lecture-4

February 8, 2019

1 Higher Order Functions

Recall that in Python, functions that we define are just objects like anything else:

```
In [2]: def meaningOfLife(x) :
            return 42 * x

        type(meaningOfLife)
```

```
Out[2]: function
```

This means that we can assign functions to variables, and then *treat those variables as functions*

```
In [3]: f = meaningOfLife
        print (f(2))
```

```
84
```

This is sort of like function pointers in C, but much cleaner (because Python doesn't force you to be super precise about types). Note that in the same way that we can bind functions to variables, we can pass functions in to other functions. For example, the function `myApply` below takes a function as its first argument, and calls that function on its second argument:

```
In [4]: def apply(fun, x) :
            return fun(x)

        apply(meaningOfLife, 3)
```

```
Out[4]: 126
```

A function that takes other functions as arguments (or *returns* a function) is called a *higher order function*.

So why are higher order functions useful? Let's take a simple example, where we want to filter an input list, say to discard values that are too high or too low:

```
In [5]: import numpy as np
        data = np.loadtxt('math_scores.txt')
        print (len(data), data[:10])
```

```
50000 [67.2 42.3 33.1 59.  41.3 38.3 44.3 50.4 53.5 43.5]
```

```
In [6]: def simpleFilter(data) :
            res = []
            for d in data :
                if d >= 40 and d <= 60 :
                    res.append(d)
            return res

        filtered1 = simpleFilter(data)
        print (len(filtered1), filtered1[:10])
```

```
31582 [42.3, 59.0, 41.3, 44.3, 50.4, 53.5, 43.5, 53.4, 52.2, 56.3]
```

But now if we want to change the range we filter out, we have to write a *new* filter function to do it. Instead, we can turn filter into a higher order function that takes as an argument a new function `p` that returns `True` if we want to keep the datum, and `False` if we don't:

```
In [7]: def myFilter(p, data) :
            res = []
            for d in data :
                if p(d) :
                    res.append(d)
            return res
```

Two things to note: First, we calling the test `p` for "predicate." Second, our filtering function is called `myFilter` because Python has a built-in function called `filter` that does more or less the same thing. (The built-in function does not return a list; it returns an *iterator*, which can often be used in similar ways.)

Now all we need to do is define a new function to find things in range, then pass it to `myFilter`:

```
In [8]: def inRange(d) :
            return True if d >= 40 and d <= 60 else False

        filtered2 = myFilter(inRange, data)

        print (len(filtered2), filtered2[:10])
```

```
31582 [42.3, 59.0, 41.3, 44.3, 50.4, 53.5, 43.5, 53.4, 52.2, 56.3]
```

Great! Now we can write arbitrary filter functions. Here's one that filters out any element where floor(element) is odd:

```
In [9]: def isEven(d) :
            return True if (int(d) % 2 == 0) else False

        onlyEven = myFilter(isEven, data)

        print (len(onlyEven), onlyEven[:10])
```

```
24936 [42.3, 38.3, 44.3, 50.4, 52.2, 56.3, 44.4, 58.8, 90.1, 52.0]
```

But let's go back to our range function. It's annoying that we have to write a brand new predicate function `p` every time we want a new range to filter. So we can take advantage of another trick of higher order functions: the ability to *return* functions from other functions.

1.1 Returning Functions and Closures

Python lets you define *nested* functions: functions that are defined inside of other functions. Because function definitions are just objects, these newly defined functions can be returned. The trick to these functions is that variables you use inside the nested functions take on their values from the enclosing function. In programming languages terminology, this is a *closure*: a function that "captures" the values of its surrounding environment, and "keeps them around" for the next time you call the function.

Let's define a function that *creates* a predicate function for filtering out a range:

```
In [10]: def createRangeP(minimum, maximum) :
             def p(d) :
                 return True if d >= float(minimum) and d <= maximum else False
             return p

         p1 = createRangeP(45, 55)
         p2 = createRangeP(45, 60)
         print (p1(40))
         print (p1(50))
         print (p1(60))
         print
         print (p2(40))
         print (p2(50))
         print (p2(60))
```

```
False
True
False
False
True
True
```

Let's unpack what happened here. The function `p` is declared as an inner function. It takes one argument, `d`. Inside this definition, it uses two variables `minimum` and `maximum` that don't exist inside `p`'s local scope. Instead, `p` *captures* those variables from its enclosing scope: the values of `minimum` and `maximum` passed in to `createRangeP`. Importantly, `p` "remembers" these values even after it has been returned, for when it is called later!

This new function, in combination with `myFilter` lets us easily filter arbitrary ranges:

```
In [11]: filtered3 = myFilter(createRangeP(20, 80), data)

         print len(filtered3), filtered3[:10]
```

```
  File "<ipython-input-11-6d4e46aa1400>", line 3
    print len(filtered3), filtered3[:10]
            ^
SyntaxError: invalid syntax
```

1.2 Lambdas

Python provides a shortcut for quickly defining functions that compute an expression and return it, called *lambdas* (called this because *lambda* is the symbol used for functions in lambda calculus). We can use this anywhere we need to define and use a function just once, but don't need to assign this function to a variable/name so we can use it again later (for this reason, these functions are often called *anonymous functions*)

```
In [12]: filtered4 = myFilter(lambda x : True if x >= 20 and x <= 80 else False, data)

         print (len(filtered4), filtered4[:10])
```

```
41051 [67.2, 42.3, 33.1, 59.0, 41.3, 38.3, 44.3, 50.4, 53.5, 43.5]
```

We could have also written createRange using lambdas:

```
In [13]: def createRangeP_lambda(minimum, maximum) :
             return lambda d : True if d >= float(minimum) and d <= maximum else False

         filtered5 = myFilter(createRangeP_lambda(20, 80), data)

         print (len(filtered5), filtered5[:10])
```

```
41051 [67.2, 42.3, 33.1, 59.0, 41.3, 38.3, 44.3, 50.4, 53.5, 43.5]
```