Higher Order Functions

Surya Duggirala

January 2017

1 Introduction

Higher order functions is one of the more complex topics within computer science to wrap your mind around. The idea is that we can nest functions within each other and make them behave a specific way. For this topic I think it's easier to think of functions as placeholders of sorts. Think about the variable "x". It represents something. It is an abstraction. We can use functions to do a very similar thing. For example:

```
>>> def func(x):
...     return x + 1
...
>>> func(1)
2
>>> # Now let's assign the function itself to a variable.
>>> encase = func
>>> encase
<function func at 0x10a4ddf28>
>>> # Although what we're about to do with higher order
>>> # functions isn't quite the same, it is a similar
>>> # idea intuitively.
>>> Let's take a look at this example
>>> def f(x):
...     def g(y):
...         return x + y
...     return g
...
>>> # So if we break this function down, we can see
>>> # that f(x) is a function in the global frame
>>> # and that it returns the function g
>>> # it does no more and no less. This is a very important
>>> # concept. You cannot forget that calling f(x) returns
>>> # the function g. Again, to reiterate. Calling function
>>> # f(x) returns another function.
>>> # So let's run through this slowly
```

```
>>> f(3)
<function f.<locals>.g at 0x107a17d08>
>>> # We returned the function g.
>>> f(3)(4)
7
>>> # Look at this syntax. It's completely different from
>>> # anything we've seen before. But thinking about it
>>> # intuitively, all we're doing is g(4) because f(x) is
>>> # just g. f(x) IS g
>>> function_g = f(3)
>>> function_g(4)
7
```

2 Deeper Understanding

So now that we've kind of got a feel for higher order function and how they work (kind of), we can go into different ways to understand this. Just a disclaimer before we dive into this: the following examples are just ways to understand higher order functions and how they work. The following functions don't necessarily represent the same things as a true higher order function.

```
>>> def f(x):
...     print(x)
...     return g
...
>>> def g(y):
...     print(y)
...     return h
...
>>> def h(z):
...     print(z)
...
>>> f(1)
1
<function g at 0x107a17d08>
>>> f(1)(2)
1
2
<function h at 0x107802f28>
>>> f(1)(2)(3)
1
2
3
>>> # Do you see what just happened?
>>> # We're using the return values to our advantage in this
```

```
>>> # scenario. h(z) doesn't return anything but f(x) and g(y)
>>> # both return functions.
>>> These functions behave similarly to the following.
>>> def f(x):
...     def g(y):
...         def h(z):
...             print(z)
...         print(y)
...         return h
...     print(x)
...     return g
...
>>> f(1)
1
<function f.<locals>.g at 0x107a17ea0>
>>> f(1)(2)
1
2
<function f.<locals>.g.<locals>.h at 0x107a6b048>
>>> f(1)(2)(3)
1
2
3
```

If you read what the function objects

$$< function\ at\ ... >$$

you'll notice that for the first example with three functions all stored in the global scope, that is to say not nested within each other, look like this

$$< function\ h\ at\ 0x107802f28 >$$

Compare that to what the nested function in the second example shows:

$$< function\ f.<locals>.g.<locals>.h\ at\ 0x107a6b048 >$$

You'll notice that the second example has a lot of periods. We'll get into this more when we talk about objects, but what helps with understanding is that these nested functions are properties of sorts of the larger function. Each time a function has a nested function, think of the parent function's property. Having and returning a nested function is no different than having and returning an integer, for example.

```
>>> def f(x):
...     print(x)
...     num = 4
...     return num
```

```
...
>>> f(5)
5
4
>>> # Compare that to this:
>>> def g(x):
...     print(x)
...     def num(y):
...         return y + x
...     return num
...
>>> g(3)
3
<function g.<locals>.num at 0x107a17ea0>
>>> func = g(3)
3
>>> func
<function g.<locals>.num at 0x107a17ea0>
>>> func(4)
7
>>> # All you're doing is returning a function rather
>>> # than a number or boolean. Super simple when you
>>> # stop overthinking it actually!
```

An important property of higher order functions is that they can only manipulate whatever is inside their scope. So for example:

```
>>> def f(x):
...     def g():
...         x += 1
...         return x
...     return g
...
>>> f(4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in g
UnboundLocalError: local variable 'x' referenced before assignment
>>> def string_builder(string):
...     def string_manip():
...         string += "Unfortunately not..."
...     return string_manip
...
>>> string_builder("Can I do this? ")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in string_manip
```

```
UnboundLocalError: local variable 'string' referenced before assignment
```

But wait, doesn't this drastically reduce the functionality of higher order functions? If we can't pass use our nested function to manipulate something outside its scope aren't we severely limiting what we can do with it? It turns out that's not the case. There's an incredibly helpful keyword in Python that lets us work with variables outside the scope of the function we're in.

```
>>> def f(x):
...     def g():
...         nonlocal x
...         x += 1
...         return x
...     return g
...
>>> f(3)()
4
>>> f(5)()
6
>>> def string_builder(string):
...     def string_manip():
...         nonlocal string
...         string += "Yes you can!"
...     return string_manip
...
>>> permissions = string_builder("Can I do this? ")()
>>> print(permissions)
Can I do this? Yes you can!
```

Do you see what's happening here? We can throw in the keyword 'nonlocal' and all of a sudden we can access elements outside the scope of the function we're currently in. But another question arises. Will this only work when we're one layer of abstraction deep? In other words, will this only work when we're only nested one function deep?

```
>>> def f(x):
...     def g():
...         def h():
...             nonlocal x
...             x += 1
...             return x
...         return h
...     return g
...
>>> f(5)()()
6
```

As you've seen, we can call it from three levels deep. We can go as deep into the

function as we want as long as the variable x invokes the nonlocal method. The catch is that it will only invoke on the most recent assignment of that variable.

```
>>> def f(x):
...     def g():
...         nonlocal x
...         x = "Goodbye World!"
...         def h():
...             nonlocal x
...             x = "Hello World!"
...             print(x)
...         return h
...     print(x)
...     return g
...
>>> f(1)()()
1
Goodbye World!
Hello World!
```

This function is interesting because we do in fact change the value of x the deeper into the function we go. The order of the calls makes it so that the value of x changes as we call more functions.

3 Conclusion

So some of the key thoughts you should be leaving with as this note closes is that higher order functions can be seen as containers as other functions. If you are still having some issues understanding how these work, I highly suggest checking out the note on return values to get a better understanding of how functions can be returned and then used. Functions work the same as any other part of programming. You can assign them to variables, you can return them, and they follow the same rules of scope that other data types in python follow. These are definitely confusing, but the biggest issue is that they're easy to overthink and then confuse. If they're taken one step at a time though, higher order functions are no more complicated than any other topic in computer science.