Chapter 16
Higher-Order Functions

16.1 Introduction

In this chapter we will explore the concept of high-order functions. These are functions that take as a parameter, or return (or both), a function. To do this we will first look into how Python represents functions in memory and explore what actually happens when we execute a Python function.

16.2 Recap on Functions in Python

Let us first recap a few things regarding functions in Python:

Functions (mostly) have a name, and when invoked (or executed) the body of code associated with the function name is run.

There are some important ideas to remember when considering functions:

- Functions can be viewed as named blocks of code and are one of the main ways in which we can organize our programs in Python.
- Functions are defined using the keyword `def` and constitute a function header (the function name and the parameters, if any, defined for that function) and the function body (what is executed when the function is run).
- Functions are invoked or executed using their name followed by round brackets '()' with or without parameters depending on how the function has been defined.

This means we can write a function such as the following `get_msg` function:

```
def get_msg():
    return 'Hello Python World!'
```

We can then call it by specifying its name and the round brackets:

J. Hunt, *A Beginners Guide to Python 3 Programming*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-35122-8_16

```
message = get_msg()
print(message)
```

This of course prints out the string `'Hello Python World!'` which is what you should expect by now.

16.3 Functions as Objects

A few chapters back we threw in something stating that if you forgot to include the round brackets then you were referencing the function itself rather than trying to execute it!

What exactly does that mean? Let's see what happens if we forgot to include the round brackets:

```
message = get_msg
print(message)
```

The output generated now is:

```
<function get_msg at 0x10ad961e0>
```

which might look very confusing at first sight.

What this is actually telling you is that you have referenced a function called `get_msg` that is located at a (hexidecimal) address in memory.

It is interesting to note that just as data has to be located in memory so does program code (so that it can be found and run), although typically data and code are located in separate areas of memory (as data tends to be short lived).

Another interesting thing to do is to find out what the *type* of `get_msg` is—hey it's a function—but what does that mean?

If we issue this statement and run it in Python:

```
print(type(get_msg))
```

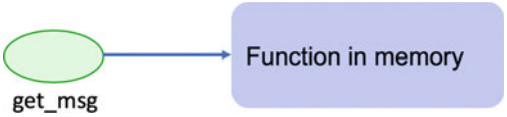
Then we will get the following:

```
<class 'function'>
```

This means that it is of the class of things that are functions just as 1 is of the class of things called integers, `'John'` is of the class of things called strings, and `42.6` is of the class of things called floating point numbers.

Taking this further it actually means that the *thing* being referenced by `get_msg` is a function object (an example or instance of the function class). This `get_msg` is really a type of variable that references (or points at) at the *function object* in memory which we can execute using the round brackets.

This is illustrated by the following diagram:



This means that when we run `get_msg()` what actually happens is we go to the `get_msg` variable and following the reference (or pointer) there to the function and then because we have the round brackets we run that function.

This has two implications:

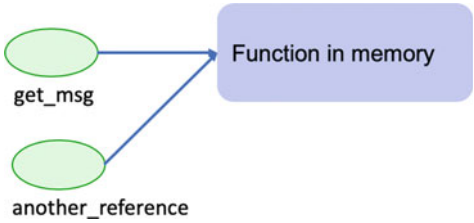
1. We can pass the reference to a function around.
2. We can make `get_msg` reference (point) at a different function.

Let us look at the first of these implications. If we assign the reference represented by `get_msg` to something else, then in effect we have an alias for this function. This is because another variable now also references the same function. For example, if we write:

```
another_reference = get_msg
print(another_reference())
```

Then the result is that the string 'Hello Python World!' is again printed out.

What this has done is to copy the reference held in `get_msg` into `another_reference` (but it is a copy of that reference and that is the address of the function in memory). Thus, we now have in memory:



So just to emphasize this—we did not make a copy of the function; only its address in memory. Thus the same value is held in both get_msg and another_reference, and both these values are references to the same function object in memory.

What does this mean and why should we care? Well it means that we can pass references to functions around within our program which can be a very useful feature that we will look at later in this chapter.

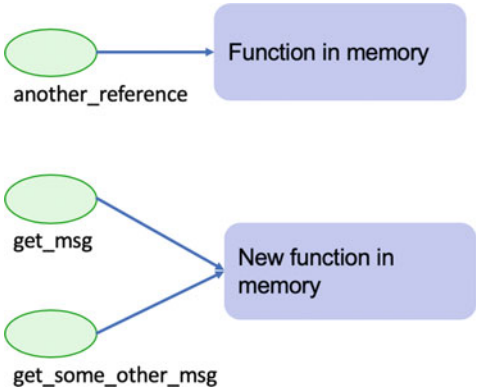
Now let us go back to the second implication mentioned above; we can reassign another function to `get_msg`.

For example, let's say we wrote this next:

```
def get_some_other_msg():
    return 'Some other message!!!'

get_msg = get_some_other_msg
print(get_msg())
```

Now `get_msg` no longer references the original functions; it now references the new function defined by `get_some_other_msg`. It means that in memory we now have



which means the result of calling `print(get_msg())` will be that the string `'Some other message!!!'` is returned and printed out (rather than the `'Hello Python World!'`).

However, notice that we did not overwrite the original function; it is still being referenced by the `another_reference` variable and indeed can still be called via this variable. For example, the code:

```
print(get_msg())
print(another_reference())
```

now generates the output:

```
Some other message!!!
Hello Python World!
```

This illustrates some of the power but also the potential confusion that comes from how functions are represented and can be manipulated in Python.

16.4 Higher-Order Function Concepts

Given that we can assign a reference into a function to a variable, then this might imply that we can also use the same approach to pass a reference to a function as an argument to another function.

This means that one function can take another function as a parameter. Such functions are known as higher-order functions and are one of the key constructs in Functional Programming.

That is, a function that takes another function as a parameter is known as a *higher-order function.*

In fact, in Python, higher-order functions are functions that do at least one of the following (and may do both):

- Take one or more functions as a parameter.
- Return as a result a function.

All other functions in Python are *first-order* functions.

Many of the functions found in the Python libraries are higher-order functions. It is a common enough pattern that once you are aware of it you will recognize it in many different libraries.

16.4.1 Higher-Order Function Example

As an abstract example, consider the following higher-order function *apply*. This function (written in pseudo-code—not a real programming language) takes an integer and a function. Within the body of the function being defined, the function passed in as a parameter is applied to the integer parameter. The result of the function being defined is then returned:

```
def apply(x, function):
    result = function(x)
    return result
```

The function `apply` is a *higher-order* function because its behaviour (and its result) will depend on the behaviour defined by another function—the one passed into it.

We could also define a function that multiplies a number by 10.0, for example:

```
def mult(y):
    return y * 10.0
```

Now we can use the function `mult` with the function apply, for example:

```
apply(5, mult)
```

This would return the value 50.0.

16.5 Python Higher-Order Functions

As we have already seen when we define a function it actually creates a function object that is referenced by the name of the function. For example, if we create the function `mult_by_two`:

```
def mult_by_two(num):
    return num * 2
```

Then this has created a function object referenced by the name `multi_by_two` that we can invoke (execute) using the round brackets '`()`'.

It is also a one parameter function that takes a number and returns a value which is twice that number.

Thus, a parameter that expects to be given a reference to a function that takes a number and returns a number can be given a reference to any function that meets this (implied) contract. This includes our `mult_by_two` function but also any of the following:

```
def mult_by_five(num):
    return num * 5

def square(num):
    return num * num

def add_one(num):
    return num + 1
```

All of the above could be used with the following higher-order function:

```
def apply(num, func):
    return func(num)
```

For example:

```
result = apply(10, mult_by_two)
print(result)
```

The output from this code is:

```
20
```

The following listing provides a complete set of the earlier sample functions and how they may be used with the apply function:

```
print(apply(10, mult_by_five))
print(apply(10, square))
print(apply(10, add_one))
print(apply(10, mult_by_two))
```

The output from this is:

```
50
100
11
20
```

16.5.1 Using Higher-Order Functions

Looking at the previous section you may be wondering why you would want to use a higher-order function or indeed why define one. After all, could you not have called one of the functions (`multi_by_five`, `square`, `add_one` or `mult_by_two`) directly by passing in the integer to used? Yes, we could have, for example, we could have done:

```
square(10)
```

And this would have exactly the same effect as calling:

```
apply(10, square)
```

The first approach would seem to be both simpler and more efficient.

The key to why higher-order functions are so powerful is to consider what would happen if we know that some function should be applied to the value 10, but we do not yet know what it is. The actual function will be provided at some point in the future. Now we are creating a reusable piece of code that will be able to apply an appropriate function to the data we have when that function is known.

For example, let us assume that we want to calculate the amount of tax someone should pay based on their salary. However, we do not know how to calculate the tax that this person must pay as it is dependent on external factors. The `calculate_tax` function could take an appropriate function that performs that calculation and provides the appropriate tax value.

The following listing implements this approach. The function `calculate_tax` does not know how to calculate the actual tax to be paid, instead a function must be provided as a parameter to the `calculate_tax` function. The function passed in takes a number and returns the result of performing the calculation. It is used with the salary parameter also passed into the `calculate_tax` function.

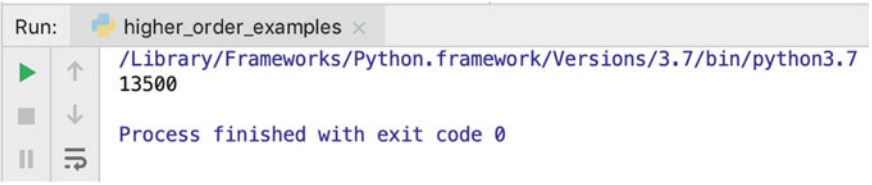
```
import math

def simple_tax_calculator(amount):
    return math.ceil(amount * 0.3)

def calculate_tax(salary, func):
    return func(salary)

print(calculate_tax(45000.0, simple_tax_calculator))
```

The `simple_tax_calculator` function defines a function that takes a number and multiplies it by `0.3` and then uses the `math.ceil` function (imported from the `math` library/module) to round it up to a whole number. A call is then made to the `calculate_tax` function passing in the float `45000.0` as the salary and a *reference* to the `simple_tax_calculator` function. Finally, it prints out the tax calculated. The result of running this program is:



Thus, the function `calculate_tax` is a reusable function that can have different tax calculation strategies defined for it.

16.5.2 Functions Returning Functions

In Python as well as passing a function into another function, functions can be returned from a function. This can be used to select among a number of different options or to create a new function based on the parameters.

For example, the following code creates a function that can be used to check whether a number is even, odd or negative based on the string passed into it:

```
def make_checker(s):
    if s == 'even':
        return lambda n: n%2 == 0
    elif s == 'positive':
        return lambda n: n >= 0
    elif s == 'negative':
        return lambda n: n < 0
    else:
        raise ValueError('Unknown request')
```

Note the use of the raise `Value Error`; for the moment we will just say that this is a way of showing that there is a problem in the code which may occur if this function is called with an in appropriate parameter value for 's'.

This function is a *factory* for functions that can be created to perform specific operations. It is used below to create three functions that can be used to validate what type a number is:

```
f1 = make_checker('even')
f2 = make_checker('positive')
f3 = make_checker('negative')
print(f1(3))
print(f2(3))
print(f3(3))
```

Of course, it is not only anonymous functions that can be returned from a function; it is also possible to return a named function. This is done by returning just the name of the function (i.e., without the round brackets).

In the following example, a named function is defined within an outer function (although it could have been defined elsewhere in the code). It is then returned from the function:

```
def make_function():
    def adder(x, y):
        return x + y
    return adder
```

We can then use this make_function to create the adder function and store it into another variable. We can now use this function in our code, for example:

```
f1 = make_function()
print(f1(3, 2))
print(f1(3, 3))
print(f1(3, 1))
```

which produces the output

```
5
6
4
```

16.6 Online Resources

Further information on higher-order functions in Python can be found using the following online resources:

1. https://en.wikipedia.org/wiki/Higher-order_function Wikipedia page on higher-order functions.
2. https://docs.python.org/3.1/library/functools.html A module to support the creation and use of higher-order functions.
3. https://www.tutorialspoint.com/functional_programming/functional_programming_higher_order_functions.htm A tutorial on higher-order functions.

16.7 Exercises

The aim of this exercise is to explore higher-order functions.

You should write a higher-order function called `my_higher_order_function(i, func)`. This function takes a parameter and a second function to apply to the parameter.

Now you should write a sample program that uses the higher-order function you just created to perform. An example of the sort of thing you might implement is given below:

```
print(my_higher_order_function(2, double))
print(my_higher_order_function(2, triple))
print(my_higher_order_function(16, square_root))
print(my_higher_order_function(2, is_prime))
print(my_higher_order_function(4, is_prime))
print(my_higher_order_function('2', is_integer))
print(my_higher_order_function('A', is_integer))
print(my_higher_order_function('A', is_letter))
print(my_higher_order_function('1', is_letter))
```

If you are using the above code as your test application then you should write each of the supporting functions; each should take a single parameter.

Sample output from this code snippet is:

```
4
8
4.0
True
False
True
False
True
False
```

Note a simple way to find the square root of a number is to use the exponent (or power of) operator and multiply by 0.5.