

TOPICS

5.1	Introduction to Functions	5.7	Introduction to Value-Returning Functions: Generating Random Numbers
5.2	Defining and Calling a Void Function	5.8	Writing Your Own Value-Returning Functions
5.3	Designing a Program to Use Functions	5.9	The <code>math</code> Module
5.4	Local Variables	5.10	Storing Functions in Modules
5.5	Passing Arguments to Functions		
5.6	Global Variables and Global Constants		

5.1

Introduction to Functions

CONCEPT: A function is a group of statements that exist within a program for the purpose of performing a specific task.

In Chapter 2 we described a simple algorithm for calculating an employee's pay. In the algorithm, the number of hours worked is multiplied by an hourly pay rate. A more realistic payroll algorithm, however, would do much more than this. In a real-world application, the overall task of calculating an employee's pay would consist of several subtasks, such as the following:

- Getting the employee's hourly pay rate
- Getting the number of hours worked
- Calculating the employee's gross pay
- Calculating overtime pay
- Calculating withholdings for taxes and benefits
- Calculating the net pay
- Printing the paycheck

Most programs perform tasks that are large enough to be broken down into several subtasks. For this reason, programmers usually break down their programs into small manageable pieces known as functions. A *function* is a group of statements that exist within a program for the purpose of performing a specific task. Instead of writing a large program as one long sequence of statements, it can be written as several small functions, each one

performing a specific part of the task. These small functions can then be executed in the desired order to perform the overall task.

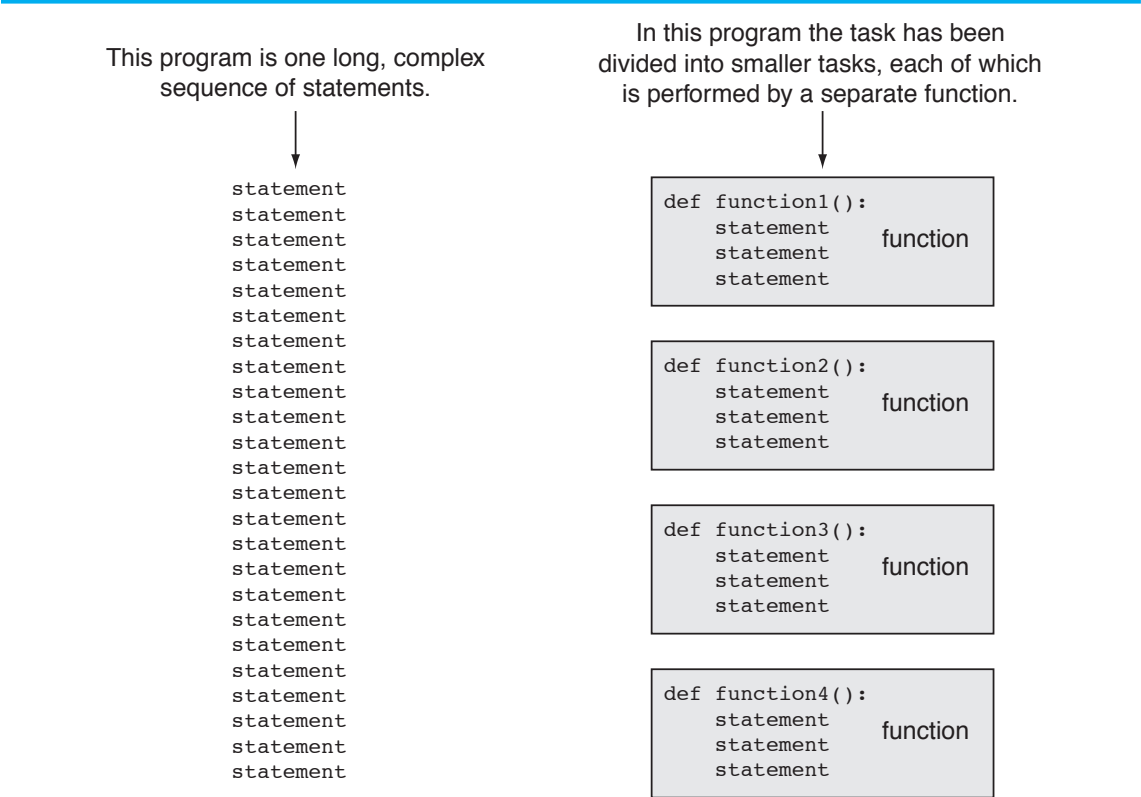
This approach is sometimes called *divide and conquer* because a large task is divided into several smaller tasks that are easily performed. Figure 5-1 illustrates this idea by comparing two programs: one that uses a long complex sequence of statements to perform a task, and another that divides a task into smaller tasks, each of which is performed by a separate function.

When using functions in a program, you generally isolate each task within the program in its own function. For example, a realistic pay calculating program might have the following functions:

- A function that gets the employee’s hourly pay rate
- A function that gets the number of hours worked
- A function that calculates the employee’s gross pay
- A function that calculates the overtime pay
- A function that calculates the withholdings for taxes and benefits
- A function that calculates the net pay
- A function that prints the paycheck

A program that has been written with each task in its own function is called a *modularized program*.

Figure 5-1 Using functions to divide and conquer a large task



Benefits of Modularizing a Program with Functions

A program benefits in the following ways when it is broken down into functions:

Simpler Code

A program's code tends to be simpler and easier to understand when it is broken down into functions. Several small functions are much easier to read than one long sequence of statements.

Code Reuse

Functions also reduce the duplication of code within a program. If a specific operation is performed in several places in a program, a function can be written once to perform that operation and then be executed any time it is needed. This benefit of using functions is known as *code reuse* because you are writing the code to perform a task once and then reusing it each time you need to perform the task.

Better Testing

When each task within a program is contained in its own function, testing and debugging becomes simpler. Programmers can test each function in a program individually, to determine whether it correctly performs its operation. This makes it easier to isolate and fix errors.

Faster Development

Suppose a programmer or a team of programmers is developing multiple programs. They discover that each of the programs perform several common tasks, such as asking for a username and a password, displaying the current time, and so on. It doesn't make sense to write the code for these tasks multiple times. Instead, functions can be written for the commonly needed tasks, and those functions can be incorporated into each program that needs them.

Easier Facilitation of Teamwork

Functions also make it easier for programmers to work in teams. When a program is developed as a set of functions that each performs an individual task, then different programmers can be assigned the job of writing different functions.

Void Functions and Value-Returning Functions

In this chapter you will learn to write two types of functions: void functions and value-returning functions. When you call a *void function*, it simply executes the statements it contains and then terminates. When you call a *value-returning function*, it executes the statements that it contains, and then it returns a value back to the statement that called it. The `input` function is an example of a value-returning function. When you call the `input` function, it gets the data that the user types on the keyboard and returns that data as a string. The `int` and `float` functions are also examples of value-returning functions. You pass an argument to the `int` function, and it returns that argument's value converted to an integer. Likewise, you pass an argument to the `float` function, and it returns that argument's value converted to a floating-point number.

The first type of function that you will learn to write is the void function.

**Checkpoint**

- 5.1 What is a function?
- 5.2 What is meant by the phrase “divide and conquer”?
- 5.3 How do functions help you reuse code in a program?
- 5.4 How can functions make the development of multiple programs faster?
- 5.5 How can functions make it easier for programs to be developed by teams of programmers?

5.2**Defining and Calling a Void Function**

CONCEPT: The code for a function is known as a function definition. To execute the function, you write a statement that calls it.

Function Names

Before we discuss the process of creating and using functions, we should mention a few things about function names. Just as you name the variables that you use in a program, you also name the functions. A function’s name should be descriptive enough so that anyone reading your code can reasonably guess what the function does.

Python requires that you follow the same rules that you follow when naming variables, which we recap here:

- You cannot use one of Python’s key words as a function name. (See Table 1-2 for a list of the key words.)
- A function 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.

Because functions perform actions, most programmers prefer to use verbs in function names. For example, a function that calculates gross pay might be named `calculate_gross_pay`. This name would make it evident to anyone reading the code that the function calculates something. What does it calculate? The gross pay, of course. Other examples of good function names would be `get_hours`, `get_pay_rate`, `calculate_overtime`, `print_check`, and so on. Each function name describes what the function does.

Defining and Calling a Function

To create a function you write its *definition*. Here is the general format of a function definition in Python:

```
def function_name():
    statement
    statement
    etc.
```



VideoNote
Defining and
Calling a function

The first line is known as the *function header*. It marks the beginning of the function definition. The function header begins with the key word `def`, followed by the name of the function, followed by a set of parentheses, followed by a colon.

Beginning at the next line is a set of statements known as a block. A *block* is simply a set of statements that belong together as a group. These statements are performed any time the function is executed. Notice in the general format that all of the statements in the block are indented. This indentation is required because the Python interpreter uses it to tell where the block begins and ends.

Let's look at an example of a function. Keep in mind that this is not a complete program. We will show the entire program in a moment.

```
def message():  
    print('I am Arthur,')  
    print('King of the Britons.')
```

This code defines a function named `message`. The `message` function contains a block with two statements. Executing the function will cause these statements to execute.

Calling a Function

A function definition specifies what a function does, but it does not cause the function to execute. To execute a function, you must *call* it. This is how we would call the `message` function:

```
message()
```

When a function is called, the interpreter jumps to that function and executes the statements in its block. Then, when the end of the block is reached, the interpreter jumps back to the part of the program that called the function, and the program resumes execution at that point. When this happens, we say that the function *returns*. To fully demonstrate how function calling works, we will look at Program 5-1.

Program 5-1 (function_demo.py)

```
1 # This program demonstrates a function.  
2 # First, we define a function named message.  
3 def message():  
4     print('I am Arthur,')  
5     print('King of the Britons.')
```

```
6  
7 # Call the message function.  
8 message()
```

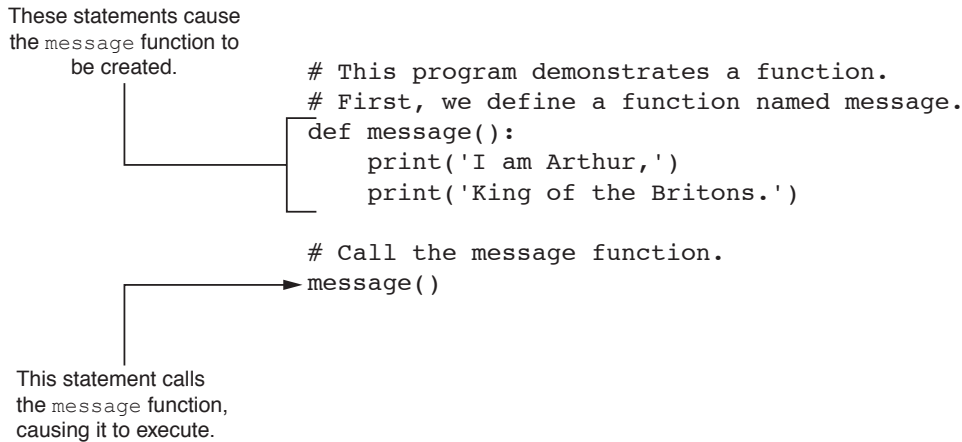
Program Output

```
I am Arthur,  
King of the Britons.
```

Let's step through this program and examine what happens when it runs. First, the interpreter ignores the comments that appear in lines 1 and 2. Then, it reads the `def` statement

in line 3. This causes a function named `message` to be created in memory, containing the block of statements in lines 4 and 5. (Remember, a function definition creates a function, but it does not cause the function to execute.) Next, the interpreter encounters the comment in line 7, which is ignored. Then it executes the statement in line 8, which is a function call. This causes the `message` function to execute, which prints the two lines of output. Figure 5-2 illustrates the parts of this program.

Figure 5-2 The function definition and the function call



Program 5-1 has only one function, but it is possible to define many functions in a program. In fact, it is common for a program to have a main function that is called when the program starts. The main function then calls other functions in the program as they are needed. It is often said that the main function contains a program's *mainline logic*, which is the overall logic of the program. Program 5-2 shows an example of a program with two functions: `main` and `message`.

Program 5-2 (two_functions.py)

```

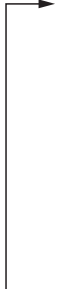
1  # This program has two functions. First we
2  # define the main function.
3  def main():
4      print('I have a message for you.')
5      message()
6      print('Goodbye!')
7
8  # Next we define the message function.
9  def message():
10     print('I am Arthur,')
11     print('King of the Britons.')
12
13 # Call the main function.
14 main()

```

Program Output

```
I have a message for you.  
I am Arthur,  
King of the Britons.  
Goodbye!
```


The definition of the `main` function appears in lines 3 through 6, and the definition of the `message` function appears in lines 9 through 11. The statement in line 14 calls the `main` function, as shown in Figure 5-3.

Figure 5-3 Calling the `main` function

```
# This program has two functions. First we  
# define the main function.  
def main():  
    print('I have a message for you.')  
    message()  
    print('Goodbye!')  
  
# Next we define the message function.  
def message():  
    print('I am Arthur,')  
    print('King of the Britons.')  
  
# Call the main function.  
main()
```

The interpreter jumps to the `main` function and begins executing the statements in its block.

The first statement in the `main` function calls the `print` function in line 4. It displays the string `'I have a message for you.'`. Then, the statement in line 5 calls the `message` function. This causes the interpreter to jump to the `message` function, as shown in Figure 5-4. After the statements in the `message` function have executed, the interpreter returns to the `main` function and resumes with the statement that immediately follows the function call. As shown in Figure 5-5, this is the statement that displays the string `'Goodbye!'`.

Figure 5-4 Calling the `message` function

```
# This program has two functions. First we  
# define the main function.  
def main():  
    print('I have a message for you.')  
    message()  
    print('Goodbye!')  
  
# Next we define the message function.  
def message():  
    print('I am Arthur,')  
    print('King of the Britons.')  
  
# Call the main function.  
main()
```

The interpreter jumps to the `message` function and begins executing the statements in its block.

Figure 5-5 The message function returns

When the message function ends, the interpreter jumps back to the part of the program that called it and resumes execution from that point.

```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the main function.
main()
```

That is the end of the main function, so the function returns as shown in Figure 5-6. There are no more statements to execute, so the program ends.

Figure 5-6 The main function returns

When the main function ends, the interpreter jumps back to the part of the program that called it. There are no more statements, so the program ends.

```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the main function.
main()
```



NOTE: When a program calls a function, programmers commonly say that the *control* of the program transfers to that function. This simply means that the function takes control of the program's execution.

Indentation in Python

In Python, each line in a block must be indented. As shown in Figure 5-7, the last indented line after a function header is the last line in the function's block.

Figure 5-7 All of the statements in a block are indented

The last indented line is the last line in the block.

```
def greeting():
    print('Good morning!')
    print('Today we will learn about functions.')
```

These statements are not in the block.

```
print('I will call the greeting function.')
greeting()
```


When you indent the lines in a block, make sure each line begins with the same number of spaces. Otherwise an error will occur. For example, the following function definition will cause an error because the lines are all indented with different numbers of spaces.

```
def my_function():  
    print('And now for')  
    print('something completely')  
    print('different.')
```

In an editor there are two ways to indent a line: (1) by pressing the Tab key at the beginning of the line, or (2) by using the spacebar to insert spaces at the beginning of the line. You can use either tabs or spaces when indenting the lines in a block, but don't use both. Doing so may confuse the Python interpreter and cause an error.

IDLE, as well as most other Python editors, automatically indents the lines in a block. When you type the colon at the end of a function header, all of the lines typed afterward will automatically be indented. After you have typed the last line of the block you press the Backspace key to get out of the automatic indentation.



TIP: Python programmers customarily use four spaces to indent the lines in a block. You can use any number of spaces you wish, as long as all the lines in the block are indented by the same amount.



NOTE: Blank lines that appear in a block are ignored.



Checkpoint

- 5.6 A function definition has what two parts?
- 5.7 What does the phrase “calling a function” mean?
- 5.8 When a function is executing, what happens when the end of the function's block is reached?
- 5.9 Why must you indent the statements in a block?

5.3

Designing a Program to Use Functions

CONCEPT: Programmers commonly use a technique known as top-down design to break down an algorithm into functions.

Flowcharting a Program with Functions

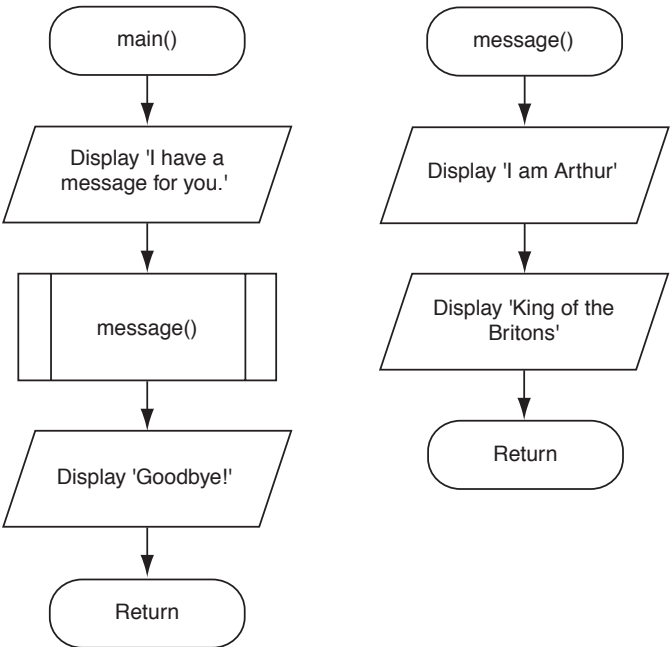
In Chapter 2 we introduced flowcharts as a tool for designing programs. In a flowchart, a function call is shown with a rectangle that has vertical bars at each side, as shown in Figure 5-8. The name of the function that is being called is written on the symbol. The example shown in Figure 5-8 shows how we would represent a call to the `message` function.

Figure 5-8 Function call symbol



Programmers typically draw a separate flowchart for each function in a program. For example, Figure 5-9 shows how the `main` function and the `message` function in Program 5-2 would be flowcharted. When drawing a flowchart for a function, the starting terminal symbol usually shows the name of the function and the ending terminal symbol usually reads `Return`.

Figure 5-9 Flowchart for Program 5-2



Top-Down Design

In this section, we have discussed and demonstrated how functions work. You’ve seen how control of a program is transferred to a function when it is called and then returns to the part of the program that called the function when the function ends. It is important that you understand these mechanical aspects of functions.

Just as important as understanding how functions work is understanding how to design a program that uses functions. Programmers commonly use a technique known as *top-down design* to break down an algorithm into functions. The process of top-down design is performed in the following manner:

- The overall task that the program is to perform is broken down into a series of subtasks.

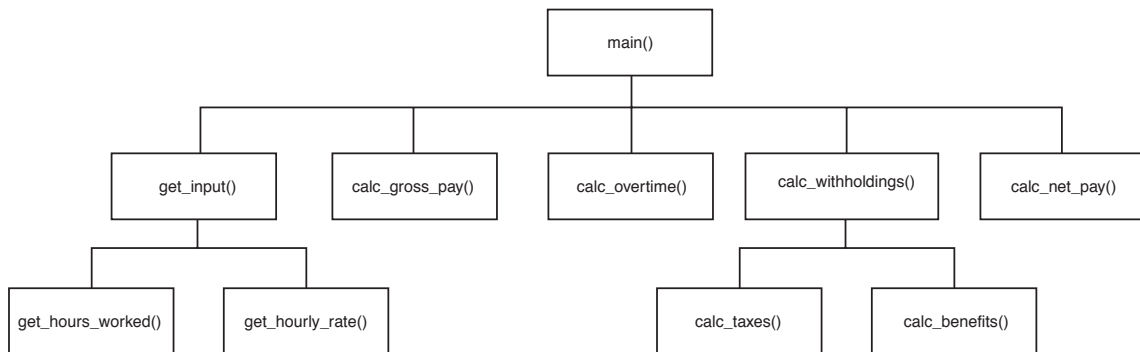
- Each of the subtasks is examined to determine whether it can be further broken down into more subtasks. This step is repeated until no more subtasks can be identified.
- Once all of the subtasks have been identified, they are written in code.

This process is called top-down design because the programmer begins by looking at the topmost level of tasks that must be performed and then breaks down those tasks into lower levels of subtasks.

Hierarchy Charts

Flowcharts are good tools for graphically depicting the flow of logic inside a function, but they do not give a visual representation of the relationships between functions. Programmers commonly use *hierarchy charts* for this purpose. A hierarchy chart, which is also known as a *structure chart*, shows boxes that represent each function in a program. The boxes are connected in a way that illustrates the functions called by each function. Figure 5-10 shows an example of a hierarchy chart for a hypothetical pay calculating program.

Figure 5-10 A hierarchy chart



The chart shown in Figure 5-10 shows the main function as the topmost function in the hierarchy. The main function calls five other functions: `get_input`, `calc_gross_pay`, `calc_overtime`, `calc_withholdings`, and `calc_net_pay`. The `get_input` function calls two additional functions: `get_hours_worked` and `get_hourly_rate`. The `calc_withholdings` function also calls two functions: `calc_taxes` and `calc_benefits`.

Notice that the hierarchy chart does not show the steps that are taken inside a function. Because they do not reveal any details about how functions work, they do not replace flowcharts or pseudocode.

In the Spotlight:

Defining and Calling Functions

Professional Appliance Service, Inc. offers maintenance and repair services for household appliances. The owner wants to give each of the company's service technicians a small handheld computer that displays step-by-step instructions for many of the repairs that they



perform. To see how this might work, the owner has asked you to develop a program that displays the following instructions for disassembling an Acme laundry dryer:

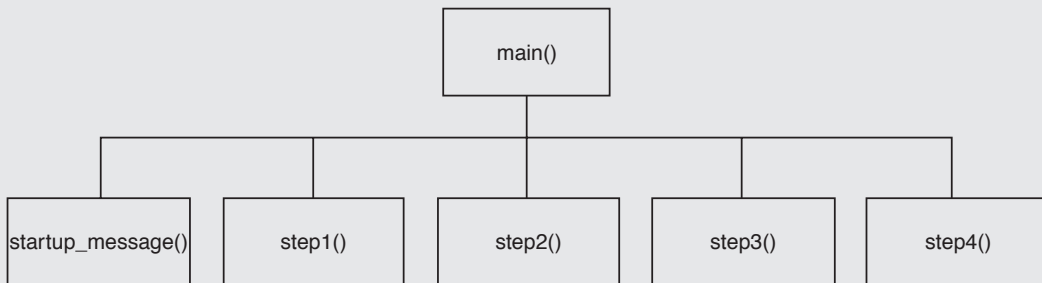
- Step 1: Unplug the dryer and move it away from the wall.
- Step 2: Remove the six screws from the back of the dryer.
- Step 3: Remove the dryer's back panel.
- Step 4: Pull the top of the dryer straight up.

During your interview with the owner, you determine that the program should display the steps one at a time. You decide that after each step is displayed, the user will be asked to press the Enter key to see the next step. Here is the algorithm in pseudocode:

Display a starting message, explaining what the program does.
Ask the user to press Enter to see step 1.
Display the instructions for step 1.
Ask the user to press Enter to see the next step.
Display the instructions for step 2.
Ask the user to press Enter to see the next step.
Display the instructions for step 3.
Ask the user to press Enter to see the next step.
Display the instructions for step 4.

This algorithm lists the top level of tasks that the program needs to perform and becomes the basis of the program's main function. Figure 5-11 shows the program's structure in a hierarchy chart.

Figure 5-11 Hierarchy chart for the program



As you can see from the hierarchy chart, the main function will call several other functions. Here are summaries of those functions:

- **startup_message**—This function will display the starting message that tells the technician what the program does.
- **step1**—This function will display the instructions for step 1.
- **step2**—This function will display the instructions for step 2.
- **step3**—This function will display the instructions for step 3.
- **step4**—This function will display the instructions for step 4.

Between calls to these functions, the main function will instruct the user to press a key to see the next step in the instructions. Program 5-3 shows the code for the program.

Program 5-3 (acme_dryer.py)

```
1  # This program displays step-by-step instructions
2  # for disassembling an Acme dryer.
3  # The main function performs the program's main logic.
4  def main():
5      # Display the start-up message.
6      startup_message()
7      input('Press Enter to see Step 1.')
8      # Display step 1.
9      step1()
10     input('Press Enter to see Step 2.')
11     # Display step 2.
12     step2()
13     input('Press Enter to see Step 3.')
14     # Display step 3.
15     step3()
16     input('Press Enter to see Step 4.')
17     # Display step 4.
18     step4()
19
20 # The startup_message function displays the
21 # program's initial message on the screen.
22 def startup_message():
23     print('This program tells you how to')
24     print('disassemble an ACME laundry dryer.')
25     print('There are 4 steps in the process.')
26     print()
27
28 # The step1 function displays the instructions
29 # for step 1.
30 def step1():
31     print('Step 1: Unplug the dryer and')
32     print('move it away from the wall.')
33     print()
34
35 # The step2 function displays the instructions
36 # for step 2.
37 def step2():
38     print('Step 2: Remove the six screws')
39     print('from the back of the dryer.')
40     print()
41
42 # The step3 function displays the instructions
43 # for step 3.
44 def step3():
45     print('Step 3: Remove the back panel')
```

(program continues)

Program 5-3 (continued)

```
46     print('from the dryer.')
47     print()
48
49     # The step4 function displays the instructions
50     # for step 4.
51     def step4():
52         print('Step 4: Pull the top of the')
53         print('dryer straight up.')
54
55     # Call the main function to begin the program.
56     main()
```

Program Output

This program tells you how to
disassemble an ACME laundry dryer.
There are 4 steps in the process.

Press Enter to see Step 1.

Step 1: Unplug the dryer and
move it away from the wall.

Press Enter to see Step 2.

Step 2: Remove the six screws
from the back of the dryer.

Press Enter to see Step 3.

Step 3: Remove the back panel
from the dryer.

Press Enter to see Step 4.

Step 4: Pull the top of the
dryer straight up.

Pausing Execution Until the User Presses Enter

Sometimes you want a program to pause so the user can read information that has been displayed on the screen. When the user is ready for the program to continue execution, he or she presses the Enter key and the program resumes. In Python you can use the `input` function to cause a program to pause until the user presses the Enter key. Line 7 in Program 5-3 is an example:

```
input('Press Enter to see Step 1.')
```

This statement displays the prompt 'Press Enter to see Step 1.' and pauses until the user presses the Enter key. The program also uses this technique in lines 10, 13, and 16.

5.4 Local Variables

CONCEPT: A local variable is created inside a function and cannot be accessed by statements that are outside the function. Different functions can have local variables with the same names because the functions cannot see each other's local variables.

Anytime you assign a value to a variable inside a function, you create a *local variable*. A local variable belongs to the function in which it is created, and only statements inside that function can access the variable. (The term *local* is meant to indicate that the variable can be used only locally, within the function in which it is created.)

An error will occur if a statement in one function tries to access a local variable that belongs to another function. For example, look at Program 5-4.

Program 5-4 (bad_local.py)

```
1  # Definition of the main function.
2  def main():
3      get_name()
4      print('Hello', name)      # This causes an error!
5
6  # Definition of the get_name function.
7  def get_name():
8      name = input('Enter your name: ')
9
10 # Call the main function.
11 main()
```

This program has two functions: `main` and `get_name`. In line 8 the `name` variable is assigned a value that is entered by the user. This statement is inside the `get_name` function, so the `name` variable is local to that function. This means that the `name` variable cannot be accessed by statements outside the `get_name` function.

The `main` function calls the `get_name` function in line 3. Then, the statement in line 4 tries to access the `name` variable. This results in an error because the `name` variable is local to the `get_name` function, and statements in the `main` function cannot access it.

Scope and Local Variables

A variable's *scope* is the part of a program in which the variable may be accessed. A variable is visible only to statements in the variable's scope. A local variable's scope is the function in which the variable is created. As you saw demonstrated in Program 5-4, no statement outside the function may access the variable.

In addition, a local variable cannot be accessed by code that appears inside the function at a point before the variable has been created. For example, look at the following function. It will cause an error because the `print` function tries to access the `val` variable, but this statement appears before the `val` variable has been created. Moving the assignment statement to a line before the `print` statement will fix this error.

```
def bad_function():
    print('The value is', val)    # This will cause an error!
    val = 99
```

Because a function's local variables are hidden from other functions, the other functions may have their own local variables with the same name. For example, look at Program 5-5. In addition to the main function, this program has two other functions: `texas` and `california`. These two functions each have a local variable named `birds`.

Program 5-5 (birds.py)

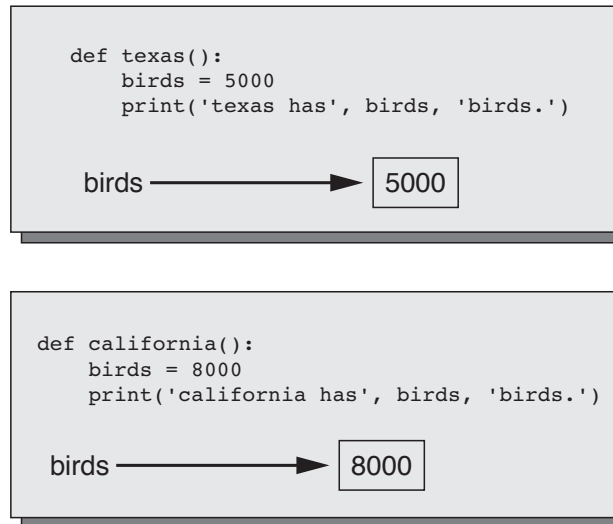
```
1  # This program demonstrates two functions that
2  # have local variables with the same name.
3
4  def main():
5      # Call the texas function.
6      texas()
7      # Call the california function.
8      california()
9
10 # Definition of the texas function. It creates
11 # a local variable named birds.
12 def texas():
13     birds = 5000
14     print('texas has', birds, 'birds.')
15
16 # Definition of the california function. It also
17 # creates a local variable named birds.
18 def california():
19     birds = 8000
20     print('california has', birds, 'birds.')
21
22 # Call the main function.
23 main()
```

Program Output

```
texas has 5000 birds.
california has 8000 birds.
```


Although there are two separate variables named `birds` in this program, only one of them is visible at a time because they are in different functions. This is illustrated in Figure 5-12. When the `texas` function is executing, the `birds` variable that is created in line 13 is visible. When the `california` function is executing, the `birds` variable that is created in line 19 is visible.

Figure 5-12 Each function has its own `birds` variable



Checkpoint

- 5.10 What is a local variable? How is access to a local variable restricted?
- 5.11 What is a variable's scope?
- 5.12 Is it permissible for a local variable in one function to have the same name as a local variable in a different function?

5.5

Passing Arguments to Functions

CONCEPT: An argument is any piece of data that is passed into a function when the function is called. A parameter is a variable that receives an argument that is passed into a function.



VideoNote
Passing Arguments
to a Function

Sometimes it is useful not only to call a function, but also to send one or more pieces of data into the function. Pieces of data that are sent into a function are known as *arguments*. The function can use its arguments in calculations or other operations.

If you want a function to receive arguments when it is called, you must equip the function with one or more parameter variables. A *parameter variable*, often simply called a *parameter*, is a special variable that is assigned the value of an argument when a function is called. Here is an example of a function that has a parameter variable:

```
def show_double(number):
    result = number * 2
    print(result)
```

This function's name is `show_double`. Its purpose is to accept a number as an argument and display the value of that number doubled. Look at the function header and notice the word `number` that appear inside the parentheses. This is the name of a parameter variable. This variable will be assigned the value of an argument when the function is called. Program 5-6 demonstrates the function in a complete program.

Program 5-6 (pass_arg.py)

```
1  # This program demonstrates an argument being
2  # passed to a function.
3
4  def main():
5      value = 5
6      show_double(value)
7
8  # The show_double function accepts an argument
9  # and displays double its value.
10 def show_double(number):
11     result = number * 2
12     print(result)
13
14 # Call the main function.
15 main()
```

Program Output

10

When this program runs, the `main` function is called in line 15. Inside the `main` function, line 5 creates a local variable named `value`, assigned the value 5. Then the following statement in line 6 calls the `show_double` function:


```
show_double(value)
```

Notice that `value` appears inside the parentheses. This means that `value` is being passed as an argument to the `show_double` function, as shown in Figure 5-13. When this statement executes, the `show_double` function will be called, and the `number` parameter will be assigned the same value as the `value` variable. This is shown in Figure 5-14.

Figure 5-13 The `value` variable is passed as an argument

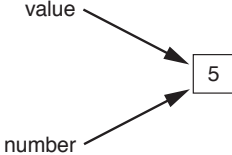
```
def main():
    value = 5
    show_double(value)

def show_double(number):
    result = number * 2
    print(result)
```


Figure 5-14 The `value` variable and the `number` parameter reference the same value

```
def main():
    value = 5
    show_double(value)

def show_double(number):
    result = number * 2
    print(result)
```



Let's step through the `show_double` function. As we do, remember that the `number` parameter variable will be assigned the value that was passed to it as an argument. In this program, that number is 5.

Line 11 assigns the value of the expression `number * 2` to a local variable named `result`. Because `number` references the value 5, this statement assigns 10 to `result`. Line 12 displays the `result` variable.

The following statement shows how the `show_double` function can be called with a numeric literal passed as an argument:

```
show_double(50)
```

This statement executes the `show_double` function, assigning 50 to the `number` parameter. The function will print 100.

Parameter Variable Scope

Earlier in this chapter, you learned that a variable's scope is the part of the program in which the variable may be accessed. A variable is visible only to statements inside the variable's scope. A parameter variable's scope is the function in which the parameter is used. All of the statements inside the function can access the parameter variable, but no statement outside the function can access it.



In the Spotlight:

Passing an Argument to a Function

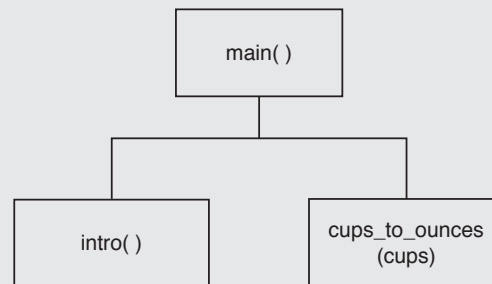
Your friend Michael runs a catering company. Some of the ingredients that his recipes require are measured in cups. When he goes to the grocery store to buy those ingredients, however, they are sold only by the fluid ounce. He has asked you to write a simple program that converts cups to fluid ounces.

You design the following algorithm:

1. *Display an introductory screen that explains what the program does.*
2. *Get the number of cups.*
3. *Convert the number of cups to fluid ounces and display the result.*

This algorithm lists the top level of tasks that the program needs to perform and becomes the basis of the program's main function. Figure 5-15 shows the program's structure in a hierarchy chart.

Figure 5-15 Hierarchy chart for the program



As shown in the hierarchy chart, the `main` function will call two other functions.

Here are summaries of those functions:

- `intro`—This function will display a message on the screen that explains what the program does.
- `cups_to_ounces`—This function will accept the number of cups as an argument and calculate and display the equivalent number of fluid ounces.

In addition to calling these functions, the `main` function will ask the user to enter the number of cups. This value will be passed to the `cups_to_ounces` function. The code for the program is shown in Program 5-7.

Program 5-7 (`cups_to_ounces.py`)

```

1  # This program converts cups to fluid ounces.
2
3  def main():
4      # display the intro screen.
```

```
5     intro()
6     # Get the number of cups.
7     cups_needed = int(input('Enter the number of cups: '))
8     # Convert the cups to ounces.
9     cups_to_ounces(cups_needed)
10
11 # The intro function displays an introductory screen.
12 def intro():
13     print('This program converts measurements')
14     print('in cups to fluid ounces. For your')
15     print('reference the formula is:')
16     print(' 1 cup = 8 fluid ounces')
17     print()
18
19 # The cups_to_ounces function accepts a number of
20 # cups and displays the equivalent number of ounces.
21 def cups_to_ounces(cups):
22     ounces = cups * 8
23     print('That converts to', ounces, 'ounces.')
24
25 # Call the main function.
26 main()
```

Program Output (with input shown in bold)

This program converts measurements
in cups to fluid ounces. For your
reference the formula is:

1 cup = 8 fluid ounces

Enter the number of cups: **4**

That converts to 32 ounces.

Passing Multiple Arguments

Often it's useful to write functions that can accept multiple arguments. Program 5-8 shows a function named `show_sum`, that accepts two arguments. The function adds the two arguments and displays their sum.

Program 5-8 (multiple_args.py)

```
1 # This program demonstrates a function that accepts
2 # two arguments.
3
4 def main():
5     print('The sum of 12 and 45 is')
```

(program continues)

Program 5-8 (continued)

```

6     show_sum(12, 45)
7
8     # The show_sum function accepts two arguments
9     # and displays their sum.
10    def show_sum(num1, num2):
11        result = num1 + num2
12        print(result)
13
14    # Call the main function.
15    main()

```

Program Output

```

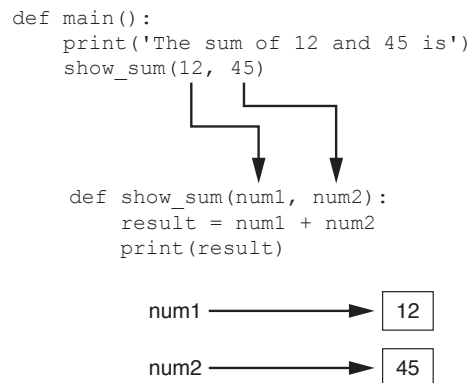
The sum of 12 and 45 is
57

```

Notice that two parameter variable names, `num1` and `num2`, appear inside the parentheses in the `show_sum` function header. This is often referred to as a *parameter list*. Also notice that a comma separates the variable names.

The statement in line 6 calls the `show_sum` function and passes two arguments: 12 and 45. These arguments are passed *by position* to the corresponding parameter variables in the function. In other words, the first argument is passed to the first parameter variable, and the second argument is passed to the second parameter variable. So, this statement causes 12 to be assigned to the `num1` parameter and 45 to be assigned to the `num2` parameter, as shown in Figure 5-16.

Figure 5-16 Two arguments passed to two parameters



Suppose we were to reverse the order in which the arguments are listed in the function call, as shown here:

```
show_sum(45, 12)
```

This would cause 45 to be passed to the `num1` parameter and 12 to be passed to the `num2` parameter. The following code shows another example. This time we are passing variables as arguments.

```
value1 = 2
value2 = 3
show_sum(value1, value2)
```

When the `show_sum` function executes as a result of this code, the `num1` parameter will be assigned the value 2 and the `num2` parameter will be assigned the value 3.

Program 5-9 shows one more example. This program passes two strings as arguments to a function.

Program 5-9 (string_args.py)

```
1 # This program demonstrates passing two string
2 # arguments to a function.
3
4 def main():
5     first_name = input('Enter your first name: ')
6     last_name = input('Enter your last name: ')
7     print('Your name reversed is')
8     reverse_name(first_name, last_name)
9
10 def reverse_name(first, last):
11     print(last, first)
12
13 # Call the main function.
14 main()
```

Program Output (with input shown in bold)

```
Enter your first name: Matt 
Enter your last name: Hoyle 
Your name reversed is
Hoyle Matt
```

Making Changes to Parameters

When an argument is passed to a function in Python, the function parameter variable will reference the argument's value. However, any changes that are made to the parameter variable will not affect the argument. To demonstrate this look at Program 5-10.

Program 5-10 (change_me.py)

```
1 # This program demonstrates what happens when you
2 # change the value of a parameter.
3
```

(program continues)

Program 5-10 (continued)

```

4  def main():
5      value = 99
6      print('The value is', value)
7      change_me(value)
8      print('Back in main the value is', value)
9
10 def change_me(arg):
11     print('I am changing the value.')
12     arg = 0
13     print('Now the value is', arg)
14
15 # Call the main function.
16 main()

```

Program Output

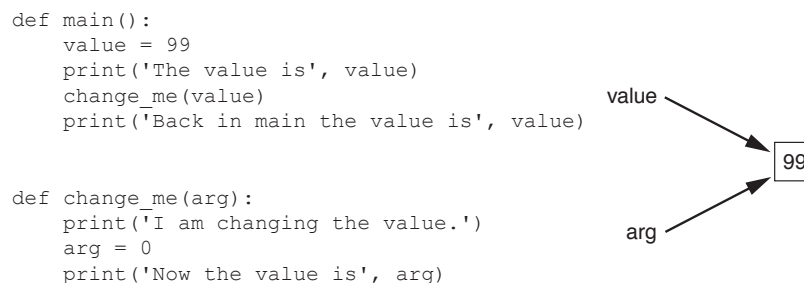
```

The value is 99
I am changing the value.
Now the value is 0
Back in main the value is 99

```

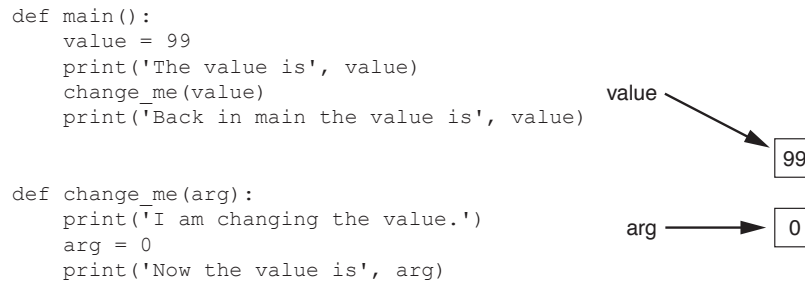
The main function creates a local variable named `value` in line 5, assigned the value 99. The statement in line 6 displays 'The value is 99'. The `value` variable is then passed as an argument to the `change_me` function in line 7. This means that in the `change_me` function the `arg` parameter will also reference the value 99. This is shown in Figure 5-17.

Figure 5-17 The value variable is passed to the `change_me` function



Inside the `change_me` function, in line 12, the `arg` parameter is assigned the value 0. This reassignment changes `arg`, but it does not affect the `value` variable in `main`. As shown in Figure 5-18, the two variables now reference different values in memory. The statement in line 13 displays 'Now the value is 0' and the function ends.

Control of the program then returns to the main function. The next statement to execute is in line 8. This statement displays 'Back in main the value is 99'. This proves that

Figure 5-18 The value variable is passed to the `change_me` function

even though the parameter variable `arg` was changed in the `change_me` function, the argument (the `value` variable in `main`) was not modified.

The form of argument passing that is used in Python, where a function cannot change the value of an argument that was passed to it, is commonly called *pass by value*. This is a way that one function can communicate with another function. The communication channel works in only one direction, however. The calling function can communicate with the called function, but the called function cannot use the argument to communicate with the calling function. Later in this chapter you will learn how to write a function that can communicate with the part of the program that called it by returning a value.

Keyword Arguments

Programs 5-8 and 5-9 demonstrate how arguments are passed by position to parameter variables in a function. Most programming languages match function arguments and parameters this way. In addition to this conventional form of argument passing, the Python language allows you to write an argument in the following format, to specify which parameter variable the argument should be passed to:

```
parameter_name=value
```

In this format, *parameter_name* is the name of a parameter variable and *value* is the value being passed to that parameter. An argument that is written in accordance with this syntax is known as a *keyword argument*.

Program 5-11 demonstrates keyword arguments. This program uses a function named `show_interest` that displays the amount of simple interest earned by a bank account for a number of periods. The function accepts the arguments `principal` (for the account principal), `rate` (for the interest rate per period), and `periods` (for the number of periods). When the function is called in line 7, the arguments are passed as keyword arguments.

Program 5-11 (keyword_args.py)

```

1  # This program demonstrates keyword arguments.
2
3  def main():
4      # Show the amount of simple interest, using 0.01 as
5      # interest rate per period, 10 as the number of periods,

```

(program continues)

Program 5-11 (continued)

```

6      # and $10,000 as the principal.
7      show_interest(rate=0.01, periods=10, principal=10000.0)
8
9      # The show_interest function displays the amount of
10     # simple interest for a given principal, interest rate
11     # per period, and number of periods.
12
13     def show_interest(principal, rate, periods):
14         interest = principal * rate * periods
15         print('The simple interest will be $', \
16               format(interest, ',.2f'), \
17               sep='')
18
19     # Call the main function.
20     main()

```

Program Output

The simple interest will be \$1000.00.

Notice in line 7 that the order of the keyword arguments does not match the order of the parameters in the function header in line 13. Because a keyword argument specifies which parameter the argument should be passed into, its position in the function call does not matter.

Program 5-12 shows another example. This is a variation of the `string_args` program shown in Program 5-9. This version uses keyword arguments to call the `reverse_name` function.

Program 5-12 (keyword_string_args.py)

```

1  # This program demonstrates passing two strings as
2  # keyword arguments to a function.
3
4  def main():
5      first_name = input('Enter your first name: ')
6      last_name = input('Enter your last name: ')
7      print('Your name reversed is')
8      reverse_name(last=last_name, first=first_name)
9
10 def reverse_name(first, last):
11     print(last, first)
12
13 # Call the main function.
14 main()

```

Program Output (with input shown in bold)

```

Enter your first name: Matt 
Enter your last name: Hoyle 
Your name reversed is
Hoyle Matt

```

Mixing Keyword Arguments with Positional Arguments

It is possible to mix positional arguments and keyword arguments in a function call, but the positional arguments must appear first, followed by the keyword arguments. Otherwise an error will occur. Here is an example of how we might call the `show_interest` function of Program 5-10 using both positional and keyword arguments:

```
show_interest(10000.0, rate=0.01, periods=10)
```

In this statement, the first argument, `10000.0`, is passed by its position to the `principal` parameter. The second and third arguments are passed as keyword arguments. The following function call will cause an error, however, because a non-keyword argument follows a keyword argument:

```
# This will cause an ERROR!
show_interest(1000.0, rate=0.01, 10)
```



Checkpoint

- 5.13 What are the pieces of data that are passed into a function called?
- 5.14 What are the variables that receive pieces of data in a function called?
- 5.15 What is a parameter variable's scope?
- 5.16 When a parameter is changed, does this affect the argument that was passed into the parameter?
- 5.17 The following statements call a function named `show_data`. Which of the statements passes arguments by position, and which passes keyword arguments?
 - a. `show_data(name='Kathryn', age=25)`
 - b. `show_data('Kathryn', 25)`

5.6

Global Variables and Global Constants

CONCEPT: A global variable is accessible to all the functions in a program file.

You've learned that when a variable is created by an assignment statement inside a function, the variable is local to that function. Consequently, it can be accessed only by statements inside the function that created it. When a variable is created by an assignment statement that is written outside all the functions in a program file, the variable is *global*. A global variable can be accessed by any statement in the program file, including the statements in any function. For example, look at Program 5-13.

Program 5-13 (global1.py)

```
1 # Create a global variable.
2 my_value = 10
3
4 # The show_value function prints
5 # the value of the global variable.
```

(program continues)

Program 5-13 (continued)

```

6  def show_value():
7      print(my_value)
8
9  # Call the show_value function.
10 show_value()

```

Program Output

```

10

```

The assignment statement in line 2 creates a variable named `my_value`. Because this statement is outside any function, it is global. When the `show_value` function executes, the statement in line 7 prints the value referenced by `my_value`.

An additional step is required if you want a statement in a function to assign a value to a global variable. In the function you must declare the global variable, as shown in Program 5-14.

Program 5-14 (global2.py)

```

1  # Create a global variable.
2  number = 0
3
4  def main():
5      global number
6      number = int(input('Enter a number: '))
7      show_number()
8
9  def show_number():
10     print('The number you entered is', number)
11
12 # Call the main function.
13 main()

```

Program Output

```

Enter a number: 55 
The number you entered is 55

```

The assignment statement in line 2 creates a global variable named `number`. Notice that inside the `main` function, line 5 uses the `global` key word to declare the `number` variable. This statement tells the interpreter that the `main` function intends to assign a value to the global `number` variable. That's just what happens in line 6. The value entered by the user is assigned to `number`.

Most programmers agree that you should restrict the use of global variables, or not use them at all. The reasons are as follows:

- Global variables make debugging difficult. Any statement in a program file can change the value of a global variable. If you find that the wrong value is being stored in a

global variable, you have to track down every statement that accesses it to determine where the bad value is coming from. In a program with thousands of lines of code, this can be difficult.

- Functions that use global variables are usually dependent on those variables. If you want to use such a function in a different program, most likely you will have to redesign it so it does not rely on the global variable.
- Global variables make a program hard to understand. A global variable can be modified by any statement in the program. If you are to understand any part of the program that uses a global variable, you have to be aware of all the other parts of the program that access the global variable.

In most cases, you should create variables locally and pass them as arguments to the functions that need to access them.

Global Constants

Although you should try to avoid the use of global variables, it is permissible to use global constants in a program. A *global constant* is a global name that references a value that cannot be changed. Because a global constant's value cannot be changed during the program's execution, you do not have to worry about many of the potential hazards that are associated with the use of global variables.

Although the Python language does not allow you to create true global constants, you can simulate them with global variables. If you do not declare a global variable with the `global` key word inside a function, then you cannot change the variable's assignment inside that function. The following *In the Spotlight* section demonstrates how global variables can be used in Python to simulate global constants.

In the Spotlight: Using Global Constants



Marilyn works for Integrated Systems, Inc., a software company that has a reputation for providing excellent fringe benefits. One of their benefits is a quarterly bonus that is paid to all employees. Another benefit is a retirement plan for each employee. The company contributes 5 percent of each employee's gross pay and bonuses to their retirement plans. Marilyn wants to write a program that will calculate the company's contribution to an employee's retirement account for a year. She wants the program to show the amount of contribution for the employee's gross pay and for the bonuses separately. Here is an algorithm for the program:

Get the employee's annual gross pay.
Get the amount of bonuses paid to the employee.
Calculate and display the contribution for the gross pay.
Calculate and display the contribution for the bonuses.

The code for the program is shown in Program 5-15.

Program 5-15 (retirement.py)

```

1  # The following is used as a global constant
2  # the contribution rate.
3  CONTRIBUTION_RATE = 0.05
4
5  def main():
6      gross_pay = float(input('Enter the gross pay: '))
7      bonus = float(input('Enter the amount of bonuses: '))
8      show_pay_contrib(gross_pay)
9      show_bonus_contrib(bonus)
10
11 # The show_pay_contrib function accepts the gross
12 # pay as an argument and displays the retirement
13 # contribution for that amount of pay.
14 def show_pay_contrib(gross):
15     contrib = gross * CONTRIBUTION_RATE
16     print('Contribution for gross pay: $', \
17           format(contrib, ',.2f'), \
18           sep='')
19
20 # The show_bonus_contrib function accepts the
21 # bonus amount as an argument and displays the
22 # retirement contribution for that amount of pay.
23 def show_bonus_contrib(bonus):
24     contrib = bonus * CONTRIBUTION_RATE
25     print('Contribution for bonuses: $', \
26           format(contrib, ',.2f'), \
27           sep='')
28
29 # Call the main function.
30 main()

```

Program Output (with input shown in bold)

```

Enter the gross pay: 80000.00 
Enter the amount of bonuses: 20000.00 
Contribution for gross pay: $4000.00
Contribution for bonuses: $1000.00

```

First, notice the global declaration in line 3:

```
CONTRIBUTION_RATE = 0.05
```

CONTRIBUTION_RATE will be used as a global constant to represent the percentage of an employee's pay that the company will contribute to a retirement account. It is a common practice to write a constant's name in all uppercase letters. This serves as a reminder that the value referenced by the name is not to be changed in the program.

The CONTRIBUTION_RATE constant is used in the calculation in line 15 (in the show_pay_contrib function) and again in line 24 (in the show_bonus_contrib function).

Marilyn decided to use this global constant to represent the 5 percent contribution rate for two reasons:

- It makes the program easier to read. When you look at the calculations in lines 15 and 24 it is apparent what is happening.
- Occasionally the contribution rate changes. When this happens, it will be easy to update the program by changing the assignment statement in line 3.



Checkpoint

- 5.18 What is the scope of a global variable?
- 5.19 Give one good reason that you should not use global variables in a program.
- 5.20 What is a global constant? Is it permissible to use global constants in a program?

5.7

Introduction to Value-Returning Functions: Generating Random Numbers

CONCEPT: A value-returning function is a function that returns a value back to the part of the program that called it. Python, as well as most other programming languages, provides a library of prewritten functions that perform commonly needed tasks. These libraries typically contain a function that generates random numbers.

In the first part of this chapter you learned about void functions. A void function is a group of statements that exist within a program for the purpose of performing a specific task. When you need the function to perform its task, you call the function. This causes the statements inside the function to execute. When the function is finished, control of the program returns to the statement appearing immediately after the function call.

A *value-returning function* is a special type of function. It is like a void function in the following ways.

- It is a group of statements that perform a specific task.
- When you want to execute the function, you call it.

When a value-returning function finishes, however, it returns a value back to the part of the program that called it. The value that is returned from a function can be used like any other value: it can be assigned to a variable, displayed on the screen, used in a mathematical expression (if it is a number), and so on.

Standard Library Functions and the `import` Statement

Python, as well as most other programming languages, comes with a *standard library* of functions that have already been written for you. These functions, known as *library functions*,

make a programmer's job easier because they perform many of the tasks that programmers commonly need to perform. In fact, you have already used several of Python's library functions. Some of the functions that you have used are `print`, `input`, and `range`. Python has many other library functions. Although we won't cover them all in this book, we will discuss library functions that perform fundamental operations.

Some of Python's library functions are built into the Python interpreter. If you want to use one of these built-in functions in a program, you simply call the function. This is the case with the `print`, `input`, `range`, and other functions that you have already learned about. Many of the functions in the standard library, however, are stored in files that are known as *modules*. These modules, which are copied to your computer when you install Python, help organize the standard library functions. For example, functions for performing math operations are stored together in a module, functions for working with files are stored together in another module, and so on.

In order to call a function that is stored in a module, you have to write an `import` statement at the top of your program. An `import` statement tells the interpreter the name of the module that contains the function. For example, one of the Python standard modules is named `math`. The `math` module contains various mathematical functions that work with floating-point numbers. If you want to use any of the `math` module's functions in a program, you should write the following `import` statement at the top of the program:

```
import math
```

This statement causes the interpreter to load the contents of the `math` module into memory and makes all the functions in the `math` module available to the program.

Because you do not see the internal workings of library functions, many programmers think of them as *black boxes*. The term “black box” is used to describe any mechanism that accepts input, performs some operation (that cannot be seen) using the input, and produces output. Figure 5-19 illustrates this idea.

Figure 5-19 A library function viewed as a black box



We will first demonstrate how value-returning functions work by looking at standard library functions that generate random numbers and some interesting programs that can be written with them. Then you will learn to write your own value-returning functions and how to create your own modules. The last section in this chapter comes back to the topic of library functions and looks at several other useful functions in the Python standard library.

Generating Random Numbers

Random numbers are useful for lots of different programming tasks. The following are just a few examples.

- Random numbers are commonly used in games. For example, computer games that let the player roll dice use random numbers to represent the values of the dice. Programs

that show cards being drawn from a shuffled deck use random numbers to represent the face values of the cards.

- Random numbers are useful in simulation programs. In some simulations, the computer must randomly decide how a person, animal, insect, or other living being will behave. Formulas can be constructed in which a random number is used to determine various actions and events that take place in the program.
- Random numbers are useful in statistical programs that must randomly select data for analysis.
- Random numbers are commonly used in computer security to encrypt sensitive data.

Python provides several library functions for working with random numbers. These functions are stored in a module named `random` in the standard library. To use any of these functions you first need to write this `import` statement at the top of your program:

```
import random
```

This statement causes the interpreter to load the contents of the `random` module into memory. This makes all of the functions in the `random` module available to your program.¹

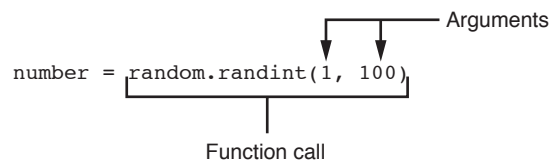
The first random-number generating function that we will discuss is named `randint`. Because the `randint` function is in the `random` module, we will need to use *dot notation* to refer to it in our program. In dot notation, the function's name is `random.randint`. On the left side of the dot (period) is the name of the module, and on the right side of the dot is the name of the function.

The following statement shows an example of how you might call the `randint` function.

```
number = random.randint(1, 100)
```

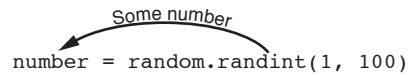
The part of the statement that reads `random.randint(1, 100)` is a call to the `randint` function. Notice that two arguments appear inside the parentheses: 1 and 100. These arguments tell the function to give an integer random number in the range of 1 through 100. (The values 1 and 100 are included in the range.) Figure 5-20 illustrates this part of the statement.

Figure 5-20 A statement that calls the `random` function



Notice that the call to the `randint` function appears on the right side of an `=` operator. When the function is called, it will generate a random number in the range of 1 through 100 and then *return* that number. The number that is returned will be assigned to the `number` variable, as shown in Figure 5-21.

¹There are several ways to write an `import` statement in Python, and each variation works a little differently. Many Python programmers agree that the preferred way to import a module is the way shown in this book.

Figure 5-21 The `random` function returns a value


```
number = random.randint(1, 100)
```

A random number in the range of
1 through 100 will be assigned to
the `number` variable.

Program 5-16 shows a complete program that uses the `randint` function. The statement in line 2 generates a random number in the range of 1 through 10 and assigns it to the `number` variable. (The program output shows that the number 7 was generated, but this value is arbitrary. If this were an actual program, it could display any number from 1 to 10.)

Program 5-16 (random_numbers.py)

```
1  # This program displays a random number
2  # in the range of 1 through 10.
3  import random
4
5  def main():
6      # Get a random number.
7      number = random.randint(1, 10)
8      # Display the number.
9      print('The number is', number)
10
11 # Call the main function.
12 main()
```

Program Output

The number is 7

Program 5-17 shows another example. This program uses a `for` loop that iterates five times. Inside the loop, the statement in line 8 calls the `randint` function to generate a random number in the range of 1 through 100.

Program 5-17 (random_numbers2.py)

```
1  # This program displays five random
2  # numbers in the range of 1 through 100.
3  import random
4
5  def main():
6      for count in range(5):
7          # Get a random number.
8          number = random.randint(1, 100)
```

```

9           # Display the number.
10          print(number)
11
12  # Call the main function.
13  main()

```

Program Output

```

89
7
16
41
12

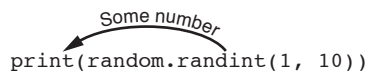
```

Both Program 5-16 and 5-17 call the `randint` function and assign its return value to the `number` variable. If you just want to display a random number, it is not necessary to assign the random number to a variable. You can send the `random` function's return value directly to the `print` function, as shown here:

```
print(random.randint(1, 10))
```

When this statement executes, the `randint` function is called. The function generates a random number in the range of 1 through 10. That value is returned and then sent to the `print` function. As a result, a random number in the range of 1 through 10 will be displayed. Figure 5-22 illustrates this.

Figure 5-22 Displaying a random number



```
print(random.randint(1, 10))
```

A random number in the range of
1 through 10 will be displayed.

Program 5-18 shows how you could simplify Program 5-17. This program also displays five random numbers, but this program does not use a variable to hold those numbers. The `randint` function's return value is sent directly to the `print` function in line 7.

Program 5-18 (random_numbers3.py)

```

1  # This program displays five random
2  # numbers in the range of 1 through 100.
3  import random
4
5  def main():
6      for count in range(5):
7          print(random.randint(1, 100))
8
9  # Call the main function.
10 main()

```

(program continues)

Program 5-18 (continued)**Program Output**

```
89
7
16
41
12
```

Experimenting with Random Numbers in Interactive Mode

To get a feel for the way the `randint` function works with different arguments, you might want to experiment with it in interactive mode. To demonstrate, look at the following interactive session. (We have added line numbers for easier reference.)

```
1 >>> import random 
2 >>> random.randint(1, 10) 
3 5
4 >>> random.randint(1, 100) 
5 98
6 >>> random.randint(100, 200) 
7 181
8 >>>
```

Let's take a closer look at each line in the interactive session:

- The statement in line 1 imports the `random` module. (You have to write the appropriate `import` statements in interactive mode, too.)
- The statement in line 2 calls the `randint` function, passing 1 and 10 as arguments. As a result the function returns a random number in the range of 1 through 10. The number that is returned from the function is displayed in line 3.
- The statement in line 4 calls the `randint` function, passing 1 and 100 as arguments. As a result the function returns a random number in the range of 1 through 100. The number that is returned from the function is displayed in line 5.
- The statement in line 6 calls the `randint` function, passing 100 and 200 as arguments. As a result the function returns a random number in the range of 100 through 200. The number that is returned from the function is displayed in line 7.

In the Spotlight:

Using Random Numbers

Dr. Kimura teaches an introductory statistics class and has asked you to write a program that he can use in class to simulate the rolling of dice. The program should randomly generate two numbers in the range of 1 through 6 and display them. In your interview with Dr. Kimura, you learn that he would like to use the program to simulate several rolls of the dice, one after the other. Here is the pseudocode for the program:

While the user wants to roll the dice:

Display a random number in the range of 1 through 6

Display another random number in the range of 1 through 6

Ask the user if he or she wants to roll the dice again



You will write a `while` loop that simulates one roll of the dice and then asks the user if another roll should be performed. As long as the user answers “y” for yes, the loop will repeat. Program 5-19 shows the program.

Program 5-19 (dice.py)

```
1  # This program the rolling of dice.
2  import random
3
4  # Constants for the minimum and maximum random numbers
5  MIN = 1
6  MAX = 6
7
8  def main():
9      # Create a variable to control the loop.
10     again = 'y'
11
12     # Simulate rolling the dice.
13     while again == 'y' or again == 'Y':
14         print('Rolling the dice ...')
15         print('Their values are:')
16         print(random.randint(MIN, MAX))
17         print(random.randint(MIN, MAX))
18
19         # Do another roll of the dice?
20         again = input('Roll them again? (y = yes): ')
21
22     # Call the main function.
23     main()
```

Program Output (with input shown in bold)

```
Rolling the dice ...
Their values are:
3
1
Roll them again? (y = yes): y 
Rolling the dice ...
Their values are:
1
1
Roll them again? (y = yes): y 
Rolling the dice ...
Their values are:
5
6
Roll them again? (y = yes): y 
```

The `randint` function returns an integer value, so you can write a call to the function anywhere that you can write an integer value. You have already seen examples where the function's return value is assigned to a variable and where the function's return value is sent to the `print` function. To further illustrate the point, here is a statement that uses the `randint` function in a math expression:

```
x = random.randint (1, 10) * 2
```

In this statement, a random number in the range of 1 through 10 is generated and then multiplied by 2. The result is a random even integer from 2 to 20 assigned to the `x` variable. You can also test the return value of the function with an `if` statement, as demonstrated in the following In the Spotlight section.



In the Spotlight:

Using Random Numbers to Represent Other Values

Dr. Kimura was so happy with the dice rolling simulator that you wrote for him, he has asked you to write one more program. He would like a program that he can use to simulate ten coin tosses, one after the other. Each time the program simulates a coin toss, it should randomly display either “Heads” or “Tails”.

You decide that you can simulate the tossing of a coin by randomly generating a number in the range of 1 through 2. You will write an `if` statement that displays “Heads” if the random number is 1, or “Tails” otherwise. Here is the pseudocode:

Repeat 10 times:

If a random number in the range of 1 through 2 equals 1 then:

Display ‘Heads’

Else:

Display ‘Tails’

Because the program should simulate 10 tosses of a coin you decide to use a `for` loop. The program is shown in Program 5-20.

Program 5-20 (coin_toss.py)

```
1 # This program simulates 10 tosses of a coin.
2 import random
3
4 # Constants
5 HEADS = 1
6 TAILS = 2
7 TOSSES = 10
8
```

```
9 def main():
10     for toss in range(TOSSES):
11         # Simulate the coin toss.
12         if random.randint(HEADS, TAILS) == HEADS:
13             print('Heads')
14         else:
15             print('Tails')
16
17 # Call the main function.
18 main()
```

Program Output

```
Tails
Tails
Heads
Tails
Heads
Heads
Heads
Heads
Tails
Heads
Tails
```

The randrange, random, and uniform Functions

The standard library's `random` module contains numerous functions for working with random numbers. In addition to the `randint` function, you might find the `randrange`, `random`, and `uniform` functions useful. (To use any of these functions you need to write `import random` at the top of your program.)

If you remember how to use the `range` function (which we discussed in Chapter 4) then you will immediately be comfortable with the `randrange` function. The `randrange` function takes the same arguments as the `range` function. The difference is that the `randrange` function does not return a list of values. Instead, it returns a randomly selected value from a sequence of values. For example, the following statement assigns a random number in the range of 0 through 9 to the `number` variable:

```
number = random.randrange(10)
```

The argument, in this case 10, specifies the ending limit of the sequence of values. The function will return a randomly selected number from the sequence of values 0 up to, but not including, the ending limit. The following statement specifies both a starting value and an ending limit for the sequence:

```
number = random.randrange(5, 10)
```

When this statement executes, a random number in the range of 5 through 9 will be assigned to `number`. The following statement specifies a starting value, an ending limit, and a step value:

```
number = random.randrange(0, 101, 10)
```

In this statement the `randrange` function returns a randomly selected value from the following sequence of numbers:

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

Both the `randint` and the `randrange` functions return an integer number. The `random` function, however, returns a random floating-point number. You do not pass any arguments to the `random` function. When you call it, it returns a random floating point number in the range of 0.0 up to 1.0 (but not including 1.0). Here is an example:

```
number = random.random()
```

The `uniform` function also returns a random floating-point number, but allows you to specify the range of values to select from. Here is an example:

```
number = random.uniform(1.0, 10.0)
```

In this statement the `uniform` function returns a random floating-point number in the range of 1.0 through 10.0 and assigns it to the `number` variable.

Random Number Seeds

The numbers that are generated by the functions in the `random` module are not truly random. Although we commonly refer to them as random numbers, they are actually *pseudorandom numbers* that are calculated by a formula. The formula that generates random numbers has to be initialized with a value known as a *seed value*. The seed value is used in the calculation that returns the next random number in the series. When the `random` module is imported, it retrieves the system time from the computer's internal clock and uses that as the seed value. The system time is an integer that represents the current date and time, down to a hundredth of a second.

If the same seed value were always used, the random number functions would always generate the same series of pseudorandom numbers. Because the system time changes every hundredth of a second, it is a fairly safe bet that each time you import the `random` module, a different sequence of random numbers will be generated. However, there may be some applications in which you want to always generate the same sequence of random numbers. If that is the case, you can call the `random.seed` function to specify a seed value. Here is an example:

```
random.seed(10)
```

In this example, the value 10 is specified as the seed value. If a program calls the `random.seed` function, passing the same value as an argument each time it runs, it will always produce the same sequence of pseudorandom numbers. To demonstrate, look at the following interactive sessions. (We have added line numbers for easier reference.)

```
1 >>> import random Enter
2 >>> random.seed(10) Enter
3 >>> random.randint(1, 100) Enter
4 58
5 >>> random.randint(1, 100) Enter
6 43
```



```
7 >>> random.randint(1, 100)   
8 58  
9 >>> random.randint(1, 100)   
10 21  
11 >>>
```

In line 1 we import the `random` module. In line 2 we call the `random.seed` function, passing 10 as the seed value. In lines 3, 5, 7, and 9 we call `random.randint` function to get a pseudorandom number in the range of 1 through 100. As you can see, the function gave us the numbers 58, 43, 58, and 21. If we start a new interactive session and repeat these statements, we get the same sequence of pseudorandom numbers, as shown here:

```
1 >>> import random   
2 >>> random.seed(10)   
3 >>> random.randint(1, 100)   
4 58  
5 >>> random.randint(1, 100)   
6 43  
7 >>> random.randint(1, 100)   
8 58  
9 >>> random.randint(1, 100)   
10 21  
11 >>>
```



Checkpoint

- 5.21 How does a value-returning function differ from the void functions?
- 5.22 What is a library function?
- 5.23 Why are library functions like “black boxes”?
- 5.24 What does the following statement do?
`x = random.randint(1, 100)`
- 5.25 What does the following statement do?
`print(random.randint(1, 20))`
- 5.26 What does the following statement do?
`print(random.randrange(10, 20))`
- 5.27 What does the following statement do?
`print(random.random())`
- 5.28 What does the following statement do?
`print(random.uniform(0.1, 0.5))`
- 5.29 When the `random` module is imported, what does it use as a seed value for random number generation?
- 5.30 What happens if the same seed value is always used for generating random numbers?

5.8

Writing Your Own Value-Returning Functions

CONCEPT: A value-returning function has a **return** statement that returns a value back to the part of the program that called it.



VideoNote
Writing a Value-
Returning Function

You write a value-returning function in the same way that you write a void function, with one exception: a value-returning function must have a **return** statement. Here is the general format of a value-returning function definition in Python:

```
def function_name():
    statement
    statement
    etc.
    return expression
```

One of the statements in the function must be a **return** statement, which takes the following form:

```
return expression
```

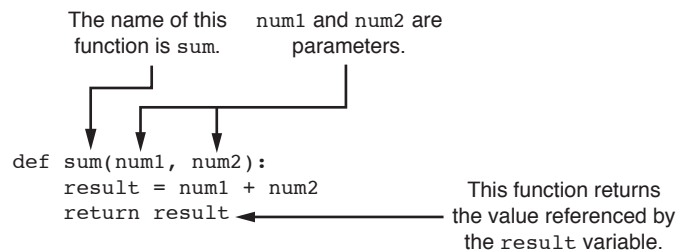
The value of the *expression* that follows the key word **return** will be sent back to the part of the program that called the function. This can be any value, variable, or expression that has a value (such as a math expression).

Here is a simple example of a value-returning function:

```
def sum(num1, num2):
    result = num1 + num2
    return result
```

Figure 5-23 illustrates various parts of the function.

Figure 5-23 Parts of the function



The purpose of this function is to accept two integer values as arguments and return their sum. Let's take a closer look at how it works. The first statement in the function's block assigns the value of `num1 + num2` to the `result` variable. Next, the **return** statement executes, which causes the function to end execution and sends the value referenced by the `result` variable back to the part of the program that called the function. Program 5-21 demonstrates the function.

Program 5-21 (total_ages.py)

```

1  # This program uses the return value of a function.
2
3  def main():
4      # Get the user's age.
5      first_age = int(input('Enter your age: '))
6
7      # Get the user's best friend's age.
8      second_age = int(input("Enter your best friend's age: "))
9
10     # Get the sum of both ages.
11     total = sum(first_age, second_age)
12
13     # Display the total age.
14     print('Together you are', total, 'years old.')
15
16 # The sum function accepts two numeric arguments and
17 # returns the sum of those arguments.
18 def sum(num1, num2):
19     result = num1 + num2
20     return result
21
22 # Call the main function.
23 main()

```

Program Output (with input shown in bold)

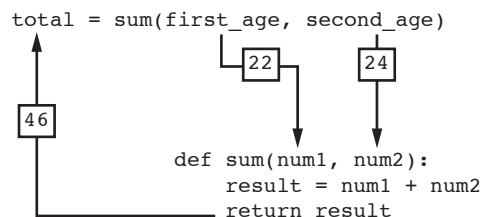
Enter your age: **22**

Enter your best friend's age: **24**

Together you are 46 years old.

In the main function, the program gets two values from the user and stores them in the `first_age` and `second_age` variables. The statement in line 11 calls the `sum` function, passing `first_age` and `second_age` as arguments. The value that is returned from the `sum` function is assigned to the `total` variable. In this case, the function will return 46. Figure 5-24 shows how the arguments are passed into the function, and how a value is returned back from the function.

Figure 5-24 Arguments are passed to the `sum` function and a value is returned



Making the Most of the `return` Statement

Look again at the `sum` function presented in Program 5-21:

```
def sum(num1, num2):
    result = num 1 + num 2
    return result
```

Notice that two things happen inside this function: (1) the value of the expression `num1 + num2` is assigned to the `result` variable, and (2) the value of the `result` variable is returned. Although this function does what it sets out to do, it can be simplified. Because the `return` statement can return the value of an expression, you can eliminate the `result` variable and rewrite the function as:

```
def sum(num1, num2):
    return num 1 + num 2
```

This version of the function does not store the value of `num1 + num2` in a variable. Instead, it takes advantage of the fact that the `return` statement can return the value of an expression. This version of the function does the same thing as the previous version, but in only one step.

How to Use Value-Returning Functions

Value-returning functions provide many of the same benefits as void functions: they simplify code, reduce duplication, enhance your ability to test code, increase the speed of development, and ease the facilitation of teamwork.

Because value-returning functions return a value, they can be useful in specific situations. For example, you can use a value-returning function to prompt the user for input, and then it can return the value entered by the user. Suppose you've been asked to design a program that calculates the sale price of an item in a retail business. To do that, the program would need to get the item's regular price from the user. Here is a function you could define for that purpose:

```
def get_regular_price():
    price = float(input("Enter the item's regular price: "))
    return price
```

Then, elsewhere in the program, you could call that function, as shown here:

```
# Get the item's regular price.
reg_price = get_regular_price()
```

When this statement executes, the `get_regular_price` function is called, which gets a value from the user and returns it. That value is then assigned to the `reg_price` variable.

You can also use functions to simplify complex mathematical expressions. For example, calculating the sale price of an item seems like it would be a simple task: you calculate the discount and subtract it from the regular price. In a program, however, a statement that performs this calculation is not that straightforward, as shown in the following example. (Assume `DISCOUNT_PERCENTAGE` is a global constant that is defined in the program, and it specifies the percentage of the discount.)

```
sale_price = reg_price - (reg_price * DISCOUNT_PERCENTAGE)
```

At a glance, this statement isn't easy to understand because it performs so many steps: it calculates the discount amount, subtracts that value from `reg_price`, and assigns the result to `sale_price`. You could simplify the statement by breaking out part of the math expression and placing it in a function. Here is a function named `discount` that accepts an item's price as an argument and returns the amount of the discount:

```
def discount(price):  
    return price * DISCOUNT_PERCENTAGE
```

You could then call the function in your calculation:

```
sale_price = reg_price - discount(reg_price)
```

This statement is easier to read than the one previously shown, and it is clearer that the discount is being subtracted from the regular price. Program 5-22 shows the complete sale price calculating program using the functions just described.

Program 5-22 (sale_price.py)

```
1  # This program calculates a retail item's  
2  # sale price.  
3  
4  # DISCOUNT_PERCENTAGE is used as a global  
5  # constant for the discount percentage.  
6  DISCOUNT_PERCENTAGE = 0.20  
7  
8  # The main function.  
9  def main():  
10     # Get the item's regular price.  
11     reg_price = get_regular_price()  
12  
13     # Calculate the sale price.  
14     sale_price = reg_price - discount(reg_price)  
15  
16     # Display the sale price.  
17     print('The sale price is $', format(sale_price, ',.2f'), sep='')  
18  
19  # The get_regular_price function prompts the  
20  # user to enter an item's regular price and it  
21  # returns that value.  
22  def get_regular_price():  
23     price = float(input("Enter the item's regular price: "))  
24     return price  
25  
26  # The discount function accepts an item's price  
27  # as an argument and returns the amount of the  
28  # discount, specified by DISCOUNT_PERCENTAGE.  
29  def discount(price):
```

(program continues)

Program 5-22 (continued)

```
30     return price * DISCOUNT_PERCENTAGE
31
32 # Call the main function.
33 main()
```

Program Output (with input shown in bold)

Enter the item's regular price: **100.00**
The sale price is \$80.00

Using IPO Charts

An IPO chart is a simple but effective tool that programmers sometimes use for designing and documenting functions. IPO stands for *input*, *processing*, and *output*, and an *IPO chart* describes the input, processing, and output of a function. These items are usually laid out in columns: the input column shows a description of the data that is passed to the function as arguments, the processing column shows a description of the process that the function performs, and the output column describes the data that is returned from the function. For example, Figure 5-25 shows IPO charts for the `get_regular_price` and `discount` functions that you saw in Program 5-22.

Figure 5-25 IPO charts for the `getRegularPrice` and `discount` functions

IPO Chart for the <code>get_regular_price</code> Function		
Input	Processing	Output
None	Prompts the user to enter an item's regular price	The item's regular price

IPO Chart for the <code>discount</code> Function		
Input	Processing	Output
An item's regular price	Calculates an item's discount by multiplying the regular price by the global constant <code>DISCOUNT_PERCENTAGE</code>	The item's discount

Notice that the IPO charts provide only brief descriptions of a function's input, processing, and output, but do not show the specific steps taken in a function. In many cases, however, IPO charts include sufficient information so that they can be used instead of a flowchart. The decision of whether to use an IPO chart, a flowchart, or both is often left to the programmer's personal preference.

In the Spotlight:

Modularizing with Functions



Hal owns a business named Make Your Own Music, which sells guitars, drums, banjos, synthesizers, and many other musical instruments. Hal's sales staff works strictly on commission. At the end of the month, each salesperson's commission is calculated according to Table 5-1.

Table 5-1 Sales commission rates

Sales This Month	Commission Rate
Less than \$10,000	10%
\$10,000–14,999	12%
\$15,000–17,999	14%
\$18,000–21,999	16%
\$22,000 or more	18%

For example, a salesperson with \$16,000 in monthly sales will earn a 14 percent commission (\$2,240). Another salesperson with \$18,000 in monthly sales will earn a 16 percent commission (\$2,880). A person with \$30,000 in sales will earn an 18 percent commission (\$5,400).

Because the staff gets paid once per month, Hal allows each employee to take up to \$2,000 per month in advance. When sales commissions are calculated, the amount of each employee's advanced pay is subtracted from the commission. If any salesperson's commissions are less than the amount of their advance, they must reimburse Hal for the difference. To calculate a salesperson's monthly pay, Hal uses the following formula:

$$\text{pay} = \text{sales} \times \text{commission rate} - \text{advanced pay}$$

Hal has asked you to write a program that makes this calculation for him. The following general algorithm outlines the steps the program must take.

1. *Get the salesperson's monthly sales.*
2. *Get the amount of advanced pay.*
3. *Use the amount of monthly sales to determine the commission rate.*
4. *Calculate the salesperson's pay using the formula previously shown. If the amount is negative, indicate that the salesperson must reimburse the company.*

Program 5-23 shows the code, which is written using several functions. Rather than presenting the entire program at once, let's first examine the main function and then each function separately. Here is the main function:

Program 5-23 (commission_rate.py) main function

```

1  # This program calculates a salesperson's pay
2  # at Make Your Own Music.
3  def main():
4      # Get the amount of sales.
5      sales = get_sales()
6
7      # Get the amount of advanced pay.
8      advanced_pay = get_advanced_pay()
9
10     # Determine the commission rate.
11     comm_rate = determine_comm_rate(sales)
12
13     # Calculate the pay.
14     pay = sales * comm_rate - advanced_pay
15
16     # Display the amount of pay.
17     print('The pay is $', format(pay, ',.2f'), sep='')
18
19     # Determine whether the pay is negative.
20     if pay < 0:
21         print('The Salesperson must reimburse')
22         print('the company.')
23

```

Line 5 calls the `get_sales` function, which gets the amount of sales from the user and returns that value. The value that is returned from the function is assigned to the `sales` variable. Line 8 calls the `get_advanced_pay` function, which gets the amount of advanced pay from the user and returns that value. The value that is returned from the function is assigned to the `advanced_pay` variable.

Line 11 calls the `determine_comm_rate` function, passing `sales` as an argument. This function returns the rate of commission for the amount of sales. That value is assigned to the `comm_rate` variable. Line 14 calculates the amount of pay, and then line 17 displays that amount. The `if` statement in lines 20 through 22 determines whether the pay is negative, and if so, displays a message indicating that the salesperson must reimburse the company. The `get_sales` function definition is next.

Program 5-23 (commission_rate.py) `get_sales` function

```

24  # The get_sales function gets a salesperson's
25  # monthly sales from the user and returns that value.
26  def get_sales():
27      # Get the amount of monthly sales.

```



```
28     monthly_sales = float(input('Enter the monthly sales: '))
29
30     # Return the amount entered.
31     return monthly_sales
32
```

The purpose of the `get_sales` function is to prompt the user to enter the amount of sales for a salesperson and return that amount. Line 28 prompts the user to enter the sales and stores the user's input in the `monthly_sales` variable. Line 31 returns the amount in the `monthly_sales` variable. Next is the definition of the `get_advanced_pay` function.

Program 5-23 (commission_rate.py) `get_advanced_pay` function

```
33 # The get_advanced_pay function gets the amount of
34 # advanced pay given to the salesperson and returns
35 # that amount.
36 def get_advanced_pay():
37     # Get the amount of advanced pay.
38     print('Enter the amount of advanced pay, or')
39     print('enter 0 if no advanced pay was given.')
40     advanced = float(input('Advanced pay: '))
41
42     # Return the amount entered.
43     return advanced
44
```

The purpose of the `get_advanced_pay` function is to prompt the user to enter the amount of advanced pay for a salesperson and return that amount. Lines 38 and 39 tell the user to enter the amount of advanced pay (or 0 if none was given). Line 40 gets the user's input and stores it in the `advanced` variable. Line 43 returns the amount in the `advanced` variable. Defining the `determine_comm_rate` function comes next.

Program 5-23 (commission_rate.py) `determine_comm_rate` function

```
45 # The determine_comm_rate function accepts the
46 # amount of sales as an argument and returns the
47 # applicable commission rate.
48 def determine_comm_rate(sales):
49     # Determine the commission rate.
50     if sales < 10000.00:
51         rate = 0.10
52     elif sales >= 10000 and sales <= 14999.99:
53         rate = 0.12
54     elif sales >= 15000 and sales <= 17999.99:
55         rate = 0.14
56     elif sales >= 18000 and sales <= 21999.99:
57         rate = 0.16
```

(program continues)

Program 5-23 (continued)

```

58     else:
59         rate = 0.18
60
61     # Return the commission rate.
62     return rate
63

```

The `determine_comm_rate` function accepts the amount of sales as an argument, and it returns the applicable commission rate for that amount of sales. The `if-elif-else` statement in lines 50 through 59 tests the `sales` parameter and assigns the correct value to the local `rate` variable. Line 62 returns the value in the local `rate` variable.

Program Output (with input shown in bold)

```

Enter the monthly sales: 14650.00 
Enter the amount of advanced pay, or
enter 0 if no advanced pay was given.
Advanced pay: 1000.00 
The pay is $758.00

```

Program Output (with input shown in bold)

```

Enter the monthly sales: 9000.00 
Enter the amount of advanced pay, or
enter 0 if no advanced pay was given.
Advanced pay: 0 
The pay is $900.00

```

Program Output (with input shown in bold)

```

Enter the monthly sales: 12000.00 
Enter the amount of advanced pay, or
enter 0 if no advanced pay was given.
Advanced pay: 2000.00 
The pay is $-560.00
The salesperson must reimburse
the company.

```

Returning Strings

So far you've seen examples of functions that return numbers. You can also write functions that return strings. For example, the following function prompts the user to enter his or her name, and then returns the string that the user entered.

```

def get_name():
    # Get the user's name.
    name = input('Enter your name: ')
    # Return the name.
    return name

```

Returning Boolean Values

Python allows you to write *Boolean functions*, which return either `True` or `False`. You can use a Boolean function to test a condition, and then return either `True` or `False` to indicate whether the condition exists. Boolean functions are useful for simplifying complex conditions that are tested in decision and repetition structures.

For example, suppose you are designing a program that will ask the user to enter a number, and then determine whether that number is even or odd. The following code shows how you can make that determination.

```
number = int(input('Enter a number: '))
if (number % 2) == 0:
    print('The number is even.')
else:
    print('The number is odd.')
```

Let's take a closer look at the Boolean expression being tested by this `if-else` statement:

```
(number % 2) == 0
```

This expression uses the `%` operator, which was introduced in Chapter 2. This is called the remainder operator. It divides two numbers and returns the remainder of the division. So this code is saying, “If the remainder of `number` divided by 2 is equal to 0, then display a message indicating the number is even, or else display a message indicating the number is odd.”

Because dividing an even number by 2 will always give a remainder of 0, this logic will work. The code would be easier to understand, however, if you could somehow rewrite it to say, “If the number is even, then display a message indicating it is even, or else display a message indicating it is odd.” As it turns out, this can be done with a Boolean function. In this example, you could write a Boolean function named `is_even` that accepts a number as an argument and returns `True` if the number is even, or `False` otherwise. The following is the code for such a function.

```
def is_even(number):
    # Determine whether number is even. If it is,
    # set status to true. Otherwise, set status
    # to false.
    if (number % 2) == 0:
        status = True
    else:
        status = False
    # Return the value of the status variable.
    return status
```

Then you can rewrite the `if-else` statement so it calls the `is_even` function to determine whether number is even:

```
number = int(input('Enter a number: '))
if is_even(number):
    print('The number is even.')
else:
    print('The number is odd.')
```

Not only is this logic easier to understand, but now you have a function that you can call in the program anytime you need to test a number to determine whether it is even.

Using Boolean Functions in Validation Code

You can also use Boolean functions to simplify complex input validation code. For instance, suppose you are writing a program that prompts the user to enter a product model number and should only accept the values 100, 200, and 300. You could design the input algorithm as follows:

```
# Get the model number.
model = int(input('Enter the model number: '))
# Validate the model number.
while model != 100 and model != 200 and model != 300:
    print('The valid model numbers are 100, 200 and 300.')
    model = int(input('Enter a valid model number: '))
```

The validation loop uses a long compound Boolean expression that will iterate as long as `model` does not equal 100 *and* `model` does not equal 200 *and* `model` does not equal 300. Although this logic will work, you can simplify the validation loop by writing a Boolean function to test the `model` variable and then calling that function in the loop. For example, suppose you pass the `model` variable to a function you write named `is_invalid`. The function returns `True` if `model` is invalid, or `False` otherwise. You could rewrite the validation loop as follows:

```
# Validate the model number.
while is_invalid(model):
    print('The valid model numbers are 100, 200 and 300.')
    model = int(input('Enter a valid model number: '))
```

This makes the loop easier to read. It is evident now that the loop iterates as long as `model` is invalid. The following code shows how you might write the `is_invalid` function. It accepts a model number as an argument, and if the argument is not 100 and the argument is not 200 and the argument is not 300, the function returns `True` to indicate that it is invalid. Otherwise, the function returns `False`.

```
def is_invalid(mod_num):
    if mod_num != 100 and mod_num != 200 and mod_num != 300:
        status = True
    else:
        status = False
    return status
```

Returning Multiple Values

The examples of value-returning functions that we have looked at so far return a single value. In Python, however, you are not limited to returning only one value. You can specify multiple expressions separated by commas after the `return` statement, as shown in this general format:

```
return expression1, expression2, etc.
```

As an example, look at the following definition for a function named `get_name`. The function prompts the user to enter his or her first and last names. These names are stored in two local variables: `first` and `last`. The `return` statement returns both of the variables.

```
def get_name():
    # Get the user's first and last names.
    first = input('Enter your first name: ')
    last = input('Enter your last name: ')

    # Return both names.
    return first, last
```

When you call this function in an assignment statement, you need to use two variables on the left side of the `=` operator. Here is an example:

```
first_name, last_name = get_name()
```

The values listed in the `return` statement are assigned, in the order that they appear, to the variables on the left side of the `=` operator. After this statement executes, the value of the `first` variable will be assigned to `first_name` and the value of the `last` variable will be assigned to `last_name`. Note that the number of variables on the left side of the `=` operator must match the number of values returned by the function. Otherwise an error will occur.



Checkpoint

5.31 What is the purpose of the `return` statement in a function?

5.32 Look at the following function definition:

```
def do_something(number):
    return number * 2
```

- What is the name of the function?
- What does the function do?
- Given the function definition, what will the following statement display?

```
print(do_something(10))
```

5.33 What is a Boolean function?

5.9

The math Module

CONCEPT: The Python standard library's `math` module contains numerous functions that can be used in mathematical calculations.

The `math` module in the Python standard library contains several functions that are useful for performing mathematical operations. Table 5-2 lists many of the functions in the `math` module. These functions typically accept one or more values as arguments, perform a mathematical operation using the arguments, and return the result. (All of the functions listed in Table 5-2 return a `float` value, except the `ceil` and `floor` functions, which return `int` values.) For example, one of the functions is named `sqrt`. The `sqrt` function accepts an argument and returns the square root of the argument. Here is an example of how it is used:

```
result = math.sqrt(16)
```

This statement calls the `sqrt` function, passing 16 as an argument. The function returns the square root of 16, which is then assigned to the `result` variable. Program 5-24 demonstrates the `sqrt` function. Notice the `import math` statement in line 2. You need to write this in any program that uses the `math` module.

Program 5-24 (square_root.py)

```

1  # This program demonstrates the sqrt function.
2  import math
3
4  def main():
5      # Get a number.
6      number = float(input('Enter a number: '))
7
8      # Get the square root of the number.
9      square_root = math.sqrt(number)
10
11     # Display the square root.
12     print('The square root of', number, 'is', square_root)
13
14     # Call the main function.
15     main()

```

Program Output (with input shown in bold)

```

Enter a number: 25 
The square root of 25.0 is 5.0

```

Program 5-25 shows another example that uses the `math` module. This program uses the `hypot` function to calculate the length of a right triangle's hypotenuse.

Program 5-25 (hypotenuse.py)

```

1  # This program calculates the length of a right
2  # triangle's hypotenuse.
3  import math
4
5  def main():
6      # Get the length of the triangle's two sides.
7      a = float(input('Enter the length of side A: '))
8      b = float(input('Enter the length of side B: '))
9
10     # Calculate the length of the hypotenuse.
11     c = math.hypot(a, b)
12
13     # Display the length of the hypotenuse.
14     print('The length of the hypotenuse is', c)
15

```

```

16 # Call the main function.
17 main()

```

Program Output (with input shown in bold)

```

Enter the length of side A: 5.0 
Enter the length of side B: 12.0 
The length of the hypotenuse is 13.0

```

Table 5-2 Many of the functions in the `math` module

math Module Function	Description
<code>acos(x)</code>	Returns the arc cosine of <code>x</code> , in radians.
<code>asin(x)</code>	Returns the arc sine of <code>x</code> , in radians.
<code>atan(x)</code>	Returns the arc tangent of <code>x</code> , in radians.
<code>ceil(x)</code>	Returns the smallest integer that is greater than or equal to <code>x</code> .
<code>cos(x)</code>	Returns the cosine of <code>x</code> in radians.
<code>degrees(x)</code>	Assuming <code>x</code> is an angle in radians, the function returns the angle converted to degrees.
<code>exp(x)</code>	Returns e^x
<code>floor(x)</code>	Returns the largest integer that is less than or equal to <code>x</code> .
<code>hypot(x, y)</code>	Returns the length of a hypotenuse that extends from (0, 0) to (<code>x</code> , <code>y</code>).
<code>log(x)</code>	Returns the natural logarithm of <code>x</code> .
<code>log10(x)</code>	Returns the base-10 logarithm of <code>x</code> .
<code>radians(x)</code>	Assuming <code>x</code> is an angle in degrees, the function returns the angle converted to radians.
<code>sin(x)</code>	Returns the sine of <code>x</code> in radians.
<code>sqrt(x)</code>	Returns the square root of <code>x</code> .
<code>tan(x)</code>	Returns the tangent of <code>x</code> in radians.

The `math.pi` and `math.e` Values

The `math` module also defines two variables, `pi` and `e`, which are assigned mathematical values for π and e . You can use these variables in equations that require their values. For example, the following statement, which calculates the area of a circle, uses `pi`. (Notice that we use dot notation to refer to the variable.)

```
area = math.pi * radius**2
```



Checkpoint

- 5.34 What `import` statement do you need to write in a program that uses the `math` module?
- 5.35 Write a statement that uses a `math` module function to get the square root of 100 and assigns it to a variable.
- 5.36 Write a statement that uses a `math` module function to convert 45 degrees to radians and assigns the value to a variable.

5.10 Storing Functions in Modules

CONCEPT: A module is a file that contains Python code. Large programs are easier to debug and maintain when they are divided into modules.

As your programs become larger and more complex, the need to organize your code becomes greater. You have already learned that a large and complex program should be divided into functions that each performs a specific task. As you write more and more functions in a program, you should consider organizing the functions by storing them in modules.

A module is simply a file that contains Python code. When you break a program into modules, each module should contain functions that perform related tasks. For example, suppose you are writing an accounting system. You would store all of the account receivable functions in their own module, all of the account payable functions in their own module, and all of the payroll functions in their own module. This approach, which is called *modularization*, makes the program easier to understand, test, and maintain.

Modules also make it easier to reuse the same code in more than one program. If you have written a set of functions that are needed in several different programs, you can place those functions in a module. Then, you can import the module in each program that needs to call one of the functions.

Let's look at a simple example. Suppose your instructor has asked you to write a program that calculates the following:

- The area of a circle
- The circumference of a circle
- The area of a rectangle
- The perimeter of a rectangle

There are obviously two categories of calculations required in this program: those related to circles, and those related to rectangles. You could write all of the circle-related functions in one module, and the rectangle-related functions in another module. Program 5-26 shows the `circle` module. The module contains two function definitions: `area` (which returns the area of a circle) and `circumference` (which returns the circumference of a circle).

Program 5-26 (circle.py)

```

1  # The circle module has functions that perform
2  # calculations related to circles.
3  import math
4
5  # The area function accepts a circle's radius as an
6  # argument and returns the area of the circle.
7  def area(radius):
8      return math.pi * radius**2
9

```



```
10 # The circumference function accepts a circle's
11 # radius and returns the circle's circumference.
12 def circumference(radius):
13     return 2 * math.pi * radius
```

Program 5-27 shows the `rectangle` module. The module contains two function definitions: `area` (which returns the area of a rectangle) and `perimeter` (which returns the perimeter of a rectangle.)

Program 5-27 (`rectangle.py`)

```
1 # The rectangle module has functions that perform
2 # calculations related to rectangles.
3
4 # The area function accepts a rectangle's width and
5 # length as arguments and returns the rectangle's area.
6 def area(width, length):
7     return width * length
8
9 # The perimeter function accepts a rectangle's width
10 # and length as arguments and returns the rectangle's
11 # perimeter.
12 def perimeter(width, length):
13     return 2 * (width + length)
```

Notice that both of these files contain function definitions, but they do not contain code that calls the functions. That will be done by the program or programs that import these modules.

Before continuing, we should mention the following things about module names:

- A module's file name should end in `.py`. If the module's file name does not end in `.py` you will not be able to import it into other programs.
- A module's name cannot be the same as a Python key word. An error would occur, for example, if you named a module `for`.

To use these modules in a program, you import them with the `import` statement. Here is an example of how we would import the `circle` module:

```
import circle
```

When the Python interpreter reads this statement it will look for the file `circle.py` in the same folder as the program that is trying to import it. If it finds the file, it will load it into memory. If it does not find the file, an error occurs.²

² Actually the Python interpreter is set up to look in various other predefined locations in your system when it does not find a module in the program's folder. If you choose to learn about the advanced features of Python, you can learn how to specify where the interpreter looks for modules.

Once a module is imported you can call its functions. Assuming that `radius` is a variable that is assigned the radius of a circle, here is an example of how we would call the `area` and `circumference` functions:

```
my_area = circle.area(radius)
my_circum = circle.circumference(radius)
```

Program 5-28 shows a complete program that uses these modules.

Program 5-28 (geometry.py)

```
1  # This program allows the user to choose various
2  # geometry calculations from a menu. This program
3  # imports the circle and rectangle modules.
4  import circle
5  import rectangle
6
7  # Constants for the menu choices
8  AREA_CIRCLE_CHOICE = 1
9  CIRCUMFERENCE_CHOICE = 2
10 AREA_RECTANGLE_CHOICE = 3
11 PERIMETER_RECTANGLE_CHOICE = 4
12 QUIT_CHOICE = 5
13
14 # The main function.
15 def main():
16     # The choice variable controls the loop
17     # and holds the user's menu choice.
18     choice = 0
19
20     while choice != QUIT_CHOICE:
21         # display the menu.
22         display_menu()
23
24         # Get the user's choice.
25         choice = int(input('Enter your choice: '))
26
27         # Perform the selected action.
28         if choice == AREA_CIRCLE_CHOICE:
29             radius = float(input("Enter the circle's radius: "))
30             print('The area is', circle.area(radius))
31         elif choice == CIRCUMFERENCE_CHOICE:
32             radius = float(input("Enter the circle's radius: "))
33             print('The circumference is', \
34                   circle.circumference(radius))
35         elif choice == AREA_RECTANGLE_CHOICE:
36             width = float(input("Enter the rectangle's width: "))
37             length = float(input("Enter the rectangle's length: "))
38             print('The area is', rectangle.area(width, length))
```

```
39         elif choice == PERIMETER_RECTANGLE_CHOICE:
40             width = float(input("Enter the rectangle's width: "))
41             length = float(input("Enter the rectangle's length: "))
42             print('The perimeter is', \
43                   rectangle.perimeter(width, length))
44         elif choice == QUIT_CHOICE:
45             print('Exiting the program...')
46         else:
47             print('Error: invalid selection.')
48
49     # The display_menu function displays a menu.
50     def display_menu():
51         print(' MENU')
52         print('1) Area of a circle')
53         print('2) Circumference of a circle')
54         print('3) Area of a rectangle')
55         print('4) Perimeter of a rectangle')
56         print('5) Quit')
57
58     # Call the main function.
59     main()
```

Program Output (with input shown in bold)

```
    MENU
1) Area of a circle
2) Circumference of a circle
3) Area of a rectangle
4) Perimeter of a rectangle
5) Quit
Enter your choice: 1  Enter
Enter the circle's radius: 10
The area is 314.159265359
    MENU
1) Area of a circle
2) Circumference of a circle
3) Area of a rectangle
4) Perimeter of a rectangle
5) Quit
Enter your choice: 2  Enter
Enter the circle's radius: 10
The circumference is 62.8318530718
    MENU
1) Area of a circle
2) Circumference of a circle
3) Area of a rectangle
4) Perimeter of a rectangle
5) Quit
```

(program output continues)

Program Output *(continued)*

```

Enter your choice: 3 
Enter the rectangle's width: 5
Enter the rectangle's length: 10
The area is 50
      MENU
1) Area of a circle
2) Circumference of a circle
3) Area of a rectangle
4) Perimeter of a rectangle
5) Quit
Enter your choice: 4 
Enter the rectangle's width: 5
Enter the rectangle's length: 10
The perimeter is 30
      MENU
1) Area of a circle
2) Circumference of a circle
3) Area of a rectangle
4) Perimeter of a rectangle
5) Quit
Enter your choice: 5 
Exiting the program ...

```

Menu-Driven Programs

Program 5-28 is an example of a menu-driven program. A *menu-driven program* displays a list of the operations on the screen, and allows the user to select the operation that he or she wants the program to perform. The list of operations that is displayed on the screen is called a *menu*. When Program 5-28 is running, the user enters 1 to calculate the area of a circle, 2 to calculate the circumference of a circle, and so forth.

Once the user types a menu selection, the program uses a decision structure to determine which menu item the user selected. An `if-elif-else` statement is used in Program 5-28 (in lines 28 through 47) to carry out the user's desired action. The entire process of displaying a menu, getting the user's selection, and carrying out that selection is repeated by a `while` loop (which begins in line 14). The loop repeats until the user selects 5 (Quit) from the menu.

Review Questions

Multiple Choice

1. A group of statements that exist within a program for the purpose of performing a specific task is a(n) _____.
 - a. block
 - b. parameter
 - c. function
 - d. expression

2. A design technique that helps to reduce the duplication of code within a program and is a benefit of using functions is _____.
 - a. code reuse
 - b. divide and conquer
 - c. debugging
 - d. facilitation of teamwork
3. The first line of a function definition is known as the _____.
 - a. body
 - b. introduction
 - c. initialization
 - d. header
4. A variable created inside a function block is known as a _____.
 - a. global variable
 - b. local variable
 - c. super variable
 - d. new variable
5. A design technique that programmers use to break down an algorithm into functions is known as _____.
 - a. top-down design
 - b. code simplification
 - c. code refactoring
 - d. hierarchical subtasking
6. A _____ is a diagram that gives a visual representation of the relationships between functions in a program.
 - a. flowchart
 - b. function relationship chart
 - c. symbol chart
 - d. hierarchy chart
7. A _____ is a variable that is created inside a function.
 - a. global variable
 - b. local variable
 - c. hidden variable
 - d. none of the above; you cannot create a variable inside a function
8. A(n) _____ is the part of a program in which a variable may be accessed.
 - a. declaration space
 - b. area of visibility
 - c. scope
 - d. mode
9. A(n) _____ is a piece of data that is sent into a function.
 - a. argument
 - b. parameter
 - c. header
 - d. packet

10. A(n) _____ is a special variable that receives a piece of data when a function is called.
 - a. argument
 - b. parameter
 - c. header
 - d. packet
11. A variable that is visible to every function in a program file is a _____.
 - a. local variable
 - b. universal variable
 - c. program-wide variable
 - d. global variable
12. When possible, you should avoid using _____ variables in a program.
 - a. local
 - b. global
 - c. reference
 - d. parameter
13. This is a prewritten function that is built into a programming language.
 - a. standard function
 - b. library function
 - c. custom function
 - d. cafeteria function
14. A global variable whose value cannot be changed is known as _____.
 - a. global constant
 - b. local variable
 - c. global variable
 - d. local constant
15. This standard library function returns a random floating-point number in the range of 0.0 up to 1.0 (but not including 1.0).
 - a. random
 - b. randint
 - c. random_integer
 - d. uniform
16. This standard library function returns a random floating-point number within a specified range of values.
 - a. random
 - b. randint
 - c. random_integer
 - d. uniform
17. This statement causes a function to end and sends a value back to the part of the program that called the function.
 - a. end
 - b. send
 - c. exit
 - d. return

18. This value initializes the formula that generates random numbers.
 - a. dummy value
 - b. random value
 - c. seed value
 - d. new value
19. This type of function returns either True or False.
 - a. Binary
 - b. `true_false`
 - c. Boolean
 - d. logical
20. This is not a function of math module.
 - a. `hypot (x, y)`
 - b. `radians (x)`
 - c. `sin (x)`
 - d. `len (x)`

True or False

1. The phrase “divide and conquer” means that all of the programmers on a team should be divided and work in isolation.
2. Void functions do not return any value when they are called.
3. The value of a global constant can only be changed within a function.
4. Calling a function and defining a function mean the same thing.
5. Boolean functions have no return type.
6. A hierarchy chart does not show the steps that are taken inside a function.
7. A statement in one function can access a local variable in another function.
8. In Python you cannot write functions that accept multiple arguments.
9. An `import` statement can be used to call a built-in function that is stored in a module.
10. You cannot have both keyword arguments and non-keyword arguments in a function call.
11. Some library functions are built into the Python interpreter.
12. You do not need to have an `import` statement in a program to use the functions in the `random` module.
13. Complex mathematical expressions can sometimes be simplified by breaking out part of the expression and putting it in a function.
14. A local variable can be accessed by all the functions in a program.
15. IPO charts provide only brief descriptions of a function’s input, processing, and output, but do not show the specific steps taken in a function.

Short Answer

1. How do functions help you to reuse code in a program?
2. Name and describe the two parts of a function definition.
3. When a function is executing, what happens when the end of the function block is reached?
4. What is a local variable? What statements are able to access a local variable?

5. What is a local variable's scope?
6. Why do global variables make a program difficult to debug?
7. Suppose you want to select a random number from the following sequence:
0, 5, 10, 15, 20, 25, 30
What library function would you use?
8. What statement do you have to have in a value-returning function?
9. What three things are listed on an IPO chart?
10. What is a Boolean function?
11. What are the advantages of breaking a large program into modules?

Algorithm Workbench

1. Write a function named `times_ten`. The function should accept an argument and display the product of its argument multiplied times 10.
2. Examine the following function header, and then write a statement that calls the function, passing 12 as an argument.

```
def show_value(quantity):
```

3. Look at the following function header:

```
def my_function(a, b, c):
```

Now look at the following call to `my_function`:

```
my_function(3, 2, 1)
```

When this call executes, what value will be assigned to `a`? What value will be assigned to `b`? What value will be assigned to `c`?

4. What will the following program display?

```
def main():
    x = 1
    y = 3.4
    print(x, y)
    change_us(x, y)
    print(x, y)

def change_us(a, b):
    a = 0
    b = 0
    print(a, b)
main()
```

5. Look at the following function definition:

```
def my_function(a, b, c):
    d = (a + c) / b
    print(d)
```

- a. Write a statement that calls this function and uses keyword arguments to pass 2 into `a`, 4 into `b`, and 6 into `c`.
- b. What value will be displayed when the function call executes?

6. Write a function named `welcome()` that asks the user to enter his or her name and displays it followed by a welcome message.
7. The following statement calls a function named `half`, which returns a value that is half that of the argument. (Assume the `number` variable references a `float` value.) Write code for the function.


```
result = half(number)
```
8. What will the following code display?


```
xdef display(x,y):
    return(x%2==0)
print(display(4,7))
print(display(7,4))
```
9. Write a function named `area` that accepts the base and perpendicular of a right-angled triangle as two arguments and returns the area of that triangle.
10. Write a function named `get_first_name` that asks the user to enter his or her first name, and returns it.

Programming Exercises



VideoNote
The Kilometer
Converter Problem

1. Kilometer Converter

Write a program that asks the user to enter a distance in kilometers, and then converts that distance to miles. The conversion formula is as follows:

$$\text{Miles} = \text{Kilometers} \times 0.6214$$

2. Sales Tax Program Refactoring

Programming Exercise #6 in Chapter 2 was the Sales Tax program. For that exercise you were asked to write a program that calculates and displays the county and state sales tax on a purchase. If you have already written that program, redesign it so the subtasks are in functions. If you have not already written that program, write it using functions.

3. How Much Insurance?

Many financial experts advise that property owners should insure their homes or buildings for at least 80 percent of the amount it would cost to replace the structure. Write a program that asks the user to enter the replacement cost of a building and then displays the minimum amount of insurance he or she should buy for the property.

4. Automobile Costs

Write a program that asks the user to enter the monthly costs for the following expenses incurred from operating his or her automobile: loan payment, insurance, gas, oil, tires, and maintenance. The program should then display the total monthly cost of these expenses, and the total annual cost of these expenses.

5. Property Tax

A county collects property taxes on the assessment value of property, which is 60 percent of the property's actual value. For example, if an acre of land is valued at \$10,000, its assessment value is \$6,000. The property tax is then 72¢ for each \$100 of the assessment

value. The tax for the acre assessed at \$6,000 will be \$43.20. Write a program that asks for the actual value of a piece of property and displays the assessment value and property tax.

6. Calories from Fat and Carbohydrates

A nutritionist who works for a fitness club helps members by evaluating their diets. As part of her evaluation, she asks members for the number of fat grams and carbohydrate grams that they consumed in a day. Then, she calculates the number of calories that result from the fat, using the following formula:

$$\text{calories from fat} = \text{fat grams} \times 9$$

Next, she calculates the number of calories that result from the carbohydrates, using the following formula:

$$\text{calories from carbs} = \text{carb grams} \times 4$$

The nutritionist asks you to write a program that will make these calculations.

7. Stadium Seating

There are three seating categories at a stadium. For a softball game, Class A seats cost \$20, Class B seats cost \$15, and Class C seats cost \$10. Write a program that asks how many tickets for each class of seats were sold, and then displays the amount of income generated from ticket sales.

8. Paint Job Estimator

A painting company has determined that for every 112 square feet of wall space, one gallon of paint and eight hours of labor will be required. The company charges \$35.00 per hour for labor. Write a program that asks the user to enter the square feet of wall space to be painted and the price of the paint per gallon. The program should display the following data:

- The number of gallons of paint required
- The hours of labor required
- The cost of the paint
- The labor charges
- The total cost of the paint job

9. Monthly Sales Tax

A retail company must file a monthly sales tax report listing the total sales for the month, and the amount of state and county sales tax collected. The state sales tax rate is 5 percent and the county sales tax rate is 2.5 percent. Write a program that asks the user to enter the total sales for the month. From this figure, the application should calculate and display the following:

- The amount of county sales tax
- The amount of state sales tax
- The total sales tax (county plus state)

10. Feet to Inches

One foot equals 12 inches. Write a function named `feet_to_inches` that accepts a number of feet as an argument and returns the number of inches in that many feet. Use the function in a program that prompts the user to enter a number of feet and then displays the number of inches in that many feet.

11. Math Quiz

Write a program that gives simple math quizzes. The program should display two random numbers that are to be added, such as:

```
247
+ 129
```

The program should allow the student to enter the answer. If the answer is correct, a message of congratulations should be displayed. If the answer is incorrect, a message showing the correct answer should be displayed.

12. Maximum of Two Values

Write a function named `max` that accepts two integer values as arguments and returns the value that is the greater of the two. For example, if 7 and 12 are passed as arguments to the function, the function should return 12. Use the function in a program that prompts the user to enter two integer values. The program should display the value that is the greater of the two.

13. Falling Distance

When an object is falling because of gravity, the following formula can be used to determine the distance the object falls in a specific time period:

$$d = \frac{1}{2}gt^2$$

The variables in the formula are as follows: d is the distance in meters, g is 9.8, and t is the amount of time, in seconds, that the object has been falling.

Write a function named `falling_distance` that accepts an object's falling time (in seconds) as an argument. The function should return the distance, in meters, that the object has fallen during that time interval. Write a program that calls the function in a loop that passes the values 1 through 10 as arguments and displays the return value.

14. Kinetic Energy

In physics, an object that is in motion is said to have kinetic energy. The following formula can be used to determine a moving object's kinetic energy:

$$KE = \frac{1}{2}mv^2$$

The variables in the formula are as follows: KE is the kinetic energy, m is the object's mass in kilograms, and v is the object's velocity in meters per second.

Write a function named `kinetic_energy` that accepts an object's mass (in kilograms) and velocity (in meters per second) as arguments. The function should return the amount of kinetic energy that the object has. Write a program that asks the user to enter values for mass and velocity, and then calls the `kinetic_energy` function to get the object's kinetic energy.

15. Test Average and Grade

Write a program that asks the user to enter five test scores. The program should display a letter grade for each score and the average test score. Write the following functions in the program:

- `calc_average`—This function should accept five test scores as arguments and return the average of the scores.

- `determine_grade`—This function should accept a test score as an argument and return a letter grade for the score based on the following grading scale:

Score	Letter Grade
90–100	A
80–89	B
70–79	C
60–69	D
Below 60	F

16. Odd/Even Counter

In this chapter, you saw an example of how to write an algorithm that determines whether a number is even or odd. Write a program that generates 100 random numbers and keeps a count of how many of those random numbers are even and how many of them are odd.

17. Prime Numbers

A prime number is a number that is only evenly divisible by itself and 1. For example, the number 5 is prime because it can only be evenly divided by 1 and 5. The number 6, however, is not prime because it can be divided evenly by 1, 2, 3, and 6.

Write a Boolean function named `is_prime` which takes an integer as an argument and returns true if the argument is a prime number, or false otherwise. Use the function in a program that prompts the user to enter a number and then displays a message indicating whether the number is prime.



TIP: Recall that the `%` operator divides one number by another and returns the remainder of the division. In an expression such as `num1 % num2`, the `%` operator will return 0 if `num1` is evenly divisible by `num2`.

18. Prime Number List

This exercise assumes that you have already written the `is_prime` function in Programming Exercise 17. Write another program that displays all of the prime numbers from 1 to 100. The program should have a loop that calls the `is_prime` function.

19. Future Value

Suppose you have a certain amount of money in a savings account that earns compound monthly interest, and you want to calculate the amount that you will have after a specific number of months. The formula is as follows:

$$F = P \times (1 + i)^t$$

The terms in the formula are:

- *F* is the future value of the account after the specified time period.
- *P* is the present value of the account.
- *i* is the monthly interest rate.
- *t* is the number of months.

Write a program that prompts the user to enter the account's present value, monthly interest rate, and the number of months that the money will be left in the account. The program should pass these values to a function that returns the future value of the account, after the specified number of months. The program should display the account's future value.

20. Random Number Guessing Game

Write a program that generates a random number in the range of 1 through 100, and asks the user to guess what the number is. If the user's guess is higher than the random number, the program should display "Too high, try again." If the user's guess is lower than the random number, the program should display "Too low, try again." If the user guesses the number, the application should congratulate the user and then generate a new random number so the game can start over.

Optional Enhancement: Enhance the game so it keeps count of the number of guesses that the user makes. When the user correctly guesses the random number, the program should display the number of guesses.

21. Rock, Paper, Scissors Game

Write a program that lets the user play the game of Rock, Paper, Scissors against the computer. The program should work as follows:

1. When the program begins, a random number in the range of 1 through 3 is generated. If the number is 1, then the computer has chosen rock. If the number is 2, then the computer has chosen paper. If the number is 3, then the computer has chosen scissors. (Don't display the computer's choice yet.)
2. The user enters his or her choice of "rock," "paper," or "scissors" at the keyboard.
3. The computer's choice is displayed.
4. A winner is selected according to the following rules:
 - If one player chooses rock and the other player chooses scissors, then rock wins. (The rock smashes the scissors.)
 - If one player chooses scissors and the other player chooses paper, then scissors wins. (Scissors cuts paper.)
 - If one player chooses paper and the other player chooses rock, then paper wins. (Paper wraps rock.)
 - If both players make the same choice, the game must be played again to determine the winner.