UNIT 5

OBJECT-ORIENTED PROGRAMMING

CHAPTER
5.1

Objects

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Describe the use of objects and recognize the difference between an object and an instance;
- Explain the principles of object-oriented programming;
- Define objects in Python and use principles of object-oriented programming for declaring methods and combining classes.

1. What Are Objects?

At this point, we've covered the basics of computing. Congratulations! So far, you've covered the principles and methods that put a man on the moon. The techniques you now know covered the state of the art of computing for its first several decades of existence.

Surprisingly, though, the principles needed to put a man on the moon were far simpler than the principles needed to load up that gif of a cat falling down on your phone. Modern cloud computing, virtual reality, and even typical web development require more complex frameworks and paradigms than the ones responsible for the early days of the space program.

Unit 5 is our "advanced topics" unit. In this unit, we're going to preview the next topics you'll cover if you decide to continue your education in computing. We're going to focus on two topics: objects and algorithms. In many ways, these two topics cover two different directions you could choose to go in computing. If you want to go into developing websites, video games, or other portions of the design side of computing, you'll be using objects a lot. If you want to go into machine learning, theory, or the more mathematical side of computing, you'll spend a lot of your time developing algorithms.

What Are Objects?

That brings us to the topic for this chapter. What *are* **objects** ? Surprise! You've actually been interacting with objects throughout the past several chapters; we've just glossed over them and promised to come back to them later. Well, later is now.

Objects are custom data types that you get to create. We usually refer to the data types themselves as **Classes** . Just like you've been using data types to represent numbers, letters, and strings, you can create data types or classes to represent people, places, and items. In creating these, you specify multiple variables to be wrapped up into one data type.

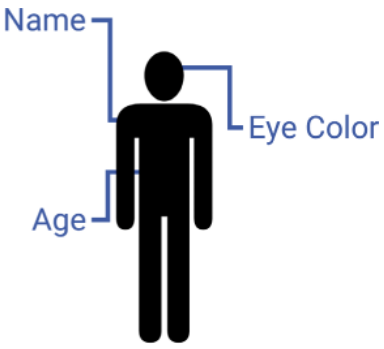
One of the biggest strengths of programming with objects is that it lets us write programs the way we think about things in the real world. We're naturally predisposed to think about the world in terms of generic objects. For example, you have the concept in you of a person. A person would be a single entity that has lots of variables assigned to them. They have a first name, a last name, and a middle name. They have a height, a hair color, and an eye color. They have a phone number and an e-mail address.

These are all variables that we could wrap up into one data type, and call that data type a `Person` class. The class tells us what variables should be specified for that type of object. A `Person` object would almost certainly have a first name and a last name, and that's the reason why we know that asking, "What's your name?" is a logical question to ask a new person. We know they should have a name because these are variables in that type. By that same principle, if we had a `Chair` object, we would find it very silly to ask, "What's your name?" to a chair because "name" probably isn't a variable we'd associate with a chair type.

Objects vs. Instances

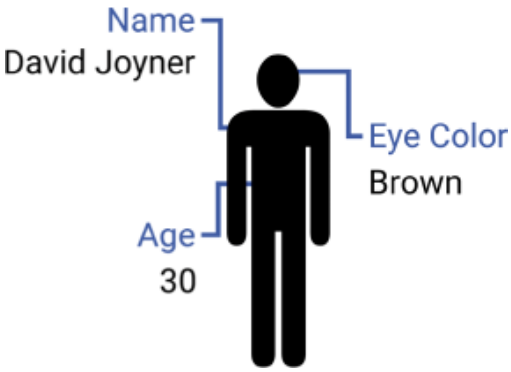
So, a class is a generic structure for a certain kind of data. If we have a `Person` class, we expect it to have a name, a height, and an eye color. These are variables we expect to exist about people.

Page 256



An **instance** , on the other hand, is a *specific* person. A `Person` class is a data type with variables for name, age, and eye color. David Joyner is an *instance* of a `Person` object with name "David Joyner", age 30, and eye color brown. "David Joyner", 30, and brown are values for these variables, just as David Joyner is an instance of an object of type `Person`. It might seem weird to differentiate the name "David Joyner" from the `Person` David Joyner, but ask yourself: if I change my name, am I suddenly a different person? No; having a name is a variable, my actual name is that variable's value, but even if I change that variable's value, the person is the same. It's just difficult to refer to people without using their names! Think about this with something like chairs, though: a chair has a serial number, which is an unchanging name of that specific chair, differentiating it from other chairs of the same type. Even if we remove a leg and change the color, the serial number remains the same.

Classes are the general description of the types of variables associated with the type. An instance of the class is a particular example of an object of that type. You know what a person is, and you know that David Joyner is an example—an instance—of a person. Similarly, you know what a chair is in general, and the chair you're probably sitting on right now is an instance of a chair. You know that chairs can have lots of variables, like a number of legs, a color, and a material; similarly, you know that the chair you're sitting on is white, wooden, and has four legs. You know that not all chairs are white with four legs, just as you know that not all people are named "David Joyner". The general concepts are classes, and the specific examples are instances of these classes.



2. Objects and Instances in Python

To get started, let's talk just about how to define classes and instances in Python. We've actually already seen these in some places, but we didn't refer to them by these terms. For this lesson, let's stick to the running `Person` example. A `Person` should have a first name, a last name, an eye color, and an age. We'll represent the names and eye color as strings and the age as a number.

Declaring a Class

Figure 5.1.1 shows how we would define our `Person` class. Line 2 here is the line that starts off the creation of a new object. This is similar to what we've seen in the past with loops and conditionals: the reserved word `class` tells the computer that it's about to see a code block contained within the class, after the class name (Person) and a colon as usual. The contents of the class is indented.

#	DeclaringaClass-1.py	Output
1	`#Define the class Person`	
2	`class Person:`	
3	`    #Create a new instance of Person`	
4	`    def __init__(self):`	
5	`        #Person's default values`	
6	`        self.firstname = "[no first name]"`	
7	`        self.lastname = "[no last name]"`	
8	`        self.eyecolor = "[no eye color]"`	
9	`        self.age = -1`	
10		

Figure 5.1.1

What do we see next? On line 4, we see… a function! A function with the strange name `__init__` and the strange parameter `self`. We'll talk more about `__init__` later, but for now, what you need to know is that `__init__` is called when we first create a new instance of this class. Think back to when we would define strings: The string class actually had an `__init__` method that was called whenever we created a new string. It just executes some code that will be needed for the rest of the program. Here, when we create a new `Person`, we want to give some default values to the names and eye color that emphasize that the real values haven't yet been supplied.

When a function is defined *inside* a class, we call it a **method** . It still works the same way: when the method is called, the lines of code are executed in order. In the `__init__()` method here in **Figure 5.1.1** , there are four lines. Notice that each line starts with the variable name `self`. `self` is a little strange: it tells Python to define the following variable (like `firstname`) for the instance as a whole. If we leave off `self`, the variable has the same scope as a variable normally would in a function: it stops existing when the function is over. So, writing `self.firstname` says, "Any time we look at this instance's `firstname`, it should be the same one!" Every method declared inside a class should have `self` as the

first parameter, and every variable for the class should be preceded by `self` every time the variable is used inside the class. `self` is a little like saying 'my'; it collects together the class's variables.

Defining the `Person` class works just like defining the functions we've seen in the past. Seeing the line `class Person` tells the computer, "Hey, you need to know this concept of a `Person`." Later on, we can actually use this concept in our code. Think of declaring a class like teaching someone a concept. If you knew someone wasn't familiar with books, you would teach them that every book has a title and an author; then when you give them a book, they would know to look for the title and author. Teaching them the idea of a book having a title and author is like declaring a class; handing them a copy of *Introduction to Computing* by David Joyner is like having them create an instance.

Once defined, though, classes work like any data type. You can use instances of them as values in lists or tuples. If some of their variables are lists, you can loop over these lists. You can even use classes as values for *other* classes. For example, we could create a `Name` class that has two variables, `firstname` and `lastname`, as shown in **Figure 5.1.2**.

#	DeclaringaClass-2.py	Output
1	`#Define the class Name`	
2	`class Name:`	
3	`    def __init__(self):`	
4	`        self.firstname = "[no first name]"`	
5	`        self.lastname = "[no last name]"`	
6		
7	`#Define the class Person`	
8	`class Person:`	
9	`    #Create a new instance of Person`	
10	`    def __init__(self):`	
11	`        #Person's default values`	
12	`        self.name = Name()`	
13	`        self.eyecolor = "[no eye color]"`	
14	`        self.age = -1`	
15		

Figure 5.1.2

In **Figure 5.1.2**, an instance of the `Name` class supplies the `firstname` and `lastname` to the `Person` class. We can use this to create extremely complex data structures that are nonetheless easy to use because everything is organized in logical ways.

Creating Instances

Now that we have the class declared, we can use it in our program! We declare a new person with the line `myPerson = Person()`, as shown on line 12 of **Figure 5.1.3**. Notice that this syntax looks like we're calling a function because of the parentheses after `Person`. This line is effectively like calling `Person.__init__()`, but Python is smart enough to let us leave out `__init__` when it's written exactly like that. So, calling `Person()` is like saying to the computer, "Give me a new instance of `Person`!" As a result, the computer creates a new instance, runs `__init__` to do the initial setup, and then returns that new instance so that we can assign it to a variable, all on line 12.

#	CreatingInstances-1.py	Output
1	`#Define the class Person`	`[no first name]`
2	`class Person:`	`[no last name]`
3	`    #Create a new instance of Person`	`[no eye color]`
4	`    def __init__(self):`	`-1`
5	`        #Person's default values`	
6	`        self.firstname = "[no first name]"`	
7	`        self.lastname = "[no last name]"`	
8	`        self.eyecolor = "[no eye color]"`	
9	`        self.age = -1`	
10		
11	`#Create a new Person and assign it to myPerson`	
12	`myPerson = Person()`	
13	`#Print myPerson's values`	
14	`print(myPerson.firstname)`	
15	`print(myPerson.lastname)`	
16	`print(myPerson.eyecolor)`	
17	`print(myPerson.age)`	
18		

Figure 5.1.3

Then, we can use `myPerson` like a normal variable. By calling `myPerson.firstname` on line 14, we access `self.firstname` from within the instance. Notice that this is similar to calling something like `myString.isupper()` to check if a string is uppercase: `firstname` and `isupper()` are contained *within* the instance, and calling them gives an answer specific to *that* instance. The difference is that `firstname` is a variable while `isupper()` is a method, but we'll talk about that in detail later.

Once we've created an instance of a class, we can also modify its variables just as if they were variables in our own program, as shown in **Figure 5.1.4**. We can print these variables by calling `print(myPerson.firstname)`, or we can modify them just by reassigning `myPerson.firstname`. Here on line 16, we reassign it to "David", then print it on line 18 to see the change. What about in our more complex example, though, where we had separate classes for `Name` and `Person`?

Page 259

#	CreatingInstances-2.py	Output
1	`#Define the class Person`	`[no first name]`
2	`class Person:`	`David`
3	`    #Create a new instance of Person`	
4	`    def __init__(self):`	
5	`        #Person's default values`	
6	`        self.firstname = "[no first name]"`	
7	`        self.lastname = "[no last name]"`	
8	`        self.eyecolor = "[no eye color]"`	
9	`        self.age = -1`	
10		
11	`#Create a new Person and assign it to myPerson`	
12	`myPerson = Person()`	
13	`#Print myPerson's firstname`	
14	`print(myPerson.firstname)`	
15	`#Change myPerson's firstname to David`	
16	`myPerson.firstname = "David"`	
17	`#Print myPerson's firstname`	
18	`print(myPerson.firstname)`	
19		

Figure 5.1.4

In **Figure 5.1.5**, instead of accessing `firstname` directly from `myPerson` (`myPerson.firstname`), we instead access `name` from `myPerson` (`myPerson.name`), and *then* access `firstname` from `name` (`myPerson.name.firstname`). `name.firstname` means "get `name`'s `firstname`". So, `myPerson.name.firstname` "get `myPerson`'s `name`'s `firstname`." We'll stick with just one class for the most part, but know we can combine them like this as well.

#	CreatingInstances-3.py	Output
1	`#Define the class Name`	`[no first name]`
2	`class Name:`	`David`
3	`    def __init__(self):`	
4	`        self.firstname = "[no first name]"`	
5	`        self.lastname = "[no last name]"`	
6		
7	`#Define the class Person`	
8	`class Person:`	
9	`    def __init__(self):`	
10	`        self.name = Name()`	
11	`        self.eyecolor = "[no eye color]"`	
12	`        self.age = -1`	
13		
14	`#Create a new Person and assign it to myPerson`	
15	`myPerson = Person()`	
16	`#Print myPerson's name's firstname`	
17	`print(myPerson.name.firstname)`	
18	`#Change myPerson's name's firstname to David`	
19	`myPerson.name.firstname = "David"`	
20	`#Print myPerson's name's firstname`	
21	`print(myPerson.name.firstname)`	

Figure 5.1.5

The usefulness of objects is that we can create multiple instances, each with their own values. For example, we could create mutliple variables to represent multiple people, and give each person a unique name, as seen in **Figure 5.1.6** .

#	CreatingInstances-4.py	Output
1	`#Define the class Person`	`myPerson1: David`
2	`class Person:`	`myPerson2: Vrushali`
3	`    #Create a new instance of Person`	
4	`    def __init__(self):`	
5	`        #Person's default values`	
6	`        self.firstname = "[no first name]"`	
7	`        self.lastname = "[no last name]"`	
8	`        self.eyecolor = "[no eye color]"`	
9	`        self.age = -1`	
10		
11	`#Create two new Persons and assign them to`	
12	`#myPerson1 and myPerson2`	
13	`myPerson1 = Person()`	
14	`myPerson2 = Person()`	
15	`myPerson1.firstname = "David"`	
16	`myPerson2.firstname = "Vrushali"`	
17		
18	`print("myPerson1: " + myPerson1.firstname)`	
19	`print("myPerson2: " + myPerson2.firstname)`	
20		

Figure 5.1.6

In **Figure 5.1.6** , we define two instances of the `Person` class, `myPerson1` and `myPerson2`. Changing the first name of `myPerson1` doesn't impact `myPerson2`, as confirmed by the `print()` statements on lines 18 and 19. We could thus create lists of lots of instances of `Person` to represent class rosters or directories, and modifying one instance wouldn't affect any of the others.

Objects vs. Dictionaries

Recall that at the end of the **Chapter 4.5** on dictionaries, we briefly discussed using dictionaries to create object-like structures. Specifically, we said that if we used the same keys across multiple dictionaries, we were effectively creating objects. Check out the similarity in **Figure 5.1.7** .

```
#  ObjectsvsDictionaries.py                                  Output
1  #Define the class Name                                    Dictionary:
2  class Name:                                               David
3      def __init__(self):                                   Instance:
4          self.firstname = "[no first name]"                David
5          self.lastname = "[no last name]"
6
7  #Define dictionaries with keys firstname and lastname
8  myNameDict = {"firstname" : "David", "lastname" : "Joyner"}
9
10 #Define instances of Name
11 myNameInst = Name()
12 myNameInst.firstname = "David"
13 myNameInst.lastname = "Joyner"
14
15 print("Dictionary: " + myNameDict["firstname"])
16 print("Instance: " + myNameInst.firstname)
17
```

Figure 5.1.7

In each case, we define an instance (of type dictionary on line 8 and of type `Person` on lines 11 to 13) that represents David Joyner. For the dictionary, it's with the key `firstname` and the value "David", and for the class it's with the class variable `firstname` and the value "David". So what's the benefit of using class instead of dictionaries? First, by defining classes, we *guarantee* (rather than *assume*) the keys are what we expect them to be. Second, classes can have methods as well as variables, while dictionaries have only variables; in other words, dictionaries only store data, whereas classes can also act on data.

In **Figure 5.1.7** , it would seem that the benefit of dictionaries is that we can define everything in one line, but there's a way we can do that with classes, too, which we'll cover next.

3. Encapsulating Methods in Classes

Part of the power of classes and instances is that they let us create data types with logical combinations of variables. We could create a `Person` class with a person's name, eye color, age, address, and telephone number. We could create a `Chair` class with a chair's color, material, number of legs, and style. We could create a `Student` class with a student's name, student ID number, and a list of their course enrollments. These course enrollments could themselves be instances of a `CourseData` class, which would have a list of the student's grades in that class. We can use objects to create complicated schemes of data that are still relatively easy to use because they're organized logically.

That's only half the power of object-oriented programming, though. The other half is that classes can contain methods —their own dedicated functions—as well as variables.

Encapsulating Methods

Encapsulation is the principle of object-oriented program that describes organizing variables and methods together into custom structures. I've saved defining the term until here because while it applies just to variables as well, methods are what make encapsulation truly powerful.

A method is a function defined inside of a class. It has all the same properties: a name, a list of parameters, some code, and optionally a return statement. The scope inside the method is defined as the normal scope of a function for that language (typically the method's parameters and any variables it defines in its own code), *plus* any variables that are visible in the instance of the class as a whole (accessed via that `self` variable). So, in the case of a `Person` class with class variables `firstname` and `lastname`, the class might have a `getFullName()` method that would return the first and last name together. Because the first and last name exist inside the instance, the method `getFullName()` could see them and manipulate them before returning the name.

Common Method Types: Constructors and Destructors

Methods can be used for anything we use functions to do. For example, if we had a class representing a person's bank account called `BankAccount`, it might have a variable to represent the current balance, and methods to check the balance and attempt a transaction.

There are four common types of methods, however, that tend to come up a lot in the implementation of classes. The first two are **constructors** and **destructors**. Oftentimes, these both have special syntax in a language to set them apart so that the computer can recognize them. A constructor contains code that will run every time a new instance is created; it's kind of like a "setup" method. For example, if a class has a list, the list needs to be initialized before it can actually be used; this is done in the constructor because it guarantees it is performed before it's needed. Or, in our `BankAccount` example, we would typically assume every newly-created bank account has no balance, so in the constructor we would initially set the balance to 0.

Since constructors are methods, they can take arguments for parameters as well. In other words, when someone is first creating an instance of a `Person` class, they could supply the first name and last name as arguments directly to the constructor. The constructor, then, would initialize the instance with these initial values. So, a constructor is a method that is automatically run whenever you create a new instance of a class; any code you want to run before the instance can be used should be placed in the constructor.

A destructor, on the other hand, is a method that deletes the instance. This is most pertinent in languages where the programmer is asked to do a lot of manual memory management, and for the most part these principles are outside the scope of this material. Generally, though, destructors can be useful if you're dealing with massive quantities of data and find yourself running out of memory: you can free some up by destroying instances when you're done with them.

Common Method Types: Getters and Setters

Getters and **setters** are simple method structures that allow code to interact with the variables inside an instance. A getter simply returns the value of a certain variable, and a setter changes the value of a variable.

Why do we need to do these through methods, though? Can't we just access these variables directly? Many times, it's best to design our code such that the variables inside it cannot be modified directly. For example, imagine again that we're writing a class to represent a `BankAccount`. Imagine this class is going to be accessed by different banks and customers. We don't want them all to just be able to change the value of `myBalance` at will: they should only be able to change it in certain ways, like in a transaction. Methods like getters and setters let us dictate the rules under which class variables can be accessed and changed.

Even if we're writing a class where we *don't* mind if the variables are accessed and changed haphazardly, we still might want to know about these changes. Because getters and setters are methods, we can build whatever code into them that we want. Imagine that we want to build a log every time someone accessed or changed a certain piece of data. By using getters and setters, we allow ourselves to run some logging code every time these methods are run. If other code was accessing the variables directly, we wouldn't be able to log when the data was accessed.

Some languages go so far as to dictate the use of getters and setters. Some languages have a concept of "privacy" in the variables and methods they encapsulate: they can determine whether a variable or method is public, where it can be accessed from outside the instance, or private, where it can only be accessed by methods within the class. So, in our `BankAccount` example, we might state that `myBalance` is a private variable, meaning it can only be accessed by methods inside the class. `getAccountBalance()`, on the other hand, would be a public method that returns `myBalance`. In that way, `myBalance` is still accessible, but it can't be modified, and `BankAccount` can log every time it is accessed.

4. Encapsulating Methods in Python

In many programming languages, class variables are defined outside of any particular method. Python, interestingly, doesn't support that: if you define a variable outside of a method, then that variable exists once and has the same value for every single instance of that class (which is called a "static" variable). Sometimes that's useful, but we won't really talk about these times here; instead, we'll focus on how to use class variables and methods the more typical way.

Constructors in Python

We've already seen a constructor in Python, and in order to define class variables in Python, we *have* to define them inside a method, preceded by that `self` variable. So, recall the code shown again in **Figure 5.1.8** .

#	DeclaringaClass-1.py	Output
1	`#Define the class Person`	
2	`class Person:`	
3	`    #Create a new instance of Person`	
4	`    def __init__(self):`	
5	`        #Person's default values`	
6	`        self.firstname = "[no first name]"`	
7	`        self.lastname = "[no last name]"`	
8	`        self.eyecolor = "[no eye color]"`	
9	`        self.age = -1`	
10		

Figure 5.1.8

On line 4, `__init__` is Python's convention for identifying constructors. Whenever a new instance of a class is created, Python goes and searches for the class's `__init__` method and runs it if it exists. If it doesn't exist, that's alright; Python just creates the instance without running any initial code. Here, running this constructor creates the variables `firstname`, `lastname`, `eyecolor`, and `age` to be seen later.

Notice, however, that `__init__` is a method, and that it has a parameter list given in parentheses on line 4. Because it's a method in a class, its first parameter must always be `self`; this is what lets the method see the variables defined for the instance. After that, however, we can define parameters as normal. If we wanted to be able to create a new person and define that person's first name and last name from the start; we could write the code shown in **Figure 5.1.9** .

#	ConstructorsinPython-1.py	Output
1	`#Define the class Person`	David
2	`class Person:`	Joyner
3	`    #Create a new instance of Person`	
4	`    def __init__(self, firstname, lastname):`	
5	`        self.firstname = firstname`	
6	`        self.lastname = lastname`	
7	`        self.eyecolor = "[no eye color]"`	
8	`        self.age = -1`	
9		
10	`#Creates a person with names David and Joyner`	
11	`myPerson = Person("David", "Joyner")`	
12	`print(myPerson.firstname)`	
13	`print(myPerson.lastname)`	
14		

Figure 5.1.9

When `Person("David", "Joyner")` is run on line 11, Python automatically goes looking for `__init_` in the `Person` class. It finds it, and pairs the first argument "David" with the first parameter `firstname`, and the second argument "Joyner" with the second parameter `lastname`. It then creates its own class variable `self.firstname` on line 5 and sets it equal to `firstname` from the parameter list, and then does the same for `lastname` on line 6. Note here that it can tell the difference between `self.firstname` and `firstname`. `self.firstname` tells it to go and check for a variable named `firstname` that is persistent for the instance, and if it doesn't find one, create one; `firstname` without `self` preceding it is known to only exist within the method, so it checks the variables defined in the parameter list.

After calling that constructor, the values of `firstname` and `lastname` within the instance `myPerson` are assigned to "David" and "Joyner", so when we print them on lines 12 and 13, these values still print.

Now that we've done this, though, can we still create an instance without these arguments? As shown in Figure 5.1.10 , no! "Positional" arguments are mandatory in functions and methods, meaning that we must supply them to run the method. If we want to preserve the ability to skip supplying arguments, we need to make the parameters optional, as shown in Figure 5.1.11 .

#	ConstructorsinPython-2.py
1	`#Define the class Person`
2	`class Person:`
3	`    #Create a new instance of Person`
4	`    def __init__(self, firstname, lastname):`
5	`        self.firstname = firstname`
6	`        self.lastname = lastname`
7	`        self.eyecolor = "[no eye color]"`
8	`        self.age = -1`
9	
10	`#Creates a new person`
11	`myPerson = Person()`
12	`print(myPerson.firstname)`
13	`print(myPerson.lastname)`
14	

Output

```
Traceback (most recent call
last):
File "...", line 11
    myPerson = Person()
TypeError: __init__() missing 2
required positional arguments:
'firstname' and 'lastname'
```

Figure 5.1.10

#	ConstructorsinPython-3.py
1	`#Define the class Person`
2	`class Person:`
3	`    #Create a new instance of Person`
4	`    def __init__(self, firstname="[no first name]",`
5	`                 lastname="[no last name"):`
6	`        self.firstname = firstname`
7	`        self.lastname = lastname`
8	`        self.eyecolor = "[no eye color]"`
9	`        self.age = -1`
10	
11	`myPerson1 = Person()`
12	`print(myPerson1.firstname)`
13	`myPerson2 = Person(firstname = "David")`
14	`print(myPerson2.firstname)`
15	`myPerson3 = Person("Vrushali")`
16	`print(myPerson3.firstname)`
17	

Output

```
[no first name]
David
Vrushali
```

Figure 5.1.11

In Figure 5.1.11 , we've defined the parameters as optional by giving them default values in the parameter list. If a given argument isn't supplied, the code assumes it should use the value from the parameter list (such as "[no first name]" for `firstname` on line 4). So, the `Person` instance on line 11 supplies no arguments, and so `myPerson1`'s first name is "[no first name]". The second `Person` instance, `myPerson2` on line 13, supplies `firstname = "David"` as an argument, and so the `firstname` parameter gets the value "David". The third `Person` instance, `myPerson3` on line 15, shows that if an argument is given where no positional parameter is located, the program assumes the argument is for the next parameter; so, even though `firstname =` is not included on line 15, the code nonetheless assumes "Vrushali" is the value for `firstname` since `firstname` is the first parameter in the list.

Destructors do exist in Python, but because Python does so much memory management on its own, you likely won't need to use them until you get to much more advanced programs.

Getters and Setters

Interestingly as well, Python does not provide privacy options for its variables and methods. There is no way to bindingly mark a variable or method in a Python class as private, meaning that other code can always access variables directly. By convention, we often precede variables that we don't *want* other classes or functions to access with a double underscore;

however, this is only a convention, meaning that other classes or functions are still able to access the data. The double underscore simply informs them that they are not intended to access the data in this way.

Part of this is "Pythonic" style, which focuses on easy access to data. This is related to the ease of Python's lists and dictionaries; other languages supply these as more traditional classes with tougher syntax. Generally, when you're developing classes in Python, it's alright to directly access variables. That's a major taboo in some languages (like Java), but it's accepted in Python.

That said, getters and setters have other purposes as well. Recall that part of the benefit of getters and setters was that they allow us to run some code whenever a variable is accessed or modified. That might be useful in a simple logging behavior, for example. To demonstrate this, let's finally implement that `BankAccount` class we've been talking about.

First, **Figure 5.1.12** shows the class itself. Notice a few things. First, notice that to create an account, we *must* have a name, but the balance is optional, as shown on lines 4 (defining the class) and 22 (creating an instance). If no balance is supplied, the computer assumes the balance is 0.0. Notice that we say 0.0 to force the computer to see this as a float, not an integer.

Page 265

GettersandSetters.py

```
1  #Define class BankAccount
2  class BankAccount:
3      #Initialize balance to 0
4      def __init__(self, name, balance = 0.0):
5          self.log("Account created!")
6          self.name = name
7          self.balance = balance
8
9      def getBalance(self): #Getter for balance
10         self.log("Balance checked at " + str(self.balance))
11         return self.balance
12
13     def setBalance(self, newBalance): #Setter for balance
14         self.log("Balance changed to " + str(newBalance))
15         self.balance = newBalance
16
17     def log(self, message): #Logging method
18         myLog = open("Log.txt", "a")
19         print(message, file = myLog)
20         myLog.close()
21
22 myBankAccount = BankAccount("David Joyner")
23 myBankAccount.setBalance(20.0)
24 print(myBankAccount.getBalance())
25
```

Output

```
20.0
```

Log.txt

```
Account created!
Balance changed to 20.0
Balance checked at 20.0
```

Figure 5.1.12

Second, notice we've supplied a getter and setter on lines 9 through 15. Within each, we have the obvious lines `return self.balance` and `self.balance = newBalance`, which get and set `balance` respectively. Notice, however, that we precede these with a call to `log()` on lines 10 and 14 with a message. This is why getters and setters can still be valuable: they allow us to trace or log program execution.

Third, notice that when we're calling `log()`, we still precede it with `self.`. `self` is still how we allow different parts of an instance to see other parts. To let the `setBalance()` method see the `log()` method, it needs to call `self.log()` to basically say, "my `log()` method". Fourth, notice that we open our file in "append" mode inside `log()` on line 18. This allows us to build a log over time in the file without storing the log in a variable within our program.

Down in the main code, we first create a bank account on line 22. We supply a name to the constructor, but no balance, so the balance is assumed to be 0. We then set the balance to 20.0 on line 23; we could have set it in the constructor, of course, but we're demonstrating the setter and getter. Then, we print the balance on line 24, seeing that it has been correctly set to 20.0. If we opened Log.txt (as shown in the bottom right), we would see three lines:

- Account created!
- Balance changed to 20.0
- Balance checked at 20.0

The constructor, getter, and setter all write to Log.txt.

Encapsulating Other Functions

Note that constructors, destructors, getters, and setters are four common paradigms for designing methods, but that certainly is not an exhaustive list of every type of method. We can create methods to do whatever we want. For example, in the `BankAccount` code, we likely don't want a `setBalance()` method because we rarely say, "Regardless of the prior value of this account, set its value to this new number." Instead, we say "deposit $20" or "withdraw $10". So, these are the methods we would likely want to create.

In **Figure 5.1.13**, instead of just changing the balance manually as in **Figure 5.1.12**, we add or subtract to or from it with the `deposit()` and `withdraw()` methods. Of course, we could have done this anyway with a call like `myBankAccount.setBalance(myBankAccount.getBalance() + 20.0)`, but we want to write methods that are as easy to call as possible. If we know we'll regularly be withdrawing from and depositing to the account, it's better to have methods to take care of that.

Page 266

EncapsulatingOtherFunctions.py

```
1  class BankAccount:
2      def __init__(self, name, balance = 0.0):
3          self.log("Account created!")
4          self.name = name
5          self.balance = balance
6
7      def getBalance(self):
8          self.log("Balance checked at " + str(self.balance))
9          return self.balance
10
11     def deposit(self, amount):
12         self.balance += amount
13         self.log("+" + str(amount) + ": " + str(self.balance))
14
15     def withdraw(self, amount):
16         self.balance -= amount
17         self.log("-" + str(amount) + ": " + str(self.balance))
18
19     def log(self, message): ...
...
24 myBankAccount = BankAccount("David Joyner")
25 myBankAccount.deposit(20.0)
26 print(myBankAccount.getBalance())
27 myBankAccount.withdraw(10.0)
28 print(myBankAccount.getBalance())
```

Output

```
20.0
10.0
```

Log.txt

```
Account created!
+20.0: 20.0
Balance checked at 20.0
-10.0: 10.0
Balance checked at 10.0
```

Figure 5.1.13

5. Advanced Topics in Classes in Python

We've talked a good bit so far about references and mutability in Python. How do these concepts play along with classes? The answer: they pretty much follow the same conventions. Immutable types are still immutable, but most types are still mutable. This can get a little tricky when we start dealing with combinations of the two, though.

Combining Classes

Let's explore this just by trying out some different combinations of things. For this running example, let's use our two classes from before, `Person` and `Name`. Let's also keep things simple by accessing the variables directly instead of using getters and setters. First, let's see how we can combine them in interesting ways.

In **Figure 5.1.14**, notice that `Person` has a `name`, `eyecolor`, and `age`. There's nothing in `Person` that dictates that the name must be of type `Name`, but that's the value we're supplying it on line 16. Had we supplied the string "David Joyner" directly, `myPerson` would have been created just fine, but the code would crash on line 17 when we tried to access the variable `firstname`, which exists in `Name` but not in string.

#	CombiningClasses.py	Output
1	`#Defines the class Person`	David
2	`class Person:`	Joyner
3	`    def __init__(self, name, eyecolor, age):`	brown
4	`        self.name = name`	30
5	`        self.eyecolor = eyecolor`	
6	`        self.age = age`	
7		
8	`#Defines the class Name`	
9	`class Name:`	
10	`    def __init__(self, firstname, lastname):`	
11	`        self.firstname = firstname`	
12	`        self.lastname = lastname`	
13		
14	`#Creates a person with eyecolor "brown", age 30, and`	
15	`#a name with firstname "David", lastname "Joyner",`	
16	`myPerson = Person(Name("David", "Joyner"), "brown", 30)`	
17	`print(myPerson.name.firstname)`	
18	`print(myPerson.name.lastname)`	
19	`print(myPerson.eyecolor)`	
20	`print(myPerson.age)`	
21		

Figure 5.1.14

Second, notice that we're initializing the argument for name *while* we're initializing `myPerson`, all on line 16. Calling `Name("David", "Joyner")` returns an instance of `Name` with `firstname` "David" and `lastname` "Joyner", which is the argument we want to pass into the constructor for `Person`. So, we can initialize `Name` right there within the constructor for `Person`. We also could have separately called `myName = Name("David", "Joyner")` and used `myName` as the argument, but since we never need `myName` on its own, we might as well create it inside `Person`'s constructor on line 16.

The result is an instance of `Person` called `myPerson`, which has a string for `eyecolor` ("brown"), an integer for `age` (30), and an instance of `Name` for `name`. That instance of `Name` has strings for its `firstname` ("David") and `lastname` ("Joyner").

Instance Assignments

So, using the instance of `Person` from **Figure 5.1.16** , let's see what happens if we assign it to another instance. What do you think happens with the code in **Figure 5.1.15** ?

Page 267

#	InstanceAssignments.py	Output
1	`class Person:`	myPerson1's
2	`    def __init__(self, name, eyecolor, age):`	eyecolor: blue
3	`        self.name = name`	myPerson2's
4	`        self.eyecolor = eyecolor`	eyecolor: blue
5	`        self.age = age`	
6		
7	`class Name:`	
8	`    def __init__(self, firstname, lastname):`	
9	`        self.firstname = firstname`	
10	`        self.lastname = lastname`	
11		
12	`myPerson1 = Person(Name("David", "Joyner"), "brown", 30)`	
13	`myPerson2 = myPerson1`	
14	`myPerson2.eyecolor = "blue"`	
15	`print("myPerson1's eyecolor: " + myPerson1.eyecolor)`	
16	`print("myPerson2's eyecolor: " + myPerson2.eyecolor)`	
17		

Figure 5.1.15

#	InstancesasArguments-1.py	Output
1	`class Person:`	DAVID
2	`    def __init__(self, name, eyecolor, age):`	JOYNER
3	`        self.name = name`	
4	`        self.eyecolor = eyecolor`	
5	`        self.age = age`	
6		
7	`class Name:`	
8	`    def __init__(self, firstname, lastname):`	
9	`        self.firstname = firstname`	
10	`        self.lastname = lastname`	
11		
12	`def capitalizeName(name):`	
13	`    name.firstname = name.firstname.upper()`	
14	`    name.lastname = name.lastname.upper()`	
15		
16	`myPerson = Person(Name("David", "Joyner"), "brown", 30)`	
17	`capitalizeName(myPerson.name)`	
18	`print(myPerson.name.firstname)`	
19	`print(myPerson.name.lastname)`	
20		

Figure 5.1.16

We create an instance of `Person` just like before on line 12. We then create a second `Person` instance, `myPerson2`, and set it equal to `myPerson1` on line 13. We then modify `myPerson2` on line 14. Does `myPerson1` also change? The output of line 15 shows it does! An instance of the `Person` class is mutable, so when we say `myPerson2 = myPerson1`, we're really just telling them to look at the same data in memory. So, if `myPerson2` changes something (like `eyecolor`), it's changing it in the same place in memory that `myPerson1` refers to. So, it changes for both.

Instances as Arguments

What happens if we pass an instance into a function? Let's imagine we have a function called `capitalizeName()` on lines 12 through 14, which converts an instance of `Name` to all caps, as shown in **Figure 5.1.16** .

`capitalizeName(myPerson.name)` passes in the instance of the `Name` object into `capitalizeName`. `Name` is mutable, meaning that `capitalizeName()` is working off the same copy of `myPerson.name`. So, when the name's capitalization changes, it *does* change it for the original copy, as seen by the `print()` statements in lines 18 and 19.

Page 268

What if, though, instead of `capitalizeName()`, we had `capitalizeString()`, and called it separately on `firstname` and `lastname`? This is shown in **Figure 5.1.17** , and the new method is on lines 12 and 13.

#	InstancesasArguments-2.py	Output
1	`class Person:`	David
2	`    def __init__(self, name, eyecolor, age):`	Joyner
3	`        self.name = name`	
4	`        self.eyecolor = eyecolor`	
5	`        self.age = age`	
6		
7	`class Name:`	
8	`    def __init__(self, firstname, lastname):`	
9	`        self.firstname = firstname`	
10	`        self.lastname = lastname`	
11		
12	`def capitalizeString(instring):`	
13	`    instring = instring.upper()`	
14		
15	`myPerson = Person(Name("David", "Joyner"), "brown", 30)`	
16	`capitalizeString(myPerson.name.firstname)`	
17	`capitalizeString(myPerson.name.lastname)`	
18	`print(myPerson.name.firstname)`	
19	`print(myPerson.name.lastname)`	
20		

Figure 5.1.17

Here, the `firstname` and `lastname` are *not* capitalized when printed on lines 18 and 19. Why? Because strings themselves are immutable, and the assignment operation within `capitalizeString()` operates on a string. So, `capitalizeString()` changes what its local copy of `instring` points at, but that local copy isn't the same as `firstname` or `lastname` in this instance of `Name`.

So, any operations we make on mutable data types propagate out of the function; any operations we make on immutable types do not. If we pass an instance of a class into a function or method, the variables of the class could be changed; if we pass only the immutable variables themselves, then the variables of the class cannot be changed.

Making Actual Copies

So what do you do if you want to make an *actual* copy of an instance, such that you can modify it separately? Although there are more efficient ways, the basic principle is that you must copy at the level of the immutable data types. In other words, setting `myPerson2` equal to `myPerson1` didn't work because it just made the two variables point at the same values. To actually copy the values, we need to do exactly that: copy the values.

Figure 5.1.18 is a good demonstration of copying the values. Instead of just setting `myPerson2` equal to `myPerson1`, we create a *new* instance of `Person`, using the *values* of `myPerson1` as the arguments on line 13. So, we tell `myPerson2` to point to a new instance of `Person` because we call `Person`'s constructor, and we populate `name`, `eyecolor`, and `age` with the same values as `myPerson1`. This creates a new instance in memory, as opposed to just another variable that points to the same instance in memory.

```
# MakingActualCopies-1.py
1  class Person:
2      def __init__(self, name, eyecolor, age):
3          self.name = name
4          self.eyecolor = eyecolor
5          self.age = age
6
7  class Name:
8      def __init__(self, firstname, lastname):
9          self.firstname = firstname
10         self.lastname = lastname
11
12 myPerson1 = Person(Name("David", "Joyner"), "brown", 30)
13 myPerson2 = Person(myPerson1.name, myPerson1.eyecolor, myPerson1.age)
14 myPerson2.eyecolor = "blue"
15 print(myPerson1.eyecolor)
16 print(myPerson2.eyecolor)
17 myPerson2.name.firstname = "Vrushali"
18 print(myPerson1.name.firstname)
19 print(myPerson2.name.firstname)
```

```
Output
brown
blue
Vrushali
Vrushali
```

Figure 5.1.18

Since `myPerson2` has its *own* variables for `eyecolor`, then when we reassign it on line 14, it doesn't affect the `eyecolor` for `myPerson1`. `eyecolor` is a string, which is immutable, so when we reassign it, it doesn't change the value in memory: it instead just points the variable `myPerson2.eyecolor` at a new value. This doesn't change the value that `myPerson1.eyecolor` points at.

Notice, though, that the same doesn't apply to `name` as written here. We passed `myPerson1.name` as the argument when constructing `myPerson2`, but `myPerson1.name` is an instance of the `Name` class. An instance of the `Name` class is mutable. That means that even though `myPerson1` and `myPerson2` have their own variables called `name`, they're pointing at the same value in memory. So, when we modify what `myPerson2.name.firstname` points at on line 17, it *also* modifies what `myPerson1.name.firstname` points at, as shown by the output of lines 18 and 19. To make a true copy, we need to actually construct a new instance of `Name` as well, as shown in **Figure 5.1.19** .

Page 269

```
# MakingActualCopies-2.py
1  class Person:
2      def __init__(self, name, eyecolor, age):
3          self.name = name
4          self.eyecolor = eyecolor
5          self.age = age
6
7  class Name:
8      def __init__(self, firstname, lastname):
9          self.firstname = firstname
10         self.lastname = lastname
11
12 myPerson1 = Person(Name("David", "Joyner"), "brown", 30)
13 myPerson2 = Person(Name(myPerson1.name.firstname, myPerson1.name.lastname),
14                     myPerson1.eyecolor, myPerson1.age)
15 myPerson2.eyecolor = "blue"
16 print(myPerson1.eyecolor)
17 print(myPerson2.eyecolor)
18 myPerson2.name.firstname = "Vrushali"
19 print(myPerson1.name.firstname)
20 print(myPerson2.name.firstname)
```

```
Output
brown
blue
David
Vrushali
```

Figure 5.1.19

Instead of just passing `myPerson1.name` as the argument to the constructor creating `myPerson2` on line 13, we instead are calling the constructor of `Name` as well to create a new instance of `Name`, too. Into that constructor, we pass `myPerson1.name.firstname` and `myPerson1.name.lastname`, which are strings and thus immutable. Then, when we modify `myPerson2`'s `name`, it does not affect `myPerson1`'s `name`, as shown by the output of lines 19 and 20.

6. Polymorphism and Inheritance and Abstraction, Oh My!

We've barely scratched the surface of what objects can do. There are many advanced concepts in designing classes, like inheritance and abstraction. We're not going to get into programming these concepts in our material, but we want to quickly preview it so that when you see it later in your computing education, they'll sound familiar. Typically, there are entire classes on object-oriented programming, and these concepts are a big part of that additional depth.

Abstraction

Let's take a quick example. What is this a picture of? Most people would probably say: a chair, and of course, they'd be right. Some people, though, might say it's a dining room chair. They'd also be right. Some people might say it's a Harvest-style dining room chair. They'd also be right. On the other end of the spectrum, some people might say it's a piece of furniture. They'd also be right. Some people might say it's a home good. They'd also be right. And some people might say, well, it's a thing. They'd also be right.

What this object "is" exists at different levels of **abstraction** , and we need different levels for different purposes. For example, it wouldn't make much sense to ask: how many legs does a home good have? But it does make sense to ask: how many legs does a dining room chair have? Certain variables only make sense at a certain level of abstraction.

Or, to take another example, imagine you want to buy a Harvest-style dining room chair. Do you go to the Harvest-style dining room chair store? Probably not. You don't even go to the chair store; you go to the furniture store. You know that's the level of abstraction at which chairs are sold: they're sold as part of general furniture sales. Yet, you rarely go shopping for furniture: you don't go to the store thinking, "I don't know if I want a chair or a table." You probably know what you want. This is an example of different levels of abstraction at work.

Polymorphism

Polymorphism is a characteristic of this idea of abstraction that was implicit in that example. Polymorphism is the ability to ask the same questions—or in software, run the same code—on a wide variety of different types of concepts. In the above example, we would likely say that chairs, tables, and beds are three different types of objects, and yet we can ask certain questions across them, like their price and their material. We can imagine a method that returns a material, and we can imagine that method being applicable to any kind of furniture. Then, we can imagine iterating over a catalog of furniture and asking, for each item, "What material is this?" Even though the data are different types of furniture, that one single question makes sense.