variable, are removed. Any values that have been assigned to them are simply forgotten. Note that `total` is *not* changed.

In Python, a function can never change the contents of a variable that was passed as an argument. When you call a function with a variable as argument, you don't actually pass the variable, just the value that it contains.

5.4 Return Values

The return statement terminates a function call and yields the function result.

You use the `return` statement to specify the result of a function. In the preceding examples, each `return` statement returned a variable. However, the `return` statement can return the value of any expression. Instead of saving the return value in a variable and returning the variable, it is often possible to eliminate the variable and return the value of a more complex expression:

```
def cubeVolume(sideLength) :
   return sideLength ** 3
```

When the `return` statement is processed, the function exits *immediately*. Some programmers find this behavior convenient for handling exceptional cases at the beginning of the function:

```
def cubeVolume(sideLength)
   if sideLength < 0 :
      return 0
   # Handle the regular case.
   . . .
```

If the function is called with a negative value for `sideLength`, then the function returns `0` and the remainder of the function is not executed. (See Figure 4.)

Every branch of a function should return a value. Consider the following incorrect function:

```
def cubeVolume(sideLength) :
   if sideLength >= 0 :
      return sideLength ** 3
   # Error—no return value if sideLength < 0
```

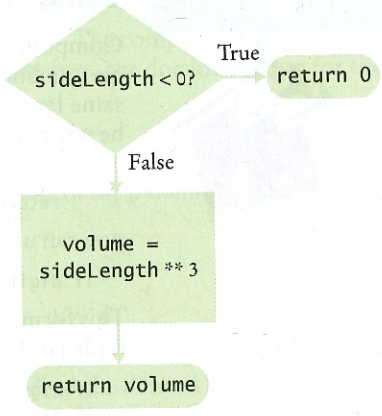


Figure 4 A `return` Statement Exits a Function Immediately

The compiler will not report this as an error. Instead, the special value `None` will be returned from the function. A correct implementation is:

```
def cubeVolume(sideLength) :
   if sideLength >= 0
      return sideLength ** 3
   else :
      return 0
```

Some programmers dislike the use of multiple return statements in a function. You can avoid multiple returns by storing the function result in a variable that you return in the last statement of the function. For example:

```
def cubeVolume(sideLength) :
   if sideLength >= 0:
      volume = sideLength ** 3
   else :
      volume = 0
   return volume
```

See `ch05/earthquake.py` in your source code for a complete program that demonstrates a function that returns a value.

13. Suppose we change the body of the `cubeVolume` function to

```
if sideLength <= 0 :
   return 0
return sideLength ** 3
```

How does this function differ from the one described in this section?

14. What does this function do?

```
def mystery (n) :
   if n % 2 == 0 :
      return True
   else :
      return False
```

15. Implement the `mystery` function of Self Check 14 with a single `return` statement.

Practice It Now you can try these exercises at the end of the chapter: R5.12, P5.20.

Special Topic 5.1

Using Single-Line Compound Statements

Compounds statements in Python are generally written across several lines. The header is on one line and the body on the following lines, with each body statement indented to the same level. When the body contains a single statement, however, compound statements may be written on a single line. For example, instead of constructing the following `if` statement:

```
if digit == 1 :
   return "one"
```

you can use the special single-line form because the body contains a single statement

```
if digit == 1 : return "one"
```

This form can be very useful in functions that select a single value from among a collection and return it. For example, the single-line form used here

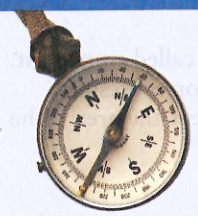
```
if digit == 1 : return "one"
if digit == 2 : return "two"
if digit == 3 : return "three"
```

```
if digit == 4 : return "four"
if digit == 5 : return "five"
if digit == 6 : return "six"
if digit == 7 : return "seven"
if digit == 8 : return "eight"
if digit == 9 : return "nine"
```

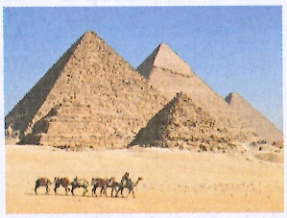
produces condensed code that is easy to read.

Sometimes, the use of single-line compound statements can be distracting or cause the reader to accidentally skip over important details. Thus, in this book, we limit its use to `if` statements that contain a `return` clause.

HOW TO 5.1 Implementing a Function



A function is a computation that can be used multiple times with different arguments, either in the same program or in different programs. Whenever a computation may be needed more than once, turn it into a function.



Problem Statement Suppose that you are helping archaeologists who research Egyptian pyramids. You have taken on the task of writing a function that determines the volume of a pyramid, given its height and base length.

Step 1 Describe what the function should do.

Provide a simple English description, such as "Compute the volume of a pyramid whose base is a square."

Step 2 Determine the function's "inputs".

Turn computations that can be reused into functions.

Make a list of *all* the parameters that can vary. It is common for beginners to implement functions that are overly specific. For example, you may know that the great pyramid of Giza, the largest of the Egyptian pyramids, has a height of 146 meters and a base length of 230 meters. You should *not* use these numbers in your calculation, even if the original problem only asked about the great pyramid. It is just as easy—and far more useful—to write a function that computes the volume of *any* pyramid. In our case, the parameters are the pyramid's height and base length.

Step 3 Determine the types of the parameter variables and the return value.

The height and base length can both be floating-point numbers. The computed volume is also a floating-point number, yielding a return type of `float`. Therefore, the documentation for the function will be

```
## Computes the volume of a pyramid whose base is square.
#  @param height a float indicating the height of the pyramid
#  @param baseLength a float indicating the length of one side of the pyramid's base
#  @return the volume of the pyramid as a float
```

and the function will be defined as

```
def pyramidVolume(height, baseLength) :
```

Step 4 Write pseudocode for obtaining the desired result.

In most cases, a function needs to carry out several steps to find the desired answer. You may need to use mathematical formulas, branches, or loops. Express your function in pseudocode.

An Internet search yields the fact that the volume of a pyramid is computed as

volume = 1/3 x height x base area

Because the base is a square, we have

base area = base length x base length

Using these two equations, we can compute the volume from the arguments.

Step 5 Implement the function body.

In our example, the function body is quite simple. Note the use of the `return` statement to return the result.

```
def pyramidVolume(height, baseLength) :
   baseArea = baseLength * baseLength
   return height * baseArea / 3
```

Step 6 Test your function.

After implementing a function, you should test it in isolation. Such a test is called a **unit test**. Work out test cases by hand, and make sure that the function produces the correct results.

For example, for a pyramid with height 9 and base length 10, we expect the area to be $1/3 \times 9 \times 100 = 300$. If the height is 0, we expect an area of 0.

```
def main() :
   print("Volume:" + pyramidVolume(9, 10)
   print("Expected: 300")
   print("Volume:" + pyramidVolume(0, 10)
   print("Expected: 0")
```

The output confirms that the function worked as expected:

```
Volume: 300
Expected: 300
Volume: 0
Expected: 0
```

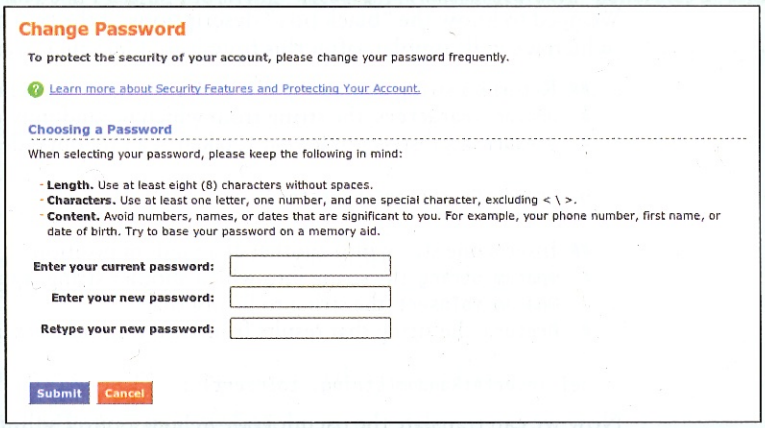
The complete program for calculating a pyramid's volume is provided below.

ch05/pyramids.py

```
1  ##
2  #  This program defines a function for calculating a pyramid's volume and
3  #  provides a unit test for the function.
4  #
5
6  def main() :
7     print("Volume:", pyramidVolume(9, 10)
8     print("Expected: 300")
9     print("Volume:", pyramidVolume(0, 10)
10    print("Expected: 0")
11
12 ## Computes the volume of a pyramid whose base is a square.
13 #  @param height a float indicating the height of the pyramid
14 #  @param baseLength a float indicating the length of one side of the pyramid's base
15 #  @return the volume of the pyramid as a float
16 #
17 def pyramidVolume(height, baseLength)
18    baseArea = baseLength * baseLength
19    return height * baseArea / 3
20
21 # Start the program.
22 main()
```

WORKED EXAMPLE 5.1 Generating Random Passwords

Problem Statement Many web sites and software packages require you to create passwords that contain at least one digit and one special character. Your task is to write a program that generates such a password of a given length. The characters should be chosen randomly.



Step 1 Describe what the function should do.

The problem description asks you to write a program, not a function. We will write a password-generating function and call it from the program's `main` function.

Let us be more precise about the function. It will generate a password with a given number of characters. We could include multiple digits and special characters, but for simplicity, we decide to include just one of each. We need to decide which special characters are valid. For our solution, we will use the following set:

```
+ - * / ? ! @ # $ % &
```

The remaining characters of the password are letters. For simplicity, we will use only lowercase letters in the English alphabet.

Step 2 Determine the function's "inputs".

There is just one parameter: the length of the password.

Step 3 Determine the types of the parameter variables and the return value.

At this point, we have enough information to document and specify the function header:

```
## Generates a random password.
#  @param length an integer that specifies the length of the password
#  @return a string containing the password of the given length with one
#  digit and one special character
#
def makePassword(length) :
```

Step 4 Write pseudocode for obtaining the desired result.

Here is one approach for making a password:

```
Make an empty string called password.
Randomly generate length - 2 letters and append them to password.
Randomly generate a digit and insert it at a random location in password.
Randomly generate a symbol and insert it at a random location in password.
```

How do we generate a random letter, digit, or symbol? How do we insert a digit or symbol in a random location? We will delegate those tasks to helper functions. Each of those functions starts a new sequence of steps, which, for greater clarity, we will place after the steps for this function.

Step 5 Implement the function body.

We need to know the "black box" descriptions of the two helper functions described in Step 4 (which we will complete after this function). Here they are:

```
## Returns a string containing one character randomly chosen from a given string.
#  @param characters the string from which to randomly choose a character
#  @return a substring of length 1, taken at a random index
#
def randomCharacter(characters) :

## Inserts one string into another at a random position.
#  @param string the string into which another string is inserted
#  @param toInsert the string to be inserted
#  @return the string that results from inserting toInsert into string
#
def insertAtRandom(string, toInsert) :
```

Now we can translate the pseudocode in Step 4 into Python:

```
def makePassword(length) :
   password = ""
   for i in range(length - 2) :
      password = password + randomCharacter("abcdefghijklmnopqrstuvwxyz")

   randomDigit = randomCharacter("0123456789")
   password = insertAtRandom(password, randomDigit)

   randomSymbol = randomCharacter("+-*/?!@#$%&")
   password = insertAtRandom(password, randomSymbol)

   return password
```

Step 6 Test your function.

Because our function depends on several helper functions, we must implement the helper functions first, as described in the following sections. (If you are impatient, you can use the technique of stubs that is described in Programming Tip 5.5.)

Here is a simple `main` function that calls the `makePassword` function:

```
def main() :
   result = makePassword(8)
   print(result)
```

Place all functions into a file named `password.py`. Add a call to `main`. Run the program a few times. Typical outputs are

```
u@taqr8f
i?fs1dgh
ot$3rvdv
```

Each output has length 8 and contains a digit and special symbol.

Repeat for the First Helper Function

Now it is time to turn to the helper function for generating a random letter, digit, or special symbol.

Step 1 Describe what the function should do.

How do we deal with the choice between letter, digit, or special symbol? Of course, we could write three separate functions, but it is better if we can solve all three tasks with a single function. We could require a parameter, such as 1 for letter, 2 for digit, and 3 for special symbol. But stepping back a bit, we can supply a more general function that simply selects a random character from *any* set. Passing the string `"abcdefghijklmnopqrstuvwxyz"` generates a random lowercase letter. To get a random digit, pass the string `"0123456789"` instead.

Now we know what our function should do. Given any string, it should return a random character in it.

Step 2 Determine the function's "inputs".

The input is any string.

Step 3 Determine the types of the parameter variables and the return value.

The input type is clearly a string, as is the return value.

The function header will be:

```
def randomCharacter(characters) :
```

Step 4 Write pseudocode for obtaining the desired result.

```
n = length of the input string, characters
r = a random integer between 0 and n - 1
return the substring of characters of length 1 that starts at r
```

Step 5 Implement the function body.

Simply translate the pseudocode into Python:

```
def randomCharacter(characters) :
   n = len(characters)
   r = randint(0, n - 1)
   return characters[r]
```

Step 6 Test your function.

Supply a program file for testing this function only:

```
from random import randint

def main() :
   for i in range(10) :
      print(randomCharacter("abcdef", end="")
   print()

def randomCharacter(characters) :
   n = len(characters) :
   r = randint(0, n - 1)
   return characters[r]

main()
```

When you run this program, you might get an output such as

```
afcdfeefac
```

This confirms that the function works correctly.

Repeat for the Second Helper Function

Finally, we implement the second helper function, which inserts a string containing a single character at a random location in a string.

Step 1 Describe what the function should do.

Suppose we have a string `"arxcsw"` and a string `"8"`. Then the second string should be inserted at a random location, returning a string such as `"ar8xcsw"` or `"arxcsw8"`. Actually, it doesn't matter that the second string has length 1, so we will simply specify that our function should insert an arbitrary string into a given string.

Step 2 Determine the function's "inputs".

The first input is the string into which another string should be inserted. The second input is the string to be inserted.

Step 3 Determine the types of the parameter variables and the return value.

The inputs are both strings, and the result is also a string. We can now fully describe our function:

```
## Inserts one string into another at a random position.
#  @param string the string into which another string is inserted
#  @param toInsert the string to be inserted
#  @return a string that results from inserting toInsert into string
#
def insertAtRandom(string, toInsert) :
```

Step 4 Write pseudocode for obtaining the desired result.

There is no predefined function for inserting a string into another. Instead, we need to find the insertion position and then "break up" the first string by taking two substrings: the characters up to the insertion position, and the characters following it.

How many choices are there for the insertion position? If `string` has length 6, there are seven choices:

1. `|arxcsw`
2. `a|rxcsw`
3. `ar|xcsw`
4. `arx|csw`
5. `arxc|sw`
6. `arxcs|w`
7. `arxcsw|`

In general, if the string has length n, there are $n + 1$ choices, ranging from 0 (before the start of the string) to n (after the end of the string).

Here is the pseudocode:

```
n = length of the string
r = a random integer between 0 and n (inclusive)
result = the characters in string from 0 to r (exclusive) + toInsert + the remainder of string
```

Step 5 Implement the function body.

Translate the pseudocode into Python:

```
def insertAtRandom(string, toInsert) :
   n = len(string)
   r = randint(0, n)
   result = ""

   for i in range(r) :
      result = result + string[i]
   result = result + toInsert
   for i in range(r, n) :
      result = result + string[i]

   return result
```

Step 6 Test your function.

Supply a program file for testing this function only:

```
from random import randint

def main() :
   for i in range(10) :
      print(insertAtRandom("arxcsw", "8")

def insertAtRandom(string, toInsert) :
   n = len(string) :
   r = randint(0, n)
   result = ""

   for i in range(r) :
      result = result + string[i]
   result = result + toInsert
   for i in range(r, n) :
      result = result + string[i]

   return result

main()
```

When you run this program, you might get an output such as

```
arxcsw8
ar8xcsw
arxc8sw
a8rxcsw
arxcsw8
ar8xcsw
arxcsw8
a8rxcsw
8arxcsw
8arxcsw
```

The output shows that the second string is being inserted at an arbitrary position, including the beginning and end of the first string.

See `password.py` in your source code for the complete program.

5.5 Functions Without Return Values

Some functions may not return a value, but they can produce output.

Sometimes, you need to carry out a sequence of instructions that does not yield a value. If that instruction sequence occurs multiple times, you will want to package it into a function.

Here is a typical example: Your task is to print a string in a box, like this:

```
-------
!Hello!
-------
```

Some functions are called because they produce output, even though they don't return a value.