Chapter 9 - Functions

9 Functions

Functions are blocks of statements that are organized in a related way and are used to perform a single repeatable task. We have seen examples of many of the Python built-in functions, such as `print()`, `len()`, `range()`, `id()`, `input()`, `int()` and those of the iterable objects but what happens when we need to write our own functions? We would need away to be able to design and build those functions.

9.1 Function Basics

Let's take a look at the syntax for creating a function:

```
def function_name( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

The rules for creating a Python function are as follows:

- Functions **must** begin with the keyword **def** (in lower case) followed by the function name, rounded brackets `()` and colon (`:`)
- Should the function have any input parameters, it should be placed within the rounded brackets.
- The first statement of a function is optional but it is normally used for the documentation string of the function. It is commonly referred to as the *docstring*.
- The `function_suite` is the block of statements that is required for the function to complete the task.
- The `return [expression]` exits the function and pass control back to the caller of the function. A return statement without an expression is the same as return `None`.
- By default, the input parameters have a **positional behaviour** meaning that the order in which they are listed is the same as the order the caller **must use** to pass data to the function

We will also be introducing a Python testing module called `doctest`. This module allows us to perform unit testing on the codes that we have written. Unit testing means to test every individual piece of code (function or class) for the correct output given a series of inputs. Test cases that utilize the `doctest` module are located within the docstrings of a function.

```
def print_info(name, age, country):
    """
    Prints the infomation of a Person.
    Inputs: name - string
            age - string
            country - string

    >>> print_info("John", 35, "Australia")
    Person name is: John
    Age is: 35
    Person is from: Australia
    >>> print_info("Gary", 19, "Canada")
    Person name is: Gary
    Age is: 19
    Person is from: Canada
```

```
16      """
17      print("Person name is:", name)
18      print("Age is:", age)
19      print("Person is from:", country)
20      return
21
22  import doctest
23  doctest.run_docstring_examples(print_info, globals(),
24                                 verbose=True, name="print_info")
```

In the example above, we have defined a function `print_info()` with 3 input parameters. Notice how Python does not require the developers to state the datatypes of the input parameters thus when the function is called, care has to be taken about which datatypes are being used to pass the information to the function.

The `docstring` for this function also contain the `doctest` test cases (lines 8 to 15) as well as the description of the what the function does (lines 3 to 6). Test cases are denoted by the triple greater than signs (`>>>`) and the expected output is placed immediately after the test case. To run the `doctest` tests, include lines 22 and 24 at the last line your code cell before executing your code. The list of arguments are as follows

- `print_info` - the string, a module, a function, or a class object to be tested.
- `globals()` - returning a dictionary containing the variables defined in the global namespace. This is used for the test execution context.
- `verbose` - is used to show a detailed output of the results even after successful tests have been complete. Default is `False` meaning that the output will only be generated in the case of failed test cases.
- `name` - name of the function or anything you would like to use. This value is used in failure messages and defaults to `NoName`.

Note that `doctest` will run all previous test cases in the Jupyter Notebook code cells thus if tests are not required for the function, remove the test cases from the function or clear your kernel's output. This is the output of the `doctest` of successful cases.

```
Finding tests in print_info
Trying:
    print_info("John", 35, "Australia")
Expecting:
    Person name is: John
    Age is: 35
    Person is from: Australia
ok
Trying:
    print_info("Gary", 19, "Canada")
Expecting:
    Person name is: Gary
    Age is: 19
    Person is from: Canada
ok
```

9.2 Argument Passing

In the previous section, we have mentioned that functions may or may not have input arguments. Without arguments, the rounded brackets are left blank but with input arguments, they are placed within the rounded brackets. In this section, we shall see the some of the more common ways of how data can be passed into a function.

9.2.1 Positional Arguments

This is the most direct way of passing data to a function. Parameters are placed within the rounded brackets as a list of comma separated parameters within the function definition like so

```
def print_info(name, age, country):
    """
    Prints the infomation of a Person.
    Inputs: name - string
            age - string
            country - string
    """
    print("Person name is: ", name)
    ...
```

and when the function is called, the caller supplies the data in to the list of parameters in the **exact** same order as in the function definition like so

```
print_info("John", 35, "Australia")
```

thus the value `John` is mapped to the parameter `name`, the value `35` is mapped to the parameter `age` and the value `Australia` is mapped to the parameter `country`. This method of passing data is rigid as the **order** and **number** of the arguments matters and it is the responsibility of the caller to supply the right type and right number of arguments as defined by the function definition.

9.2.2 Keyword Arguments

Keyword arguments allows the function caller to specify the data in the form of `<keyword> = <value>` pairs. A `<keyword>` is the name of the input parameter of a function thus if we use the function from before, `name`, `age` and `country` are keywords and the caller of the function changes to

```
# different ways to use keyword arguments
print_info(age=35, name="John", country="Australia")
# mixed with positional arguments
print_info("John", country="Australia", age=35)
```

This method frees the caller of the order of input arguments as required in the previous section for passing data to a function. However, the **number** of input arguments must still match the number of input parameters as defined by the function definition. Keyword arguments can also be mixed with positional arguments but bear in mind that keyword arguments **must** always be located after all positional arguments.

9.2.3 Default Parameters

Default parameters are input parameters in a function definition that has default values. They have the form of `<keyword> = <value>` in the function definition. These input parameters are regarded as **optional** parameters because if no data is provided, the default value is used. An example is shown below.

```python
def print_info(name, age, country="USA", salary=2000):
    """
    Prints the infomation of a Person.
    Inputs: name - string
            age - string
            country - string, default="USA"
            salary - float/integer, default=2000
    """
    print("Person name is: ", name)
    ...
```

In the above code snippet, `salary` has a default value of `2000`. Default parameters **must** also be located after all positional parameters are listed in the function definition. The caller of this function can then use both the positional and/or keyword arguments methods to call the function like so

```python
# with keyword arguments
print_info(age=35, name="John", country="Australia")
# mixed with positional arguments
print_info("John", country="Australia", age=35)
# with the optional default parameters
print_info("John", 35, salary=5000 )
```

Do note that since the parameters `name` and `age` are positional parameters, the function caller **has** to provide data to those arguments before using the keyword arguments method to supply data to the other optional arguments. In addition, values for default parameters should be immutable objects as mutable objects would produce buggy result due to passing by reference (covered in the section *Passing by Value or Reference or Object Reference?*).

9.3 Return values

Should the caller of the function be expecting a return value from the function, the `return` statement within the function cannot be missing or without an expression. If it is, a `None` will be returned. Functions can also return multiple values, if required.

```python
def assign_grade(score):
    """
    Assign a letter grade to the given score
    Inputs: score - integer

    >>> assign_grade(63)
    (68, 'A')
    >>> assign_grade(46)
    (51, 'D')
    """
    grade = ""
    score_adjust = score + 5   # sneaky adjustment
    if score_adjust < 50:
        grade = "F"
    elif score_adjust >= 50 and score_adjust < 65:
        grade = "D"
    else:
        grade = "A"
    # returning 2 values
    return score_adjust, grade

import doctest
doctest.run_docstring_examples(assign_grade, globals(),
                               verbose=True, name="assign_grade")
```

the output is:

```
Finding tests in assign_grade
Trying:
    assign_grade(63)
Expecting:
    (68, 'A')
ok
Trying:
    assign_grade(46)
Expecting:
    (51, 'D')
ok
```

9.4 Anonymous Function

In general anonymous functions are functions without a name. Recall that normal functions are defined using the `def` keyword but with anonymous functions, the `lambda` keyword is used.

Syntax

```
lambda <arguments>: <expression>
```

A simple usage is as follows:

```python
# single argument
double_num = lambda x: x*2
double_num(2)

# multiple arguments
add_nums = lambda x,y: f'The sum of {x} and {y} is {x+y}'
add_nums(8,9)
```

The properties of `lambda` functions are:

- it can only contain expressions **NOT** statements.
- it's written as a single line of execution.
- it does not support type annotations. (not covered)
- it can be immediately invoked.

No Statements

`lambda` functions cannot contain any statements such as `return`, `pass`, `continue`, `raise`, etc will result in a syntax error. For example, we want to skip the processing of values larger than `5`, we could use an `if` expression like below.

```
if x >5:
    continue
```

the `lambda` equivalent would be:

```
do_nothing = lambda x: continue if x > 5
do_nothing(8)
```

but the output is

```
  File "<ipython-input-14-5b0d5572a735>", line 1
    do_nothing = lambda x: continue if x>5
                           ^
SyntaxError: invalid syntax
```

Single Expression

As we saw earlier, the `lambda` syntax accepts an expression. We can chain multiple expressions to form a single expression using the rounded brackets `()`. For example, we would like to check if 2 integer numbers are equal or which is greater. In normal code, it would look like the following:

```
if x > y:
    return x
elif y > x:
    return y
else:
    return 'The numbers are equal'
```

the `lambda` equivalent would be:

```
eq_or_gt = lambda x,y: x if x > y else (y if y > x else 'The numbers are equal')
eq_or_gt(5,9)
```

Type Annotation

This is beyond the scope of this course therefore we will not go through in detail what type annotations are but a brief description is that it is used to explicitly define the datatype of the function parameter and its output.

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

The above code means that the function `greeting` has an argument `name` that is expected to be of type `str` and it returns a value also of type `str`. There is no `lambda` equivalent of type annotations.

Immediately Invoked

This means that a `lambda` function can be immediately executed upon creation. It uses the following form

```python
(lambda x: x * x)(3)
```

This is generally not used in practice as it makes the `lambda` function a "one-time-use-only" function. Remember that previously, we assign the `lambda` function to a variable and we were then able to use continuously use the `lambda` function like a normal function. This feature of `lambda` function is meant to be used in higher-order functions where functions accept other functions as arguments and return one or more functions. For example, sorting a list of words via the last letter of the word.

```python
# sorting a string by the last letter of the word
words = ['banana', 'pie', 'Washington', 'book']
sorted(words, key=lambda x: x[-1])
```

Caveat to note about `lambda` functions is that when they are the cause of an error, the traceback messages returned are not as precise as normal functions as `lambda` do not have names provided to them.

```python
div_zero = lambda x: x/0
div_zero(5)
```

the error messages would be

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-42465ed3616a> in <module>
      1 div_zero = lambda x: x/0
----> 2 div_zero(5)

<ipython-input-2-42465ed3616a> in <lambda>(x)
----> 1 div_zero = lambda x: x/0
      2 div_zero(5)

ZeroDivisionError: division by zero
```

9.5 `None` Object

With functions, the `None` object is used very often as it is used as temporary assignment to an input parameter especially when we do not know the specific value the input parameter can have. In addition, functions without a `return` statement or an absence of an expression in the `return` statement also returns a `None` object. As such, we would need to know how to detect the `None` object.

```
a = None
if a:
    print("There's something")
else:
    print("There's nothing")
```

Executing the above code will output `There's nothing`. This would appear that the `None` type was evaluated in a Boolean context but that is wrong as `None` is always `False`. To check if the value is a `None` object, we have to use the `is` identity operator.

```
a = None
if a is None:
    print("It's of NoneType")
else:
    print("It's something else")

# output: It's of NoneType
```

Let's place it in a function and run some tests as proof.

```
def what_is(val):
    """
    Testing a for None type value.
    Input: val - any datatype

    >>> what_is(None)
    None is both None and False
    >>> what_is(85)
    85 is True
    >>> what_is(0)
    0 is False
    >>> what_is('abc')
    abc is True
    """
    if val is None:
        if not val:
            print(val, "is both None and False")
    elif val:
        print(val, "is True")
    else:
        print(val, "is False")


import doctest
doctest.run_docstring_examples(what_is, globals(), verbose=True, name="what_is")
```

the results are

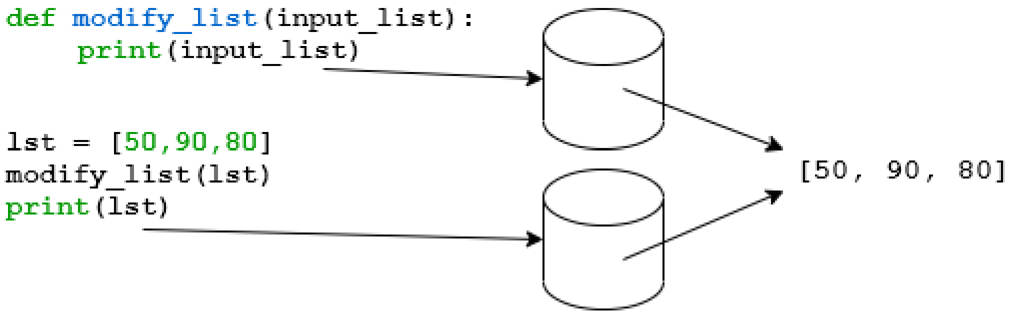
```
Finding tests in what_is
Trying:
    what_is(None)
Expecting:
    None is both None and False
ok
```

```
Trying:
    what_is(85)
Expecting:
    85 is True
ok
Trying:
    what_is(0)
Expecting:
    0 is False
ok
Trying:
    what_is('abc')
Expecting:
    abc is True
ok
```

9.6 Passing by Object Reference

For those of who have programming background experience, you could be wondering does Python pass arguments to functions by Value or Reference? The answer is neither! Python passes by **Object Reference** (or some call it passing by assignment).

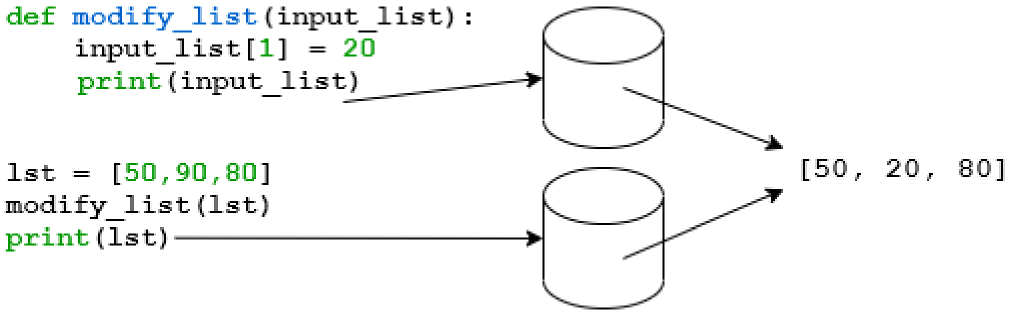
Passing by Object Reference
What is passing by object reference? It means that when an argument is passed to a function, it will receive a reference to the object but it **will not** receive the "container" that houses the object. The function will create its own container (refer to figure 1 below).



```
def modify_list(input_list):
    print(input_list)

lst = [50,90,80]
modify_list(lst)
print(lst)
```

Figure 1: Passing by Object Reference in Python.

Because both function and function caller refer to the same object in memory but in different containers, any operation carried out on the `input_list` gets reflected in the `lst` as well (refer to figure 2 below). In simple terms, different containers are storing the same object or we can also say the same object is stored in multiple different containers.



```
def modify_list(input_list):
    input_list[1] = 20
    print(input_list)

lst = [50,90,80]
modify_list(lst)
print(lst)
```

Figure 2: Modifying a elements in a `list`.

```python
def modify_list(input_list):
    '''
    Modify the elements in a list
    Inputs:
        input_list - a list to modify
    '''
    print("before list change:", input_list)
    # modify element in the list
    input_list[1] = 20
    print("after list change:", input_list)

lst = [50,90,80]
modify_list(lst)
print("list outside function:", lst)
```

The key point is **different names = different containers**.

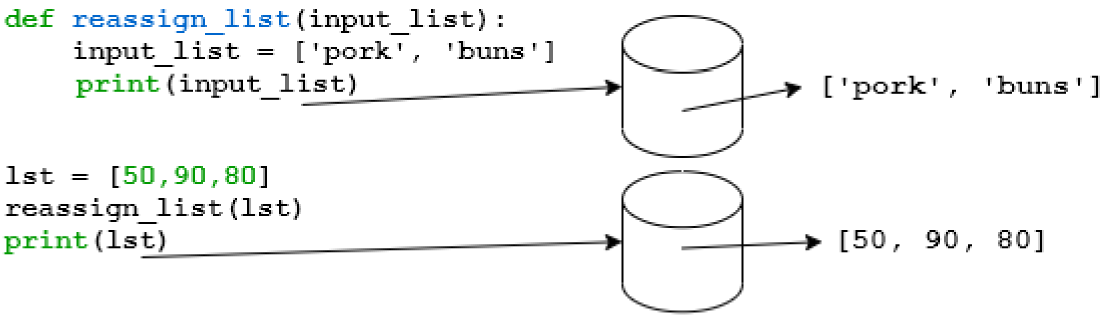
Lets see what happens when we reassign a variable within a function.

```python
def reassign_list(input_list):
    '''
    Reassign the incoming list with a totally different list
    Inputs:
        input_list - a list to modify
    '''

    print('Initial address of input_list', id(input_list))
    # reassign the incoming list with new data
    input_list = ['pork', 'buns']
    print('Final address of input_list', id(input_list))


lst = [50,90,80]
print('Initial address of lst', id(lst))
reassign_list(lst)
print('Final address of lst', id(lst))
print(lst)
```

the output is:

```
Initial address of lst 1981188357888
Initial address of input_list 1981188357888
Final address of input_list 1981188359488
Final address of lst 1981188357888
[50, 90, 80]
```

Now when we reassign the variable in the function (by placing another totally different content into the container), it does not bother the function caller (ie nothing gets changed), refer to figure 8 below.



```
def reassign_list(input_list):
    input_list = ['pork', 'buns']
    print(input_list)

lst = [50,90,80]
reassign_list(lst)
print(lst)
```

```
['pork', 'buns']
[50, 90, 80]
```

Figure 8: Reassigning an object's value.

However, if we wanted the `lst` variable to change, we have to **return** the new value in `input_list` then assign it to `lst` to make the changes permanent like in the code example below.

```
def reassign_list(input_list):
    '''
    Reassign the incoming list with a totally different list
    Inputs:
        input_list - a list to modify
    '''

    print('Initial address of input_list', id(input_list))
    # reassign the incoming list with new data
    input_list = ['pork', 'buns']
    print('Final address of input_list', id(input_list))
    return input_list


lst = [50,90,80]
print('Initial address of lst', id(lst))
lst = reassign_list(lst)
print('Final address of lst', id(lst))
print(lst)
```

```
Initial address of lst 1981188637376
Initial address of input_list 1981188637376
Final address of input_list 1981188637952
Final address of lst 1981188637952
['pork', 'buns']
```

9.7 `import` Statement

Within this chapter, we have seen an `import` statement being used to import the `doctest` library. So what exactly does the `import` statement do? In simple terms, the `import` statement gives your current workspace access to codes from another library, package or module.

Let's break down some terms:

- **Library** - is an umbrella term that loosely means "a bundle of code." These can have tens or even hundreds of individual modules that can provide a wide range of functionality. A library is a collection of packages.
- **Package** - is basically a directory with a collection of related modules that work together to provide certain functionality. These modules are contained within a folder and can be imported just like any other modules.
- **Module** - is a Python file that's intended to be imported into scripts or other modules. It often defines members like classes, functions, and variables intended to be used in other files that import it.

There are 2 general syntax for importing a package or a module

```
# method 1
import <package_or_module_name>[.<module_name>]
# method 2
from <package_name>[.<module_name>] import <module_or_function_name>
```

The main reason that we want to import specific modules is because the Python interpreter requires time and memory space to load the requested modules. Loading of large packages without specifying specific modules can significantly increase the execution time and memory used by the scripts.

For example, if we want to use the function `randint()` from the `random` module in the Python standard library, we can either use

`import random`
or
`from random import randint`

In this case, the first `import` statement will import **all** functions from the `random` module where as the second import statement will **only** import the `randint()` function for use in the program. There is also a difference in how the functions are called based on how they are imported.

```
# method 1
import random
random.randint(0,50)

# method 2
from random import randint
randint(0,50)
```

Aliasing Imported Modules

Imported modules can also be aliased using the `as` keyword. What this means is that the name of the import module can be changed if you have already used the same name for something else in your program or would like to abbreviate a commonly used module.

The general syntax is

```
import <module_name> as <alias_name>
# or
from <module_name> import <module_name> as <alias_name>
```

9.8 References

1. Sturtz, Mar 2020, Defining Your Own Python Function, https://realpython.com/defining-your-own-python-function/
2. Python - Functions, https://www.tutorialspoint.com/python/python_functions.htm
3. Lubanovic, 2019, 'Chapter 9. Functions' in Introducing Python, 2nd Edition, O`Reilly Media, Inc.
4. Heaton, R, 2014, Is Python pass-by-reference or pass-by-value?, https://robertheaton.com/2014/02/09/pythons-pass-by-object-reference-as-explained-by-philip-k-dick/
5. Mogyorosi, M, 2020, Pass by Reference in Python: Background and Best Practices, https://realpython.com/python-pass-by-reference/
6. Burgaud A, How to Use Python Lambda Functions, https://realpython.com/python-lambda/
7. Tagliaferri L, 2017, How To Import Modules in Python 3, https://www.digitalocean.com/community/tutorials/how-to-import-modules-in-python-3
8. Nzomo M, Absolute vs Relative Imports in Python, https://realpython.com/absolute-vs-relative-python-imports/