Chapter 5 (second part): Functions

These slides have been prepared based on the book titled "Starting Out with Python" by Tony Gaddis.

Lecturer: F. Kuzey Edes-Huyal

Bahcesehir University

November 7, 2023

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.

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.
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`.

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(f'The number you entered is {number}.')
11
12  # Call the main function.
13  main()
```

Program Output

```
Enter a number: 55 Enter
The number you entered is 55
```

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.

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 keyword 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.

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.

Program 5-15 (retirement.py)

```
1  # The following is used as a global constant to represent
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(f'Contribution for gross pay: ${contrib:,.2f}.')
17
18 # The show_bonus_contrib function accepts the
19 # bonus amount as an argument and displays the
20 # retirement contribution for that amount of pay.
21 def show_bonus_contrib(bonus):
22     contrib = bonus * CONTRIBUTION_RATE
23     print(f'Contribution for bonuses: ${contrib:,.2f}.')
24
25 # Call the main function.
26 main()
```

Program Output (with input shown in bold)

```
Enter the gross pay: 80000.00 Enter
Enter the amount of bonuses: 20000.00 Enter
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 `sho_ pay_contrib` function) and again in line 22 (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.

Introduction to Value-Returning Functions: Generating Random Numbers

In the first part of this chapter, you learned about void functions. A **void function** is a groupof 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 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.

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 about which you have already learned.

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.

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.

Generating Random Numbers

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

- Random numbers are commonly used in games.
- Random numbers are useful in simulation programs.
- 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
```

The first random-number generating function that we will discuss is named `randint`. 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 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.

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
 9      # Display the number.
10      print(f'The number is {number}.')
11
12  # Call the main function.
13  main()
```

Program Output

```
The number is 7.
```

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

Program Output

```
89
7
16
41
12
```

Calling Functions from an F-String

A function call can be used as a placeholder in an f-string. Here is an example:

```
print(f'The number is {random.randint(1, 100)}.')
```

This statement will display a message such as: `The number is 58`.

F-strings are especially helpful when you want to format the result of a function call.

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.

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

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 assignsa random number in therange of 0 through 9 to the number variable:

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

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.
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.

```
>> import random Enter
>> random.seed(10) Enter
>> random.randint(1, 100) Enter
58
>> random.randint(1, 100) Enter
43
```

As you can see, the function gave us the numbers 58, 43 and 58. If we start a new interactive session and repeat these statements, we get the same sequence of pseudorandom numbers.

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
```

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(f'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
Enter your best friend's age: 24 Enter
Together you are 46 years old.
```

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 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.

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
```

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)`
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)`

```
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(f'The sale price is ${sale_price:,.2f}.')
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):
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 Enter
The sale price is $80.00.
```

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 you saw in Program 5-22.

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

The `get_regular_price` Function		
Input	Processing	Output
None	Prompts the user to enter an item's regular price	The item's regular price

The `discount` Function		
Input	Processing	Output
An item's regular price	Calculates an item's discount by multiplying the regular price by the global constant `DISCOUNT_PERCENTAGE`	The item's discount

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, 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
```

A function can also return an f-string. When a function returns an f-string, the Python interpreter will evaluate any placeholders and format specifiers that the f-string contains, and it will return the formatted result. Here is an example:

```
def dollar_format(value):
    return f'{value:,.2f}'
```

Returning Boolean Values

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
```

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
```

```
# 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.

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.
```

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()
```

Returning None from a Function

Python has a special built-in value named None that is used to represent no value. Sometimes it is useful to return None from a function to indicate that an error has occurred. For example, consider the following function:

```
def divide(num1, num2):
    return num1 / num2
```

The divide function takes two arguments, num1 and num2, and returns the result of num1 divided by num2. However, an error will occur if num2 is equal to zero because division by zero is not possible.

To prevent the program from crashing, we can modify the function to,

```
def divide(num1, num2):
    if num2 == 0:
        result = None
    else:
        result = num1 / num2
    return result
```

Program 5-24 (none_demo.py)

```
1  # This program demonstrates the None keyword.
2
3  def main():
4      # Get two numbers from the user.
5      num1 = int(input('Enter a number: '))
6      num2 = int(input('Enter another number: '))
7
8      # Call the divide function.
9      quotient = divide(num1, num2)
10
11     # Display the result.
12     if quotient is None:
13         print('Cannot divide by zero.')
14     else:
15         print(f'{num1} divided by {num2} is {quotient}.')
16
17 # The divide function divides num1 by num2 and
18 # returns the result. If num2 is 0, the function
19 # returns None.
20 def divide(num1, num2):
21     if num2 == 0:
22         result = None
23     else:
24         result = num1 / num2
25     return result
26
27 # Execute the main function.
28 main()
```

Program Output (with Input Shown in Bold)

```
Enter a number: 10 Enter
Enter another number: 0 Enter
```

The `math` Module

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)
```

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

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

Program 5-25 (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(f'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 Enter
The square root of 25.0 is 5.0.
```

Storing Functions in 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 5-27 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 5-27 (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
```

(program continues)

Program 5-27 *(continued)*

```
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 5-28 (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 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 keyword. 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.

Once a module is imported you can call its functions. Assuming 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)
```

Conditionally Executing the `main` Function in a Module

When a module is imported, the Python interpreter executes the statements in the module just as if the module were a standalone program. For example, when we import the circle.py module shown in Program 5-27, the following things happen:

- The math module is imported.
- A function named area is defined.
- A function named circumference is defined.

When we import the rectangle.py module shown in Program 5-28, the following things happen:

- A function named area is defined.
- A function named perimeter is defined.

When programmers create modules, they do not usually intend for those modules to be run as standalone programs. Modules are typically intended to be imported into other programs. For that reason, most modules only define things such as functions.

However, it is possible to create a Python module that can be either run as a standalone program or imported into another program. For example, suppose Program A defines several useful functions that you would like to use in Program B. Therefore, you would like to import Program A into Program B. However, you do not want Program A to start executing its main function when you import it. You simply want it to define its functions without executing any of them. To accomplish this, you need to write code in Program A that determines how the file is being used. Is it being run as a standalone program? Or is it being imported into another program? The answer will determine whether the main function in Program A is supposed to execute.

Fortunately, Python provides a way to make this determination. When the Python interpreter is processing a source code file, it creates a special variable named `__name__`. (The variable's name begins with two underscore characters and ends with two underscore characters.) If the file is being imported as a module, the `__name__` variable will be set to the name of the module. Otherwise, if the file is being executed as a standalone program, the `__name__` variable will be set to the string '`__main__`'. You can use the value of the `__name__` variable to determine whether the main function should execute or not. If the `__name__` variable is equal to '`__main__`', you should execute the main function because the file is being executed as a standalone program.

Program 5-30 (rectangle2.py)

```
1  # The area function accepts a rectangle's width and
2  # length as arguments and returns the rectangle's area.
3  def area(width, length):
4      return width * length
5
6  # The perimeter function accepts a rectangle's width
7  # and length as arguments and returns the rectangle's
8  # perimeter.
9  def perimeter(width, length):
10     return 2 * (width + length)
11
12 # The main function is used to test the other functions.
13 def main():
14     width = float(input("Enter the rectangle's width: "))
15     length = float(input("Enter the rectangle's length: "))
16     print('The area is', area(width, length))
17     print('The perimeter is', perimeter(width, length))
18
19 # Call the main function ONLY if the file is being run as
20 # a standalone program.
21 if __name__ == '__main__':
22     main()
```

The rectangle2.py program defines an area function, a perimeter function, and a main function. Then, the if statement in line 21 tests the value of the `_ _name_ _variable`. If the variable is equal to '`_ _main_ _`', the statement in line 22 calls the main function. Otherwise, if the `_ _name_ _variable` is set to any other value, the main function is not executed. The technique shown in Program 5-30 is a good practice to use any time you have a main function in a Python source file. It ensures that when you import the file it will behave as a module, and when you execute the file directly, it will behave as a standalone program. Form this point forward in the book, we will use this technique anytime we show you an example that uses a main function.

Possible Quiz Question 1

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, then displays the minimum amount of insurance he or she should buy for the property.

Possible Quiz Question 2

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:

calories from fat = fat grams * 9

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

calories from carbs = carb grams * 4

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

Possible Quiz Question 3

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.

Today's quiz question

Create a function that accepts the user's first name and last name as input and displays the reversed version. Utilize keyword arguments instead of positional ones.