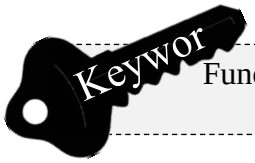
Chapter 4: Functions

What we will learn:

- ✓ Definition of a function
- ✓ Syntax of functions
- ✓ Return values from a function

What you need to know before:

- ✓ Variables
- ✓ Expressions
- ✓ Statements
- ✓ Values



Function Return

4.1 Definition

Functions are blocks of codes that perform specific tasks. It allows us to reuse it and carry out a set of computations more than once. Functions take an input and can return an output. They can also take in multiple inputs or no input, have multiple outputs or no outputs at all. Python provides a lot of useful functions for us, such as `print()`, `type()`, `abs()`, these functions are called built-in because they are supplied as part of the language. We can declare and define our own function to do what we want them to do. Functions that are defined by a user are called user-defined functions. This chapter will focus on user-defined functions.

4.1.1 Why Do We Use Functions?

- Using functions makes our program code more readable. When we define a function, we are actually naming certain sets of instruction that are performing a specific task and therefore making our program code easier to read.
- Functions can reduce our program code significantly by putting repetitive code into a function.
- Change or adjustments can be made in one place.
- Functions facilitate modular design. We can break down a large problem into manageable chunks, work on these chunks in functions then put the entire working piece together for a working solution.
- Functions can be placed into modules so that they can be reused.

4.1.2 Function Syntax

Syntax:

```
def <name>(<ParameterList>):
    <block>
    return <expression>, <expression>
```

Where *<name>* follows the same conventions as a variable.

<ParameterList> represents the inputs to the function separated by comma. The return statement is used to specify the output. By default, all functions return a value. If one is not specified using the return keyword then *None* is automatically returned.

return <expression>, <expression>, ...

Hint: Functions must begin with a letter the alphabet and cannot contain a space.

Example:

```
>>> def sum(a, b):
        return a+b

>>> sum(12, 10)
22
>>>
```

Whenever we express what the function should do we call this **defining a function**. The function in the example is defining the function sum. The `a` and `b` are referred to as **parameters** and are only valid variables within the definition of the function; the range of the area that the variable is valid is called the **scope of the variable**. Note that `a` and `b` are place holders and are only valid within the definition.

Whenever we use a function, this is referred to as **calling a function**. In the example above, `sum(12, 10)` is a call to the function with arguments 12 and 10.

4.1.3 Returning Values

We can define functions that just do something without explicitly returning a value; for example, the following function will take a name and say hello twice.

```
>>> def hello_twice(name):
        print("Hello "+name)
        print("Hello "+name)


>>> print(hello_twice("Ahmed"))
Hello Ahmed
Hello Ahmed
None
>>>
```

Notice that after calling the function, `None` is printed as the returned value. It turns out that because we did not use the return keyword in the function definition `None` is returned automatically.

4.2 Parameters and Arguments

Some books will use parameters and arguments interchangeably; however, there is a difference. It turns out that parameters are the place holders that are used in defining the function, whereas the arguments are the actual values that are supplied during the function call. Parameters are specific to function definition; in fact the parameters are data types (defines acceptable range and operations) whereas an argument is an instance of the parameter and is used in the function call. Notice that the argument can change with each function call, and is normally executed at runtime.

Python also uses a technique called operator overloading, where operators such as the + can be defined to perform different operations based on the operands that surround it. Since + is defined for strings also (namely concatenation), the following will result in sum concatenating the strings that it receives.

Hint: In *sum(a,b)*, *a* and *b are the* parameters while *sum(4,5)*, 4 and 5 are arguments.

```
>>> sum('Hello', ' Bobby')
'Hello Bobby'
```

This is a very powerful concept in object-oriented programming and Python, being an object-oriented language, utilises these features.

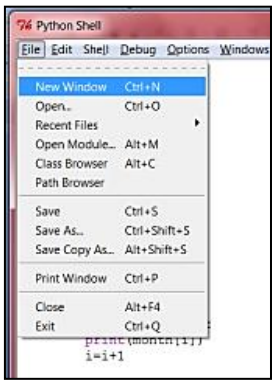
4.3 Saving Your Program Code

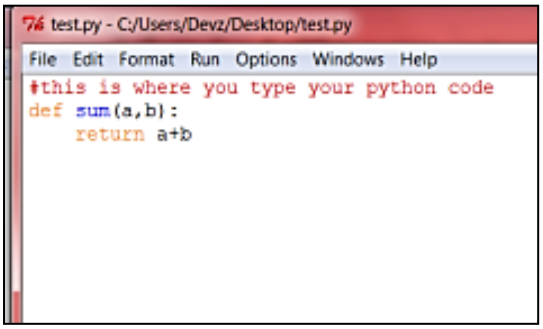
Python can operate in two modes, namely the interactive mode and the script mode. You know that you are in script mode if you have the three chevrons (>>>) and the blinking cursor. In this mode, you type and at the end of the statement Python interprets and produces a result. All of the examples so far have been in interactive mode. You can create a Python file, type all your program code here and then interpret this file. This allows us to make changes and rerun the script. The instructions below will demonstrate how to create a script file.

From the Python interpreter, you can *select file → new window* to start a new window that you can type your program code into.

Save this file with a suitable file name and *.py* as the file extension; this will tell the interpreter that this file is a Python file. Once you finish typing your program code, you can execute your program by selecting *run → run module*. Your output will be displayed in the Python interpreter.

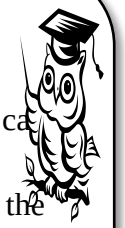
Example: Creating a new script file.





After typing your script you can run it and make changes.

Hint: If we need to add other numbers we simply call the sum function. Python is very dynamic and can reuse the function with strings. For example, calling the sum function with two strings will return both strings concatenated as + is defined for strings.



4.4 Program Flow

A program is executed from top to bottom and line by line unless the flow is diverted by some control structure (we will examine control structures in Chapter 6). It turns out that functions do not affect the program flow. When a function is called, the execution jumps to the definition of the program and replaces the argument(s) with the parameter(s) (recall that the parameters are place holders whereas arguments are the actual values that are sent to the program). The body of the program is executed then execution continues after the line that calls the function.

This process seems pretty straightforward, but when you imagine that a function can call another function who in turn can call other functions, keeping track of where you are can be quite complex. Further, in Python you can send a function as an argument to another function. It turns out that Python has a very efficient method of handling this situation. Here are two simple examples that should demonstrate this point, but you can imagine that these could be more complex calculations.

Imagine that we have a function called "larger" that returns the larger of two given numbers, then:

```
>>> x = 12
>>> y = 7
>>> z = 79
>>> print(larger(larger(x,y),z))
79
>>>
```

This will print the largest of the three numbers. In the example above, you can see that the inner `larger(x,y)` was an argument to the outer larger function. The outer larger function would execute an instance of the larger function returning the value 12, as 12 is larger than 7; then the outer larger would evaluate `larger(12,79)` and finally return 79 to the print function.