5.2 Implementing and Testing Functions

In this section, you will learn how to implement a function from a given specification, and how to call it with test inputs.

5.2.1 Implementing a Function

We will start with a very simple example: a function to compute the volume of a cube with a given side length.

The `cubeVolume` *function uses a given side length to compute the volume of a cube.*

When defining a function, you provide a name for the function and a variable for each argument.

When writing this function, you need to

- Pick a name for the function (`cubeVolume`).
- Define a variable for each argument (`sideLength`). These variables are called the **parameter variables.**

Put all this information together along with the `def` reserved word to form the first line of the function's definition:

```
def cubeVolume(sideLength) :
```

This line is called the **header** of the function. Next, specify the *body* of the function. The body contains the statements that are executed when the function is called.

The volume of a cube of side length s is $s \times s \times s = s^3$. However, for greater clarity, our parameter variable has been called `sideLength`, not s, so we need to compute `sideLength ** 3`.

We will store this value in a variable called `volume`:

```
volume = sideLength ** 3
```

In order to return the result of the function, use the `return` statement:

```
return volume
```

The `return` *statement gives the function's result to the caller.*

A function is a compound statement, which requires the statements in the body to be indented to the same level. Here is the complete function:

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

5.2.2 Testing a Function

In the preceding section, you saw how to write a function. If you run a program containing just the function definition, then nothing happens. After all, nobody is calling the function.

In order to test the function, your program should contain

- The definition of the function.
- Statements that call the function and print the result.

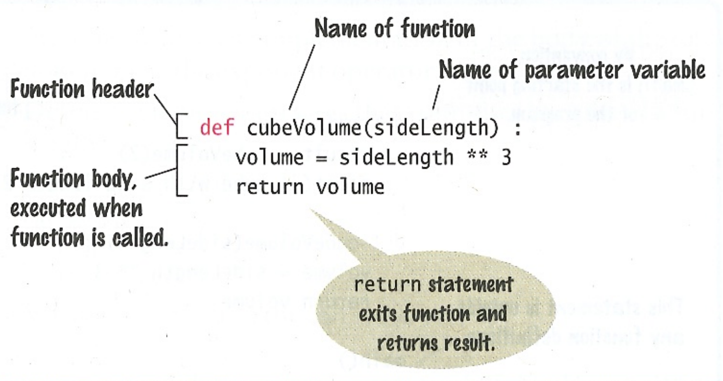
Here is such a program:

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

result1 = cubeVolume(2)
result2 = cubeVolume(10)

print("A cube with side length 2 has volume", result1)
print("A cube with side length 10 has volume", result2)
```

Note that the function returns different results when it is called with different arguments. Consider the call `cubeVolume(2)`. The argument `2` corresponds to the `sideLength` parameter variable. Therefore, in this call, `sideLength` is `2`. The function computes `sideLength ** 3`, or `2 ** 3`. When the function is called with a different argument, say `10`, then the function computes `10 ** 3`.

Syntax 5.1 Function Definition

Syntax `def` *functionName*(*parameterName*$_1$, *parameterName*$_2$, . . .) :
   statements


Name of function
Name of parameter variable
Function header
Function body, executed when function is called.
return statement exits function and returns result.

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

5.2.3 Programs that Contain Functions

When you write a program that contains one or more functions, you need to pay attention to the order of the function definitions and statements in the program.

Have another look at the program of the preceding section. Note that it contains

- The definition of the `cubeVolume` function.
- Several statements, two of which call that function.

As the Python interpreter reads the source code, it reads each function definition and each statement. The statements in a function definition are not executed until the function is called. Any statement not in a function definition, on the other hand, is executed as it is encountered. Therefore, it is important that you define each function before you call it. For example, the following will produce a compile-time error

```
print(cubeVolume(10))

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

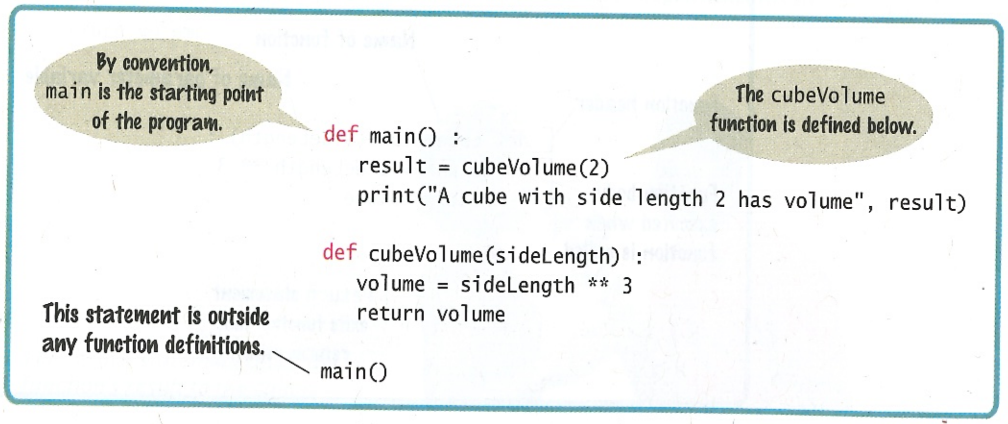
The compiler does not know that the `cubeVolume` function will be defined later in the program.

However, a function can be called from within another function before the former has been defined. For example, the following is perfectly legal:

```
def main() :
   result = cubeVolume(2)
   print("A cube with side length 2 has volume", result)

def cubeVolume(sideLength) :
   volume = sideLength ** 3
   return volume

main()
```

Note that the `cubeVolume` function is called from within the `main` function even though `cubeVolume` is defined after `main`. To see why this is not a problem, consider the flow of execution. The definitions of the `main` and `cubeVolume` functions are processed. The statement in the last line is not contained in any function. Therefore, it is executed directly. It calls the `main` function. The body of the `main` function executes, and it calls `cubeVolume`, which is now known.

Syntax 5.2 Program with Functions


By convention, main is the starting point of the program.
The cubeVolume function is defined below.
This statement is outside any function definitions.

```
def main() :
   result = cubeVolume(2)
   print("A cube with side length 2 has volume", result)

def cubeVolume(sideLength) :
   volume = sideLength ** 3
   return volume

main()
```

When defining and using functions in Python, it is good programming practice to place all statements into functions, and to specify one function as the starting point. In the previous example, the `main` function is the point at which execution begins. Any legal name can be used for the starting point, but we chose `main` because it is the required function name used by other common languages.

Of course, we must have one statement in the program that calls the `main` function. That statement is the last line of the program, `main()`.

The complete program including comments is provided below. Note that both functions are in the same file. Also note the comment that describes the behavior of the `cubeVolume` function. (Programming Tip 5.1 describes the format of the comment.)

ch05/cubes.py

```python
##
#  This program computes the volumes of two cubes.
#

def main() :
   result1 = cubeVolume(2)
   result2 = cubeVolume(10)
   print("A cube with side length 2 has volume", result1)
   print("A cube with side length 10 has volume", result2)

## Computes the volume of a cube.
#  @param sideLength the length of a side of the cube
#  @return the volume of the cube
#
def cubeVolume(sideLength) :
   volume = sideLength ** 3
   return volume

# Start the program.
main()
```

Program Run

```
A cube with side length 2 has volume 8
A cube with side length 10 has volume 1000
```

SELF CHECK

5. What is the value of `cubeVolume(3)`?
6. What is the value of `cubeVolume(cubeVolume(2))`?
7. Provide an alternate implementation of the body of the `cubeVolume` function that does not use the exponent operator.
8. Define a function `squareArea` that computes the area of a square of a given side length.
9. Consider this function:

```python
def mystery(x, y) :
   result = (x + y) / (y - x)
   return result
```

What is the result of the call `mystery(2, 3)`?

Practice It Now you can try these exercises at the end of the chapter: R5.1, R5.2, P5.5, P5.22.

Programming Tip 5.1

Function Comments

Whenever you write a function, you should *comment* its behavior. Comments are for human readers, not compilers. Various individuals prefer different layouts for function comments. In this book, we will use the following layout:

```
## Computes the volume of a cube.
#  @param sideLength the length of a side of the cube
#  @return the volume of the cube
#
def cubeVolume(sideLength) :
   volume = sideLength ** 3
   return volume
```

Function comments explain the purpose of the function, the meaning of the parameter variables and return value, as well as any special requirements.

This particular documentation style is borrowed from the Java programming language. It is supported by a wide variety of documentation tools such as Doxygen (www.doxygen.org), which extracts the documentation in HTML format from the Python source.

Each line of the function comment begins with a hash symbol (#) in the first column. The first line, which is indicated by two hash symbols, describes the purpose of the function. Each @param clause describes a parameter variable and the @return clause describes the return value.

There is an alternative (but, in our opinion, somewhat less descriptive) way of documenting the purpose of a Python function. Add a string, called a "docstring", as the first statement of the function body, like this:

```
def cubeVolume(sideLength) :
   "Computes the volume of a cube."
   volume = sideLength ** 3
   return volume
```

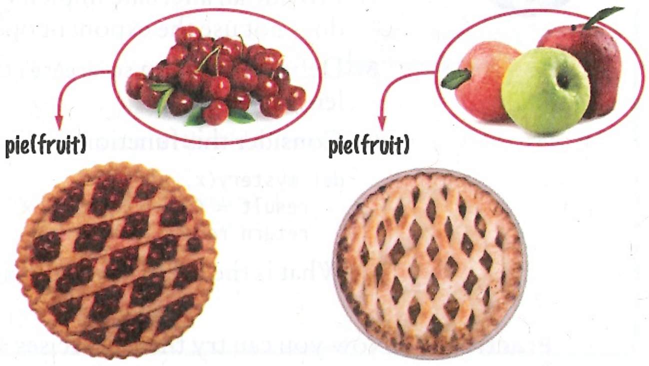
We don't use this style, but many Python programmers do.

Note that the function comment does not document the implementation (*how* the function does what it does) but rather the design (*what* the function does, its inputs, and its results). The comment allows other programmers to use the function as a "black box".

5.3 Parameter Passing

Parameter variables hold the arguments supplied in the function call.

In this section, we examine the mechanism of parameter passing more closely. When a function is called, variables are created for receiving the function's arguments. These variables are called **parameter variables**. (Another commonly used term is **formal parameters**.) The values that are supplied to the function when it is called are the **arguments** of the call. (These values are also commonly called the **actual parameters**.) Each parameter variable is initialized with the corresponding argument.



A recipe for a fruit pie may say to use any kind of fruit. Here, "fruit" is an example of a parameter variable. Apples and cherries are examples of arguments.