**Chapter 6 More on Programming**

**6.1 User-Defined Functions**

We have seen many built-in functions in Python. You can also write your own functions, which are called ***user-defined functions***. Related user-defined functions can be stored in user-defined modules, which can then be imported into programs for use.

When we write a function, we write the ***function definition***. The function can then be called, much like built-in functions.

The general form of a function definition is:

def functionname(parameters):

""" Documentation string. """

# Function body

The first line is called the ***function header***. It consists of the reserved word **def**, the name of the function, ***parameters*** (if any) in parentheses, followed by a colon. The rest of the function, which is indented, is the ***function body***. This typically begins with a ***documentation string*** (***docstring***), in triple quotes. The docstring should be sentence(s) describing what the function does, beginning with a capitalized word and ending with a period. After the docstring there are the statements in the body of the function. It is common to indent the body of the function 4 spaces.

The function is called by giving the name of the function, and passing arguments in parentheses that will correspond to the parameters in the function header:

functionname(arguments)

Sometimes the arguments in the function call are referred to as ***actual parameters*** and the parameters in the function header are referred to as ***formal parameters***.

Although this is somewhat arbitrary, we will begin with two types of functions:

* Functions that calculate and return a value
* Functions that accomplish a task

Two main reasons for writing our own functions are:

* To be able to reuse code by calling the function whenever it is needed
* To write ***modular programs***

If there is code that will be repeated at different times, perhaps using different values, it can be made into a function and then called whenever needed.

Modular programs are programs that consist of a series of functions that do the actual work. The functions are called by a ***main program***, which can be implemented as either a script or function.

**6.1.1 Functions with Return**

We have seen examples of built-in functions that calculate and return a value, such as the **round** function. User-defined functions can also accomplish this using the **return** statement.

As an example, we will write a function that returns the area of a square. In order to calculate the area of a square, the function needs the length of each side of the square, so the side length must be passed as an argument to the function. Here is a function definition for a function named *sqrarea* that accomplishes this:

*def sqrarea(sidelen):*

*"""This function calculates the area of a square."""*

*sqar = sidelen\*\*2*

*return sqar*

The function can then be called by using the name of the function and passing a number for the length of a side of the square:

*>>> sqrarea(3)*

9

In the function call, the side length, 3, was passed to the parameter *sidelen* in the function. The function then squared this, and ***returned*** the result. We say that when the function is called, ***control*** is sent to the function, and the function begins executing. The **return** statement not only returns the value (in this case, 9), but also sends control out of the function.

In general a docstring describes what a function does and can be displayed using the **help** function:

*>>> help(sqrarea)*

Help on function sqrarea in module \_\_main\_\_:

sqrarea(sidelen)

This function calculates the area of a square.

The variable *sqar*, which was used in the function body, was not necessary. The **return** statement could just return an expression:

*def sqrarea(sidelen):*

*"""This function calculates the area of a square."""*

*return sidelen\*\*2*

The value returned from the function would normally be stored in a variable, e.g.

*>>> mysquare = sqrarea(3)*

*>>> mysquare*

9

In this example, one argument was passed to the function. For a function to calculate the area of a rectangle, both the length and width of the rectangle must be passed to the function, so there will be two parameters in the function header and two arguments passed in the function call:

*def rectarea(rlen, rwid):*

*"""This function calculates the area of a rectangle."""*

*return rlen \* rwid*

*>>> rectarea(2, 4)*

8

The values of the arguments are passed to the corresponding parameters in the function, so the first argument, 2, is passed to the first parameter *rlen*, and the second argument 4 is passed to the second parameter *rwid*.

Notice that the **return** statement returned the result of an expression; an intermediate variable could be used but is not necessary.

In Python, the **return** statement can only return one object. If it is desired to have a function return more than one thing, they can be stored in one data structure (such as a **tuple** or **list**), and that can be returned. In the following example, a function calculates and returns both the area and the perimeter of a square, by storing both in a list.

*def sqrarea\_perim(sidelen):*

*"""Calculates the area and perimeter of a square."""*

*sqar = sidelen\*\*2*

*sqperim = 4 \* sidelen*

*results = [sqar, sqperim]*

*return results*

*>>> sqrarea\_perim(11)*

[121, 44]

Using a tuple instead of a list is perhaps easier to understand, and it is easier to use the results.

*def sqrarea\_perim(sidelen):*

*"""Calculates the area and perimeter of a square."""*

*sqar = sidelen\*\*2*

*sqperim = 4 \* sidelen*

*return sqar, sqperim*

*>>> area, perim = sqrarea\_perim(11)*

*>>> print(area, perim)*

121 44

**6.1.2 Functions with no Return**

In many cases functions calculate and return values, but in some cases functions just accomplish a task, such as printing.

For example, the following function receives as parameters a string and an integer n, and prints the string n times in a row on one line. After that, it moves the cursor down.

*def printstrs(instr, n):*

*"""This function prints instr n times."""*

*for i in range(n):*

*print(instr,end='')*

*print()*

Here are two examples of calling the function:

*>>> printstrs('x',3)*

xxx

*>>> printstrs("Hi", 5)*

HiHiHiHiHi

Notice that the function does not have a **return** statement, since it is not calculating anything. However, all functions return something regardless of whether there is a **return** statement or not. The default value that is returned is a special value **None**, as we can see by passing a call to the function *printstrs* to the **print** function. In the following example, ‘aa’ and then the newline gets printed by the function, and then the result returned by the function call, **None**, is printed once the function stops executing and returns control.

*>>> print(printstrs('a',2))*

aa

None

The *printstrs* function prints the line ‘aa’ and returns **None**. The **print** function then prints the returned value, **None**. This is equivalent to:

*>>> x = printstrs('a',2)*

*>>> print(x)*

It is not always necessary to pass arguments to functions (whether they explicitly return something or not). For example, we may wish to have a function that just prints a set of instructions.

*def printinstruct():*

*"""This function just prints stuff."""*

*print('Please sit up')*

*print('Do not slouch!')*

*print('Chew with your mouth closed')*

*>>> printinstruct()*

Please sit up

Do not slouch!

Chew with your mouth closed

A function like this that just prints helps to facilitate modular programs. Of course, the instructions printed by a function would normally refer to actions that the user must take when running a program!

**6.1.3 Scope**

The ***scope*** of a variable or object is where that object is valid. For functions, everything that is defined in the function is valid only in that function. We say that variables defined in a function are ***local*** to that function, meaning that their scope is the function. Formally, what happens is that every function has its own ***symbol table***, which contains the names of the variables and their values. Once a function is called and control is sent to the function, its symbol table is created. The function’s symbol table only exists while the function is executing.

For example, for the *sqarea* function, the symbol table contains the local variable *sqar*, as well as the parameter *sidelen*.

*def sqrarea(sidelen):*

*"""This function calculates the area of a square."""*

*sqar = sidelen\*\*2*

*return sqar*

After the function is called, attempting to reference either *sqar* or *sidelen* will result in an error. This is because after the function stops executing, the symbol table, the variable *sqar*, and the parameter *sidelen* no longer exist.

*>>> area = sqrarea(4)*

*>>> print(sqar)*

NameError: name 'sqar' is not defined

**6.1.4 Lambda Functions**

***Anonymous functions***, called ***lambda functions*** in Python, are very short one line functions that do not require a formal function definition using **def**. Lambda functions are created using the keyword **lambda**. The general form is:

lambda argument(s): expression

The function begins with the reserved word **lambda**, then argument(s) followed by a colon and then an expression that uses the arguments; the result of the expression is returned.

If the lambda function is stored in a variable, the variable can be used to call the function. For example,

*>>> times3 = lambda num: num \* 3*

*>>> times3(11)*

33

One advantage of a lambda function is that it is shorter than a function definition and does not require the header with **def** or the **return** statement. A disadvantage is that the “body” of the function is confined to one simple expression.