CHAPTER 2 Input, Processing, and Output

TOPICS

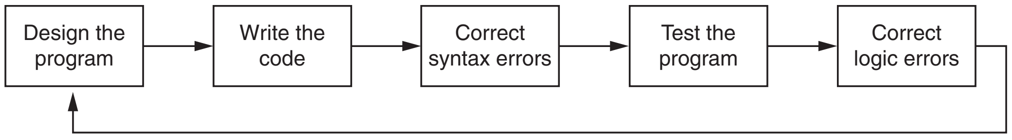
2.1 Designing a Program

CONCEPT: Programs must be carefully designed before they are written. During the design process, programmers use tools such as pseudocode and flowcharts to create models of programs.

The Program Development Cycle

In Chapter 1, you learned that programmers typically use high-level languages such as Python to create programs. There is much more to creating a program than writing code, however. The process of creating a program that works correctly typically requires the five phases shown in Figure 2-1. The entire process is known as the *program development cycle*.

Figure 2-1 The program development cycle



Let's take a closer look at each stage in the cycle.

1. **Design the Program.** All professional programmers will tell you that a program should be carefully designed before the code is actually written. When programmers begin a

new project, they should never jump right in and start writing code as the first step. They start by creating a design of the program. There are several ways to design a program, and later in this section, we will discuss some techniques that you can use to design your Python programs.

2. **Write the Code.** After designing the program, the programmer begins writing code in a high-level language such as Python. Recall from Chapter 1 that each language has its own rules, known as syntax, that must be followed when writing a program. A language's syntax rules dictate things such as how keywords, operators, and punctuation characters can be used. A syntax error occurs if the programmer violates any of these rules.
3. **Correct Syntax Errors.** If the program contains a syntax error, or even a simple mistake such as a misspelled keyword, the compiler or interpreter will display an error message indicating what the error is. Virtually all code contains syntax errors when it is first written, so the programmer will typically spend some time correcting these. Once all of the syntax errors and simple typing mistakes have been corrected, the program can be compiled and translated into a machine language program (or executed by an interpreter, depending on the language being used).
4. **Test the Program.** Once the code is in an executable form, it is then tested to determine whether any logic errors exist. A *logic error* is a mistake that does not prevent the program from running, but causes it to produce incorrect results. (Mathematical mistakes are common causes of logic errors.)
5. **Correct Logic Errors.** If the program produces incorrect results, the programmer *debugs* the code. This means that the programmer finds and corrects logic errors in the program. Sometimes during this process, the programmer discovers that the program's original design must be changed. In this event, the program development cycle starts over and continues until no errors can be found.

More About the Design Process

The process of designing a program is arguably the most important part of the cycle. You can think of a program's design as its foundation. If you build a house on a poorly constructed foundation, eventually you will find yourself doing a lot of work to fix the house! A program's design should be viewed no differently. If your program is designed poorly, eventually you will find yourself doing a lot of work to fix the program.

The process of designing a program can be summarized in the following two steps:

1. Understand the task that the program is to perform.
2. Determine the steps that must be taken to perform the task.

Let's take a closer look at each of these steps.

Understand the Task That the Program Is to Perform

It is essential that you understand what a program is supposed to do before you can determine the steps that the program will perform. Typically, a professional programmer gains this understanding by working directly with the customer. We use the term *customer* to describe the person, group, or organization that is asking you to write a program. This could be a customer in the traditional sense of the word, meaning someone who is paying you to write a program. It could also be your boss, or the manager of a department within your company. Regardless of whom it is, the customer will be relying on your program to perform an important task.

To get a sense of what a program is supposed to do, the programmer usually interviews the customer. During the interview, the customer will describe the task that the program should perform, and the programmer will ask questions to uncover as many details as possible about the task. A follow-up interview is usually needed because customers rarely mention everything they want during the initial meeting, and programmers often think of additional questions.

The programmer studies the information that was gathered from the customer during the interviews and creates a list of different software requirements. A *software requirement* is simply a single task that the program must perform in order to satisfy the customer. Once the customer agrees that the list of requirements is complete, the programmer can move to the next phase.

TIP: If you choose to become a professional software developer, your customer will be anyone who asks you to write programs as part of your job. As long as you are a student, however, your customer is your instructor! In every programming class that you will take, it's practically guaranteed that your instructor will assign programming problems for you to complete. For your academic success, make sure that you understand your instructor's requirements for those assignments and write your programs accordingly.

Determine the Steps That Must Be Taken to Perform the Task

Once you understand the task that the program will perform, you begin by breaking down the task into a series of steps. This is similar to the way you would break down a task into a series of steps that another person can follow. For example, suppose someone asks you how to boil water. You might break down that task into a series of steps as follows:

1. Pour the desired amount of water into a pot.
2. Put the pot on a stove burner.
3. Turn the burner to high.
4. Watch the water until you see large bubbles rapidly rising. When this happens, the water is boiling.

This is an example of an *algorithm,* which is a set of well-defined logical steps that must be taken to perform a task. Notice the steps in this algorithm are sequentially ordered. Step 1 should be performed before step 2, and so on. If a person follows these steps exactly as they appear, and in the correct order, he or she should be able to boil water successfully.

A programmer breaks down the task that a program must perform in a similar way. An algorithm is created, which lists all of the logical steps that must be taken. For example, suppose you have been asked to write a program to calculate and display the gross pay for an hourly paid employee. Here are the steps that you would take:

1. Get the number of hours worked.
2. Get the hourly pay rate.
3. Multiply the number of hours worked by the hourly pay rate.
4. Display the result of the calculation that was performed in step 3.

Of course, this algorithm isn't ready to be executed on the computer. The steps in this list have to be translated into code. Programmers commonly use two tools to help them accomplish this: pseudocode and flowcharts. Let's look at each of these in more detail.

Pseudocode

Because small mistakes like misspelled words and forgotten punctuation characters can cause syntax errors, programmers have to be mindful of such small details when writing code. For this reason, programmers find it helpful to write a program in pseudocode (pronounced "sue doe code") before they write it in the actual code of a programming language such as Python.

The word "pseudo" means fake, so *pseudocode* is fake code. It is an informal language that has no syntax rules and is not meant to be compiled or executed. Instead, programmers use pseudocode to create models, or "mock-ups," of programs. Because programmers don't have to worry about syntax errors while writing pseudocode, they can focus all of their attention on the program's design. Once a satisfactory design has been created with pseudocode, the pseudocode can be translated directly to actual code. Here is an example of how you might write pseudocode for the pay calculating program that we discussed earlier:

```
Input the hours worked
Input the hourly pay rate
Calculate gross pay as hours worked multiplied by pay rate
Display the gross pay
```

Each statement in the pseudocode represents an operation that can be performed in Python. For example, Python can read input that is typed on the keyboard, perform mathematical calculations, and display messages on the screen.

Flowcharts

Flowcharting is another tool that programmers use to design programs. A *flowchart* is a diagram that graphically depicts the steps that take place in a program. Figure 2-2 shows how you might create a flowchart for the pay calculating program.

Notice there are three types of symbols in the flowchart: ovals, parallelograms, and a rectangle. Each of these symbols represents a step in the program, as described here:

- The ovals, which appear at the top and bottom of the flowchart, are called *terminal symbols*. The *Start* terminal symbol marks the program's starting point, and the *End* terminal symbol marks the program's ending point.
- Parallelograms are used as *input symbols* and *output symbols*. They represent steps in which the program reads input or displays output.
- Rectangles are used as *processing symbols*. They represent steps in which the program performs some process on data, such as a mathematical calculation.

The symbols are connected by arrows that represent the "flow" of the program. To step through the symbols in the proper order, you begin at the *Start* terminal and follow the arrows until you reach the *End* terminal.

Checkpoint

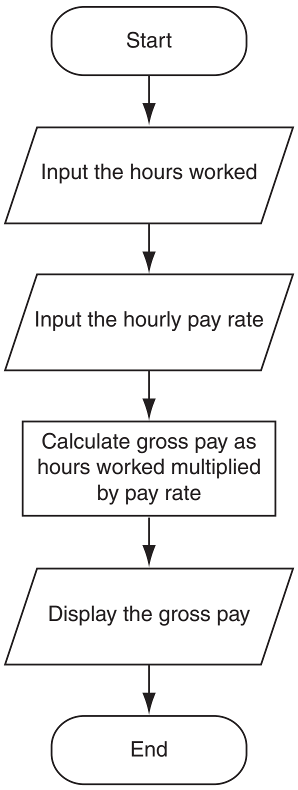
2.1 Who is a programmer's customer?

2.2 What is a software requirement?

2.3 What is an algorithm?

2.4 What is pseudocode?

Figure 2-2 Flowchart for the pay calculating program


Start
Input the hours worked
Input the hourly pay rate
Calculate gross pay as hours worked multiplied by pay rate
Display the gross pay
End

2.5 What is a flowchart?

2.6 What do each of the following symbols mean in a flowchart?
- Oval
- Parallelogram
- Rectangle

2.2 Input, Processing, and Output

CONCEPT: Input is data that the program receives. When a program receives data, it usually processes it by performing some operation with it. The result of the operation is sent out of the program as output.

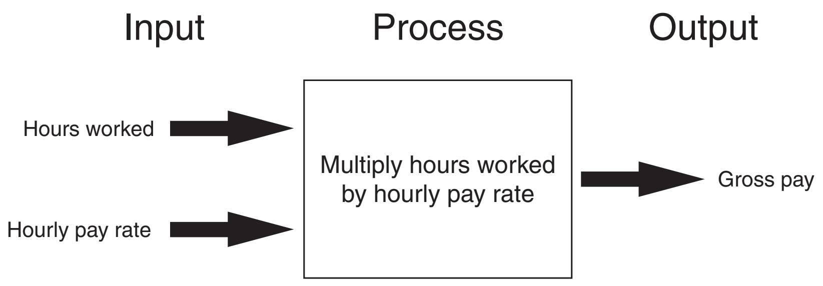
Computer programs typically perform the following three-step process:

1. Input is received.
2. Some process is performed on the input.
3. Output is produced.

Input is any data that the program receives while it is running. One common form of input is data that is typed on the keyboard. Once input is received, some process, such as a mathematical calculation, is usually performed on it. The results of the process are then sent out of the program as output.

Figure 2-3 illustrates these three steps in the pay calculating program that we discussed earlier. The number of hours worked and the hourly pay rate are provided as input. The program processes this data by multiplying the hours worked by the hourly pay rate. The results of the calculation are then displayed on the screen as output.

Figure 2-3 The input, processing, and output of the pay calculating program



In this chapter, we will discuss basic ways that you can perform input, processing, and output using Python.

2.3 Displaying Output with the `print` Function

CONCEPT: You use the `print` function to display output in a Python program.

VideoNote
The `print` Function

A *function* is a piece of prewritten code that performs an operation. Python has numerous built-in functions that perform various operations. Perhaps the most fundamental built-in function is the `print` function, which displays output on the screen. Here is an example of a statement that executes the `print` function:

```
print('Hello world')
```

In interactive mode, if you type this statement and press the Enter key, the message *Hello world* is displayed. Here is an example:

```
>>> print('Hello world') Enter
Hello world
>>>
```

When programmers execute a function, they say that they are *calling* the function. When you call the `print` function, you type the word `print`, followed by a set of parentheses. Inside the parentheses, you type an *argument*, which is the data that you want displayed on the screen. In the previous example, the argument is `'Hello world'`. Notice the quote marks are not displayed when the statement executes. The quote marks simply specify the beginning and the end of the text that you wish to display.

Suppose your instructor tells you to write a program that displays your name and address on the computer screen. Program 2-1 shows an example of such a program, with the output that it will produce when it runs. (The line numbers that appear in a program listing in this book are *not* part of the program. We use the line numbers in our discussion to refer to parts of the program.)

Program 2-1 **(output.py)**

```
1  print('Kate Austen')
2  print('123 Full Circle Drive')
3  print('Asheville, NC 28899')
```

Program Output

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

It is important to understand that the statements in this program execute in the order that they appear, from the top of the program to the bottom. When you run this program, the first statement will execute, followed by the second statement, and followed by the third statement.

Strings and String Literals

Programs almost always work with data of some type. For example, Program 2-1 uses the following three pieces of data:

```
'Kate Austen'
'123 Full Circle Drive
'Asheville, NC 28899'
```

These pieces of data are sequences of characters. In programming terms, a sequence of characters that is used as data is called a *string*. When a string appears in the actual code of a program, it is called a *string literal*. In Python code, string literals must be enclosed in quote marks. As mentioned earlier, the quote marks simply mark where the string data begins and ends.

In Python, you can enclose string literals in a set of single-quote marks (') or a set of double-quote marks ("). The string literals in Program 2-1 are enclosed in single-quote marks, but the program could also be written as shown in Program 2-2.

Program 2-2 **(double_quotes.py)**

```
1  print("Kate Austen")
2  print("123 Full Circle Drive")
3  print("Asheville, NC 28899")
```

Program Output

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

If you want a string literal to contain either a single-quote or an apostrophe as part of the string, you can enclose the string literal in double-quote marks. For example, Program 2-3 prints two strings that contain apostrophes.

Program 2-3 (apostrophe.py)

```
print("Don't fear!")
print("I'm here!")
```

Program Output

```
Don't fear!
I'm here!
```

Likewise, you can use single-quote marks to enclose a string literal that contains double-quotes as part of the string. Program 2-4 shows an example.

Program 2-4 (display_quote.py)

```
print('Your assignment is to read "Hamlet" by tomorrow.')
```

Program Output

```
Your assignment is to read "Hamlet" by tomorrow.
```

Python also allows you to enclose string literals in triple quotes (either `"""` or `'''`). Triple-quoted strings can contain both single quotes and double quotes as part of the string. The following statement shows an example:

```
print("""I'm reading "Hamlet" tonight.""")
```

This statement will print

```
I'm reading "Hamlet" tonight.
```

Triple quotes can also be used to surround *multiline strings*, something for which single and double quotes cannot be used. Here is an example:

```
print("""One
Two
Three""")
```

This statement will print

```
One
Two
Three
```

Checkpoint

2.7 Write a statement that displays your name.

2.8 Write a statement that displays the following text:

```
Python's the best!
```

2.9 Write a statement that displays the following text:

```
The cat said "meow."
```

2.4 Comments

CONCEPT: Comments are notes of explanation that document lines or sections of a program. Comments are part of the program, but the Python interpreter ignores them. They are intended for people who may be reading the source code.

Comments are short notes placed in different parts of a program, explaining how those parts of the program work. Although comments are a critical part of a program, they are ignored by the Python interpreter. Comments are intended for any person reading a program's code, not the computer.

In Python, you begin a comment with the # character. When the Python interpreter sees a # character, it ignores everything from that character to the end of the line. For example, look at Program 2-5. Lines 1 and 2 are comments that briefly explain the program's purpose.

Program 2-5 (comment1.py)

```
1  # This program displays a person's
2  # name and address.
3  print('Kate Austen')
4  print('123 Full Circle Drive')
5  print('Asheville, NC 28899')
```

Program Output

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

Programmers commonly write end-line comments in their code. An *end-line comment* is a comment that appears at the end of a line of code. It usually explains the statement that appears in that line. Program 2-6 shows an example. Each line ends with a comment that briefly explains what the line does.

Program 2-6 (comment2.py)

```
print('Kate Austen')                # Display the name.
print('123 Full Circle Drive')      # Display the address.
print('Asheville, NC 28899')        # Display the city, state, and ZIP.
```

Program Output

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

As a beginning programmer, you might be resistant to the idea of liberally writing comments in your programs. After all, it can seem more productive to write code that actually does something! It is crucial that you take the extra time to write comments, however. They will almost certainly save you and others time in the future when you have to modify or debug the program. Large and complex programs can be almost impossible to read and understand if they are not properly commented.

2.5 Variables

CONCEPT: A variable is a name that represents a storage location in the computer's memory.

Programs usually store data in the computer's memory and perform operations on that data. For example, consider the typical online shopping experience: you browse a website and add the items that you want to purchase to the shopping cart. As you add items to the shopping cart, data about those items is stored in memory. Then, when you click the checkout button, a program running on the website's computer calculates the cost of all the items you have in your shopping cart, applicable sales taxes, shipping costs, and the total of all these charges. When the program performs these calculations, it stores the results in the computer's memory.

Programs use variables to store data in memory. A *variable* is a name that represents a value in the computer's memory. For example, a program that calculates the sales tax on a purchase might use the variable name `tax` to represent that value in memory. And a program that calculates the distance between two cities might use the variable name `distance` to represent that value in memory. When a variable represents a value in the computer's memory, we say that the variable *references* the value.

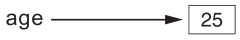
Creating Variables with Assignment Statements

You use an *assignment statement* to create a variable and make it reference a piece of data. Here is an example of an assignment statement:

```
age = 25
```

After this statement executes, a variable named age will be created, and it will reference the value 25. This concept is shown in Figure 2-4. In the figure, think of the value 25 as being stored somewhere in the computer's memory. The arrow that points from age to the value 25 indicates that the variable name age references the value.

Figure 2-4 The age variable references the value 25



An assignment statement is written in the following general format:

```
variable = expression
```

The equal sign (=) is known as the *assignment operator*. In the general format, *variable* is the name of a variable and *expression* is a value, or any piece of code that results in a value. After an assignment statement executes, the variable listed on the left side of the = operator will reference the value given on the right side of the = operator.

To experiment with variables, you can type assignment statements in interactive mode, as shown here:

```
>>> width = 10 Enter
>>> length = 5 Enter
>>>
```

The first statement creates a variable named width and assigns it the value 10. The second statement creates a variable named length and assigns it the value 5. Next, you can use the print function to display the values referenced by these variables, as shown here:

```
>>> print(width) Enter
10
>>> print(length) Enter
5
>>>
```

When you pass a variable as an argument to the print function, you do not enclose the variable name in quote marks. To demonstrate why, look at the following interactive session:

```
>>> print('width') Enter
width
>>> print(width) Enter
10
>>>
```

In the first statement, we passed 'width' as an argument to the print function, and the function printed the string *width*. In the second statement, we passed width (with no quote marks) as an argument to the print function, and the function displayed the value referenced by the width variable.

In an assignment statement, the variable that is receiving the assignment must appear on the left side of the = operator. As shown in the following interactive session, an error occurs if the item on the left side of the = operator is not a variable:

```
>>> 25 = age Enter
SyntaxError: can't assign to literal
>>>
```

The code in Program 2-7 demonstrates a variable. Line 2 creates a variable named `room` and assigns it the value 503. The statements in lines 3 and 4 display a message. Notice line 4 displays the value that is referenced by the `room` variable.

Program 2-7 **(variable_demo.py)**

```
1  # This program demonstrates a variable.
2  room = 503
3  print('I am staying in room number')
4  print(room)
```

Program Output

```
I am staying in room number
503
```

Program 2-8 shows a sample program that uses two variables. Line 2 creates a variable named `top_speed`, assigning it the value 160. Line 3 creates a variable named `distance`, assigning it the value 300. This is illustrated in Figure 2-5.

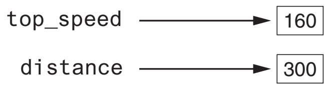
Program 2-8 **(variable_demo2.py)**

```
1  # Create two variables: top_speed and distance.
2  top_speed = 160
3  distance = 300
4
5  # Display the values referenced by the variables.
6  print('The top speed is')
7  print(top_speed)
8  print('The distance traveled is')
9  print(distance)
```

Program Output

```
The top speed is
160
The distance traveled is
300
```

Figure 2-5 Two variables



WARNING! You cannot use a variable until you have assigned a value to it. An error will occur if you try to perform an operation on a variable, such as printing it, before it has been assigned a value.

Sometimes a simple typing mistake will cause this error. One example is a misspelled variable name, as shown here:

```
temperature = 74.5    # Create a variable
print(tempereture)    # Error! Misspelled variable name
```

In this code, the variable `temperature` is created by the assignment statement. The variable name is spelled differently in the `print` statement, however, which will cause an error. Another example is the inconsistent use of uppercase and lowercase letters in a variable name. Here is an example:

```
temperature = 74.5    # Create a variable
print(Temperature)    # Error! Inconsistent use of case
```

In this code, the variable `temperature` (in all lowercase letters) is created by the assignment statement. In the `print` statement, the name `Temperature` is spelled with an uppercase T. This will cause an error because variable names are case sensitive in Python.

NOTE: Internally, Python variables work differently than variables in most other programming languages. In most programming languages, a variable is a memory location that holds a value. In those languages, when you assign a value to a variable, the value is stored in the variable's memory location.

In Python, however, a variable is a memory location that holds the *address* of another memory location. When you assign a value to a Python variable, that value is stored in a location that is separate from the variable. The variable will hold the address of the memory location that holds the value. That is why, in Python, instead of saying that a variable "holds" a value, we say that a variable "references" a variable.

Variable Naming Rules

Although you are allowed to make up your own names for variables, you must follow these rules:

- You cannot use one of Python's keywords as a variable name. (See Table 1-2 for a list of the keywords.)
- A variable name cannot contain spaces.

- The first character must be one of the letters a through z, A through Z, or an underscore character (_).
- After the first character you may use the letters a through z or A through Z, the digits 0 through 9, or underscores.
- Uppercase and lowercase characters are distinct. This means the variable name `ItemsOrdered` is not the same as `itemsordered`.

In addition to following these rules, you should always choose names for your variables that give an indication of what they are used for. For example, a variable that holds the temperature might be named `temperature`, and a variable that holds a car's speed might be named `speed`. You may be tempted to give variables names such as `x` and `b2`, but names like these give no clue as to what the variable's purpose is.

Because a variable's name should reflect the variable's purpose, programmers often find themselves creating names that are made of multiple words. For example, consider the following variable names:

```
grosspay
payrate
hotdogssoldtoday
```

Unfortunately, these names are not easily read by the human eye because the words aren't separated. Because we can't have spaces in variable names, we need to find another way to separate the words in a multiword variable name and make it more readable to the human eye.

One way to do this is to use the underscore character to represent a space. For example, the following variable names are easier to read than those previously shown:

```
gross_pay
pay_rate
hot_dogs_sold_today
```

This style of naming variables is popular among Python programmers, and is the style we will use in this book. There are other popular styles, however, such as the *camelCase* naming convention. camelCase names are written in the following manner:

- The variable name begins with lowercase letters.
- The first character of the second and subsequent words is written in uppercase.

For example, the following variable names are written in camelCase:

```
grossPay
payRate
hotDogsSoldToday
```

NOTE: This style of naming is called camelCase because the uppercase characters that appear in a name may suggest a camel's humps.

Table 2-1 lists several sample variable names and indicates whether each is legal or illegal in Python.

Table 2-1 Sample variable names

Variable Name	Legal or Illegal?
`units_per_day`	Legal
`dayOfWeek`	Legal
`3dGraph`	Illegal. Variable names cannot begin with a digit.
`June1997`	Legal
`Mixture#3`	Illegal. Variable names may only use letters, digits, or underscores.

Displaying Multiple Items with the `print` Function

If you refer to Program 2-7, you will see that we used the following two statements in lines 3 and 4:

```
print('I am staying in room number')
print(room)
```

We called the `print` function twice because we needed to display two pieces of data. Line 3 displays the string literal `'I am staying in room number'`, and line 4 displays the value referenced by the `room` variable.

This program can be simplified, however, because Python allows us to display multiple items with one call to the `print` function. We simply have to separate the items with commas as shown in Program 2-9.

Program 2-9 (variable_demo3.py)

```
1  # This program demonstrates a variable.
2  room = 503
3  print('I am staying in room number', room)
```

Program Output

```
I am staying in room number 503
```

In line 3, we passed two arguments to the `print` function. The first argument is the string literal `'I am staying in room number'`, and the second argument is the `room` variable. When the `print` function executed, it displayed the values of the two arguments in the order that we passed them to the function. Notice the `print` function automatically printed a space separating the two items. When multiple arguments are passed to the `print` function, they are automatically separated by a space when they are displayed on the screen.

Variable Reassignment

Variables are called "variable" because they can reference different values while a program is running. When you assign a value to a variable, the variable will reference that value until you assign it a different value. For example, look at Program 2-10. The statement in line 3 creates a variable named `dollars` and assigns it the value 2.75. This is shown in the top part of Figure 2-6. Then, the statement in line 8 assigns a different value, 99.95, to the `dollars` variable. The bottom part of Figure 2-6 shows how this changes the `dollars` variable. The old value, 2.75, is still in the computer's memory, but it can no longer be used because it isn't referenced by a variable. When a value in memory is no longer referenced by a variable, the Python interpreter automatically removes it from memory through a process known as *garbage collection*.

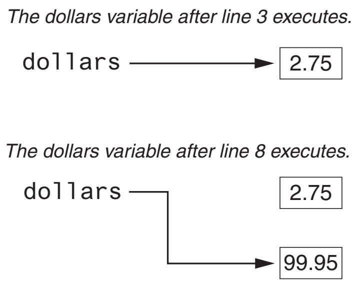
Program 2-10 (variable_demo4.py)

```
1  # This program demonstrates variable reassignment.
2  # Assign a value to the dollars variable.
3  dollars = 2.75
4  print('I have', dollars, 'in my account.')
5
6  # Reassign dollars so it references
7  # a different value.
8  dollars = 99.95
9  print('But now I have', dollars, 'in my account!')
```

Program Output

```
I have 2.75 in my account.
But now I have 99.95 in my account!
```

Figure 2-6 Variable reassignment in Program 2-10



The dollars variable after line 3 executes.
dollars
2.75
The dollars variable after line 8 executes.
dollars
2.75
99.95

Numeric Data Types and Literals

In Chapter 1, we discussed the way that computers store data in memory. (See Section 1.3) You might recall from that discussion that computers use a different technique for storing real numbers (numbers with a fractional part) than for storing integers. Not only are these types of numbers stored differently in memory, but similar operations on them are carried out in different ways.

Because different types of numbers are stored and manipulated in different ways, Python uses *data types* to categorize values in memory. When an integer is stored in memory, it is classified as an `int`, and when a real number is stored in memory, it is classified as a `float`.

Let's look at how Python determines the data type of a number. Several of the programs that you have seen so far have numeric data written into their code. For example, the following statement, which appears in Program 2-9, has the number 503 written into it:

```
room = 503
```

This statement causes the value 503 to be stored in memory, and it makes the `room` variable reference it. The following statement, which appears in Program 2-10, has the number 2.75 written into it:

```
dollars = 2.75
```

This statement causes the value 2.75 to be stored in memory, and it makes the `dollars` variable reference it. A number that is written into a program's code is called a *numeric literal*. When the Python interpreter reads a numeric literal in a program's code, it determines its data type according to the following rules:

- A numeric literal that is written as a whole number with no decimal point is considered an `int`. Examples are `7`, `124`, and `-9`.
- A numeric literal that is written with a decimal point is considered a `float`. Examples are `1.5`, `3.14159`, and `5.0`.

So, the following statement causes the number 503 to be stored in memory as an `int`:

```
room = 503
```

And the following statement causes the number 2.75 to be stored in memory as a `float`:

```
dollars = 2.75
```

When you store an item in memory, it is important for you to be aware of the item's data type. As you will see, some operations behave differently depending on the type of data involved, and some operations can only be performed on values of a specific data type.

As an experiment, you can use the built-in `type` function in interactive mode to determine the data type of a value. For example, look at the following session:

```
>>> type(1) Enter
<class 'int'>
>>>
```

In this example, the value 1 is passed as an argument to the `type` function. The message that is displayed on the next line, `<class 'int'>`, indicates that the value is an `int`. Here is another example:

```
>>> type(1.0) Enter
<class 'float'>
>>>
```

In this example, the value 1.0 is passed as an argument to the `type` function. The message that is displayed on the next line, `<class 'float'>`, indicates that the value is a `float`.

WARNING! You cannot write currency symbols, spaces, or commas in numeric literals. For example, the following statement will cause an error:

```
value = $4,567.99 # Error!
```

This statement must be written as:

```
value = 4567.99   # Correct
```

Storing Strings with the `str` Data Type

In addition to the `int` and `float` data types, Python also has a data type named `str`, which is used for storing strings in memory. The code in Program 2-11 shows how strings can be assigned to variables.

Program 2-11 (string_variable.py)

```
# Create variables to reference two strings.
first_name = 'Kathryn'
last_name = 'Marino'

# Display the values referenced by the variables.
print(first_name, last_name)
```

Program Output

```
Kathryn Marino
```

Reassigning a Variable to a Different Type

Keep in mind that in Python, a variable is just a name that refers to a piece of data in memory. It is a mechanism that makes it easy for you, the programmer, to store and retrieve data. Internally, the Python interpreter keeps track of the variable names that you create and the pieces of data to which those variable names refer. Any time you need to retrieve one of those pieces of data, you simply use the variable name that refers to it.

A variable in Python can refer to items of any type. After a variable has been assigned an item of one type, it can be reassigned an item of a different type. To demonstrate, look at the following interactive session. (We have added line numbers for easier reference.)

```
1  >>> x = 99 Enter
2  >>> print(x) Enter
3  99
4  >>> x = 'Take me to your leader' Enter
5  >>> print(x) Enter
6  Take me to your leader
7  >>>
```

The statement in line 1 creates a variable named `x` and assigns it the `int` value 99. Figure 2-7 shows how the variable `x` references the value 99 in memory. The statement in line 2 calls the `print` function, passing `x` as an argument. The output of the `print` function is shown in line 3. Then, the statement in line 4 assigns a string to the `x` variable. After this statement executes, the `x` variable no longer refers to an `int`, but to the string `'Take me to your leader'`. This is shown in Figure 2-8. Line 5 calls the `print` function again, passing `x` as an argument. Line 6 shows the `print` function's output.

Figure 2-7 The variable x references an integer

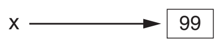
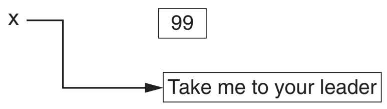


Figure 2-8 The variable x references a string



Checkpoint

2.10 What is a variable?

2.11 Which of the following are illegal variable names in Python, and why?

```
x
99bottles
july2009
theSalesFigureForFiscalYear
r&d
grade_report
```

2.12 Is the variable name `Sales` the same as `sales`? Why or why not?

2.13 Is the following assignment statement valid or invalid? If it is invalid, why?

```
72 = amount
```

2.14 What will the following code display?

```
val = 99
print('The value is', 'val')
```

2.15 Look at the following assignment statements:

```
value1 = 99
value2 = 45.9
value3 = 7.0
value4 = 7
value5 = 'abc'
```

After these statements execute, what is the Python data type of the values referenced by each variable?

2.16 What will be displayed by the following program?

```
my_value = 99
my_value = 0
print(my_value)
```

2.6 Reading Input from the Keyboard

CONCEPT: Programs commonly need to read input typed by the user on the keyboard. We will use the Python functions to do this.

VideoNote
Reading Input from the Keyboard

Most of the programs that you will write will need to read input and then perform an operation on that input. In this section, we will discuss a basic input operation: reading data that has been typed on the keyboard. When a program reads data from the keyboard, usually it stores that data in a variable so it can be used later by the program.

In this book, we use Python's built-in `input` function to read input from the keyboard. The `input` function reads a piece of data that has been entered at the keyboard and returns that piece of data, as a string, back to the program. You normally use the `input` function in an assignment statement that follows this general format:

```
variable = input(prompt)
```

In the general format, *`prompt`* is a string that is displayed on the screen. The string's purpose is to instruct the user to enter a value; *`variable`* is the name of a variable that references the data that was entered on the keyboard. Here is an example of a statement that uses the `input` function to read data from the keyboard:

```
name = input('What is your name? ')
```

When this statement executes, the following things happen:

- The string `'What is your name? '` is displayed on the screen.
- The program pauses and waits for the user to type something on the keyboard and then to press the Enter key.
- When the Enter key is pressed, the data that was typed is returned as a string and assigned to the `name` variable.

To demonstrate, look at the following interactive session:

```
>>> name = input('What is your name? ') Enter
What is your name?  Holly  Enter
>>> print(name) Enter
Holly
>>>
```

When the first statement was entered, the interpreter displayed the prompt `'What is your name? '` and waited for the user to enter some data. The user entered **`Holly`** and pressed the Enter key. As a result, the string `'Holly'` was assigned to the `name` variable. When the second statement was entered, the interpreter displayed the value referenced by the `name` variable.

Program 2-12 shows a complete program that uses the `input` function to read two strings as input from the keyboard.

Program 2-12 (string_input.py)

```
# Get the user's first name.
first_name = input('Enter your first name: ')

# Get the user's last name.
last_name = input('Enter your last name: ')

# Print a greeting to the user.
print('Hello', first_name, last_name)
```

Program Output (with input shown in bold)

```
Enter your first name: Vinny Enter
Enter your last name: Brown Enter
Hello Vinny Brown
```

Take a closer look in line 2 at the string we used as a prompt:

```
'Enter your first name: '
```

Notice the last character in the string, inside the quote marks, is a space. The same is true for the following string, used as prompt in line 5:

```
'Enter your last name: '
```

We put a space character at the end of each string because the `input` function does not automatically display a space after the prompt. When the user begins typing characters, they appear on the screen immediately after the prompt. Making the last character in the prompt a space visually separates the prompt from the user's input on the screen.

Reading Numbers with the `input` Function

The `input` function always returns the user's input as a string, even if the user enters numeric data. For example, suppose you call the `input` function, type the number 72, and press the Enter key. The value that is returned from the `input` function is the string `'72'`. This can be a problem if you want to use the value in a math operation. Math operations can be performed only on numeric values, not strings.

Fortunately, Python has built-in functions that you can use to convert a string to a numeric type. Table 2-2 summarizes two of these functions.

Table 2-2 Data conversion functions

Function	Description
`int(`*`item`*`)`	You pass an argument to the `int()` function and it returns the argument's value converted to an `int`.
`float(`*`item`*`)`	You pass an argument to the `float()` function and it returns the argument's value converted to a `float`.

For example, suppose you are writing a payroll program and you want to get the number of hours that the user has worked. Look at the following code:

```
string_value = input('How many hours did you work? ')
hours = int(string_value)
```

The first statement gets the number of hours from the user and assigns that value as a string to the `string_value` variable. The second statement calls the `int()` function, passing `string_value` as an argument. The value referenced by `string_value` is converted to an `int` and assigned to the `hours` variable.

This example illustrates how the `int()` function works, but it is inefficient because it creates two variables: one to hold the string that is returned from the `input` function, and another to hold the integer that is returned from the `int()` function. The following code shows a better approach. This one statement does all the work that the previously shown two statements do, and it creates only one variable:

```
hours = int(input('How many hours did you work? '))
```

This one statement uses *nested function* calls. The value that is returned from the `input` function is passed as an argument to the `int()` function. This is how it works:

- It calls the `input` function to get a value entered at the keyboard.
- The value that is returned from the `input` function (a string) is passed as an argument to the `int()` function.
- The `int` value that is returned from the `int()` function is assigned to the `hours` variable.

After this statement executes, the `hours` variable is assigned the value entered at the keyboard, converted to an `int`.

Let's look at another example. Suppose you want to get the user's hourly pay rate. The following statement prompts the user to enter that value at the keyboard, converts the value to a `float`, and assigns it to the `pay_rate` variable:

```
pay_rate = float(input('What is your hourly pay rate? '))
```

This is how it works:

- It calls the `input` function to get a value entered at the keyboard.
- The value that is returned from the `input` function (a string) is passed as an argument to the `float()` function.
- The `float` value that is returned from the `float()` function is assigned to the `pay_rate` variable.

After this statement executes, the `pay_rate` variable is assigned the value entered at the keyboard, converted to a `float`.

Program 2-13 shows a complete program that uses the `input` function to read a string, an `int`, and a `float`, as input from the keyboard.

Program 2-13 (`input.py`)

```
# Get the user's name, age, and income.
name = input('What is your name? ')
age = int(input('What is your age? '))
income = float(input('What is your income? '))

# Display the data.
print('Here is the data you entered:')
print('Name:', name)
print('Age:', age)
print('Income:', income)
```

Program Output (with input shown in bold)

```
What is your name? Chris Enter
What is your age? 25 Enter
What is your income? 75000.0
Here is the data you entered:
Name: Chris
Age: 25
Income: 75000.0
```

Let's take a closer look at the code:

- Line 2 prompts the user to enter his or her name. The value that is entered is assigned, as a string, to the `name` variable.
- Line 3 prompts the user to enter his or her age. The value that is entered is converted to an `int` and assigned to the `age` variable.
- Line 4 prompts the user to enter his or her income. The value that is entered is converted to a `float` and assigned to the `income` variable.
- Lines 7 through 10 display the values that the user entered.

The `int()` and `float()` functions work only if the item that is being converted contains a valid numeric value. If the argument cannot be converted to the specified data type, an error known as an exception occurs. An *exception* is an unexpected error that occurs while a program is running, causing the program to halt if the error is not properly dealt with. For example, look at the following interactive mode session:

```
>>> age = int(input('What is your age? ')) Enter
What is your age?  xyz  Enter
Traceback (most recent call last):
    File "<pyshell#81>", line 1, in <module>
         age = int(input('What is your age? '))
ValueError: invalid literal for int() with base 10: 'xyz'
>>>
```

NOTE: In this section, we mentioned the user. The *user* is simply any hypothetical person that is using a program and providing input for it. The user is sometimes called the *end user*.

Checkpoint

2.17 You need the user of a program to enter a customer's last name. Write a statement that prompts the user to enter this data and assigns the input to a variable.

2.18 You need the user of a program to enter the amount of sales for the week. Write a statement that prompts the user to enter this data and assigns the input to a variable.

2.7 Performing Calculations

CONCEPT: Python has numerous operators that can be used to perform mathematical calculations.

Most real-world algorithms require calculations to be performed. A programmer's tools for performing calculations are *math operators*. Table 2-3 lists the math operators that are provided by the Python language.

Programmers use the operators shown in Table 2-3 to create math expressions. A *math expression* performs a calculation and gives a value. The following is an example of a simple math expression:

```
12 + 2
```

Table 2-3 Python math operators

Symbol	Operation	Description
+	Addition	Adds two numbers
–	Subtraction	Subtracts one number from another
*	Multiplication	Multiplies one number by another
/	Division	Divides one number by another and gives the result as a floating-point number
//	Integer division	Divides one number by another and gives the result as a whole number
%	Remainder	Divides one number by another and gives the remainder
**	Exponent	Raises a number to a power

The values on the right and left of the + operator are called *operands*. These are values that the + operator adds together. If you type this expression in interactive mode, you will see that it gives the value 14:

```
>>> 12 + 2 Enter
14
>>>
```

Variables may also be used in a math expression. For example, suppose we have two variables named hours and pay_rate. The following math expression uses the * operator to multiply the value referenced by the hours variable by the value referenced by the pay_rate variable:

```
hours * pay_rate
```

When we use a math expression to calculate a value, normally we want to save that value in memory so we can use it again in the program. We do this with an assignment statement. Program 2-14 shows an example.

Program 2-14 (simple_math.py)

```
# Assign a value to the salary variable.
salary = 2500.0

# Assign a value to the bonus variable.
bonus = 1200.0

# Calculate the total pay by adding salary
# and bonus. Assign the result to pay.
pay = salary + bonus

# Display the pay.
print('Your pay is', pay)
```

Program Output

```
Your pay is 3700.0
```

Line 2 assigns 2500.0 to the `salary` variable, and line 5 assigns 1200.0 to the `bonus` variable. Line 9 assigns the result of the expression `salary + bonus` to the `pay` variable. As you can see from the program output, the `pay` variable holds the value 3700.0.

In the Spotlight:

Calculating a Percentage

If you are writing a program that works with a percentage, you have to make sure that the percentage's decimal point is in the correct location before doing any math with the percentage. This is especially true when the user enters a percentage as input. Most users enter the number 50 to mean 50 percent, 20 to mean 20 percent, and so forth. Before you perform any calculations with such a percentage, you have to divide it by 100 to move its decimal point two places to the left.

Let's step through the process of writing a program that calculates a percentage. Suppose a retail business is planning to have a storewide sale where the prices of all items will be 20 percent off. We have been asked to write a program to calculate the sale price of an item after the discount is subtracted. Here is the algorithm:

1. *Get the original price of the item.*
2. *Calculate 20 percent of the original price. This is the amount of the discount.*
3. *Subtract the discount from the original price. This is the sale price.*
4. *Display the sale price.*

In step 1, we get the original price of the item. We will prompt the user to enter this data on the keyboard. In our program we will use the following statement to do this. Notice the value entered by the user will be stored in a variable named `original_price`.

```
original_price = float(input("Enter the item's original price: "))
```

In step 2, we calculate the amount of the discount. To do this, we multiply the original price by 20 percent. The following statement performs this calculation and assigns the result to the `discount` variable:

```
discount = original_price * 0.2
```

In step 3, we subtract the discount from the original price. The following statement does this calculation and stores the result in the `sale_price` variable:

```
sale_price = original_price − discount
```

Last, in step 4, we will use the following statement to display the sale price:

```
print('The sale price is', sale_price)
```

Program 2-15 shows the entire program, with example output.

Program 2-15 (sale_price.py)

```
# This program gets an item's original price and
# calculates its sale price, with a 20% discount.

```

(program continues)

Program 2-15 (*continued*)

```
# Get the item's original price.
original_price = float(input("Enter the item's original price: "))

# Calculate the amount of the discount.
discount = original_price * 0.2

# Calculate the sale price.
sale_price = original_price - discount

# Display the sale price.
print('The sale price is', sale_price)
```

Program Output (with input shown in bold)

```
Enter the item's original price: 100.00 Enter
The sale price is 80.0
```

Floating-Point and Integer Division

Notice in Table 2-3 that Python has two different division operators. The / operator performs floating-point division, and the // operator performs integer division. Both operators divide one number by another. The difference between them is that the / operator gives the result as a floating-point value, and the // operator gives the result as a whole number. Let's use the interactive mode interpreter to demonstrate:

```
>>> 5 / 2 Enter
2.5
>>>
```

In this session, we used the / operator to divide 5 by 2. As expected, the result is 2.5. Now let's use the // operator to perform integer division:

```
>>> 5 // 2 Enter
2
>>>
```

As you can see, the result is 2. The // operator works like this:

- When the result is positive, it is *truncated*, which means that its fractional part is thrown away.
- When the result is negative, it is rounded *away from zero* to the nearest integer.

The following interactive session demonstrates how the // operator works when the result is negative:

```
>>> -5 // 2 Enter
-3
>>>
```

Operator Precedence

You can write statements that use complex mathematical expressions involving several operators. The following statement assigns the sum of 17, the variable x, 21, and the variable y to the variable answer:

```
answer = 17 + x + 21 + y
```

Some expressions are not that straightforward, however. Consider the following statement:

```
outcome = 12.0 + 6.0 / 3.0
```

What value will be assigned to outcome? The number 6.0 might be used as an operand for either the addition or division operator. The outcome variable could be assigned either 6.0 or 14.0, depending on when the division takes place. Fortunately, the answer can be predicted because Python follows the same order of operations that you learned in math class.

First, operations that are enclosed in parentheses are performed first. Then, when two operators share an operand, the operator with the higher *precedence* is applied first. The precedence of the math operators, from highest to lowest, are:

1. Exponentiation: `**`
2. Multiplication, division, and remainder: `* / // %`
3. Addition and subtraction: `+ -`

Notice the multiplication (`*`), floating-point division (`/`), integer division (`//`), and remainder (`%`) operators have the same precedence. The addition (`+`) and subtraction (`-`) operators also have the same precedence. When two operators with the same precedence share an operand, the operators execute from left to right.

Now, let's go back to the previous math expression:

```
outcome = 12.0 + 6.0 / 3.0
```

The value that will be assigned to outcome is 14.0 because the division operator has a higher *precedence* than the addition operator. As a result, the division takes place before the addition. The expression can be diagrammed as shown in Figure 2-9.

Figure 2-9 Operator precedence

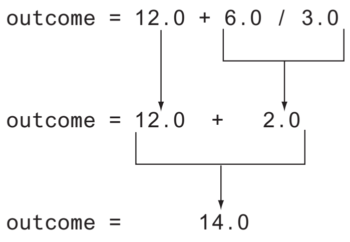


Table 2-4 shows some other sample expressions with their values.

Table 2-4 Some expressions

Expression	Value
`5 + 2 * 4`	13
`10 / 2 - 3`	2.0
`8 + 12 * 2 - 4`	28
`6 - 3 * 2 + 7 - 1`	6

NOTE: There is an exception to the left-to-right rule. When two `**` operators share an operand, the operators execute right-to-left. For example, the expression `2**3**4` is evaluated as `2**(3**4)`.

Grouping with Parentheses

Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others. In the following statement, the variables `a` and `b` are added together, and their sum is divided by 4:

```
result = (a + b) / 4
```

Without the parentheses, however, `b` would be divided by 4 and the result added to `a`. Table 2-5 shows more expressions and their values.

Table 2-5 More expressions and their values

Expression	Value
`(5 + 2) * 4`	28
`10 / (5 - 3)`	5.0
`8 + 12 * (6 - 2)`	56
`(6 - 3) * (2 + 7) / 3`	9.0

In the Spotlight:

Calculating an Average

Determining the average of a group of values is a simple calculation: add all of the values then divide the sum by the number of values. Although this is a straightforward calculation, it is easy to make a mistake when writing a program that calculates an average. For example, let's assume that the variables `a`, `b`, and `c` each hold a value and we want to calculate the average of those values. If we are careless, we might write a statement such as the following to perform the calculation:

```
average = a + b + c / 3.0
```

Can you see the error in this statement? When it executes, the division will take place first. The value in `c` will be divided by 3, then the result will be added to `a + b`. That is not the correct way to calculate an average. To correct this error, we need to put parentheses around `a + b + c`, as shown here:

```
average = (a + b + c) / 3.0
```

Let's step through the process of writing a program that calculates an average. Suppose you have taken three tests in your computer science class, and you want to write a program that will display the average of the test scores. Here is the algorithm:

1. *Get the first test score.*
2. *Get the second test score.*
3. *Get the third test score.*
4. *Calculate the average by adding the three test scores and dividing the sum by 3.*
5. *Display the average.*

In steps 1, 2, and 3 we will prompt the user to enter the three test scores. We will store those test scores in the variables `test1`, `test2`, and `test3`. In step 4, we will calculate the average of the three test scores. We will use the following statement to perform the calculation and store the result in the `average` variable:

```
average = (test1 + test2 + test3) / 3.0
```

Last, in step 5, we display the average. Program 2-16 shows the program.

Program 2-16 (test_score_average.py)

```
# Get three test scores and assign them to the
# test1, test2, and test3 variables.
test1 = float(input('Enter the first test score: '))
test2 = float(input('Enter the second test score: '))
test3 = float(input('Enter the third test score: '))

# Calculate the average of the three scores
# and assign the result to the average variable.
average = (test1 + test2 + test3) / 3.0

# Display the average.
print('The average score is', average)
```

Program Output (with input shown in bold)

```
Enter the first test score: 90 Enter
Enter the second test score: 80 Enter
Enter the third test score: 100 Enter
The average score is 90.0
```

The Exponent Operator

Two asterisks written together (`**`) is the exponent operator, and its purpose is to raise a number to a power. For example, the following statement raises the `length` variable to the power of 2 and assigns the result to the `area` variable:

```
area = length**2
```

The following session with the interactive interpreter shows the values of the expressions `4**2`, `5**3`, and `2**10`:

```
>>> 4**2 Enter
16
>>> 5**3 Enter
125
>>> 2**10 Enter
1024
>>>
```

The Remainder Operator

In Python, the `%` symbol is the remainder operator. (This is also known as the *modulus operator.*) The remainder operator performs division, but instead of returning the quotient, it returns the remainder. The following statement assigns 2 to `leftover`:

```
leftover = 17 % 3
```

This statement assigns 2 to leftover because 17 divided by 3 is 5 with a remainder of 2. The remainder operator is useful in certain situations. It is commonly used in calculations that convert times or distances, detect odd or even numbers, and perform other specialized operations. For example, Program 2-17 gets a number of seconds from the user, and it converts that number of seconds to hours, minutes, and seconds. For example, it would convert 11,730 seconds to 3 hours, 15 minutes, and 30 seconds.

Program 2-17 `(time_converter.py)`

```
# Get a number of seconds from the user.
total_seconds = float(input('Enter a number of seconds: '))

# Get the number of hours.
hours = total_seconds // 3600

# Get the number of remaining minutes.
minutes = (total_seconds // 60) % 60

# Get the number of remaining seconds.
seconds = total_seconds % 60

# Display the results.
print('Here is the time in hours, minutes, and seconds:')
print('Hours:', hours)
print('Minutes:', minutes)
print('Seconds:', seconds)
```

Program Output (with input shown in bold)

```
Enter a number of seconds: 11730 Enter
Here is the time in hours, minutes, and seconds:
```

```
Hours: 3.0
Minutes: 15.0
Seconds: 30.0
```

Let’s take a closer look at the code:

- Line 2 gets a number of seconds from the user, converts the value to a `float`, and assigns it to the `total_seconds` variable.
- Line 5 calculates the number of hours in the specified number of seconds. There are 3600 seconds in an hour, so this statement divides `total_seconds` by 3600. Notice we used the integer division operator (`//`) operator. This is because we want the number of hours with no fractional part.
- Line 8 calculates the number of remaining minutes. This statement first uses the `//` operator to divide `total_seconds` by 60. This gives us the total number of minutes. Then, it uses the `%` operator to divide the total number of minutes by 60 and get the remainder of the division. The result is the number of remaining minutes.
- Line 11 calculates the number of remaining seconds. There are 60 seconds in a minute, so this statement uses the `%` operator to divide the `total_seconds` by 60 and get the remainder of the division. The result is the number of remaining seconds.
- Lines 14 through 17 display the number of hours, minutes, and seconds.

Converting Math Formulas to Programming Statements

You probably remember from algebra class that the expression $2xy$ is understood to mean 2 times x times y. In math, you do not always use an operator for multiplication. Python, as well as other programming languages, requires an operator for any mathematical operation. Table 2-6 shows some algebraic expressions that perform multiplication and the equivalent programming expressions.

Table 2-6 Algebraic expressions

Algebraic Expression	Operation Being Performed	Programming Expression
$6B$	6 times B	`6 * B`
(3)(12)	3 times 12	`3 * 12`
$4xy$	4 times x times y	`4 * x * y`

When converting some algebraic expressions to programming expressions, you may have to insert parentheses that do not appear in the algebraic expression. For example, look at the following formula:

$$x = \frac{a + b}{c}$$

To convert this to a programming statement, $a + b$ will have to be enclosed in parentheses:

```
x = (a + b)/c
```

Table 2-7 shows additional algebraic expressions and their Python equivalents.

Table 2-7 Algebraic and programming expressions

Algebraic Expression	Python Statement
$y = 3\frac{x}{2}$	`y = 3 * x / 2`
$z = 3bc + 4$	`z = 3 * b * c + 4`
$a = \frac{x + 2}{b - 1}$	`a = (x + 2) / (b - 1)`

In the Spotlight:

Converting a Math Formula to a Programming Statement

Suppose you want to deposit a certain amount of money into a savings account and leave it alone to draw interest for the next 10 years. At the end of 10 years, you would like to have \$10,000 in the account. How much do you need to deposit today to make that happen? You can use the following formula to find out:

$$P = \frac{F}{(1 + r)^n}$$

The terms in the formula are as follows:

- P is the present value, or the amount that you need to deposit today.
- F is the future value that you want in the account. (In this case, F is \$10,000.)
- r is the annual interest rate.
- n is the number of years that you plan to let the money sit in the account.

It would be convenient to write a computer program to perform the calculation because then we can experiment with different values for the variables. Here is an algorithm that we can use:

1. *Get the desired future value.*
2. *Get the annual interest rate.*
3. *Get the number of years that the money will sit in the account.*
4. *Calculate the amount that will have to be deposited.*
5. *Display the result of the calculation in step 4.*

In steps 1 through 3, we will prompt the user to enter the specified values. We will assign the desired future value to a variable named `future_value`, the annual interest rate to a variable named `rate`, and the number of years to a variable named `years`.

In step 4, we calculate the present value, which is the amount of money that we will have to deposit. We will convert the formula previously shown to the following statement. The statement stores the result of the calculation in the `present_value` variable.

```
present_value = future_value / (1.0 + rate)**years
```

In step 5, we display the value in the `present_value` variable. Program 2-18 shows the program.

Program 2-18 (future_value.py)

```python
# Get the desired future value.
future_value = float(input('Enter the desired future value: '))

# Get the annual interest rate.
rate = float(input('Enter the annual interest rate: '))

# Get the number of years that the money will appreciate.
years = int(input('Enter the number of years the money will grow: '))

# Calculate the amount needed to deposit.
present_value = future_value / (1.0 + rate)**years

# Display the amount needed to deposit.
print('You will need to deposit this amount:', present_value)
```

Program Output

```
Enter the desired future value: 10000.0 Enter
Enter the annual interest rate: 0.05 Enter
Enter the number of years the money will grow: 10 Enter
You will need to deposit this amount: 6139.13253541
```

NOTE: Unlike the output shown for this program, dollar amounts are usually rounded to two decimal places. Later in this chapter, you will learn how to format numbers so they are rounded to a specified number of decimal places.

Mixed-Type Expressions and Data Type Conversion

When you perform a math operation on two operands, the data type of the result will depend on the data type of the operands. Python follows these rules when evaluating mathematical expressions:

- When an operation is performed on two `int` values, the result will be an `int`.
- When an operation is performed on two `float` values, the result will be a `float`.
- When an operation is performed on an `int` and a `float`, the `int` value will be temporarily converted to a `float` and the result of the operation will be a `float`. (An expression that uses operands of different data types is called a *mixed-type expression*.)

The first two situations are straightforward: operations on `int`s produce `int`s, and operations on `float`s produce `float`s. Let's look at an example of the third situation, which involves mixed-type expressions:

```python
my_number = 5 * 2.0
```

When this statement executes, the value 5 will be converted to a `float` (5.0) then multiplied by 2.0. The result, 10.0, will be assigned to `my_number`.

The `int` to `float` conversion that takes place in the previous statement happens implicitly. If you need to explicitly perform a conversion, you can use either the `int()` or `float()` functions. For example, you can use the `int()` function to convert a floating-point value to an integer, as shown in the following code:

```
fvalue = 2.6
ivalue = int(fvalue)
```

The first statement assigns the value 2.6 to the `fvalue` variable. The second statement passes `fvalue` as an argument to the `int()` function. The `int()` function returns the value 2, which is assigned to the `ivalue` variable. After this code executes, the `fvalue` variable is still assigned the value 2.6, but the `ivalue` variable is assigned the value 2.

As demonstrated in the previous example, the `int()` function converts a floating-point argument to an integer by truncating it. As previously mentioned, that means it throws away the number's fractional part. Here is an example that uses a negative number:

```
fvalue = −2.9
ivalue = int(fvalue)
```

In the second statement, the value −2 is returned from the `int()` function. After this code executes, the `fvalue` variable references the value −2.9, and the `ivalue` variable references the value −2.

You can use the `float()` function to explicitly convert an `int` to a `float`, as shown in the following code:

```
ivalue = 2
fvalue = float(ivalue)
```

After this code executes, the `ivalue` variable references the integer value 2, and the `fvalue` variable references the floating-point value 2.0.

Breaking Long Statements into Multiple Lines

Most programming statements are written on one line. If a programming statement is too long, however, you will not be able to view all of it in your editor window without scrolling horizontally. In addition, if you print your program code on paper and one of the statements is too long to fit on one line, it will wrap around to the next line and make the code difficult to read.

Python allows you to break a statement into multiple lines by using the *line continuation character,* which is a backslash (`\`). You simply type the backslash character at the point you want to break the statement, then press the Enter key.

For example, here is a statement that performs a mathematical calculation and has been broken up to fit on two lines:

```
result = var1 * 2 + var2 * 3 + \
         var3 * 4 + var4 * 5
```

The line continuation character that appears at the end of the first line tells the interpreter that the statement is continued on the next line.

Python also allows you to break any part of a statement that is enclosed in parentheses into multiple lines without using the line continuation character. For example, look at the following statement:

```
print("Monday's sales are", monday,
      "and Tuesday's sales are", tuesday,
      "and Wednesday's sales are", wednesday)
```

The following code shows another example:

```
total = (value1 + value2 +
         value3 + value4 +
         value5 + value6)
```

Checkpoint

2.19 Complete the following table by writing the value of each expression in the Value column:

Expression	Value
`6 + 3 * 5`	______
`12 / 2 - 4`	______
`9 + 14 * 2 - 6`	______
`(6 + 2) * 3`	______
`14 / (11 - 4)`	______
`9 + 12 * (8 - 3)`	______

2.20 What value will be assigned to `result` after the following statement executes?

```
result = 9 // 2
```

2.21 What value will be assigned to `result` after the following statement executes?

```
result = 9 % 2
```

2.8 String Concatenation

CONCEPT: String concatenation is the appending of one string to the end of another.

A common operation that performed on strings is concatenation, which means to append one string to the end of another string. In Python, we use the + operator to concatenate strings. The + operator produces a string that is the combination of the two strings used as its operands. The following interactive session shows an example:

```
>>> message = 'Hello ' + 'world' Enter
>>> print(message) Enter
Hello world
>>>
```

The first statement combines the strings `'Hello '` and `'world'` to produce the string `'Hello world'`. The string `'Hello world'` is then assigned to the `message` variable. The second statement displays the string.

Program 2-19 further demonstrates string concatenation.

Program 2-19 **(concatenation.py)**

```
1  # This program demonstrates string concatenation.
2  first_name = input('Enter your first name: ')
3  last_name = input('Enter your last name: ')
4
5  # Combine the names with a space between them.
6  full_name = first_name + ' ' + last_name
7
8  # Display the user's full name.
9  print('Your full name is ' + full_name)
```

Program Output (with input shown in bold)

```
Enter your first name: Alex Enter
Enter your last name: Morgan Enter
Your full name is Alex Morgan
```

Let's take a closer look at the program. Lines 2 and 3 prompt the user to enter his or her first and last names. The user's first name is assigned to the `first_name` variable and the user's last name is assigned to the `last_name` variable.

Line 6 assigns the result of a string concatenation to the `full_name` variable. The string that is assigned to the `full_name` variable begins with the value of the `first_name` variable, followed by a space (`' '`), followed by the value of the `last_name` variable. In the example program output, the user entered *Alex* for the first name and *Morgan* for the last name. As a result, the string `'Alex Morgan'` was assigned to the `full_name` variable. The statement in line 9 displays the value of the `full_name` variable.

String concatenation can be useful for breaking up a string literal so a lengthy call to the `print` function can span multiple lines. Here is an example:

```
print('Enter the amount of ' +
      'sales for each day and ' +
      'press Enter.')
```

This statement will display the following:

```
Enter the amount of sales for each day and press Enter.
```

Implicit String Literal Concatenation

When two or more string literals are written adjacent to each other, separated only by spaces, tabs, or newline characters, Python will implicitly concatenate them into a single string. For example, look at the following interactive session:

```
>>> my_str = 'one' 'two' 'three'
>>> print(my_str)
onetwothree
```

In the first line, the string literals `'one'`, `'two'` , and `'three'`are separated only by a space. As a result, Python concatenates them into the single string `'onetwothree'`. The string is then assigned to the `my_str` variable.

Implicit string literal concatenation is commonly used to break up long string literals across multiple lines. Here is an example:

```
print('Enter the amount of '
      'sales for each day and '
      'press Enter.')
```

This statement will display the following:

```
Enter the amount of sales for each day and press Enter.
```

Checkpoint

2.22 What is string concatenation?

2.23 After the follow statement executes, what value will be assigned to the `result` variable?

```
result = '1' + '2'
```

2.24 After the follow statement executes, what value will be assigned to the `result` variable?

```
result = 'h' 'e' 'l' 'l' 'o'
```

2.9 More About the `print` Function

So far, we have discussed only basic ways to display data. Eventually, you will want to exercise more control over the way data appear on the screen. In this section, you will learn more details about the Python `print` function, and you'll see techniques for formatting output in specific ways.

Suppressing the `print` Function's Ending Newline

The `print` function normally displays a line of output. For example, the following three statements will produce three lines of output:

```
print('One')
print('Two')
print('Three')
```

Each of the statements shown here displays a string and then prints a *newline character*. You do not see the newline character, but when it is displayed, it causes the output to advance to the next line. (You can think of the newline character as a special command that causes the computer to start a new line of output.)

If you do not want the `print` function to start a new line of output when it finishes displaying its output, you can pass the special argument `end=' '` to the function, as shown in the following code:

```
print('One', end=' ')
print('Two', end=' ')
print('Three')
```

Notice in the first two statements, the argument `end=' '` is passed to the `print` function. This specifies that the `print` function should print a space instead of a newline character at the end of its output. Here is the output of these statements:

```
One Two Three
```

Sometimes, you might not want the `print` function to print anything at the end of its output, not even a space. If that is the case, you can pass the argument `end=''` to the `print` function, as shown in the following code:

```
print('One', end='')
print('Two', end='')
print('Three')
```

Notice in the argument `end=''` there is no space between the quote marks. This specifies that the `print` function should print nothing at the end of its output. Here is the output of these statements:

```
OneTwoThree
```

Specifying an Item Separator

When multiple arguments are passed to the `print` function, they are automatically separated by a space when they are displayed on the screen. Here is an example, demonstrated in interactive mode:

```
>>> print('One', 'Two', 'Three') Enter
One Two Three
>>>
```

If you do not want a space printed between the items, you can pass the argument `sep=''` to the `print` function, as shown here:

```
>>> print('One', 'Two', 'Three', sep='') Enter
OneTwoThree
>>>
```

You can also use this special argument to specify a character other than the space to separate multiple items. Here is an example:

```
>>> print('One', 'Two', 'Three', sep='*') Enter
One*Two*Three
>>>
```

Notice in this example, we passed the argument `sep='*'` to the `print` function. This specifies that the printed items should be separated with the * character. Here is another example:

```
>>> print('One', 'Two', 'Three', sep='~~~') Enter
One~~~Two~~~Three
>>>
```

Escape Characters

An *escape character* is a special character that is preceded with a backslash (`\`), appearing inside a string literal. When a string literal that contains escape characters is printed, the escape characters are treated as special commands that are embedded in the string.

For example, `\n` is the newline escape character. When the `\n` escape character is printed, it isn't displayed on the screen. Instead, it causes output to advance to the next line. For example, look at the following statement:

```
print('One\nTwo\nThree')
```

When this statement executes, it displays

```
One
Two
Three
```

Python recognizes several escape characters, some of which are listed in Table 2-8.

Table 2-8 Some of Python's escape characters

Escape Character	Effect
`\n`	Causes output to be advanced to the next line.
`\t`	Causes output to skip over to the next horizontal tab position.
`\'`	Causes a single quote mark to be printed.
`\"`	Causes a double quote mark to be printed.
`\\`	Causes a backslash character to be printed.

The `\t` escape character advances the output to the next horizontal tab position. (A tab position normally appears after every eighth character.) The following statements are illustrative:

```
print('Mon\tTues\tWed')
print('Thur\tFri\tSat')
```

This statement prints `Monday`, then advances the output to the next tab position, then prints `Tuesday`, then advances the output to the next tab position, then prints `Wednesday`. The output will look like this:

```
Mon     Tues    Wed
Thur    Fri     Sat
```

You can use the `\'` and `\"` escape characters to display quotation marks. The following statements are illustrative:

```
print("Your assignment is to read \"Hamlet\" by tomorrow.")
print('I\'m ready to begin.')
```

These statements display the following:

```
Your assignment is to read "Hamlet" by tomorrow.
I'm ready to begin.
```

You can use the `\\` escape character to display a backslash, as shown in the following:

```
print('The path is C:\\temp\\data.')
```

This statement will display

```
The path is C:\temp\data.
```

Checkpoint

2.25 How do you suppress the `print` function's ending newline?

2.26 How can you change the character that is automatically displayed between multiple items that are passed to the `print` function?

2.27 What is the `'\n'` escape character?

2.10 Displaying Formatted Output with F-strings

CONCEPT: F-strings are a special type of string literal that allow you to format values in a variety of ways.

NOTE: F-strings were introduced in Python 3.6. If you are using an earlier version of Python, consider using the `format` function instead. See Appendix F for more information.

F-strings, or *formatted string literals*, give you an easy way to format the output that you want to display with the `print` function. With an f-string, you can create messages that contain the contents of variables, and you can format numbers in a variety of ways.

An f-string is a string literal that is enclosed in quotation marks and prefixed with the letter `f`. Here is a very simple example of an f-string:

```
f'Hello world'
```

This looks like an ordinary string literal, except that it is prefixed with the letter `f`. If we want to display an f-string, we pass it to the `print` function as shown in the following interactive session:

```
>>> print(f'Hello world') Enter
Hello world
```

F-strings are much more powerful than regular string literals, however. An f-string can contain placeholders for variables and other expressions. For example, look at the following interactive session:

```
>>> name = 'Johnny' Enter
>>> print(f'Hello {name}.') Enter
Hello Johnny.
```