Chapter 12
Functions in Python

12.1 Introduction

As discussed in the last chapter, when you build an application of any size you will want to break it down into more manageable units, these units can then be worked on separately, tested and maintained separately. One way in which these units can be defined is as Python functions.

This chapter will introduce Python functions, how they are defined, how they can be referenced and executed. It also considers how parameters work in Python functions and how values can be returned from functions. It also introduces lambda or anonymous functions.

12.2 What are Functions?

In Python functions are groups of related statements that can be called together, that typically perform a specific task and which may or may not take a set of parameters or return a value.

Functions can be defined in one place and called or invoked in another. This helps to make code more modular and easier to understand.

It also means that the same function can be called multiple times or in multiple locations. This helps to ensure that although a piece of functionality is used in multiple places, it is only defined once and only needs to be maintained and tested in one location.

© Springer Nature Switzerland AG 2023

J. Hunt, *A Beginners Guide to Python 3 Programming*, Undergraduate Topics
in Computer Science, https://doi.org/10.1007/978-3-031-35122-8_12

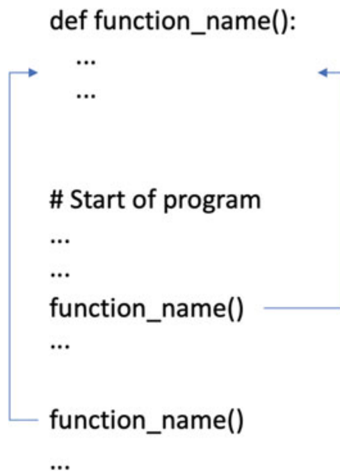
12.3 How Functions Work

We have said what they are and a little bit about why they might be good but not really how they work.

When a function is called (or invoked) the flow of control a program jumps from where the function was called to the point where the function was defined. The body of the function is then executed before control returns back to where it was called from.

As part of this process, all the values that were in place when the function was called are stored away (on something called the stack) so that if the function defines its own versions, they do not overwrite each other.

The invocation of a function is illustrated below:



Each time when the call is made to `function_name()` the program flow jumps to the body of the function and executes the statements there. Once the function finishes it returns to the point at which the function was called.

In the above this happens twice as the function is called at two separate points in the program.

12.4 Types of Functions

Technically speaking there are two types of functions in Python: *built-in* functions and *user-defined* functions.

Built-in functions are those provided by the language, and we have seen several of these already. For example, both `print()` and `input()` are built-in functions. We did not need to define them ourselves as they are provided by Python.

In contrast *user-defined* functions are those written by developers. We will be defining user-defined functions in the rest of this chapter, and it is likely that in many cases, most of the programs that you will write will include user-defined functions of one sort or another.

12.5 Defining Functions

The basic syntax of a function is illustrated below:

```
def function_name(parameter list):
    """docstring"""
    statement
    statement(s)
```

This illustrates several things:

1. All (named) functions are defined using the *keyword* def; this indicates the start of a function definition. A keyword is a part of the syntax of the Python language and cannot be redefined and is not a function.
2. A function can have a name which uniquely identifies it; you can also have anonymous functions, but we will leave those until later in this chapter.
3. The naming conventions that we have been adopting for variables are also applied to functions, and they are all lowercase with the different elements of the function name separated by '_'.
4. A function can (optionally) have a list of parameters which allow data to be passed into the function. These are optional as not all functions need to be supplied with parameters.
5. A colon is used to mark the end of the *function header* and the start of the *function body*. The function header defines the signature of the function (what is called and the parameters it takes). The function body defines what the function does.
6. An optional documentation string (the `docstring`) can be provided that describes what the function does. We typically use the triple double quote string format as this allows the documentation string to go over multiple lines if required.
7. One or more Python statements make up the function body. These are indented relative to the function definition. All lines that are indented are part of the function until a line which is intended at the same level as the `def` line.
8. It is common to use 4 spaces (not a tab) to determine how much to indent the body of a function by.

12.5.1 An Example Function

The following is one of the simplest functions you can write; it takes no parameters and has only a single statement that prints out the message 'Hello World':

```
def print_msg():
    print('Hello World!')
```

This function is called print_msg, and when called (also known as invoked) it will run the body of the function which will print out the string, for example,

```
print_msg()
```

will generate the output

```
Hello World!
```

Be careful to include the round brackets () when you call the function. This is because if you just use the function's name then you are merely referring to the location in memory where the function is stored, and you are not invoking it.

We could modify the function to make it a little more general and reusable by providing a parameter. This parameter could be used to supply the message to be printed out, for example:

```
def print_my_msg(msg):
    print(msg)
```

Now the print_my_msg function takes a single parameter, and this parameter becomes a variable which is available within the body of the function. However, this parameter only exists within the body of the function; it is not available outside of the function.

This now means that we can call the print_my_msg function with a variety of different messages:

```
print_my_msg('Hello World')
print_my_msg('Good day')
print_my_msg('Welcome')
print_my_msg('Ola')
```

The output from calling this function with each of these strings being supplied as the parameter is:

```
Hello World
Good day
Welcome
Ola
```

12.6 Returning Values from Functions

It is very common to want to return a value from a function. In Python this can be done using the `return` statement. Whenever a `return` statement is encountered within a function then that function will terminate and return any values following the return keyword.

This means that if a value is provided, then it will be made available to any calling code.

For example, the following defines a simple function that squares whatever value has been passed to it:

```
def square(n):
    return n * n
```

When we call this function, it will multiply whatever it is given by itself and then return that value. The returned value can then be used at the point that the function was invoked, for example:

```
# Store result from square in a variable
result = square(4)
print(result)
# Send the result from square immediately to another function
print(square(5))
# Use the result returned from square in a conditional expression
if square(3) < 15:
   print('Still less than 15')
```

When this code is run, we get:

```
16
25
Still less than 15
```

It is also possible to return multiple values from a function; for example, in this `swap` function the order in which the parameters are supplied is swapped when they are returned:

```
def swap(a, b):
    return b, a
```

We can then assign the values returned to variables at the point when the function is called:

```
a = 2
b = 3
x, y = swap(a, b)
print(x, ',', y)
```

which produces

```
3 , 2
```

In actual fact the result returned from the swap function is what is called a *tuple* which is a simple way to grouping data together. This means that we could also have written:

```
z = swap(a, b)
print(z)
```

which would have printed the tuple out:

```
(3, 2)
```

We will look at tuples more when we consider collections of data.

12.7 Docstring

So far our example functions have not included any documentation strings (the *docstring*). This is because the docstring is *optional,* and the functions we have written have been very simple.

However, as functions become more complex and may have multiple parameters the documentation provided can become more important.

The *docstring* allows the function to provide some guidance on what is expected in terms of the data passed into the parameters, potentially what will happen if the data is incorrect, as well as what the purpose of the function is in the first place.

In the following example, the *docstring* is being used to explain the behaviour of the function and how the parameter is used.

```
def get_integer_input(message):
    """
    This function will display the message to the user
    and request that they input an integer.

    If the user enters something that is not a number
    then the input will be rejected
    and an error message will be displayed.

    The user will then be asked to try again."""

    value_as_string = input(message)
    while not value_as_string.isnumeric():
        print('The input must be an integer')
        value_as_string = input(message)

    return int(value_as_string)
```

When used, this method will guarantee that a valid integer will be returned to the calling code:

```
age = get_integer_input('Please input your age: ')
print('age is', age)
```

An example of what happens when this is run is given below:

```
Please input your age: John
The input must be an integer
Please input your age: 21
age is 21
```

The *docstring* can be read directly from the code but is also available to IDEs such as PyCharm so that they can provide information about the function. It is even available to the programmer via a very special property of the function called `__doc__` that is accessible via the name of the function using the dot notation:

```
print(get_integer_input.__doc__)
```

which generates

```
    This function will display the message to the user
    and request that they input an integer.

    If the user enters something that is not a number
    then the input will be rejected
    and an error message will be displayed.

    The user will then be asked to try again.
```

12.8 ReStructured Text

It is also possible to place *formatting* commands within a docstring that can be picked up by a tool such as PyCharm to improve the layout and information presented. There are several options available for this such as Google style docstrings and NumPy style docstrings (see https://betterprogramming.pub/3-different-docstring-formats-for-python-d27be81e0d68). However we are going to look at reStructured Text style docstrings as this is the style recommended by the Python organization itself.

ReStructured Text (aka ReST) is intended to be an easy-to-read markup syntax used to add additional meaning to the text within a docstring. Like many other simple markup languages it uses inline markup such as '*' or a '#' to indicate emphasis or lists, etc. and as such is intended to be much simpler and easier to use than more complex markup languages such as HTML.

As a simple example, consider the following code:

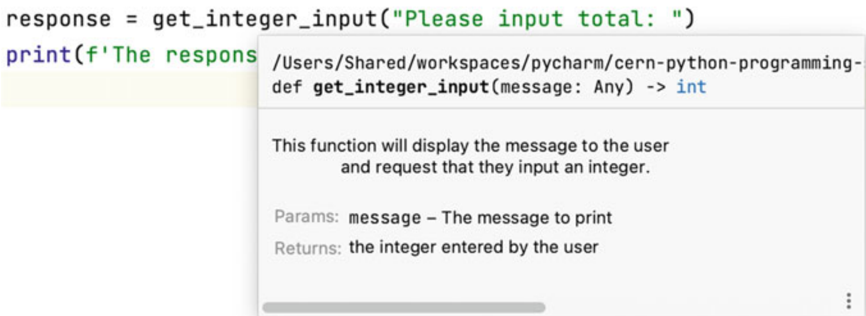
```
def get_integer_input(message):
    """
    This function will display the message to the user
    and request that they input an integer.

    :param message: The message to print
    :returns: the integer entered by the user

    """
    value_as_string = input(message)
    while not value_as_string.isnumeric():
        print('The input must be an integer')
        value_as_string = input(message)
    return int(value_as_string)
```

This function has a docstring containing some reStructured Text formatting, namely the `:param` and `:returns` directives.

A directive is an item of *meta* information that will be used by the ReST parser to generate the formatted output.

- In this case the `:param` directive indicates that `message` is a parameter to the function and the information after the final : provides a description of that parameter.
- In turn the `:returns:` directive indicates that this function returns a value and the text after this described the meaning of the returned value.

PyCharm uses this information and generates reference documentation that can be viewed in the pop-up displayed when the programmer hovers over a call to the `get_integer_input` function. This is displayed below:



As you can see the `params` and `returns` directives are displayed in such a way as to make it easy to read and understand the information provided.

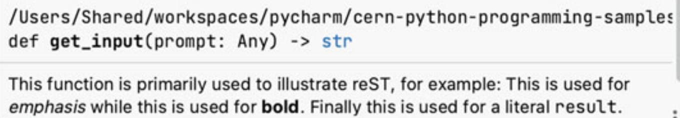
It is also possible to use ReST markup to emphasize text or to make the text bold or to indicate that a value is a literal:

- Emphasis is indicated using an asterisk '*', for example: *emphasis*.
- Bold is indicated using a double asterisk '**', for example: **strong**.
- Literal values, variables and small code samples can be indicated using two back quotes, for example: "literal".

For example:

```
def get_input(prompt):
      """This function is primarily used to illustrate ReST, for
example:
     This is used for *emphasis* while this is used for **bold**.
Finally
    this is used for a literal "result".
    """
    result = input(prompt)
    return result
```

This is rendered by PyCharm as shown below:



Note that if asterisks or back quotes appear within the text and could be confused with inline markup delimiters, they should be escaped with a backslash, for example: *.

There are a few additional restrictions on this markup that you should be aware of, including:

- The markup cannot be nested; that is, you cannot nest a bold element within an emphasis element.
- The content may not start or end with whitespace: * text* is wrong, as is *text *.
- It must be separated from surrounding text by non-word characters. It is thus necessary to use a backslash escaped space to work around that for example: thisis\ *one*\ word.

It is also possible to create lists within ReST, bulleted lists are represented by placing an asterisk at the start of a line, and numbered lists are represented by a '#.'; it is also possible to explicitly number a list. Examples of these are shown below:

```
* This is a bulleted list.
* It has two items, the second
  item uses two lines.

1. This is a numbered list.
2. It has two items too.

#. This is a numbered list.
#. It has two items too.
```

For example, when used in a function docstring:

```
def get_more_input(prompt):
    """This function is used to illustrate lists, for example:

    * This is a bulleted list.
    * It has two items, the second item uses two lines.

    1. This is a numbered list.
    2. It has two items too.

    #. This is a numbered list.
    #. It has two items as well.

    """
    result = input(prompt)
    return result
```

Nested lists are possible, but be aware that they must be separated from the parent list items by blank lines, for example:

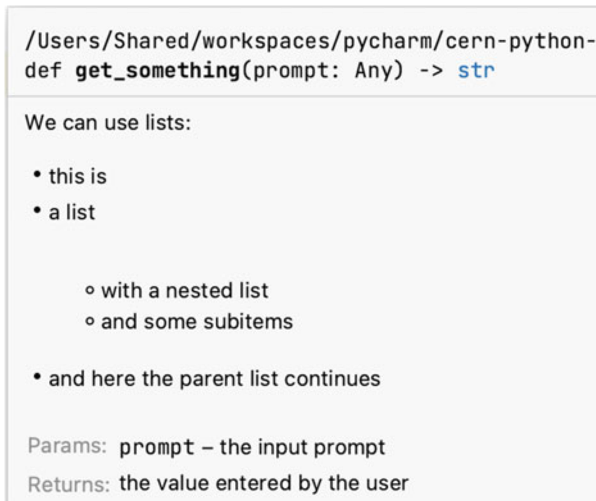
```
* this is
* a list

  * with a nested list
  * and some subitems

* and here the parent list continues
```

As an example, lists and sub lists are used below within a function docstring that also uses a couple of directives for the parameter and the return value.

```
def get_something(prompt):
    """
```

```
    We can use lists:

    * this is
    * a list

        * with a nested list
        * and some subitems

    * and here the parent list continues

    :param prompt: the input prompt
    :return: the value entered by the user
    """
    result = input(prompt)
    return result
```

This is illustrated below:



Section headers are created by underlining the section title with a punctuation character, at least as long as the text:

```
This is a heading
=================
```

Finally, if you want to embed some source code as an example within a docstring you can use the special marker ::. This creates a literal block that will not be processed as formatted text but must be indented and with a gap following the ::. For example:

```
def get_another_thing(prompt):
    """
```

```
    To use this function see the code block::

        result = get_another_thing("please input data: ")
        print(result)

    This is back to normal text
    """
    result = input(prompt)
    return result
```

PyCharm renders this as:

```
/Users/Shared/workspaces/pycharm/cern-python-programming
def get_another_thing(prompt: Any) -> str

To use this function see the code block:

result = get_another_thing("please input data: ")
print(result)

This is back to normal text
```

12.9 Function Parameters

Before we go any further it is worth clarifying some terminology associated with passing data into functions. This terminology relates to the parameters defined as part of the function header and the data passed into the function via these parameters:

- A *parameter* is a variable defined as part of the function header and is used to make data available within the function itself.
- An *argument* is the actual value or data passed into the function when it is called. The data will be held within the parameters.

Unfortunately many developers use these terms interchangeably, but it is worth being clear on the distinction.

12.9.1 Multiple Parameter Functions

So far the functions we have defined have only had zero or one parameter; however that was just a choice. We could easily have defined a function which defined two or more parameters. In these situations, the parameter list contains a list of parameter names separated by a comma.

For example,

```
def greeter(name, message):
    print('Welcome', name, '-', message)

greeter('Eloise', 'Hope you like Rugby')
```

Here the greeter function taken defines two parameters: name and message. These parameters (which are local to the function and cannot be seen outside of the function) are then used within the body of the function.

The output is

```
Welcome Eloise - Hope you like Rugby
```

You can have any number of parameters defined in a function (prior to Python 3.7 there was a limit of 256 parameters—although if you have this many then probably you have a major problem with the design of your function—however this limit has now gone).

12.9.2 Default Parameter Values

Once you have one or more parameters you may want to provide *default* values for some or all of those parameters, particularly for ones which might not be used in most cases.

This can be done very easily in Python; all that is required is that the default value must be declared in the function header along with the parameter name.

If a value is supplied for the parameter, then it will override the default. If no value is supplied when the function is called, then the default will be used.

For example, we can modify the `greeter()` function from the previous section to provide a default message such as 'Live Long and Prosper'.

```
def greeter(name, message = 'Live Long and Prosper'):
    print('Welcome', name, '-', message)

greeter('Eloise')
greeter('Eloise', 'Hope you like Rugby')
```

Now we can call the `greeter()` function with one or two arguments.

When we run this example, we will get:

```
Welcome Eloise - Live Long and Prosper
Welcome Eloise - Hope you like Rugby
```

As you can see from this in the first example (where only one argument was provided) the default message was used. However, in the second example where a message was provided, along with the name, then that message was used instead of the default.

Note we can use the terms *mandatory* and *optional* for the parameters in `greeter()`. In this case

- name is a mandatory field/parameter.
- message is an optional field/parameter (as it has a default value).

One subtle point to note is that any number of parameters in a function's parameter list can have a default value; however once one parameter has a default value all remaining parameters to the right of that parameter must also have default values. For example, we could *not* define the greeter function as

```
def greeter(message = 'Live Long and Prosper', name):
    print('Welcome', name, '-', message)
```

As this would generate an *error* indicating that name must have a default value as it comes after (to the right) of a parameter with a default value.

12.9.3 Named Arguments

So far we have relied on the position of a value to be used to determine which parameter that value is assigned to. In many cases this is the simplest and cleanest option.

However, if a function has several parameters, some of which have default values, it may become impossible to rely on using the position of a value to ensure it is given to the correct parameter (because we may want to use some of the default values instead).

For example, let us assume we have a function with four parameters

```
def greeter(name,
            title = 'Dr',
            prompt = 'Welcome',
            message = 'Live Long and Prosper'):
    print(prompt, title, name, '-', message)
```

This now raises the question how do we provide the `name` and the `message` arguments when we would like to have the default `title` and `prompt`?

The answer is to use *named* arguments (or *keyword* arguments). In this approach we provide the name of the parameter we want an argument/value to be assigned to; position is no longer relevant. For example:

```
greeter(message = 'We like Python', name = 'Lloyd')
```

In this example we are using the default values for `title` and `prompt` and have changed the order of `message` and `name`. This is completely legal and results in the following output:

```
Welcome Dr Lloyd - We like Python
```

We can actually mix *positional* and *named* arguments in Python, for example:

```
greeter('Lloyd', message = 'We like Python')
```

Here 'John' is bound to the `name` parameter as it is the first parameter, but 'We like Python' is bound to `message` parameter as it is a named argument.

However, you cannot place positional arguments after a named argument, so we cannot write:

```
greeter(name='John', 'We like Python')
```

As this will result in Python generating an error.

12.9.4 Arbitrary Arguments

In some cases, you do not know how many arguments will be supplied when a function is called. Python allows you to pass an arbitrary number of arguments into a function and then process those arguments inside the function.

To define a parameter list as being of arbitrary length, a parameter is marked with an asterisk (`*`). For example:

```
def greeter(*args):
    for name in args:
        print('Welcome', name)

greeter('John', 'Denise', 'Phoebe', 'Adam', 'Gryff', 'Jasmine')
```

This generates

```
Welcome John
Welcome Denise
Welcome Phoebe
Welcome Adam
Welcome Gryff
Welcome Jasmine
```

Note that this is another use of the `for` loop; but this time it is a sequence of strings rather than a sequence of integers that is being used.

12.9.5 *Positional and Keyword Arguments*

Some functions in Python are defined such that the arguments to the methods can either be provided using a variable number of positional or keyword arguments. Such functions have two arguments `*args` and `**kwargs` (for positional arguments and keyword arguments).

They are useful if you do not know exactly how many of either position or keyword arguments are going to be provided.

For example, the function `my_function` takes both a variable number of positional and keyword arguments:

```
def my_function(*args, **kwargs):
    for arg in args:
        print('arg:', arg)
    for key in kwargs.keys():
        print('key:', key, 'has value: ', kwargs[key])
```

This can be called with any number of arguments of either type:

```
my_function('John', 'Denise', daughter='Phoebe', son='Adam')
print('-' * 50)
my_function('Paul',  'Fiona',  son_number_one='Andrew',  son_
number_two='James', daughter='Joselyn')
```

which produces the output:

```
arg: John
arg: Denise
key: son has value: Adam
key: daughter has value: Phoebe
--------------------------------------------------
arg: Paul
arg: Fiona
key: son_number_one has value: Andrew
key: son_number_two has value: James
key: daughter has value: Joselyn
```

Also note that the keywords used for the arguments are not fixed.

You can also define methods that only use one of the `*args` and `**kwargs` depending on your requirements (as we saw with the `greeter()` function above), for example:

```
def named(**kwargs):
    for key in kwargs.keys():
        print('arg:', key, 'has value:', kwargs[key])

named(a=1, b=2, c=3)
```

In this case, the `named` function only supports the provision of keyword arguments. Its output in the above case is:

```
arg: a has value: 1
arg: c has value: 3
arg: b has value: 2
```

In general, these facilities are most likely to be used by those creating libraries as they allow for great flexibility in how the library can be used.

12.10 Anonymous Functions

All the functions we have defined in this chapter have had a *name* that they can be referenced by, such as `greeter` or `get_integer_input`. This means that we can reference and reuse these functions as many times as we like.

However, in some cases we want to create a function and use it only once; giving it a name for this one time can *pollute* the namespace of the program (i.e., there are lots of names around) and also means that someone might call it when we don't expect them to.

Python therefore has another option when defining a function; it is possible to define an *anonymous* function. In Python an anonymous function is one that does not have a name and can only be used at the point that it is defined.

Anonymous functions are defined using the keyword `lambda`, and for this reason they are also known as lambda functions.

The syntax used to define an anonymous function is:

```
lambda arguments: expression
```

Anonymous functions can have any number of arguments but only one expression (that is a statement that returns a value) as their body. The expression is executed, and the value generated from it is returned as the result of the function.

As an example, let us define an anonymous function that will square a number:

```
double = lambda i : i * i
```

In this example the lambda definition indicates that there is one parameter to the anonymous function ('i') and that the body of the function is defined after the colon ':' which multiples `i * i`; the value of which is returned as the result of the function. The whole anonymous function is then stored into a variable called double.

We can store the anonymous function into the variable as all functions are instances of the class function and can be referenced in this way (we just haven't done this so far).

To invoke the function, we can access the reference to the function held in the variable `double` and then use the round brackets to cause the function to be executed, passing in any values to be used for the parameters:

```
print(double(10))
```

When this is executed the value `100` is printed out.

Other examples of lambda/anonymous functions are given below (illustrating that an anonymous function can take any number of arguments):

```
func0 = lambda: print('no args')
func1 = lambda x: x * x
func2 = lambda x, y: x * y
func3 = lambda x, y, z: x + y + z
```

These can be used as shown below:

```
func0()
print(func1(4))
print(func2(3, 4))
print(func3(2, 3, 4))
```

The output from this code snippet is:

```
no args
16
12
9
```

12.11 Online Resources

See the Python Standard Library documentation for:

- https://docs.python.org/3/library/functions.html for a list of built-in functions in Python.
- https://www.w3schools.com/python/python_functions.asp the W3 Schools brief introduction to Python functions.
- https://www.w3schools.com/python/python_lambda.asp a short summary of lambda functions.
- https://devguide.python.org/documentation/markup for information on reStructured Text style docstrings.

12.12 Exercises

For this chapter the exercises involve the *number_guess_game* you created in the last chapter:

Take the number guess game and break it up into a number of functions. There is not necessarily a right or wrong way to do this; look for functions that are meaningful to you within the code, for example:

1. You could create a function to obtain input from the user.
2. You could create another function that will implement the main game playing loop.
3. You could also provide a function that will print out a message indicating if the player won or not.
4. You could create a function to print a welcome message when the game starts up.