Higher Order Functions

You're used to seeing functions defined in Python:

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

print(meaningOfLife(7))
```

```
294
```

But what you may not be used to is that functions in Python are just like any other piece of data. That means that you can assign them to variables. And when you do, you can treat those variables as though they are just accessing the function:

In [3]:
```
f = meaningOfLife
print(f(7))
```

```
294
```

Because functions act like any other data in Python, we call them *first class*. This means that we can, for example, pass them in as arguments to another function.

In [4]:
```
def foo(fun, x) :
    return 2 * fun(x)
```

Note that `foo` here has behavior that changes based on what `fun` is. If we pass different functions in to `foo` , it will do different things. We call `foo` a *higher order function.*

In [5]:
```
foo(f, 7)
```

Out[5]:
```
588
```

In [6]:
```
def other(x) :
    return 39 * x

foo(other, 7)
```

Out[6]:
```
546
```

Filter

One of the best uses of higher order functions is to build generic helper functions that do different things based on the function you pass in to it.

Suppose we want to write a *filter* function, that only keeps data within a certain range:

In [7]:

```
import numpy as np
data = np.loadtxt('inp.txt')
print(len(data))
```

```
1000
```

In [8]:

```
print(data[:10])
```

```
[52.157428   51.96758751 87.96353863 34.12761386 51.95632867 19.4177820
6
 98.80807704 64.30580945  2.87186666 15.67694042]
```

In [9]:

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

In [10]:

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

```
198
[52.15742800208103, 51.967587510202094, 51.95632866658776, 45.852293669
658565, 50.3288024879517, 59.43570924203889, 50.450650101455984, 57.843
559326164865, 48.88091472895692, 51.43408334637033]
```

But now we want to change the filter to keep data in a different range. It looks like we have to rewrite the function:

In [11]:

```
def simpleFilter(data) :
    res = []
    for d in data :
        if d >= 60 and d <= 80 :
            res.append(d)
    return res

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

```
199
[64.30580945497486, 66.8073779287948, 75.05560206968737, 60.33039319445
793, 64.22457990535646, 76.46131601487568, 69.2851653520291, 64.1867070
5576149, 72.68123303005324, 68.23044311467166]
```

One option is to add some additional parameters to our filter function. For example, we could add parameters to define the lower and upper bounds of the range we want to filter:

In [12]:

```
def simpleFilter(data, lo, hi) :
    res = []
    for d in data :
        if d >= lo and d <= hi :
            res.append(d)
    return res

filtered = simpleFilter(data, 40, 60)
print(len(filtered))
print(filtered[:10])
```

```
198
[52.15742800208103, 51.967587510202094, 51.95632866658776, 45.852293669
658565, 50.3288024879517, 59.43570924203889, 50.450650101455984, 57.843
559326164865, 48.88091472895692, 51.43408334637033]
```

In [13]:

```
filtered = simpleFilter(data, 60, 80)
print(len(filtered))
print(filtered[:10])
```

```
199
[64.30580945497486, 66.8073779287948, 75.05560206968737, 60.33039319445
793, 64.22457990535646, 76.46131601487568, 69.2851653520291, 64.1867070
5576149, 72.68123303005324, 68.23044311467166]
```

But that is not satisfying. What if we want to create a filter that does something entirely different, like keep numbers that are *outside* a range, or keep numbers that are *even*? We cannot use `simpleFilter` anymore. We would need to write something different each time we wanted to do a different *kind* of filtering.

So can we do better? What if we take advantage of higher order functions? Suppose we write a filter that *keeps data that pass a test* and then *pass the test to the filter*? Let's write some simple tests:

In [14]:

```
def inRange40_60(d) :
    return True if d >= 40 and d <= 60 else False
```

In [15]:

```
inRange40_60(45)
```

Out[15]:

```
True
```

In [16]:

```
inRange40_60(85)
```

Out[16]:

```
False
```

In [17]:

```
def inRange60_80(d) :
    return True if d >= 60 and d <= 80 else False
```

In [18]:

```
inRange60_80(65)
```

Out[18]:

```
True
```

In [19]:

```
inRange60_80(85)
```

Out[19]:

```
False
```

Now we can write a filter that accepts a test `p` (`p` here stands for `predicate`):

In [20]:

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

In [21]:

```
filtered1 = higherOrderFilter(data, inRange40_60)
print(len(filtered1))
```

```
198
```

In [22]:

```
filtered2 = higherOrderFilter(data, inRange60_80)
print(len(filtered2))
```

```
199
```

In [23]:

```
def outOfRange(d) :
    return True if d < 40 or d > 60 else False

filtered3 = higherOrderFilter(data, outOfRange)
print(len(filtered3))
```

```
802
```

Returning functions from functions

Now we have a completely generic function. But suppose we want to simplify the process of creating tests? Instead of defining a new function from each test, what if we can write a function that *defines new functions* for us? To do this, we will take advantage of returning functions from functions:

In [24]:
```
def createRangeP(lo, hi) :
    def p(d) :
        return True if d >= lo and d <= hi else False
    return p
```

It can be a little hard to understand what `createRangeP` is doing, so let's look at a couple of examples:

In [25]:
```
p1 = createRangeP(40, 60)
p2 = createRangeP(60, 80)
```

In [26]:
```
p1(45)
```

Out[26]:
```
True
```

In [27]:
```
p1(65)
```

Out[27]:
```
False
```

In [28]:
```
p2(45)
```

Out[28]:
```
False
```

In [29]:
```
p2(65)
```

Out[29]:
```
True
```

Think about what happens when `createRangeP` runs. When it does, it *defines a new function called* `p`. That function has specific values for `lo` and `hi` (because we passed them in to `createRangeP`, so `p` is specialized for that particular range. We then *return* this newly created function. Note that we *have not actually run* `p` *yet*. Instead, `p` is now a function that runs a test on its input argument, `x`. We then run it later, as we did above.

We can now use the newly created functions in our filter:

In [30]:
```
len(higherOrderFilter(data, p1))
```

Out[30]:
```
198
```

```
In [31]: len(higherOrderFilter(data, p2))
```

```
Out[31]: 199
```

We can also skip the step of assigning the result of `createRangeP` to a variable:

```
In [32]: len(higherOrderFilter(data, createRangeP(45, 75)))
```

```
Out[32]: 309
```

In class, we looked at a couple of other uses of returning functions from a function. For example, here is a function that takes in two tests (`fun1` and `fun2`) and returns a *new* test that returns true if both `fun1` and `fun2` pass:

```
In [33]: def createAnd(fun1, fun2) :
             def p(x) :
                 return fun1(x) and fun2(x)
             return p

         def keepEven(x) :
             return (int(x) % 2 == 0)

         andP = createAnd(keepEven, createRangeP(45, 75))
         len(higherOrderFilter(data, andP))
```

```
Out[33]: 148
```

As promised, here's a version of `createAnd` that takes in a whole list of functions and creates a new test that returns true if all of the functions are true. And as a bonus, a `createOr` :

```
In [34]: def createAndL(funcList) :
             def p(x) :
                 res = True
                 for f in funcList :
                     res = res and f(x)
                 return res
             return p

         def createOrL(funcList) :
             def p(x) :
                 res = False
                 for f in funcList :
                     res = res or f(x)
                 return res
             return p
```

Map and Reduce

Map and reduce are two of the most common higher-order functions. Map takes a list and a function and returns a new list where each element of the new list is an element from the first list with the function applied to it:

In [35]:
```
def myMap(inp, f) :
    res = []
    for i in inp :
        res.append(f(i))
    return res
```

In [36]:
```
def sq(x) : return x * x
```

In [37]:
```
small = [5, 1, 3, 7, 4, 8, 9]
```

In [38]:
```
myMap(small, sq)
```

Out[38]:
```
[25, 1, 9, 49, 16, 64, 81]
```

Instead of defining a new function every time we want to use it in a higher order function, we can use a *lambda* to define a function at the same time we need it:

In [39]:
```
squared = myMap(small, lambda x : x * x)
print(squared)
```

```
[25, 1, 9, 49, 16, 64, 81]
```

Reduce takes a list and combines together all the elements by calling a function `f` over and over that combines the numbers (e.g., adds them together):

In [40]:
```
def myReduce(inp, f, start) :# f(curr, i) -> curr'
    curr = start
    for i in inp :
        curr = f(curr, i)
    return curr
```

In [41]:
```
sums = myReduce(small, lambda curr, i : curr + i, 0)
print (sums)
```

```
37
```

In [42]:
```
product = myReduce(small, lambda curr, i: curr*i, 1)
print(small)
print(product)
```

```
[5, 1, 3, 7, 4, 8, 9]
30240
```

In [43]:

```
product = myReduce(small, lambda curr, i: i / curr, 1)
print(small)
print(product)
```

```
[5, 1, 3, 7, 4, 8, 9]
9.642857142857142
```

In [44]:

```
def average(inp) :
    return (myReduce(inp, lambda curr, i : curr + i, 0) / len(inp))
```

In [45]:

```
average(small)
```

Out[45]:

```
5.285714285714286
```

We can combine map and reduce to compute more complicated things:

In [46]:

```
def variance(inp) :
    avg = average(inp)
    diffs = myMap(inp, lambda x : x - avg)
    sq_diffs = myMap(diffs, lambda x : x * x)
    return average(sq_diffs)
```

In [47]:

```
variance(small)
```

Out[47]:

```
7.061224489795919
```

In [48]:

```
variance(data)
```

Out[48]:

```
831.4194789188293
```

List comprehensions

If you read a lot of Python code, you won't often see people using `map` and `filter` , because the same thing can be done more concisely using *list comprehensions*:

In [49]:

```
[sq(d) for d in data][:10]
```

Out[49]:

```
[2720.3972957922665,
 2700.6301516305125,
 7737.584127983336,
 1164.6940275730635,
 2699.4600885104887,
 377.0502601833694,
 9763.036087880562,
 4135.237129659535,
 8.247618136083487,
 245.76646092950406]
```

Read this "inside out": for each d in data, apply the function `sq(d)` , and put the results into an output list (note that `data` itself does not change).

We can also combine this with filter:

```
In [50]: [sq(d) for d in data if keepEven(d)][:10]
```

```
Out[50]: [2720.3972957922665,
          1164.6940275730635,
          9763.036087880562,
          4135.237129659535,
          8.247618136083487,
          4463.22574572082,
          3639.756342997896,
          694.6542710674776,
          8244.861709563314,
          2532.988359871253]
```

Which now says: for each d in data, if `keepEven(d)` is true, compute `sq(d)` and put the result in an output list.

```
In [ ]: 
```