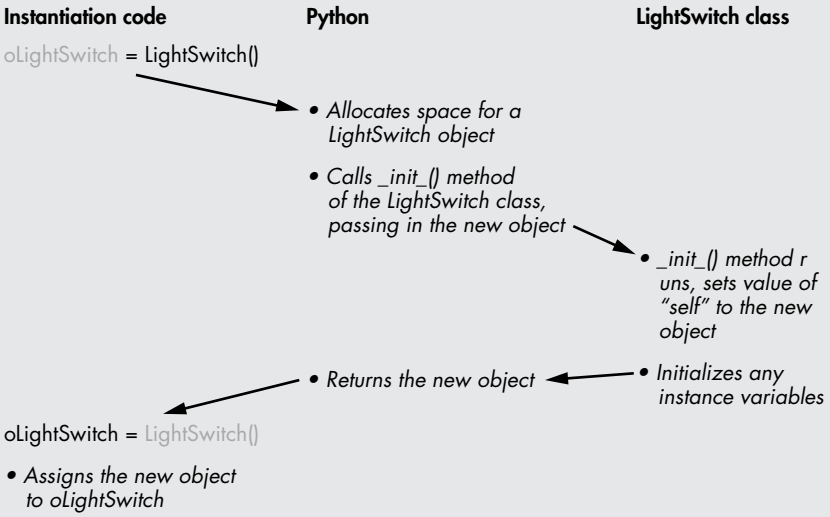


Figure 2-3: The process of instantiating an object

The process consists of five steps:

1. Our code asks Python to create an object from the `LightSwitch` class.
2. Python allocates space in memory for a `LightSwitch` object, then calls the `__init__()` method of the `LightSwitch` class, passing in the newly created object.
3. The `__init__()` method of the `LightSwitch` class runs. The new object is assigned to the parameter `self`. The code of `__init__()` initializes any instance variables in the object (in this case, the instance variable `self.switchIsOn`).
4. Python returns the new object to the original caller.
5. The result of the original call is assigned into the variable `oLightSwitch`, so it now represents the object.

You can make a class available in two ways: you can place the code of the class in the same file with the main program, or you can put the code of the class in an external file and use an `import` statement to bring in the contents of the file. I'll show the first approach in this chapter and the second

approach in Chapter 4. The only rule is that the class definition must precede any code that instantiates an object from the class.

Calling Methods of an Object

After creating an object from a class, to call a method of the object you use the generic syntax:

```
<object>.<methodName>(<any additional arguments>)
```

Listing 2-3 contains the `LightSwitch` class, code to instantiate an object from the class, and code to turn that `LightSwitch` object on and off by calling its `turnOn()` and `turnOff()` methods.

File: OO_LightSwitch_with_Test_Code.py

```
# OO_LightSwitch

class LightSwitch():
    def __init__(self):
        self.switchIsOn = False

    def turnOn(self):
        # turn the switch on
         self.switchIsOn = True

    def turnOff(self):
        # turn the switch off
         self.switchIsOn = False

    def show(self):  # added for testing
        print(self.switchIsOn)

# Main code
oLightSwitch = LightSwitch()  # create a LightSwitch object

#  Calls to methods
oLightSwitch.show()
oLightSwitch.turnOn()
oLightSwitch.show()
oLightSwitch.turnOff()
oLightSwitch.show()
oLightSwitch.turnOn()
oLightSwitch.show()
```

Listing 2-3: The `LightSwitch` class and test code to create an object and call its methods

First we create a `LightSwitch` object and assign it to the variable `oLightSwitch`. We then use that variable to call other methods available in the `LightSwitch` class. We would read these lines as "`oLightSwitch` dot `show`," "`oLightSwitch` dot `turnOn`," and so on. If we run this code, it will output:

```
False
True
False
True
```

Recall that in this class there is a single instance variable named `self.switchIsOn`, but its value is remembered and easily accessed when different methods of the same object run.

Creating Multiple Instances from the Same Class

One of the key features of OOP is that you can instantiate as many objects as you want from a single class, in the same way that you can make endless cakes from a Bundt cake pan.

So, if you want two light switch objects, or three, or more, you can just create additional objects from the `LightSwitch` class like so:

```
oLightSwitch1 = LightSwitch()  # create a light switch object
oLightSwitch2 = LightSwitch()  # create another light switch object
```

The important point here is that each object that you create from a class maintains *its own version* of the data. In this case, `oLightSwitch1` and `oLightSwitch2` each have their own instance variable, `self.switchIsOn`. Any changes you make to the data of one object will not affect the data of another object. You can call any of the methods in the class with either object.

The example in Listing 2-4 creates two light switch objects, and calls methods on the different objects.

File: OO_LightSwitch_Two_Instances.py

```
# OO_LightSwitch

class LightSwitch():
--- snipped code of LightSwitch class, as in Listing 2-3 ---

# Main code
oLightSwitch1 = LightSwitch()  # create a LightSwitch object
oLightSwitch2 = LightSwitch()  # create another LightSwitch object

#  Test code
oLightSwitch1.show()
oLightSwitch2.show()
oLightSwitch1.turnOn() # Turn switch 1 on
# Switch 2 should be off at start, but this makes it clearer
oLightSwitch2.turnOff()
oLightSwitch1.show()
oLightSwitch2.show()
```

Listing 2-4: Create two instances of a class and call methods of each

Here's the output when this program is run:

```
False
False
True
False
```

The code tells `oLightSwitch1` to turn itself on and tells `oLightSwitch2` to turn itself off. Notice that the code in the class has no global variables. Each `LightSwitch` object gets its own set of any instance variables (just one in this case) defined in the class.

While this may not seem like a huge improvement over two simple global variables which could be used to do the same thing, the implications of this technique are enormous. You'll get a better sense of this in Chapter 4, where I'll discuss how to create and maintain a large number of instances made from a class.

Python Data Types Are Implemented as Classes

It might not surprise you that all built-in data types in Python are implemented as classes. Here is a simple example:

```
>>> myString = 'abcde'
>>> print(type(myString))
<class 'str'>
```

We assign a string value to a variable. When we call the `type()` function and print the results, we see that we have an instance of the `str` string class. The `str` class gives us a number of methods we can call with strings, including `myString.upper()`, `myString.lower()`, `myString.strip()`, and so on.

Lists work in a similar way:

```
>>> myList = [10, 20, 30, 40]
>>> print(type(myList))
<class 'list'>
```

All lists are instances of the `list` class, which has many methods including `myList.append()`, `myList.count()`, `myList.index()`, and so on.

When you write a class, you are defining a new data type. Your code provides the details by defining what data it maintains and what operations it can perform. After creating an instance of your class and assigning it to a variable, you can use the `type()` built-in function to determine the class used to create it, just like with a built-in data type. Here we instantiate a `LightSwitch` object and print out its data type:

```
>>> oLightSwitch = LightSwitch()
>>> print(type(oLightSwitch))
<class 'LightSwitch'>
```

Just like with Python's built-in data types work, we can then use the variable `oLightSwitch` to call the methods available in the `oLightSwitch` class.

Definition of an Object

To summarize this section, I'll give my formal definition of an object.

object	Data, plus code that acts on that data, over time.

A class defines what an object will look like when you instantiate one. An object is a set of instance variables and the code of the methods in the class from which the object was instantiated. Any number of objects can be instantiated from a class, and each has its own set of instance variables. When you call a method of an object, the method runs and uses the set of instance variables in that object.

Building a Slightly More Complicated Class

Let's build on the concepts introduced so far and work through a second, slightly more complicated example in which we'll make a dimmer switch class. A dimmer switch has an on/off switch, but it also has a multi-position slider that affects the brightness of the light.

The slider can move through a range of brightness values. To make things straightforward, our dimmer digital slider has 11 positions, from 0 (completely off) through 10 (completely on). To raise or lower the brightness of the bulb to the maximum extent, you must move the slider through every possible setting.

This `DimmerSwitch` class has more functionality than our `LightSwitch` class and needs to remember more data—namely:

- The switch state (on or off)
- Brightness level (0 to 10)

And here are the behaviors a `DimmerSwitch` object can perform:

- Turn on
- Turn off
- Raise level
- Lower level
- Show (for debugging)

The `DimmerSwitch` class uses the standard template shown earlier in Listing 2-2: it starts with a `class` statement and a first method named `__init__()`, then defines a number of additional methods, one for each of the behaviors listed. The full code for this class is presented in Listing 2-5.

File: DimmerSwitch.py

```
# DimmerSwitch class

class DimmerSwitch():
    def __init__(self):
        self.switchIsOn = False
        self.brightness = 0

    def turnOn(self):
        self.switchIsOn = True

    def turnOff(self):
        self.switchIsOn = False

    def raiseLevel(self):
        if self.brightness < 10:
            self.brightness = self.brightness + 1

    def lowerLevel(self):
        if self.brightness > 0:
            self.brightness = self.brightness - 1

    # Extra method for debugging
    def show(self):
        print(Switch is on?', self.switchIsOn)
        print('Brightness is:', self.brightness)
```

Listing 2-5: The slightly more complicated `DimmerSwitch` class

In this `__init__()` method we have two instance variables: the familiar `self.switchIsOn` and a new one, `self.brightness`, which remembers the brightness level. We assign starting values to both instance variables. All other methods can access the current value of each of these. In addition to `turnOn()` and `turnOff()`, we also include two new methods for this class: `raiseLevel()` and `lowerLevel()`, which do exactly what their names imply. The `show()` method is used during development and debugging and just prints the current values of the instance variables.

The main code in Listing 2-6 tests our class by creating a `DimmerSwitch` object (`oDimmer`), then calling the various methods.

File: OO_DimmerSwitch_with_Test_Code.py

```
# DimmerSwitch class with test code

class DimmerSwitch():
--- snipped code of DimmerSwitch class, as in Listing 2-5 ---

# Main code
oDimmer = DimmerSwitch()

# Turn switch on, and raise the level 5 times
```

```
oDimmer.turnOn()
oDimmer.raiseLevel()
oDimmer.raiseLevel()
oDimmer.raiseLevel()
oDimmer.raiseLevel()
oDimmer.raiseLevel()
oDimmer.show()

# Lower the level 2 times, and turn switch off
oDimmer.lowerLevel()
oDimmer.lowerLevel()
oDimmer.turnOff()
oDimmer.show()

# Turn switch on, and raise the level 3 times
oDimmer.turnOn()
oDimmer.raiseLevel()
oDimmer.raiseLevel()
oDimmer.raiseLevel()
oDimmer.show()
```

Listing 2-6: `DimmerSwitch` class with test code

When we run this code, the resulting output is:

```
Switch is on? True
Brightness is: 5
Switch is on? False
Brightness is: 3
Switch is on? True
Brightness is: 6
```

The main code creates the `oDimmer` object, then makes calls to the various methods. Each time we call the `show()` method, the on/off state and the brightness level are printed. The key thing to remember here is that `oDimmer` represents an object. It allows access to all methods in the class from which it was instantiated (the `DimmerSwitch` class), *and* it has a set of all instance variables defined in the class (`self.switchIsOn` and `self.brightness`). Again, instance variables maintain their values between calls to methods of an object, so the `self.brightness` instance variable is incremented by one for each call to `oDimmer.raiseLevel()`.

Representing a More Complicated Physical Object as a Class

Let's consider a more complicated physical object: a television. With this more complicated example, we'll take a closer look at how arguments work in classes.

A television requires much more data than a light switch to represent its state, and has more behaviors. To create a `TV` class, we must consider how a user would typically use a TV and what the TV would have to remember. Let's look at some of the important buttons on a typical TV remote (Figure 2-4).

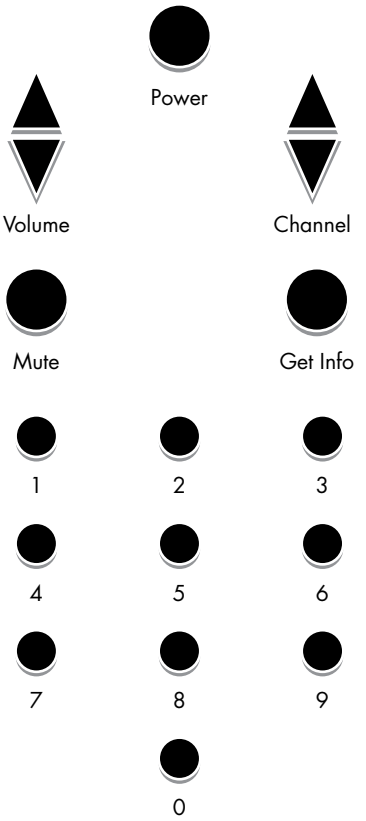


Figure 2-4: A simplified TV remote

From this we can determine that to keep track of its state, a `TV` class would have to maintain the following data:

- Power state (on or off)
- Mute state (is it muted?)
- List of channels available
- Current channel setting
- Current volume setting
- Range of volume levels available

And the actions that the TV must provide include:

- Turn the power on and off
- Raise and lower the volume
- Change the channel up and down
- Mute and unmute the sound
- Get information about the current settings
- Go to a specified channel

The code for our TV class is shown in Listing 2-7. We include the __init__() method for initialization, followed by a method for each of the behaviors.

File: TV.py

```
# TV class

class TV():
    def __init__(self): ❶
        self.isOn = False
        self.isMuted = False
        # Some default list of channels
        self.channelList = [2, 4, 5, 7, 9, 11, 20, 36, 44, 54, 65]
        self.nChannels = len(self.channelList)
        self.channelIndex = 0
        self.VOLUME_MINIMUM = 0   # constant
        self.VOLUME_MAXIMUM = 10  # constant
        self.volume = self.VOLUME_MAXIMUM //   # integer divide

    def power(self): ❷
        self.isOn = not self.isOn   # toggle

    def volumeUp(self):
        if not self.isOn:
            return
        if self.isMuted:
            self.isMuted = False  # changing the volume while muted unmutes the sound
        if self.volume < self.VOLUME_MAXIMUM:
            self.volume = self.volume + 1

    def volumeDown(self):
        if not self.isOn:
            return
        if self.isMuted:
            self.isMuted = False  # changing the volume while muted unmutes the sound
        if self.volume > self.VOLUME_MINIMUM:
            self.volume = self.volume - 1

    def channelUp(self): ❸
        if not self.isOn:
            return
        self.channelIndex = self.channelIndex + 1
        if self.channelIndex > self.nChannels:
            self.channelIndex = 0  # wrap around to the first channel

    def channelDown(self): ❹
        if not self.isOn:
            return
        self.channelIndex = self.channelIndex - 1
        if self.channelIndex < 0:
            self.channelIndex = self.nChannels - 1  # wrap around to the top channel

    def mute(self):
        if not self.isOn:
```

```
            return
        self.isMuted = not self.isMuted

    def setChannel(self, newChannel):
        if newChannel in self.channelList:
            self.channelIndex = self.channelList.index(newChannel)
        # if the newChannel is not in our list of channels, don't do anything

    def showInfo(self): ❺
        print()
        print('TV Status:')
        if self.isOn:
            print('    TV is: On')
            print('    Channel is:', self.channelList[self.channelIndex])
            if self.isMuted:
                print('    Volume is:', self.volume, '(sound is muted)')
            else:
                print('    Volume is:', self.volume)
        else:
            print('    TV is: Off')
```

Listing 2-7: The TV class with many instance variables and methods

The `__init__()` method ❶ creates all the instance variables used in all the methods, and assigns reasonable starting values to each. Technically, you can create an instance variable inside any method; however, it is a good programming practice to define all instance variables in the `__init__()` method. This avoids the risk of an error when attempting to use an instance variable in a method before it's been defined.

The `power()` method ❷ represents what happens when you push the power button on a remote. If the TV is off, pushing the power button turns it on; if the TV is on, pushing the power button turns it off. To code this behavior I've used a *toggle*, which is a Boolean that's used to represent one of two states and can easily be switched between them. With this toggle, the `not` operator switches the `self.isOn` variable from `True` to `False`, or from `False` to `True`. The code of the `mute()` method does a similar thing with the `self.muted` variable toggling between muted and not-muted, but first has to check that the TV is on. If the TV is off, calling the `mute()` method has no effect.

One interesting thing to note is that we don't really keep track of the current channel. Instead, we keep track of the *index* of the current channel, which allows us to get the current channel at any time by using `self.channelList[self.channelIndex]`.

The `channelUp()` ❸ and `channelDown()` ❹ methods basically increment and decrement the channel index, but there is also some clever code in them to allow for wrap-around. If you're currently at the last index in the channel list and the user asks to go to the next channel up, the TV goes to the first channel in the list. If you're at the first index in the channel list and the user asks to go the next channel down, the TV goes to the last channel in the list.

The `showInfo()` method ❺ prints out the current status of the TV based on the values of the instance variables (on/off, current channel, current volume setting, and mute setting).

In Listing 2-8, we'll create a TV object and call methods of that object.

File: OO_TV_with_Test_Code.py

```
# TV class with test code

--- snipped code of TV class, as in Listing 2-7 ---

# Main code
oTV = TV()  # create the TV object

# Turn the TV on and show the status
oTV.power()
oTV.showInfo()

# Change the channel up twice, raise the volume twice, show status
oTV.channelUp()
oTV.channelUp()
oTV.volumeUp()
oTV.volumeUp()
oTV.showInfo()

# Turn the TV off, show status, turn the TV on, show status
oTV.power()
oTV.showInfo()
oTV.power()
oTV.showInfo()

# Lower the volume, mute the sound, show status
oTV.volumeDown()
oTV.mute()
oTV.showInfo()

# Change the channel to 11
oTV.setChannel(11)
oTV.mute()
oTV.showInfo()
```

Listing 2-8: TV class with test code

When we run this code, here is what we get as output:

```
TV Status:
    TV is: On
    Channel is: 2
    Volume is: 5

TV Status:
    TV is: On
    Channel is: 5
    Volume is: 7

TV Status:
    TV is: Off
```

```
TV Status:
    TV is: On
    Channel is: 5
    Volume is: 7

TV Status:
    TV is: On
    Channel is: 5
    Volume is: 6 (sound is muted)

TV Status:
    TV is: On
    Channel is: 11
    Volume is: 6
```

All of the methods are working correctly, and we get the expected output.

Passing Arguments to a Method

When calling any function, the number of arguments must match the number of parameters listed in the matching `def` statement:

```
def myFunction(param1, param2, param3):
    # body of function

# call to a function:
myFunction(argument1, argument2, argument3)
```

The same rule applies with methods and method calls. However, you may notice that whenever we make a call to a method, it appears that we are specifying one less argument than the number of parameters. For example, the definition of the `power()` method in our `TV` class looks like this:

```
def power(self):
```

This implies that the `power()` method is expecting one value to be passed in, and whatever is passed in will be assigned to the variable `self`. Yet when we started by turning on the TV in Listing 2-8, we made this call:

```
oTV.power()
```

When we make the call, we don't explicitly pass anything inside the parentheses.

This may seem even stranger in the case of the `setChannel()` method. The method is written to accept two parameters:

```
def setchannel(self, newchannel):
    if newChannel in self.channelList:
        self.channelIndex = self.channelList.index(newChannel)
```

But we called `setChannel()` like this:

```
oTV.setChannel(11)
```

It appears that only one value is being passed in.

You might expect Python to generate an error here, due to a mismatch in the number arguments (1) and the number of parameters (2). In practice, Python is doing a bit of behind-the-scenes work to make the syntax easier to follow.

Let's examine this. Earlier, I said that to make a call to a method of an object, you use the following generic syntax:

```
<object>.<method>(<any additional arguments>)
```

Python takes the `<object>` you specify in the call and rearranges it to become the first argument. Any values in the parentheses of the method call are considered to be the subsequent argument(s). Thus, Python makes it appear that you wrote this instead:

```
<method of object>(<object>, <any additional arguments>)
```

Figure 2-5 shows how this works in our example code, again using the `setChannel()` method of the `TV` class.

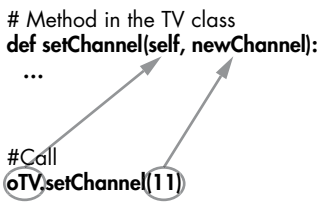


Figure 2-5: Calling a method

Although it looks like we're only providing one argument here (for `newChannel`), there are really two arguments passed in—`oTV` and `11`—and the method provides two parameters to receive these values (`self` and `newChannel`, respectively). Python rearranges the arguments for us when the call is made. This may seem odd at first, but it will become second nature very quickly. Writing the call with the object first makes it much easier for a programmer to see which object is being acted on.

This is a subtle but important feature. Remember that the object (in this case, `oTV`) keeps the current settings of all of its instance variables. Passing the object as the first argument allows the method to run with the values of the instance variables of that object.

Multiple Instances

Every method is written with `self` as the first parameter, so the `self` variable receives the object used in each call. This has a major implication: it allows

any method within a class to work with *different* objects. I'll explain how this works using an example.

In Listing 2-9 we'll create two TV objects and save them in two variables, oTV1 and oTV2. Each TV object has a volume setting, a channel list, a channel setting, and so on. We'll make calls to a number of different methods of the different objects. At the end, we'll call the showInfo() method on each TV object to see the resulting settings.

File: OO_TV_TwoInstances.py

```
# Two TV objects with calls to their methods
class TV():
--- snipped code of TV class, as in Listing 2-7 ---
# Main code
oTV1 = TV()  # create one TV object
oTV2 = TV()  # create another TV object

# Turn both TVs on
oTV1.power()
oTV2.power()

# Raise the volume of TV1
oTV1.volumeUp()
oTV1.volumeUp()

# Raise the volume of TV2
oTV2.volumeUp()
oTV2.volumeUp()
oTV2.volumeUp()
oTV2.volumeUp()
oTV2.volumeUp()

# Change TV2's channel, then mute it
oTV2.setChannel(44)
oTV2.mute()

# Now display both TVs
oTV1.showInfo()
oTV2.showInfo()
```

Listing 2-9: Creating two instances of the TV class and calling methods of each

If we run this code, it will generate the following output:

```
Status of TV:
    TV is: On
    Channel is: 2
    Volume is: 7

Status of TV:
    TV is: On
    Channel is: 44
    Volume is: 10 (sound is muted)
```

Each TV object maintains its own set of the instance variables defined in the class. This way, each TV object's instance variables can be manipulated independently of any other TV object.

Initialization Parameters

The ability to pass arguments to method calls also works when instantiating an object. So far, when we've created our objects, we've always set their instance variables to constant values. However, you'll often want to create different instances with different starting values. For example, imagine we want to instantiate different TVs and identify them using their brand name and location. This way, we can differentiate between a Samsung television in the family room and a Sony television in the bedroom. Constant values would not work for us in this situation.

To initialize an object with different values, we add parameters to the definition of the __init__() method, like this:

```
# TV class

class TV():
    def __init__(self, brand, location):  # pass in a brand and location for the TV
        self.brand = brand
        self.location = location
        --- snipped remaining initialization of TV ---
        ...
```

In all methods, parameters are local variables, so they literally go away when the method ends. For example, in the __init__() method of the TV class shown here, brand and location are local variables that will disappear when the method ends. However, we often want to save values that are passed in via parameters to use them in other methods.

In order to allow an object to remember initial values, the standard approach is to store any values passed in into instance variables. Since instance variables have object scope, they can be used in other methods in the class. The Python convention is that the name of the instance variable should be the same as the parameter name, but prefixed with self and a period, as in:

```
def __init__(self, someVariableName):
    self.someVariableName = someVariableName
```

In the TV class, the line after the def statement tells Python to take the value of the brand parameter and assign it to an instance variable named self.brand. The next line does the same thing with the location parameter and the instance variable self.location. After these assignments, we can use self.brand and self.location in other methods.

Using this approach, we can create multiple objects from the same class, but start each off with different data. So, we can create our two TV objects like this:

```
oTV1 = TV('Samsung', 'Family room')
oTV2 = TV('Sony', 'Bedroom')
```

When executing the first line, Python first allocates space for a TV object. Then it rearranges the arguments as discussed in the previous section, and calls the __init__() method of the TV class with three arguments: the newly allocated oTV1 object, the brand, and the location.

When initializing the oTV1 object, self.brand is set to the string 'Samsung' and self.location is set to the string 'Family room'. When initializing oTV2, its self.brand is set to the string 'Sony', and its self.location gets set to the string 'Bedroom'.

If we modify the showInfo() method to report the name and location of the TV:

File: OO_TV_TwoInstances_with_Init_Params.py

```
def showInfo(self):
        print()
        print('Status of TV:', self.brand)
        print('    Location:', self.location)
        if self.isOn:
        ...
```

we'll see this as output:

```
Status of TV: Sony
    Location: Family room
    TV is: On
    Channel is: 2
    Volume is: 7

Status of TV: Samsung
    Location: Bedroom
    TV is: On
    Channel is: 44
    Volume is: 10 (sound is muted)
```

We made the same method calls as in the previous example in Listing 2-9. The difference is that each TV object is now initialized with a brand and a location, and you can now see that information printed in response to each call to the modified showInfo() method.

Classes in Use

Using everything we've learned in this chapter, we can now create classes and build multiple independent instances from those classes. Here are a few examples of how we might use this:

- Say we wanted to model a student in a course. We could have a `Student` class that has instance variables for `name`, `emailAddress`, `currentGrade`, and so on. Each `Student` object we create from this class would have its own set of these instance variables, and the values given to the instance variables would be different for each student.
- Consider a game where we have multiple players. A player could be modeled by a `Player` class with instance variables for `name`, `points`, `health`, `location`, and so on. Each player would have the same capabilities, but the methods could work differently based on the different values in the instance variables.
- Imagine an address book. We could create a `Person` class with instance variables for `name`, `address`, `phoneNumber`, and `birthday`. We could create as many objects from the `Person` class as we want, one for each person we know. The instance variables in each `Person` object would contain different values. We could then write code to search through all the `Person` objects and retrieve information about the one or ones we are looking for.

In future chapters, I will explore this concept of instantiating multiple objects from a single class, and give you tools to help manage a collection of objects.

OOP as a Solution

Toward the end of Chapter 1, I mentioned three problems that are inherent in procedural coding. Hopefully, after working through the examples in this chapter you can now see how object-oriented programming solves all of those problems:

1. A well-written class can be easily reused in many different programs. Classes do not need to access global data. Instead, objects provide code and data at the same level.
2. Object-oriented programming can greatly reduce the number of global variables required because a class provides a framework in which data and code that acts on the data exist in one grouping. This also tends to make code easier to debug.
3. Objects created from a class only have access to their own data—their set of the instance variables in the class. Even when you have multiple objects created from the same class, they do not have access to each other's data.

Summary

In this chapter I provided an introduction to object-oriented programming by demonstrating the relationship between a class and an object. The class defines the shape and capabilities of an object. An object is a single instance of a class that has its own set of all the data defined in the instance variables of the class. Each piece of data you want an object to contain is stored in an instance variable, which has object scope, meaning that it is available within all methods defined in the class. All objects created from the same class get their own set of all the instance variables, and because these may contain different values, calling the methods on different objects can result in different behavior.

I showed how you create an object from a class, typically through an assignment statement. After instantiating an object, you can use it to make calls to any method defined in the class of that object. I also showed how you can instantiate multiple objects from the same class.

In this chapter, the demonstration classes implemented physical objects (light switches, TVs). This is a good way to start understanding the concepts of a class and an object. However, in future chapters I will introduce objects that do not represent physical objects.