Functions

Functions are reusable pieces of programs. They allow you to give a name to a block of statements, allowing you to run that block using the specified name anywhere in your program and any number of times. This is known as *calling* the function. We have already used many built-in functions such as `len` and `range`.

The function concept is probably *the* most important building block of any non-trivial software (in any programming language), so we will explore various aspects of functions in this chapter.

Functions are defined using the `def` keyword. After this keyword comes an *identifier* name for the function, followed by a pair of parentheses which may enclose some names of variables, and by the final colon that ends the line. Next follows the block of statements that are part of this function. An example will show that this is actually very simple:

Example (save as `function1.py`):

```
def say_hello():
    # block belonging to the function
    print('hello world')
# End of function

say_hello()  # call the function
say_hello()  # call the function again
```

Output:

```
$ python function1.py
hello world
hello world
```

How It Works

We define a function called `say_hello` using the syntax as explained above. This function takes no parameters and hence there are no variables declared in the parentheses. Parameters to functions are just input to the function so that we can pass in different values to it and get back corresponding results.

Notice that we can call the same function twice which means we do not have to write the same code again.

Function Parameters

A function can take parameters, which are values you supply to the function so that the function can *do* something utilising those values. These parameters are just like variables except that the values of these variables are defined when we call the function and are already assigned values when the function runs.

Parameters are specified within the pair of parentheses in the function definition, separated by commas. When we call the function, we supply the values in the same way. Note the terminology used - the names given in the function definition are called *parameters* whereas the values you supply in the function call are called *arguments*.

Example (save as `function_param.py`):

```
def print_max(a, b):
    if a > b:
        print(a, 'is maximum')
    elif a == b:
        print(a, 'is equal to', b)
    else:
        print(b, 'is maximum')

# directly pass literal values
print_max(3, 4)

x = 5
y = 7

# pass variables as arguments
print_max(x, y)
```

Output:

```
$ python function_param.py
4 is maximum
7 is maximum
```

How It Works

Here, we define a function called `print_max` that uses two parameters called `a` and `b`. We find out the greater number using a simple `if..else` statement and then print the bigger number.

The first time we call the function `print_max`, we directly supply the numbers as arguments. In the second case, we call the function with variables as arguments. `print_max(x, y)` causes the value of argument `x` to be assigned to parameter `a` and the value of

argument `y` to be assigned to parameter `b`. The `print_max` function works the same way in both cases.

Local Variables

When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function - i.e. variable names are *local* to the function. This is called the *scope* of the variable. All variables have the scope of the block they are declared in starting from the point of definition of the name.

Example (save as `function_local.py`):

```
x = 50


def func(x):
    print('x is', x)
    x = 2
    print('Changed local x to', x)


func(x)
print('x is still', x)
```

Output:

```
$ python function_local.py
x is 50
Changed local x to 2
x is still 50
```

How It Works

The first time that we print the *value* of the name *x* with the first line in the function's body, Python uses the value of the parameter declared in the main block, above the function definition.

Next, we assign the value `2` to `x`. The name `x` is local to our function. So, when we change the value of `x` in the function, the `x` defined in the main block remains unaffected.

With the last `print` statement, we display the value of `x` as defined in the main block, thereby confirming that it is actually unaffected by the local assignment within the previously called function.

The `global` statement

If you want to assign a value to a name defined at the top level of the program (i.e. not inside any kind of scope such as functions or classes), then you have to tell Python that the name is not local, but it is *global*. We do this using the `global` statement. It is impossible to assign a value to a variable defined outside a function without the `global` statement.

You can use the values of such variables defined outside the function (assuming there is no variable with the same name within the function). However, this is not encouraged and should be avoided since it becomes unclear to the reader of the program as to where that variable's definition is. Using the `global` statement makes it amply clear that the variable is defined in an outermost block.

Example (save as `function_global.py`):

```
x = 50


def func():
    global x

    print('x is', x)
    x = 2
    print('Changed global x to', x)


func()
print('Value of x is', x)
```

Output:

```
$ python function_global.py
x is 50
Changed global x to 2
Value of x is 2
```

How It Works

The `global` statement is used to declare that `x` is a global variable - hence, when we assign a value to `x` inside the function, that change is reflected when we use the value of `x` in the main block.

You can specify more than one global variable using the same `global` statement e.g. `global x, y, z`.

Default Argument Values

For some functions, you may want to make some parameters *optional* and use default values in case the user does not want to provide values for them. This is done with the help of default argument values. You can specify default argument values for parameters by appending to the parameter name in the function definition the assignment operator (`=`) followed by the default value.

Note that the default argument value should be a constant. More precisely, the default argument value should be immutable - this is explained in detail in later chapters. For now, just remember this.

Example (save as `function_default.py`):

```
def say(message, times=1):
    print(message * times)

say('Hello')
say('World', 5)
```

Output:

```
$ python function_default.py
Hello
WorldWorldWorldWorldWorld
```

How It Works

The function named `say` is used to print a string as many times as specified. If we don't supply a value, then by default, the string is printed just once. We achieve this by specifying a default argument value of `1` to the parameter `times` .

In the first usage of `say` , we supply only the string and it prints the string once. In the second usage of `say` , we supply both the string and an argument `5` stating that we want to *say* the string message 5 times.

> *CAUTION*
>
> Only those parameters which are at the end of the parameter list can be given default argument values i.e. you cannot have a parameter with a default argument value preceding a parameter without a default argument value in the function's parameter list.
>
> This is because the values are assigned to the parameters by position. For example, `def func(a, b=5)` is valid, but `def func(a=5, b)` is *not valid*.

Keyword Arguments

If you have some functions with many parameters and you want to specify only some of them, then you can give values for such parameters by naming them - this is called *keyword arguments* - we use the name (keyword) instead of the position (which we have been using all along) to specify the arguments to the function.

There are two advantages - one, using the function is easier since we do not need to worry about the order of the arguments. Two, we can give values to only those parameters to which we want to, provided that the other parameters have default argument values.

Example (save as `function_keyword.py`):

```
def func(a, b=5, c=10):
    print('a is', a, 'and b is', b, 'and c is', c)

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Output:

```
$ python function_keyword.py
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

How It Works

The function named `func` has one parameter without a default argument value, followed by two parameters with default argument values.

In the first usage, `func(3, 7)`, the parameter `a` gets the value `3`, the parameter `b` gets the value `7` and `c` gets the default value of `10`.

In the second usage `func(25, c=24)`, the variable `a` gets the value of 25 due to the position of the argument. Then, the parameter `c` gets the value of `24` due to naming i.e. keyword arguments. The variable `b` gets the default value of `5`.

In the third usage `func(c=50, a=100)`, we use keyword arguments for all specified values. Notice that we are specifying the value for parameter `c` before that for `a` even though `a` is defined before `c` in the function definition.

VarArgs parameters

Sometimes you might want to define a function that can take *any* number of parameters, i.e. **var**iable number of **arg**uments, this can be achieved by using the stars (save as `function_varargs.py`):

```
def total(a=5, *numbers, **phonebook):
    print('a', a)

    #iterate through all the items in tuple
    for single_item in numbers:
        print('single_item', single_item)

    #iterate through all the items in dictionary
    for first_part, second_part in phonebook.items():
        print(first_part,second_part)

total(10,1,2,3,Jack=1123,John=2231,Inge=1560)
```

Output:

```
$ python function_varargs.py
a 10
single_item 1
single_item 2
single_item 3
Inge 1560
John 2231
Jack 1123
```

How It Works

When we declare a starred parameter such as `*param`, then all the positional arguments from that point till the end are collected as a tuple called 'param'.

Similarly, when we declare a double-starred parameter such as `**param`, then all the keyword arguments from that point till the end are collected as a dictionary called 'param'.

We will explore tuples and dictionaries in a later chapter.

The `return` statement

The `return` statement is used to *return* from a function i.e. break out of the function. We can optionally *return a value* from the function as well.

Example (save as `function_return.py`):

```
def maximum(x, y):
    if x > y:
        return x
    elif x == y:
        return 'The numbers are equal'
    else:
        return y

print(maximum(2, 3))
```

Output:

```
$ python function_return.py
3
```

How It Works

The `maximum` function returns the maximum of the parameters, in this case the numbers supplied to the function. It uses a simple `if..else` statement to find the greater value and then *returns* that value.

Note that a `return` statement without a value is equivalent to `return None`. `None` is a special type in Python that represents nothingness. For example, it is used to indicate that a variable has no value if it has a value of `None`.

Every function implicitly contains a `return None` statement at the end unless you have written your own `return` statement. You can see this by running `print(some_function())` where the function `some_function` does not use the `return` statement such as:

```
def some_function():
    pass
```

The `pass` statement is used in Python to indicate an empty block of statements.

> TIP: There is a built-in function called `max` that already implements the 'find maximum' functionality, so use this built-in function whenever possible.

DocStrings

Python has a nifty feature called *documentation strings*, usually referred to by its shorter name *docstrings*. DocStrings are an important tool that you should make use of since it helps to document the program better and makes it easier to understand. Amazingly, we can even get the docstring back from, say a function, when the program is actually running!

Example (save as `function_docstring.py`):

```
def print_max(x, y):
    '''Prints the maximum of two numbers.

    The two values must be integers.'''
    # convert to integers, if possible
    x = int(x)
    y = int(y)

    if x > y:
        print(x, 'is maximum')
    else:
        print(y, 'is maximum')

print_max(3, 5)
print(print_max.__doc__)
```

Output:

```
$ python function_docstring.py
5 is maximum
Prints the maximum of two numbers.

    The two values must be integers.
```

How It Works

A string on the first logical line of a function is the *docstring* for that function. Note that DocStrings also apply to modules and classes which we will learn about in the respective chapters.

The convention followed for a docstring is a multi-line string where the first line starts with a capital letter and ends with a dot. Then the second line is blank followed by any detailed explanation starting from the third line. You are *strongly advised* to follow this convention for all your docstrings for all your non-trivial functions.

We can access the docstring of the `print_max` function using the `__doc__` (notice the *double underscores*) attribute (name belonging to) of the function. Just remember that Python treats *everything* as an object and this includes functions. We'll learn more about objects in the chapter on classes.

If you have used `help()` in Python, then you have already seen the usage of docstrings! What it does is just fetch the `__doc__` attribute of that function and displays it in a neat manner for you. You can try it out on the function above - just include `help(print_max)` in your program. Remember to press the `q` key to exit `help`.

Automated tools can retrieve the documentation from your program in this manner. Therefore, I *strongly recommend* that you use docstrings for any non-trivial function that you write. The `pydoc` command that comes with your Python distribution works similarly to `help()` using docstrings.

Summary

We have seen so many aspects of functions but note that we still haven't covered all aspects of them. However, we have already covered most of what you'll use regarding Python functions on an everyday basis.

Next, we will see how to use as well as create Python modules.