Chapter 5

Introduction to Objects

In the coming chapters, we want to unleash some of the power of Python. To access this power we have to introduce something that is syntactically different from what we have seen so far. In order to understand the reason for this change in syntax, it is best to have a simple understanding of *objects* and *object oriented programming*.

Computer languages, considered as a whole, differ widely in how they allow programmers to construct a program. There are some languages that permit object oriented programming, often abbreviated OOP. Python is one such language (C++, Java, and C# are other popular OOP languages). As will be described, OOP provides a way of organizing programs that is more similar to the way people think (i.e., more so than if one uses a non-OOP approach). Thus, in many instances an OOP approach provides the clearest and simplest route to a solution. However, as you might imagine, when something attempts to reflect the way humans think, there can be many subtleties and several layers of complexity. We do *not* want to explore any of these subtleties and complexities here.

At the conclusion of this chapter you should have a sufficiently clear understanding of a couple of aspects of OOP—to the point where you should be comfortable with the material in chapters that follow. But, there is no expectation that this chapter will enable you to write your own programs that fully exploit the power of OOP. (So, if some questions about OOP linger at the end of this chapter, it is to be expected and should not be a concern. At the end of the chapter we will revisit the "take away" message of this material.)

5.1 Overview of Objects and Classes

In previous chapters we considered data such as strings, integers, and `floats`. We also considered functions that can accept data via a parameter (or parameters) and can return data via a `return` statement. Functions are created to accomplish particular tasks and are invariably restricted in the type of data they can accept. For example, if we write a function to capitalize the characters in a string, it doesn't make sense to pass numeric data to this function. As another example, if a function calculates the absolute value, it doesn't make sense to pass this function a string.

When you pause to think about data and functions, you quickly realize they are related in some way. The fact that only certain forms of data make sense with certain functions ties data and functions together conceptually. In OOP there is still the ability to work with traditional data and

From the file: `object-basics.tex`

functions, but OOP introduces an additional construct known as an *object*. An object is a collection of data *and* functions (we'll see how to create the collection in a moment). Using objects we can more closely associate functions with the type of data on which the functions can legitimately operate. To distinguish traditional data from the data contained in an object, we call the data in an object the object's *attributes*. And, to distinguish traditional functions from the functions contained in an object, we call the functions in an object the object's *methods*. So, instead of saying an object is a collection of data and functions, it is more proper to say an object is a collection of attributes and methods.[1]

In Python, as we shall see, essentially everything is an object! The data we have seen so far are actually objects of a given type (such as `str`, `int`, or `float`). Functions that we create are objects of a different type. Built-in functions are objects of yet another type. And so on. Replacing the word "type" with the word "class" in these sentences leads us to the slightly more technical statement: All objects are members of a *class*.

What does it mean to say all objects are members of a class? This is actually closely aligned with the way we naturally think of the world and the objects in it. In the real world, we automatically think in terms of "groups" of "things." Yes, you are unique, but you are still a member of the group of humans. We may say you are a unique *instance* of a human, but you still belong to the group of humans. Your clothing may be one-of-a-kind; perhaps you made your favorite dress yourself. Nevertheless, that dress is still a member of, or instance of, the group of dresses.

The ability to group things allows us to make sense of the world more easily than could be done without these associations. Knowing a thing is a member of a group allows us to make various assumptions about that thing. For example, if we know a "thing" is a human, we know he or she shares attributes with all other members of the group of humans, such as blood type, hair color, gender, and visual acuity. (Although all members of the group share the same attributes, an individual has specific values for these attributes that are his or her own, such as a blood type of AB positive, black hair, female, and 20/15 vision.) Knowing something is a car, we know it has a different set of attributes, such as number of seats, headroom, and miles per gallon. Dresses have attributes such as type of fabric, size, sleeve length, etc. This grouping also gives us information about the way in which something functions or interacts with the world. We wouldn't expect a human to behave in the same way as a car or a dress. Things in the human, car, and dress groups all have their own ways of interacting with other things.

If we go back to the previous two paragraphs and replace the word "thing" with "object" and "group" with "class," we have the start of viewing the world in an OOP way. Instead of "all things are members of a group" we have "all objects are members of a class." A class defines what the attributes are that members of the class share. However, when it comes to an individual object (i.e., an instance of a class), the values of these attributes are those of that particular object. Again, even though people share common attributes, the values of attributes such as blood type, hair color, gender, visual acuity, etc., can vary significantly from one individual to the next.

With that said, in the next section we show how to define a class (a "group") and create an instance of the class (a "thing") in Python.

[1] One thing that can be somewhat confusing about OOP is that different languages use different terms for the same underlying concept. For example, what we are calling a method in Python, would be called a *member function* in C++. Also, different authors may use different terms to describe the same things even in the same language!

5.2 Creating a Class: Attributes

In OOP, the programmer uses a *class statement* to provide something of a blueprint for the attributes an object has (we'll turn our attention to an object's methods in the next section). Let's use a slightly contrived scenario to help illustrate the creation of a class. Assume a programmer works for a hospital and needs to develop a program that deals with patients. These patients have only three attributes: name, age, and malady. To tell Python what constitutes a patient, the programmer creates a class with a statement that follows the template shown in Listing 5.1.[2]

Listing 5.1 Basic template for creating a class.

```
1 class <ClassName>:
2     <body>
```

Line 1 provides the class header which starts with the keyword `class`. This is followed by `<ClassName>` which is the name of the class and can be any valid identifier.[3] The header is terminated with a colon. The header is followed by the `<body>` which provides statements that specify the attributes and methods in the class (we defer consideration of methods to the next section). As was the case for a function definition, the `<body>` must be indented. The body can consist of any number of statements and is terminated by halting indentation of the code (or by entering a blank line in the interactive environment or in IDLE).

Although there are various ways to implement a `Patient` class, a programmer might use the `class` statement shown in lines 1 through 4 of Listing 5.2.[4] This statement creates the attributes `name`, `age`, and `malady` simply by using the assignment operator to assign default values to these identifiers. Keep in mind that this `class` statement itself does *not* actually create a `Patient`. Rather, it tells us what attributes a `Patient` has. In some ways this is similar to what we do with functions. We define a function with a `def` statement, but this doesn't invoke the function. After the function is defined, we are free to call it as needed. The creation of a class is somewhat similar. After we create a class with a `class` statement, we are free to call the class to create as many objects (of that class) as needed. (Creating an object is often called *instantiation* and we will use the terms object and *instance* synonymously.) The rest of the code in Listing 5.2 is discussed in more detail following the listing.

Listing 5.2 `class` statement that defines the `Patient` class where a `Patient` has a `name`, `age`, and `malady`. Each of these attributes is given a default value.

```
1 >>> class Patient:
2 ...     name = "Jane Doe"
3 ...     age = 0
```

[2] A more complicated `class` statement is available that allows one class to inherit properties from another, but we have no need for it here.

[3] The class identifier is usually written using the "CapWords" convention where the first letter of each word is capitalized and, of course, there are no spaces between the words (since there can be no spaces in an identifier).

[4] A construct like this would almost certainly *not* be used in practice, but this implementation is useful for the sake of illustration. In subsequent examples in this chapter we provide improved `class` statements.

```
4  ...     malady = "healthy"
5  ...
6  >>> # Create an instance of a Patient with identifier of sally.
7  >>> sally = Patient()
8  >>> # Show sally's name attribute.
9  >>> sally.name
10 'Jane Doe'
11 >>> # Set the name, age, and malady attributes for sally.
12 >>> sally.name = "Sally Smith"
13 >>> sally.age = 21
14 >>> sally.malady = "bruised ego"
15 >>> # Show sally's name, age, and malady attributes.
16 >>> print(sally.name, sally.age, sally.malady, sep="; ")
17 Sally Smith; 21; bruised ego
18 >>> def show(patient):
19 ...     print(patient.name, patient.age, patient.malady, sep="; ")
20 ...
21 >>> show(sally)
22 Sally Smith; 21; bruised ego
```

Following the `class` statement, a `Patient` is created in line 7 and assigned to the identifier `sally`. `sally` is simply a variable, but of a new data type, i.e., a `Patient`. We say that `sally` *is* a `Patient` or `sally` is an instance of the `Patient` class. In general, to create an instance of a class, we provide the class name followed by parentheses (similar to how we call a function—later we will see what one might put in the parentheses). Thus, the right side of line 7 instructs Python to create a `Patient` and the assignment operator associates this `patient` with the identifier `sally`.

In order to access attributes of an object, we use what is sometimes called *dot notation*: we write the variable name then a "dot" and then the attribute.[5] This is illustrated in line 9 where we access `sally`'s `name` attribute. More technically, in this context the period (or dot) is actually serving as the *access attribute operator* (which we may refer to as either the access operator or the attribute operator).

We can access an attribute either to see what it is, as is done in line 9, or to assign a value to it, as is done in lines 12 through 14 where new values are assigned to `name`, `age`, and `malady`. The output from the `print()` statement in line 16 shows that all of `sally`'s attributes have been changed from the default values.

If we want to display several `Patients`, it could be cumbersome to write several `print()` statements such as in line 16. To help streamline displaying `Patients`, we could write a function as shown in lines 18 and 19. The `show()` function takes a `Patient` as a parameter and then displays the `Patient`'s attributes. Note that the variable name used for the formal parameter, `patient`, can be any valid identifier, but `patient` is nicely descriptive of the fact that this function should be passed a `Patient` object as the actual parameter. Lines 21 and 22 show what happens when `sally` is passed to this function.

[5] This dot notation actually works with literals too as will be seen later.

The `show()` function defined in lines 18 and 19 is quite specialized—it is only designed to work with `Patients`. Thus, it would be more logical if this function were bundled with the `Patient` class. Again, when a function is incorporated into a class, it is known as a method. We consider how to create a method in the next section.

Referring to Listing 5.2, with the introduction of the access attribute operator, it is tempting to think that something like `sally.name` is a valid identifier. However, *it is not.* The rules that dictate what constitutes a valid identifier have not changed from before (see Sec. 2.6). Thus, `sally` is a valid identifier: it is a `Patient` object. And, `name` is a valid identifier: it is an attribute of the `Patient` object. However, `sally.name` is not a valid identifier. This does not imply that `sally.name` is not a legitimate lvalue. In fact, it is a valid lvalue—we can assign values to it. But, we cannot, for example, use `sally.name` as the name of a function or the name of an integer variable.

5.3 Creating a Class: Methods

In lines 18 and 19 of Listing 5.2 a function is defined to show a `Patient`'s attributes. Although this function only works for `Patients`, it is not truly affiliated with the `Patient` class—the function stands on its own. We could, perhaps, accidentally call the `show()` function with a string as the parameter. If we did, an error would occur (since strings don't have attributes of `name`, `age`, and `malady`).

Instead of defining a function such that it exists outside the class, we can include a function as a method within the class if we move the associated `def` statement into the `class` statement. By doing this, we can ensure the method only operates on the objects for which it is designed to operate. A method is almost exactly like a function but there are a couple of notable differences: one affects how we define a method and the other affects how we invoke it. Let us first consider the difference in invocation.

Similar to the `show()` function in Listing 5.2, assume we have written a *method* that shows the attributes of a `Patient`. To distinguish this from the `show()` function, let's call this method `display()`. In Listing 5.2, the *function* to show the object's attributes is invoked with an argument of `sally` using this statement:

```
show(sally)
```

Since `display()` is a method, where it is defined in the `class` statement, we invoke it using this statement:

```
sally.display()
```

We'll discuss this in more detail in a moment, but we want to state up front that this invocation *does* pass `sally` as a parameter to the `display()` method.[6] To invoke a method, we provide an instance of the class (which is `sally` here) and then, using dot notation, specify the method. The parentheses are necessary. `sally` "knows" her attributes but she also "knows" her methods; hence the dot notation allows us to access not only the attributes, but also the methods within `sally`'s class.

[6]The fact that `sally` is passed as a parameter to the `display()` method is done implicitly when the method is called, but, as will be shown soon, is indicated explicitly in the method definition.

Notice that in the invocation of the method it appears that the method has no parameters. However, this is *not* the case. When working with methods, Python always passes the instance on which the method is invoked (`sally` in this example) as the first parameter of the method. When defining the method (with a `def` statement), the convention is to call this first (formal) parameter `self`. An example helps to clarify this.

In Listing 5.3 the `class` statement starts with the same attributes in Listing 5.2. For the sake of readability, line 5 is semi-blank (in that there is an indentation but nothing else on the line). This is followed by the definition of the method `display()` (which is slightly fancier than the `show()` function in Listing 5.2 but ultimately displays the same data).

The template for a method is the same as the template for a function as given in Listing 4.2. The one thing that must be kept in mind, though, is that Python always provides an instance of the class as the first parameter to the method (in this case the method only has a single parameter—later we will consider a method with multiple parameters). In some sense, although we might write `sally.display()`, Python effectively translates this to `display(sally)`. Thus, when we write `sally.display()`, line 6 instructs Python that the formal parameter `self` is assigned the actual parameter `sally` and then the body of the method is executed. There is nothing unique about the identifier `self`. Any valid identifier will work. Although it seems `patient` would be a more descriptive identifier in this particular case than `self`, the convention is to use `self` and we will follow this convention. Discussion of Listing 5.3 continues below the listing.

Listing 5.3 A class statement that includes a method definition. The first parameter of a method is always the object on which the method is invoked. Although any valid identifier can be used for this parameter, it is convention to call this first parameter `self`.

```
1  >>> class Patient:
2  ...     name = "Jane Doe"
3  ...     age = 0
4  ...     malady = "healthy"
5  ...
6  ...     def display(self):
7  ...         print("Name    = ", self.name)
8  ...         print("Age     = ", self.age)
9  ...         print("Malady  = ", self.malady)
10 ...
11 >>> samuel = Patient()
12 >>> samuel.name = "Samuel Sneed"
13 >>> samuel.age = 18
14 >>> samuel.malady = "broken heart"
15 >>> samuel.display()
16 Name    =  Samuel Sneed
17 Age     =  18
18 Malady  =  broken heart
```

After the close of the class statement, a `Patient` is created in line 11 and assigned to `samuel` (i.e., the variable `samuel` is an instance of the `Patient` class). In lines 12 and 13, new values

are assigned to samuel's name, age, and malady. Then, in line 15, the display() method is invoked.

5.4 The dir() Function

When we create a class, we create a new type of data. (We can see this if we use the type() function to check on the type of one of the Patient's we created above.) Given that all data in Python are actually objects of classes means that all data are collections of attributes and methods. This is indeed the case and the built-in function dir() provides a way to see this collection, albeit not necessarily in the prettiest form. You can think of dir() as providing a ***dir***ectory of sorts.

Listing 5.4 is a continuation of Listing 5.3 in which the Patient class is defined and samuel is an instance of this class. In line 19 we use the type() function to ask what the type is of the Patient class. In line 20 we see that a Patient's type is type! This tells us that a Patient is its own (new) form of data.

In line 21 we ask for samuel's type. We see that samuel is a Patient but we are also given some additional information. This additional information relates to where the class is defined, but this doesn't concern us now.[7] In lines 23 and 24, type() shows that samuel's age attribute is an integer. Lines 25 and 26 show that samuel.display is a method. The rest of this code is discussed following the listing.

Listing 5.4 Illustration that a class is a type of data (and hence a new class is a new type of data). Also, this code demonstrates the use of the dir() function. This is a continuation of Listing 5.3.

```
19 >>> type(Patient)
20 <class 'type'>
21 >>> type(samuel)
22 <class '__main__.Patient'>
23 >>> type(samuel.age)
24 <class 'int'>
25 >>> type(samuel.display)
26 <class 'method'>
27 >>> dir(samuel)
28 ['__class__', '__delattr__', '__dict__', '__doc__', '__eq__',
29   '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
30   '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
31   '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
32   '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'age',
33   'malady', 'name', 'display']
```

In line 27 dir() is used to show all the attributes and methods associated with Patient. We are shown a collection of strings.[8] Notice the last four strings correspond to the attributes and the

[7] Since we used the interactive environment to define the Patient class, this class is said to have been created in __main__ which is Python's name for the "top-level script environment."

[8] This particular collection is known as a list and will be described in a later chapter.

method we put in the `Patient class` statement (these are shown in slanted bold type to help differentiate them from the other strings in the list).

All the other strings in this list are attributes or methods associated with the class, but we did not establish this connection. This is something Python did for us. We will mostly ignore these other entries, but there is one notable exception which is also shown in slanted bold type: `__init__`. This is a method that allows us to initialize an object when it is created. We will illustrate the use of this method in the next section. Before doing so, we consider another example of the use of the `dir()` function.

We now know that integers, `floats`, and strings are instances of classes. As a preview of what is discussed in detail in the chapter on strings, let's create a string and then use `dir()` to see its list of attributes and methods, as is done in Listing 5.5. In line 1 the string `s` is created. In line 2 the `dir()` function is applied to this string which reveals a rather lengthy list!

Listing 5.5 Using `dir()` to list the attributes and methods of a string.

```
1  >>> s = "hello!"
2  >>> dir(s)
3  ['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
4    '__eq__', '__format__', '__ge__', '__getattribute__',
5    '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
6    '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__',
7    '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
8    '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
9    '__subclasshook__', 'capitalize', 'center', 'count', 'encode',
10   'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',
11   'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier',
12   'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
13   'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
14   'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
15   'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
16   'swapcase', 'title', 'translate', 'upper', 'zfill']
17 >>> help(s.upper)
18 Help on built-in function upper:
19
20 upper(...)
21     S.upper() -> str
22
23     Return a copy of S converted to uppercase.
24
25 >>> s.upper()
26 'HELLO!'
```

One of the strings in the list returned by `dir()` is highlighted. This is the `upper()` method. We already know that `help()` can be used to provide information on various things, including built-in functions. Similarly, we can often use `help()` to obtain information about a class's methods. In line 17, we see how we can obtain help on the `upper()` method: We provide an instance, a

dot, and then the method of interest (without parentheses—if we included parentheses we would invoke the method). The output in lines 18 through 24 indicates that this method returns a copy of the string, but all in uppercase. Lines 25 and 26 show this works as advertised!

We will not provide further examples of the `dir()` function here, but it is worth mentioning that one doesn't have to create a variable first in order to see the list of attributes and methods associated with a class. One can obtain the same output shown in Listing 5.5 by writing, for instance, `dir(str)` (where the class name is the parameter) or `dir("hello!")` or even just `dir("")` (where the latter provides an empty string as the parameter). If you are curious about the attributes and methods for an integer or `float`, you can, for example, type `dir(42)` or `dir(1.0)`, respectively.

Before moving on to the next section of this chapter, we want to mention that methods can be applied to literals. This is illustrated in Listing 5.6. If you enter `dir(int)` you will see that one of the methods in the integer class is `bit_length()`. This returns the minimum number of binary digits (bits) needed to represent an integer. However, if we want to know the "bit length" of the number `42`, we cannot write `42.bit_length()` because "`42.`" looks like a `float`. Instead, we have to enclose the numeric literal in parentheses as shown below.

Listing 5.6 Demonstration that methods can be applied to literals.

```
1 >>> "hello!".upper()   # Convert "hello!" to uppercase.
2 'HELLO!'
3 >>> # Determine minimum number of bits to represent 42.
4 >>> (42).bit_length()
5 6
```

5.5 The `__init__()` Method

We now return to the `__init__()` method mentioned in connection with Listing 5.4.[9] (Depending on the display device you use to read this material, it may not be obvious, but this method starts and ends with *two* underscores.) If a programmer defines this method, it is called automatically when an object is created. Hence `__init__()` serves as the ***init***ialization method. Using this method we can set the values of an object's attributes when the object is created (this is in contrast to the examples in Figs. 5.2 and 5.3 where the objects are created first and then their attributes are set using additional statements). Any parameters given to the `class` are passed along to `__init__()`. This is best illustrated with an example.

In Listing 5.7 we have a new `Patient class` statement. It may appear we have eliminated the `name`, `age`, and `malady` attributes. However, we have just moved their creation to statements inside the `__init__()` method that starts on line 2. In this example the `__init__()` method takes four parameters. In general `__init__()` must have at least one parameter but could have arbitrarily more. We have no choice about the first parameter: Python dictates that it is the object under creation, i.e., it is the object itself. The other three parameters are used to specify the desired

[9] Technically `__init__()` is a "method wrapper" but as far as we are concerned it is indistinguishable from a method.

name, age, and malady. The values that are passed as parameters are assigned to the attributes of the object itself in lines 4 through 6. When we have a statement such as

```
self.<attribute> = <value>
```

this automatically creates the attribute (if it didn't exist before) and assigns "<value>" to it. If the attribute previously existed, its old value is replaced with the new value to the right of the assignment operator.

Listing 5.7 A new implementation of the Patient class (previous implementations were given in Figs. 5.2 and 5.3). Here the __init__() method is used to initialize attributes when an object is created. The display() method is unchanged from Listing 5.3. A cure() method has also been added to the class.

```
1  >>> class Patient:
2  ...     def __init__(self, name, age, malady):
3  ...         """Initialize the Patient's name, age, and malady."""
4  ...         self.name   = name
5  ...         self.age    = age
6  ...         self.malady = malady
7  ...
8  ...     def display(self):
9  ...         """Display the Patient's attributes."""
10 ...         print("Name   = ", self.name)
11 ...         print("Age    = ", self.age)
12 ...         print("Malady = ", self.malady)
13 ...
14 ...     def cure(self):
15 ...         """Cure the Patient."""
16 ...         self.malady = "healthy"
17 ...
18 >>> sally = Patient("Sally Smith", 21, "bruised ego")
19 >>> sally.display()
20 Name   =  Sally Smith
21 Age    =  21
22 Malady =  bruised ego
23 >>> sally.cure()
24 >>> sally.display()
25 Name   =  Sally Smith
26 Age    =  21
27 Malady =  healthy
```

After defining the Patient class, we create the Patient sally in line 18. In this case, we set her name, age, and malady when she is created. Although we are not explicitly invoking the __init__() method, Python calls it for us. All the parameters given to Patient are passed along to __init__(). So, although it isn't strictly true, when we write

```
sally = Patient("Sally Smith", 21, "bruised ego")
```

in many respects we can think of it as being equivalent to a function call that looks like

```
__init__(sally, "Sally Smith", 21, "bruised ego")
```

Line 19 uses the `display()` method to show `sally`'s attributes. In line 23 we use the new `cure()` method to bring `sally` back to a healthy state. Line 24 again uses `display()` to show that `sally` has now recovered.

5.6 Operator Overloading

If you are shown a plus sign and asked what it does or what it represents, the first answer that springs to mind will probably pertain to the addition of two numbers. But, when you think about it a bit more, you realize the plus sign can mean different things in different contexts. For example, if you see `+5.234`, the plus sign is not indicating a sum: rather, it is indicating `5.234` is a positive number. Graffiti that says "*pat* + *chris*" indicates the existence of a couple where there is a lot of room for further interpretation. So, we should recognize that not only is the plus sign a convenient and familiar symbol, it can mean different things in different contexts.

Although we haven't dwelt on it, we've already seen that symbols in computer languages can mean different things in different "contexts." We've seen that a plus sign can mean addition (with the need for two operands) but it can also indicate sign (with a single operand). We've seen that a minus sign can mean subtraction, but it can also indicate sign (as in `-5.234`) or produce a change in sign (as in `-x`).

We've seen that a plus sign can be put between two `int`s or two `float`s or between an `int` and a `float`. We typically think of these operations as simply the sum of two numbers, but because of the way numbers are represented in a computer and the implementation of the underlying hardware, summing two `int`s requires actions on the part of the computer that are different from the actions involved in summing two `float`s. Also, when we sum an `int` and `float`, the computer must first *promote* the `int` to a `float` and then perform the sum. In Python, when we place the plus sign between two operands, the interpreter is aware of the class to which the operands belong (or, said another way, the types of the operands). Thus, Python will do "the right thing" whether we are summing `int`s or `float`s or anything else. In some cases, "the right thing" is to report an error if, for example, we try to sum an `int` and a string.

When we say a symbol can mean different things in different contexts, we take it as understood that the "context" is determined by the associated operand or operands for the symbol. For example, when a plus sign appears between two integers, the context is one involving integers and the plus sign needs to be interpreted in a way that is appropriate for integers. When a plus sign appears between an integer and a `float`, the context is now one of mixed types and the plus sign must be interpreted in a way that is appropriate for these specific types. With OOP, it is typically possible to specify what a symbol means or does in a given context, e.g., the programmer can dictate what should be done when a plus sign is used in a particular context. Defining a new interpretation of a symbol and how it should behave in a given context is known as *operator overloading*. We won't consider how one implements operator overload, but we do want to be aware of its existence.[10]

[10]Although the details aren't import to us and hence won't be covered, we will say that when objects appear to the left and right of a plus sign, Python will call the `__add__()` method for the class of the object to the left of the operand and provide both objects as parameters to this method. So, for example, an expression such as "`obj1 + obj2`" is

We have seen (line 1 of Listing 3.2) that we cannot "add" a string and a numeric value. The fact that such an operation produces an error intuitively makes sense. But, can we "add" two strings? Can we "multiply" a string by an integer or by a `float`? At this point we can only guess the answers, but we can certainly try these operations and see what happens. The results, shown in Listing 5.8, may seem odd at first, but with a bit of explanation (which is provided below the listing) they make sense and demonstrate some of the power and flexibility of Python.

Listing 5.8 Demonstration of operator overloading.

```
1  >>> s1 = "Hello" + " World!" # Concatenation.
2  >>> print(s1)
3  Hello World!
4  >>> s2 = "&" * 70            # Repetition.
5  >>> print(s1)
6  &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
7  >>> 10 * "EI" + "-OH!"       # Repetition and concatenation.
8  'EIEIEIEIEIEIEIEIEIEI-OH!'
9  >>> s3 = "#" * 70.0
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in <module>
12 TypeError: can't multiply sequence by non-int of type 'float'
```

The expression on the right side of line 1 has a plus sign with two string operands. When Python sees a "sum" of two strings, the plus sign is interpreted as the *concatenation* operator, i.e., it produces a new string that is obtained by *adding* the second string to the end of the first string. (To concatenate things is to link them together.) The output of the `print()` statement in line 2 shows that `s1` is indeed the concatenation of the two strings.

The expression on the right side of line 4 has the multiplication symbol with operands of a string and an integer. In this case Python interprets the multiplication as implying repetition. So, in this case the variable `s2` is assigned the string value consisting of `&` repeated 70 times. Lines 7 and 8 show another example of the repetition and concatenation operators.

Line 9 shows Python does not allow multiplication of a string and `float`. It makes sense to repeat a string an integer number of times but it generally doesn't make sense to repeat a string a fractional amount. Thus, Python requires that repetition be expressed as the product of a string and an `int`.

In the coming chapters we exploit the operator overloading already built into Python. This overloading is part of the power of OOP—the language is aware of the classes of the operands and will do what is appropriate for the given combination of operands and symbols.

5.7 Take-Away Message

This chapter seemingly provides quite a bit of material and the initial exposure to OOP nomenclature can be, shall we say, challenging. *However*, there are really just three messages you need to

evaluated by invoking `__add__(obj1, obj2)` where `__add__()` is defined in `obj1`'s class statement. There are similar *special methods* associated with the other symbols, such as `__mul__()` for the `*` symbol.

take away from this chapter. First, you should remember that instead of always invoking functions with an expression such as

```
<func>(<params>)
```

we will sometimes work with methods for which we write expressions of the form

```
<object>.<method>(<params>)
```

where <object> is a variable, literal data, or, in fact, any expression that evaluates to an object. Second, remember that the `dir()` function can be useful for reminding you of the attributes and methods in a particular class. And, third, remember that the meaning of a symbol can change depending on the class (or type) of the operands.

5.8 Chapter Summary

An *object* is a collection of *attributes* (data) and *methods* (functions).

Objects are said to be instances of a *class*.

A **`class`** statement provides a blueprint for creating objects.

In Python all data are objects. An object's type corresponds to its class.

To access an object's attributes or methods, one writes the object followed by the *access attribute operator*, i.e., a dot (`.`), followed by the desired attribute or method. (Keep in mind that the dot is an operator and cannot be part of an identifier.)

`dir()`: provides a listing of the attributes and methods of an object.

Operators, such as the plus sign, can be *overloaded* so that the operation performed depends on the types of the operands. For example, strings can be *concatenated* if they are "added" together.