BUILDING BLOCKS OF PROGRAMMING

LEARNING OBJECTIVES

- Explain good programming practice guidelines
- Describe variables and commonly used data types in Python
- Write code statements that assign values to variables
- Classify types of errors that can occur in Python code
- Develop functions that contain code segments
- Write code statements using Python functions

INTRODUCTION

Programming in the Python language uses instructions to tell the computer what actions you want performed. This chapter begins by discussing good programming practice. We then explain the basic elements of Python code, beginning with Python keywords, objects, operators and delimiters, data types, and variables. We explain comments and assignment statements next. Python code statements must comply with strict syntax rules in Python and the incorrect specification of code statements result in errors. This chapter explains the three types of errors that often result. Following the discussion of error types, we introduce functions. The packaging of Python code into smaller components helps to organize code and simplify the identification and resolution of errors. We also illustrate commonly used Python built-in functions and their usage. The chapter concludes with the development of a code module used by Python code in a different file.

GOOD PROGRAMMING PRACTICE

Many references provide advice and recommendations for best practices in programming. We will discuss some basics that can influence your development practices as you learn more about Python. Best practices do not just apply to Python code, and in fact, there are entire books on the subject (i.e., Martin, 2009).

To begin, consider who else will be reading your code. Is it a coworker or classmate with whom you might work on the same project? Is it someone grading your homework or project? On the other hand, is it your future self, as you may be working on a long-term project? Consider whether your code will be easy to follow and understand to whomever may be reading your code after you write it (Kaefer, 2018).

To facilitate easy-to-understand code, be sure to use descriptive variable names. While variable names like `a`, `i`, or `x` may be simple to type and illustrate concepts, consider naming your variables after what they represent or their use. For example, `customer_name` is better than `c`, and `survey_respondant_age` is better than `x`. Be sure to add plenty of helpful comments in your code. Leave comments before functions to explain what they do. Include comments throughout code that may be complex or counterintuitive on a first glance. Above all, be consistent. Avoid using different styles throughout your code and follow any style guidelines that may be in place in your organization.

Good programming practice is not simply about the readability of your code. A programmer should always consider whether there is a better way to do something. Does your code require doing the same thing repeatedly? A loop is a logical response (we cover loops in Chapter 4). Also, consider breaking the code into functions and having a loop call the function. These are the basics of writing modular code. As the number of lines of code increases, you may want to break it up into modules. The more you interact with code from different sources and read code written by different people, the more clear it will be why following good practices results in better communication among programmers and fewer headaches when trying to debug problems (Martin, 2009; van Rossum, Warsaw, & Coghlan, 2001).

Lessons learned: In this section, we learned about some practices to follow when developing code, which becomes very important when you work with other people on programming projects.

BASIC ELEMENTS OF PYTHON CODE

The Python programming language contains basic elements for specifying instructions. When a Python instruction is executed, the interpreter processes the instruction as a command, which is telling the processor what action it needs to perform. Some basic commands include assigning a value to a variable, adding two values together, and, as we saw in our first Python program in Chapter 1, printing a message to the console for a user to read. In order for the interpreter to operate efficiently, a limited number of basic elements are used to specify instructions. This section briefly introduces elements, including Python keywords, objects and classes, operators and delimiters, data types, and variables.

Python Keywords

Python keywords are identifiers that are reserved words that the interpreter uses for very specific purposes. As we will see in the chapter, we can create and name variables and functions as part of the instructions that we are writing. However, we are not allowed to use any of the Python reserved words for the names of variables or functions that we are creating. We list some

commonly used Python keywords in Table 2.1. We will use all the keywords listed in Table 2.1 within this book. The official list of all Python keywords is in the Python documentation (Python Software Foundation, 2019, "Keywords").

TABLE 2.1 ⬢ COMMONLY USED PYTHON KEYWORDS

`False`	`Except`
`None`	`for`
`True`	`from`
`and`	`global`
`as`	`if`
`break`	`import`
`continue`	`in`
`def`	`is`
`elif`	`not`
`else`	`or`

Objects and Classes

Objects are the building blocks of Python, which is an object-oriented programming language. Objects or relations between objects represent all data in a Python program (Python Software Foundation, 2019, "Data Model"). Python classes provide all the standard features of object-oriented programming, and classes provide a means of bundling data and functionality together (Python Software Foundation, 2019, "Classes"). In other words, you use **classes** to create objects and objects can have functions associated with them. Classes are used to create different types of objects that have specific data types and specific actions associated with them. For example, an object can be numeric and an action that can be performed with that object is to add another number to it. A different class of object would be one that had text-based values. An action that may be associated with that type of object could be to make it all uppercase (made up of all capital letters). This action would not apply to a numeric valued object. A **method** is an action that you can perform to or with an object. We will explain methods in more detail in the next chapter and see examples of different methods that are used with different types of objects. To learn more about the inner workings of Python, please see The Python Language Reference found at **https://docs.python.org/3/reference/index.html**.

Operators and Delimiters

Operators and delimiters are special symbols that the Python interpreter uses to perform operations and to separate items. Both operators and delimiters work with other elements. Table 2.2 shows commonly used operators and delimiters. We will use these delimiters and operators within this book. For example, the "+" operator is used when we want to add two values together and the "," delimiter is used to separate two items. The interpreter needs to have something (either an operator or a delimiter) between two distinct items, or it will not understand what it is being instructed to do with those items. The Python Language Reference specifies additional Python operators and delimiters (**https://docs.python.org/3/reference/index.html**).

TABLE 2.2 ⬢ COMMONLY USED PYTHON OPERATORS AND DELIMITERS

Operators	Delimiters
+	(
-	)
*	[
**	]
/	{
<	}
>	,
<=	:
>=	;
= =	.
! =	=

Data Types

A **data type** determines what kind of values a piece of data can have and what kind of operations you can perform on the data. Table 2.3 presents the commonly used data types in Python.

TABLE 2.3 ⬢ PYTHON DATA TYPES

Data Type (Python Type)	Description
Boolean (bool)	Used to store either `True` or `False` values.
Integer (int)	Used to store integers, which are whole numbers.
Float (float)	Used to store floating-point numbers, which can have decimals.
String (str)	Used to store text values such as "12 Main St., Anytown, USA."

Boolean data-type variables can store either `True` or `False` logical values. We will use variables of this data type in Chapter 4 when evaluating conditions that involve comparisons. Programmers use **Integer** data-type variables to store whole numbers. Variables of this data type are very useful for counting. **Float** data-type variables are also known as floating-point variables and can store decimal values. This type of variable is useful for situations where there can be fractional values, which often result when dividing numbers. Programmers use **string** data-type variables to store text that consists of letters, special symbols, and numbers. You enclose strings in quotes to reference them in code. Strings are very useful for composing messages to communicate to users and for creating labels used when formatting output results.

Variables

We use **variables** to store and access values that we are working with. The values of variables can change, or vary, as a program executes. There are some restrictions with naming of variables, including (1) they cannot have the same name as a Python keyword, (2) they cannot have any spaces or operators or delimiters within them, and (3) they cannot begin with a number.

You do not formally define variables in Python (which most programming languages require). In Python, you create a variable when you first use it. The variable name that you use when something is created refers to an object that is constructed for the purpose that you are specifying. The value that you store in a variable determines the variable's data type and the actions that can be performed to or with the corresponding object that it refers to.

Lessons learned: In this section, we learned about basic elements of Python code, including Python keywords, objects and classes, variables and data types, and operators and delimiters. From this point onward, we will combine these basic elements into various statements that make up our code.

PYTHON CODE STATEMENTS

Now that we are familiar with basic elements of Python code, we can begin to write instructions using these elements. However, there are several additional details to discuss first, including comments and variable assignment.

Comments

Comments are very important for documenting Python code. **Comments** in Python code begin with a pound sign (#), and the interpreter does not process them. Comments explain how the code works for those reading the code and can be an entire line or just a note at the end of a line of code. Multiple line comments (also known as documentation strings) can be written beginning with three quotes (“““) and ending with three quotes.

Variable Assignment

Assignment statements specify what value something is to take. You read assignment statements in Python code from right to left, storing the value on the right-hand side of the equal sign in the variable on the left-hand side of the equal sign, following the syntax:

variable_name = value

For example, `taxi_number = 333` for an integer variable or `my_name = "John Doe"` for a string variable. If you want one variable to contain the same values as another variable, you can do so like in the following code statement: `other_survey_values = survey_values`.

Code Examples

We now present a couple of code examples to illustrate the use of the elements just discussed. Figure 2.1 illustrates some Python code statements in a file (Fig 2_1 Python code example 1.py).

FIGURE 2.1 ⬢ PYTHON CODE EXAMPLE 1

```
1  # This is a comment line that begins with a pound sign "#"
2
3  # Example 1a: assigning a string value to a variable creates a string object type variable
4  trip_id = "da7a62fce04"    # This assignment is assigning a string to a variable named trip_id
5  print("The data type for the trip_id variable is: ", type(trip_id))
6
7  # Example 1b: assigning an integer value to a variable creates an integer object type variable
8  trip_seconds = 180
9  print("The data type for the trip_seconds variable is: ", type(trip_seconds))
```

Several details in the Python code in Figure 2.1 are fundamental to writing programs in Python. Lines 1, 3, and 5 have comments that take up an entire line, and line 4 shows an example of a comment that makes a note at the end of a line of code. Line 4 is an assignment statement in which the right-hand side is a value in quotes. The value within quotation marks is a string-type value. When the code in line 4 executes, the `trip_id` variable will be a string. Line 5 illustrates the use of two built-in functions to report the data type of the just created `trip_id` variable. Specifically, a type function is within a `print` function. We already used the Python built-in `print` function in the example presented in Chapter 1. The **type function** in Python determines the data type of an object. Line 8 of the Python code in Figure 2.1 is an assignment statement in which the right-hand side is a number. When the code in line 8 executes, the `trip_seconds` variable will be an integer. The `type` function is also in line 9, which reports the data type for the just constructed `trip_seconds` variable. The output of these code statements is in Figure 2.2.

FIGURE 2.2 ⬢ OUTPUT FROM EXECUTION OF PYTHON CODE IN FIGURE 2.1

```
================ RESTART: I:/Fig 2_1 Python code example 1.py ==
The data type for the trip_id variable is:  <class 'str'>
The data type for the trip_seconds variable is:  <class 'int'>
```

Examining the output shown in Figure 2.2, we see that the data type for the `trip_id` variable is a string and the data type for the `trip_seconds` variable is an integer. The output messages specifically indicate that `trip_id` is of class "str" and that `trip_seconds` is of class "int."

Our next example is like the first example but illustrates the construction of float and Boolean variables. Line 2 in Figure 2.3 assigns the value 1.1 to the variable `trip_miles` and line 3 prints out the data type of the `trip_miles` variable. Line 6 assigns the value `True` to the variable `trip_completed`. Note that `True` is a Python keyword shown in Table 2.1 and is in blue font in Figure 2.3. Also note that there are not parentheses around `True`, because if there were, it would be a string instead of the logical value `True`. Line 7 then prints out the data type of the `trip_completed` variable.

FIGURE 2.3 ⬢ PYTHON CODE EXAMPLE 2

```
1  # Example 2a: assigning a float value to a variable creates a float object type variable
2  trip_miles = 1.1
3  print("The data type for the trip_miles variable is: ", type(trip_miles))
4
5  # Example 2b: assigning a boolean value to a variable creates a boolean object type variable
6  trip_completed = True
7  print("The data type for the trip_completed variable is: ", type(trip_completed))
```

Examining the output shown in Figure 2.4, we see that the data type for the `trip_miles` variable is in fact a float and the data type for the `trip_miles` variable is a Boolean.

FIGURE 2.4 ⬢ OUTPUT FROM EXECUTING CODE EXAMPLE 2

```
================ RESTART: I:/Fig 2_3 Python code example 2.py ====
The data type for the trip_miles variable is:  <class 'float'>
The data type for the trip_completed variable is:  <class 'bool'>
```

STOP, CODE, AND UNDERSTAND!

SCU 2.1 Variable Assignment

Download the file "SCU 2_1.py" from the companion website and save it either on your computer or on a removable storage device. Open the file in the Python IDLE editor and execute the program to see that it creates a variable that is a float data type. Add quotation marks around the value 12.5 in the assignment statement in the line indicated so that the type of variable created is a string data type. Execute the modified program to verify that the revised code creates a variable with a string data type.

```
1  # modify the following line of code so that a string data type variable is created
2  variable_created = 12.5
3  print("The data type of the variable created is: ", type(variable_created))
```

Mathematical Expressions

In the previous section, we saw several examples where assignment statements created variables of different data types based on the values assigned to them. In addition to literal values, we use mathematical expressions to assign values to variables. When writing mathematical expressions, we use arithmetic operators, shown in Table 2.4. **Arithmetic operators** include symbols to add, subtract, multiply, divide, and exponentiate (raise to a power) values.

TABLE 2.4 ⬢ PYTHON ARITHMETIC OPERATORS

Symbol	Purpose
+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Integer division
**	Exponentiation

When writing mathematical expressions, you must be careful with the order of operations, which follow the PEMDAS rule (**P**arenthesis first, then **E**xponentiation, next **M**ultiplication and **D**ivision, and then **A**ddition and **S**ubtraction). Figure 2.5 illustrates how adding two numbers and multiplying by a third number can lead to two different results.

Figure 2.6 shows the output that results from the execution of the Python code in Figure 2.5. The first result (`result1`) of the value 20 reflects the fact that the multiplication of the second and third numbers occurs first before the addition of the first number to the result of the multiplication. The second result (`result2`) of the value 24 reflects the fact that the addition of the first two numbers occurs before the multiplication of the intermediate result and the third number. This example illustrates why we must be careful when applying mathematical expressions. When in doubt, we can use extra parentheses to make sure that specific calculations occur before others within an expression.

FIGURE 2.5 ⬢ PEMDAS EXAMPLE

```
# Create a few variables with numeric values
first_number = 4
second_number = 8
third_number = 2

# Two assignment statements with mathematical formulas using the variables
result1 = first_number + second_number * third_number
result2 = (first_number + second_number) * third_number

# The following line of code displays the values of the results
print("Result1: ",result1, "Result2: ", result2)
```

FIGURE 2.6 ⬢ OUTPUT FROM EXECUTION OF PEMDAS EXAMPLE

```
========== RESTART: I:\Fig 2_5 PEMDAS example.py ==
Result1:  20 Result2:  24
```

STOP, CODE, AND UNDERSTAND!

SCU 2.2 Mathematical Expressions

Download the file "SCU 2_2.py" from the companion website and save it either on your computer or on a removable storage device. Open the file in the Python IDLE editor and add parentheses around two values, a subtraction operator, and an exponentiation operator to the code below so that the code prints the value 25. Execute the modified program after the change to verify that the revised code runs and produces the correct result.

```
# Add parentheses, a subtraction operator, and an exponentiation operator
#  so the following code prints the value 25.
print(  8  3  2  )
```

Lessons learned: In this section, we learned about how to use variables in assignment statements, using `print` statements to visualize our results. We also learned how to use the Python `type` function to determine and report the data type of an object in a Python program and why the order of operations is important when we have Python code that performs calculations.

ERRORS

Writing Python code often results in three types of errors: syntax errors, exceptions, and logic errors. We explain and illustrate each of these error types in the following sections.

Syntax Errors

Syntax errors occur when the Python code does not follow the rules that dictate how to write Python code statements. The interpreter identifies syntax errors and highlights the cause of the syntax error in red font in the Python shell window, as shown in Figure 2.7. When a syntax error exists in Python code, the code will not execute until you resolve the syntax error. The cause of the syntax error in this example is that there is nothing combining the string and the variable symbol within the print statement (such as a comma ",").

FIGURE 2.7 ⬢ PYTHON SYNTAX ERROR

```
value_entered = input("please enter a number between 1 and 100: ")

print("The value that you entered was: " value_entered)
```

STOP, CODE, AND UNDERSTAND!

SCU 2.3 Syntax Errors

Download the file "SCU 2_3.py" from the companion website and save it either on your computer or on a removable storage device. Open the file in the Python IDLE editor and execute the program to see that it throws a syntax error when run. Read the text of the error and fix the code as necessary. Execute the modified program after the change to verify that the revised code runs and produces the correct result.

```
1  # The following code throws a syntax error. Read the error message and fix the code.
2  print "I would like to order a taxi."
```

Exceptions

Exceptions occur during Python code execution when you attempt an action that is not possible or not allowed. Figure 2.8 illustrates an example of Python code that results in an exception. Figure 2.9 has the output that corresponds to the execution of the code in Figure 2.8. Due to the fact that the `print` function was "Print" and not "print," a NameError exception occurs. The Python interpreter reports that the "name 'Print' is not defined," or in other words, there is no function with that name. Python programmers encounter this type of exception frequently due to Python's case sensitivity (discussed in Chapter 1).

FIGURE 2.8 ⬢ PYTHON CODE WITH EXCEPTION

```
1
2  value_entered = input("please enter a number between 1 and 100: ")
3
4  Print("The value that you entered was: ", value_entered)
```

FIGURE 2.9 ⬢ OUTPUT FOR PYTHON CODE WITH EXCEPTION

```
============= RESTART: I:\Fig 2_8 Python code with exception.py ========
please enter a number between 1 and 100: 50
Traceback (most recent call last):
  File "I:\Fig 2_8 Python code with exception.py", line 4, in <module>
    Print("The value that you entered was: ", value_entered)
NameError: name 'Print' is not defined
```

Figure 2.10 illustrates another Python code example that results in a different type of exception. Figure 2.11 has the output that corresponds to the execution of the code in Figure 2.10. The exception that occurs is a `TypeError` exception, which occurs because line 3 of the Python code in Figure 2.10 attempted to add an integer valued variable and a string valued variable together. The Python interpreter reports which line (line 3) was involved and prints out that line in the error message "`result = first_number + second_number`." In addition, the explanation is "unsupported operand type(s) for +: 'int' and 'str.'" It has been our experience that beginning Python programmers find these error messages puzzling (and sometimes frustrating), but as they learn the needed terminology and gain experience, the error messages become more helpful and resolving the issues becomes much easier.

FIGURE 2.10 ⬢ PYTHON CODE WITH TYPEERROR EXCEPTION

```
1  first_number = 10
2  second_number = "Ten"
3  result = first_number + second_number
```

FIGURE 2.11 ⬢ OUTPUT FOR PYTHON CODE WITH TYPEERROR EXCEPTION

```
======== RESTART: I:\Fig 2_10 Python code with TypeError exception.py ========
Traceback (most recent call last):
  File "I:\Fig 2_10 Python code with TypeError exception.py", line 3, in <module>
    result = first_number + second_number
```

Table 2.5 identifies some common Python built-in exceptions. We will see examples of these and other exceptions later in the book. Chapter 5 has an example that uses an installed package to show a variety of exceptions that can occur (and how to resolve them) as Python programs become more involved. References for other Python built-in exceptions are in the official Python documentation (Python Software Foundation, 2019, "Built-in Exceptions").

TABLE 2.5 ⬢ SOME COMMON PYTHON BUILT-IN EXCEPTIONS

Python Built-in Exception	Description
`IndexError`	Occurs when a subscript of a sequence has a value outside the range of the sequence.
`NameError`	Occurs when you use an undefined name.
`TypeError`	Occurs because of a type mismatch.
`ValueError`	Occurs when you pass an argument to a function of the correct type but whose value is improper.
`ZeroDivisionError`	Occurs when you divide by zero.

STOP, CODE, AND UNDERSTAND!

SCU 2.4 Exceptions

Download the file "SCU 2_4.py" from the companion website and save it either on your computer or on a removable storage device. Open the file in the Python IDLE editor and execute the program to see that it results in an exception. Remove the quotation marks around the value in the assignment statement in the line indicated to resolve the issue. Execute the modified program to verify that the revised code executes properly and that no exception occurs.

```
first_number = 10
# Remove the quotation marks in the following line to resolve issue
second_number = "20"
result = first_number + second_number
print ("The result is: ", result)
```

Logic Errors

The third category of errors, logic errors, is often the most difficult to identify. A **logic error** occurs when a program executes without terminating with an error condition but produces incorrect results. Logic errors can result from using an incorrect operator in an equation or using parentheses in the wrong location. An example of Python code with a logic error is in Figure 2.12.

FIGURE 2.12 ⬢ PYTHON CODE WITH LOGIC ERROR

```
# Create a few variables with numeric values
first_number = 4
second_number = 8

# Calculate the average of the numbers by adding them and dividing by 2
average = first_number + second_number /2

# The following line of code displays the result
print("The average of ",first_number," and ",second_number," is ",average)
```

FIGURE 2.13 ⬢ OUTPUT FOR PYTHON CODE WITH LOGIC ERROR

```
========== RESTART: I:\Fig 2_12 Logic error example.py ===
The average of  4  and  8  is  8.0
```

This example demonstrates how easy it is to develop code that executes but does not produce the correct results. Figure 2.13 prints out the message that the average of 4 and 8 is 8! The true average of 4 and 8 is 6 and results when you add the two numbers together prior to dividing the sum by 2. Reviewing the order of operations discussed earlier, we need to put parentheses around the addition of `first_number` and `second_number` to correct this error, which adds the numbers together first. As it is now (without those parentheses), we first divide `second_number` by 2 (resulting in the value 4) and then add that result to `first_number` (resulting in a final value of 8), which is incorrect. To ensure that code executes with the correct results, programmers need to develop test cases and verify that the expected outcomes for each test case do in fact occur.

STOP, CODE, AND UNDERSTAND!

SCU 2.5 Logic Errors

Download the file "SCU 2_5.py" from the companion website and save it either on your computer or on a removable storage device. Open the file in the Python IDLE editor and execute the program to see that it runs but does not have the correct result. Add parentheses around the two values being added in the assignment statement in the line indicated to resolve the issue. Execute the modified program to verify that the revised code runs and produces the correct result.

```
1  first_number = 4
2  second_number = 8
3
4  # Modify the following line by putting parenthesis around the two variables added
5  average = first_number + second_number /2
6
7  print("The average of ", first_number, " and ", second_number, " is ", average)
```

Lessons learned: In this section, we learned there are three types of errors in Python: syntax errors, exceptions, and logic errors. Syntax errors are the easiest types of errors to resolve and result when we do not follow the rules for correctly specifying Python code. Exceptions are the types of errors that occur when we attempt to do something in Python that is not possible or not allowed. Logic errors occur when code executes without terminating with an error message but has incorrect results. We will encounter each of these error types often as we develop code but will learn and become better at diagnosing them and improving our code.

FUNCTIONS

Python Built-in Functions

Python comes with over 60 built-in functions (Python Software Foundation, 2019, "Built-in Functions"). For example, the `max()` function will return the maximum value of a list of numbers. If you execute the Python code statement "`max(1, 2)`," the value 2 is returned. A list of built-in functions may be found at **https://docs.python.org/3/library/functions.html**. Table 2.6 presents some of the commonly used built-in functions along with a brief description of each.

TABLE 2.6 COMMONLY USED PYTHON BUILT-IN FUNCTIONS

Built-in Function	Description
`float`(*value*)	Returns a floating-point equivalent of *value*, which is a string or numeric value such as an integer.
`input`(*message*)	Prompts user to enter something and returns the value they entered as a string.

`int`(*value*)	Returns an integer corresponding to *value*, which is a string or floating-point value. Truncates the decimal component of *value*.
`len`(*object*)	Returns the number of items in an object, where *object* can be a string, tuple, list, or other collection.
`max`(*arguments*)	Returns the largest item from a set of arguments.
`min`(*arguments*)	Returns the smallest item from a set of arguments.
`round`(*number* [,*numDigits*])	Returns *number* rounded to *numDigits* precision after the decimal point. When we omit *numDigits*, the `round` function returns the nearest integer.
`str`(*value*)	Returns the string version of a *value*.
`sum`(*sequence* [,*start*])	Returns the sum of a *sequence* (optionally beginning at a specified *start* location).
`type`(*object*)	Returns the data type of an object.

We already have made use of the `type` function in the examples shown in Figures 2.1 and 2.3 earlier in the chapter. An example from our Taxi Trips data set will help to illustrate the usage of some of the other Python built-in functions. Referring to Table 1.2 and Table 1.3, several fields in the Taxi Trips data set involve taxi trip cost-related information (*Fare and Tips*). These fields have the data type "Money" in the SODA API (which is discussed later in the textbook), but there is not a Python basic data type "Money." If we assign the value $4.75 to a variable, an error will occur, because the Python interpreter doesn't recognize the usage of a dollar sign symbol. We will address this in the next chapter after we go further in depth with compound data types and strings.

For now, let us examine a Python program that prompts the user to enter in amounts for the taxi fare and tip amount for a taxi trip. The Python code in Figure 2.14 does this in lines 3 and 4 and then reports back to the user the amounts entered (in lines 5 and 6) as well as the data types of the variables (in lines 7 and 8). Note that code lines that span more than one physical line in the file have special symbols on the left-hand side in Figure 2.14. The output of these statements is in Figure 2.15. The data types for each of the variables are strings (even though the values entered appear to be numbers). In order to add these two values together, we must convert the values to a data type that supports addition. Line 11 of the code in Figure 2.14 shows one way to do this, by using the `float` built-in function to convert the string data-type representations of the numeric values into float data-type values and then adding those values and assigning the result of the addition to the variable `trip_total`. The `trip_total` variable will be a float data-type variable. An additional conversion is necessary to report the result back to the user. As we saw in Figures 2.10 and 2.11, if we try to combine two different types with the "+" operator, a `TypeError` exception will result.

When we use the "+" operator with two or more string data-type arguments, we concatenate the string components into a larger string. The string operation **concatenation** combines multiple strings into one string. For example, concatenating the two strings "Hey," and "Taxi!" and assigning the result to a variable would be performed using the statement `variable_name = "Hey," + "Taxi!"` Line 12 of the Python code in Figure 2.14 uses the `str` built-in function to convert the `trip_total` float value into a string and then uses the "+" operator to concatenate the "$" character with the string value. The last line of the output in Figure 2.18 displays the outcome of this `print` statement.

FIGURE 2.14 ⬢ PYTHON CODE TO ADD UP TRIP COSTS

```
# First prompt user to enter trip cost components to string variables
fare = input("Please enter the taxi fare: ")
tip = input("Please enter the tip amount: ")

print("The amounts entered for fare and tip was: ",
      fare, tip)
print("The data types for each are: ",
      type(fare), type(tip))

# now add up float values and assign to trip_total variable
trip_total = float(fare)+float(tip)
print ("The total trip cost is: ","$" + str(trip_total))
```

FIGURE 2.15 ⬢ OUTPUT FROM EXECUTION OF PYTHON CODE TO ADD UP TRIP COSTS

```
========== RESTART: I:\Fig 2_14 program to add up trip costs.py ==
Please enter the taxi fare: 5.25
Please enter the tip amount: 2.00
The amounts entered for fare and tip was:  5.25 2.00
The data types for each are:  <class 'str'> <class 'str'>
The total trip cost is:  $7.25
```

STOP, CODE, AND UNDERSTAND!

SCU 2.6 Use a Built-in Function

Download the file "SCU 2_6.py" from the companion website and save it either on your computer or on a removable storage device. Open the file in the Python IDLE editor and add a function call around the string to print the length. Hint: Use the Python built-in function `len`. Execute the modified program after the change to verify that the revised code runs and produces the correct result.

```
# Add function call around the string to print the length. Hint: the function is called 'len'.
print("length of string")
```

User-Defined Functions

We have used small Python code examples up until this point to illustrate basic coding elements of Python. As Python programs become more complex and involve more lines of code, errors can be more difficult to identify, and the code can be more difficult to modify and to test. To address these issues, a programming best practice is to package code into small units, which are often one printed page or less. In Python, we accomplish this by placing portions of code into **functions,** which are subroutines of code developed to perform specific tasks.

Function Syntax

Functions play a very important role in Python programming, and as we will see in the coming chapters, the use of packages depends significantly on the use of the functions that are within those packages. To thoroughly understand how functions work, we need to carefully examine function syntax. We show the syntax for defining a function in Python in Figure 2.16.

FIGURE 2.16 ⬢ PYTHON FUNCTION SYNTAX

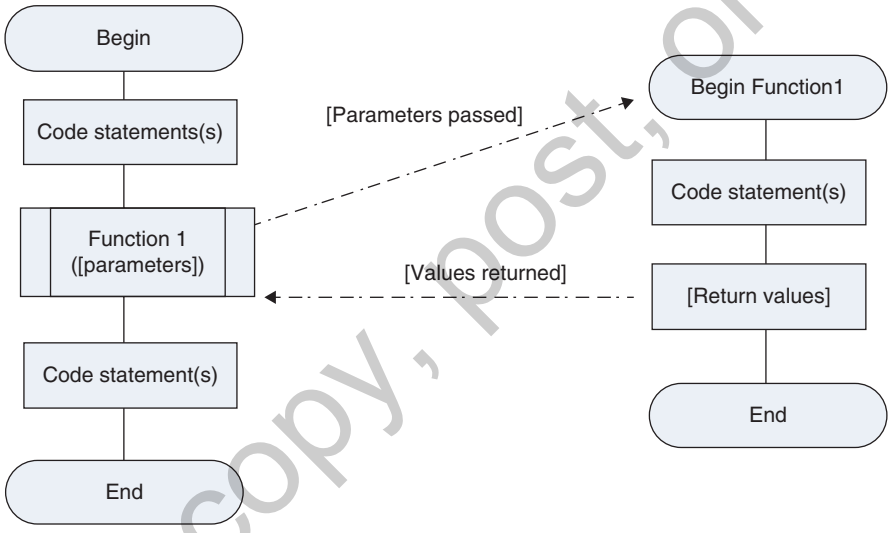
def *function_name* **(**[*parameter(s)*]**):**

code_statements

[**return** *value(s)*]

Syntax specifications involve several prominent features, including Python keywords in bold font, user-specified information in italicized font, and optional components specified within square brackets. The first line of every function definition in Python begins with the keyword **`def`**, followed by the function name and a set of parentheses in which you can specify an optional set of parameters, and then ends with a colon. The indented code statements that follow are part of the function. Optionally, a function can return one or more values. If you return more than one value, you need to separate each value with a comma. The specification of the function ends when a nonindented line of code occurs or if you reach the end of the file. We show a flowchart of the logic in using a function in Figure 2.17 to help visualize how a function works.

FIGURE 2.17 ⬢ FLOWCHART OF FUNCTION USAGE



The logic shown in Figure 2.17 shows that when you encounter a code statement that uses a function, you pass control to the function code. You can optionally pass parameters to the function, which the function uses in its processing. The function comprises code statements and optionally may return values back to the calling code. When the function ends, the control of code execution resumes in the calling code.

Using Functions

The simplest form of a function is one that does not have any parameters and does not return any values. Figure 2.18 shows an example of such a function that prints out a message. Functions in Python only execute when you invoke them. To invoke a function, you use the function name, along with the specification of function parameters (if there are any). Because the function defined in lines 1 and 2 of the Python code in Figure 2.18 does not have any parameters, empty parentheses follow the function name.

FIGURE 2.18 ⬢ SIMPLE FUNCTION WITH NO PARAMETERS

```
1  def hello():
2      print("Hello, John!")
3
4  hello()
```

Figure 2.19 illustrates the output that results when executing the Python code in Figure 2.18.

FIGURE 2.19 ⬢ OUTPUT OF PROGRAM IN FIGURE 2.18

```
========= RESTART: I:\Fig 2_18 simple function with no parameters.py ==
Hello, John!
```

Functions become more dynamic when you pass parameters to them. Actions performed by the statements in a function can change when it receives different information. Figure 2.20 shows an example of such a function that receives a name and prints out a different message depending on what name it receives.

FIGURE 2.20 ⬢ SIMPLE FUNCTION WITH PARAMETERS

```
1  def hello(name):
2      print("Hello, " + name + "!")
3
4  hello("John")
5  hello("Jane")
```

Figure 2.21 shows the output that results from executing the Python code in Figure 2.20. As you can see, the code prints two different messages, because we pass a different name to the function each time. This simple example illustrates the ability of functions to do different things depending on what information they receive.

FIGURE 2.21 ⬢ OUTPUT OF PROGRAM IN FIGURE 2.20

```
========== RESTART: I:\Fig 2_20 simple function with parameter.py ===
Hello, John!
Hello, Jane!
```

When functions have parameters, the possibility of errors that can occur increases. Two common errors are specifying the incorrect number of parameters and using incorrect data types of parameters. The Python code in Figure 2.22 illustrates such an error, where line 4 invokes the function but attempts to pass two parameters.

FIGURE 2.22 ⬢ ERROR USING FUNCTION WITH INCORRECT PARAMETERS

```
1  def hello(name):
2      print("Hello, " + name + "!")
3
4  hello("John", "Joe")
5  hello("Jane")
```

Figure 2.23 illustrates the error that occurs when executing the code in Figure 2.22. The error message is very helpful. The message identifies the line number and shows the code that caused the error. The last line of the error message specifies that a TypeError occurred and that the specific function takes one positional argument (parameter), and the function received two.

FIGURE 2.23 ⬢ OUTPUT OF PROGRAM IN FIGURE 2.22

```
=== RESTART: I:\Fig 2_22 error using function with incorrect parameters.py ===
Traceback (most recent call last):
  File "I:\Fig 2_22 error using function with incorrect parameters.py", line 4, in <module>
    hello("John", "Joe")
TypeError: hello() takes 1 positional argument but 2 were given
```

Functions can become even more useful when they return a value or values. To illustrate how returning a value works, the next example returns a string from the function and then prints the result that the function returns.

FIGURE 2.24 ⬢ FUNCTION RETURNING A VALUE

```
1  def hello(name):
2      return("Hello, " + name + "!")
3
4  message = hello("John")
5  print(message)
6  print(hello("Jane"))
```

The Python code in Figure 2.24 invokes the function two different times in two different ways. The code in line 4 invokes the function and assigns the value that the function returns to a variable named message. Then line 5 uses the `print` function to print out the value of the variable message. Line 6 uses the user-defined function within the `print` function. The output of this code execution is in Figure 2.25, illustrating that the different approaches yield similar results.

FIGURE 2.25 ⬢ OUTPUT OF PROGRAM WITH FUNCTION RETURNING A VALUE

```
============= RESTART: I:\Fig 2_24 function returning a value.py ==
Hello, John!
Hello, Jane!
```

Figure 2.26 illustrates an example of a function that receives two parameters and returns a value that is based on the data it receives.

FIGURE 2.26 ⬢ THE `find _ average` FUNCTION

```
1  # The following is the definition of the function Find_Average
2  def find_average(first_number, second_number):
3      return (first_number + second_number)/2
4
5  # The following is executable code
6  # First create a few variables with numeric values
7  number1 = 4
8  number2 = 8
9
10 # The following line of code displays the values of the results
11 print("The average of the two numbers is: ",find_average(number1, number2))
```

The Python code following the function definition in Figure 2.26 creates two integer variables in lines 7 and 8 and then invokes the `find_average` function within the code in line 11. Note that the two arguments that the function receives have different variable names than the names of the arguments used in the definition of the function. The arguments passed can take other forms as well, such as actual values or the result of some operation or calculation. The function only has one line of code, which is to return the value that results from dividing the sum of the two received arguments by the value 3. The output from executing the code in Figure 2.26 is in Figure 2.27.

FIGURE 2.27 ⬢ OUTPUT FROM EXECUTION OF `find _ average` FUNCTION

```
========== RESTART: I:\Fig 2_26 Find_Average function.py ==
The average of the two numbers is:  6.0
```

Function Location in Code

When Python code is in a text file, you can define functions anywhere within the file and in any order when there are multiple functions. However, it is important that the function definition occurs prior to the use of the function, and so a convention commonly followed

STOP, CODE, AND UNDERSTAND!

SCU 2.7 Modify a User-Defined Function

Download the file "SCU 2_7.py" from the companion website and save it either on your computer or on a removable storage device. Open the file in the Python IDLE editor and modify the function "`my _ function`" so it prints the average of x and y. Execute the modified program after the change to verify that the revised code runs and produces the correct result.

```
1  # modify the user-defined function so it prints the average of x and y:
2  def my_function(x, y):
3      print(x + y)
4
5  my_function(4, 5)
```

when writing Python code is to have all the functions defined at the beginning of the file and, if any executable lines of code in the file are not part of a function, to have those lines of code at the very end.

Lessons learned: In this section, we learned about a few functions that Python already has defined and that one common use of built-in functions is converting object data types from one type to another. We also learned about user-defined functions and how functions can have arguments (values passed to them) as well as return values. We saw how we must be careful when specifying arguments when we call functions or else errors will result.

PYTHON INSIGHT

Python uses indentation to differentiate levels of code. **Indentation** refers to the number of spaces and/or tabs at the beginning of a line of Python code. As we have already seen, function definition uses indentation to indicate what code is within the function. We will use indentation to an even greater degree in the next chapter when we cover control logic and loops, which use different blocks of code. Python 3 does not allow mixing of tabs and spaces for indentation (van Rossum et al., 2001). The recommendation is to use spaces for indenting (specifically four), unless one is working with code that already is using tabs for indentation, in which case tabs should be used consistently for indentation. This reduces the processing of the Python interpreter needed to treat all the possible combinations of tabs and spaces the same. To help visually ensure the proper indentation, text editors such as Notepad++ have features to show white space (spaces and tabs). In Notepad++, you can turn this feature on and off through the menu selections View/Show Symbol and clicking on the selection: "Show White Space and TAB."

USING MODULES OF PYTHON CODE

As discussed in Chapter 1, Guido van Rossum's vision for Python was that users could create their own coding modules and make those coding modules available to others. The previous section detailed how to create functions and to use them in Python code. In addition to using functions created within a Python file, you can use functions that others develop that are in other files. There are numerous modules available for Python, and users can even create their own. There are also built-in modules that come with Python. For example, the random module lets you generate pseudo-random numbers, which you can use to add an element of chance to a program or generate simulation data. Additionally, the fractions module enables the programmer to perform arithmetic on fractions stored as strings like "4/5," and the math module provides functions like sin, cos, and tan. We explain and give an example of using several useful packages later in this book. A searchable index of Python modules is available at **https://pypi.org/**.

We use functions in modules by importing modules. The Python code in Figures 2.28 and 2.29 illustrate how you accomplish this.

FIGURE 2.28 ⬢ MODULE WITH `find _ average` FUNCTION

```
1  # The following is the definition of the function find_average
2  def find_average(first_number, second_number):
3      return (first_number + second_number)/2
```

FIGURE 2.29 ⬢ FIND AVERAGE USING FUNCTION IN MODULE

```
1  import Fig_2_28_Find_Average_Module as fam
2
3  # The following is executable code
4  # First create a few variables with numeric values
5  number1 = 4
6  number2 = 8
7
8  # The following line of code displays the values of the results
9  print("The average of the two numbers is: ",fam.find_average(number1, number2))
```

Figure 2.28 is a Python module that has the `find_average` function defined within it. The Python module name uses extra underscore characters so that it does not have any spaces in it. This is important, because when you import Python modules, the file name referenced can't have any spaces in it. Figure 2.29 shows how to import and use a function from a different Python code module. Line 1 of Figure 2.29 uses the `import` statement to import the code in the Python module and creates the alias "fam" for the module. Line 9 of the Python code in Figure 2.29 uses the function in the imported module by referencing the alias, which indicates that the function is in that module. Figure 2.30 shows the output of executing the code in Figure 2.29, showing the same output as seen in Figure 2.27. We will make extensive use of Python code modules beginning in Chapter 4.

FIGURE 2.30 ⬢ OUTPUT FROM EXECUTION OF CODE USING FUNCTION IN MODULE

```
============= RESTART: I:\Fig 2_29 Find_Average_Using_Module.py ===
The average of the two numbers is:  6.0
```

STOP, CODE, AND UNDERSTAND!

SCU 2.8 Using Functions in Modules

Download the two files "SCU 2_8.py" and "UserFunction.py" from the companion website and save them either on your computer or on a removable storage device. Open the file "SCU 2_8.py" in the Python IDLE editor and insert a line of code to import the UserFunction module so that the code will report the average of the two numbers entered by the user using the function `my _ function`, which is in the file "UserFunction.py." Execute the modified program after the change to verify that the revised code runs and produces the correct result.

```
1  def my_function(x, y):
2      return((int(x) + int(y)) / 2)
```

```
1  # Add a line to import the UserFunction module with the alias UF:
2
3
4  value1 = input("Please enter a value: ")
5  value2 = input("Please enter a second value: ")
6
7  print("The average of the values entered is: ",UF.my_function(value1, value2))
```

Lessons learned: In this section, we learned about how we can import Python code from different modules of code so our code can use their functions. As we will see throughout the book, this is an important mechanism for using Python code that is in packages.

Chapter Summary

This chapter introduced the building blocks of Python programming, including Python data types, coding statements, and functions. We first learned about some practices to follow when developing code, which becomes very important when you work with other people on programming projects. We next learned about basic elements of Python code, including Python keywords, objects and classes, variables and data types, and operators and delimiters, which are combined into various statements that make up Python code.

We then learned about how to use variables in assignment statements, using `print` statements to visualize our results. We also learned how to use the Python `type` function to determine and report the data type of an object in a Python program and why the order of operations is important when we have Python code that performs calculations. We also learned there are three types of errors in Python: syntax errors, exceptions, and logic errors. Syntax errors are the easiest types of errors to resolve and result when we do not follow the rules for correctly specifying Python code. Exceptions are the types of errors that occur when we attempt to do something in Python that is not possible or not allowed. Logic errors occur when code executes without terminating with an error message but has incorrect results.

We also learned about a few functions that Python already has defined and that one common use of built-in functions is converting object data types from one type to another. Next we learned about user-defined functions and how functions can have arguments (values passed to them) as well as return values. We saw how we must be careful when specifying arguments when we call functions or else errors will result. Next, we learned about how we can import Python code from different modules of code so our code can use their functions. As we will see throughout the book, this is an important mechanism for using Python code that is in packages.

In the next chapter, we will delve deeper into compound data types, including lists, tuples, and dictionaries. We will see how useful these are when working with data and develop an even deeper understanding of the capabilities of the Python programming language. We will also discuss techniques for handling exceptions and resolving logic errors in our code.

Glossary

Arithmetic operators Symbols used to add, subtract, multiply, divide, and exponentiate (raise to a power) values.

Assignment statements Python code statements used to specify what value something is to take. Read from right to left, in which you assign the value on the right-hand side of the equal sign to the variable on the left-hand side of the equal sign.

Boolean Data type that can have either `True` or `False` logical values.

Classes Used to create objects.

Comments Begin with a pound sign (#). The interpreter does not process comments.

Concatenation Operation that combines multiple strings into one string.

Data type Specific attributes that determine the kind of values a piece of data can have and the kind of operations performed on the data.

`def` Python keyword that is at the beginning of every function definition.

Exception Occurs during Python code execution when you attempt an action that is not possible or not allowed.

Float Data type, which can have decimal values.

Function Subroutine of code developed to perform specific tasks.

Indentation The number of spaces and/or tabs at the beginning of a line of Python code.

Integer Data type that can store whole numbers.

Logic error Occurs when a program executes without terminating with an error condition but produces incorrect results.

Method An action that can be done to or with an object.

Objects The building blocks of Python. Objects or relations between objects represent all data in a Python program.

String Data type, which can have text made up of letters, characters, and numbers. Enclosed in quotes when referenced in code.

Syntax error Occurs when Python code does not follow the rules that dictate the specification of Python code statements.

`type` function Used to determine the data type of an object.

Variable Used to store and access values.

End-of-Chapter Exercises

2.1 Write Python code that uses the `input` built-in function to ask the user to enter a whole number between 1 and 100. The `input` function always returns a string value, so use the `int` built-in function to convert the value entered to an integer data type and square the number that the user entered using the exponentiation operator. Print a message to the user stating the value that they entered and the square of the value that they entered.

2.2 Write Python code that uses the `input` built-in function to ask the user to enter the year they were born as a four-digit number. The `input` function always returns a string value, so use the `int` built-in function to convert the year value entered to an integer data type and subtract the year entered from the current year (i.e., 2019). Print a message to the user stating the value that they entered and their calculated age. Why might the calculated age not be correct?

2.3 Write Python code that uses the `input` built-in function to ask the user to enter a decimal formatted number between 1 and 100. The `input` function always returns a string value, so use the `float` built-in function to convert the value entered to a float data type and square the number that the user entered using the exponentiation operator. Print a message to the user stating the value that they entered and the square of the value that they entered.

2.4 Modify the code in Exercise 2.3 to round the values reported to the user to two decimal places (use the `round` built-in function).

2.5 Write Python code that uses the `input` built-in function to ask the user to enter a sentence of their choosing. Use the `len` built-in function to determine how many characters were in the string entered and report this information back to the user.

2.6 Write Python code that uses the `input` built-in function to ask the user to enter a weight in pounds. The `input` function always returns a string value, so use the `float` built-in function to convert the value entered to a float data type and determine the equivalent weight in kilograms (you can use the conversion factor that 1 pound = 0.453592 kilograms). Print a message to the user stating

the weight in pounds that they entered and the equivalent weight in kilograms.

2.7 Write Python code that uses the `input` built-in function to ask the user to enter a temperature in the Fahrenheit temperature scale. The `input` function always returns a string value, so use the `float` built-in function to convert the value entered to a float data type and determine the equivalent temperature in the Celsius temperature scale (use the conversion factor °C = (°F – 32) × (5/9)). Print a message to the user stating the temperature in Fahrenheit that they entered and the equivalent temperature in Celsius. You can verify that your code executes properly by entering in 32°F (equivalent is 0°C) and 212°F (equivalent is 100°C).

References

Kaefer, P. (2018). Code like it matters: Writing code that's readable and shareable. SAS Global Forum Proceedings. Retrieved from **https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/2520-2018.pdf**

Martin, R. C. (2009). *Clean code: A handbook of agile software craftsmanship*. Upper Saddle River, NJ: Prentice Hall.

Python Software Foundation. (2019, June 17). Python 3.7.3 documentation. Retrieved from **https://docs.python.org/3/**

van Rossum, G., Warsaw, B., & Coghlan, N. (2001, July 5). PEP 8—Style guide for Python code. Retrieved from **https://www.python.org/dev/peps/pep-0008/**

Visit study.sagepub.com/researchmethods/statistics/kaefer-intro-to-python for data sets and code to accompany this text!