0.1 Something cool with higher order functions

In the last lecture, we looked at higher order functions: the idea that you can write functions that use other functions as arguments or as return values. One cool consequence of higher order functions: you don’t need multi-argument functions anymore: you only ever need functions that accept one argument.

You might think this is trivial: if I want to write a function that takes two integers, just write a function that accepts a structure with two integer fields. This is more subtle than that: we will not use any notion of “tuples”: pieces of data that actually represent multiple pieces of data.

Consider a simple function of two arguments:

```python
In [2]: def myFun(x, y):
    return 3 * x + y

print(myFun(3, 8))
```

17

We can write a version of this function that only ever accepts one argument at a time. What we’re going to do is take advantage of closures (remember Lecture 4) to write a function that takes the first argument, then returns a new function that incorporates the first argument and accepts the second argument. We can then call this new function on the second argument to produce the same result as the original function.

```python
In [4]: def myFunCurry(x):
    def inner(y):
        #this function takes the second argument!
        return 3 * x + y
    return inner

print(myFunCurry(3)(8))
```

17

Let’s deconstruct what happened. When we call myFun(3), we’re getting back a new function that closed over 3:
In [6]: inter = myFunCurry(3)
   print (inter)

<function myFunCurry.<locals>.inner at 0x10e377ae8>

This function is the same as if we had written a function that substituted in 3 for x:

In [8]: def inter2(y):
   return 3 * 3 + y
   print (inter2)

<function inter2 at 0x10e377f28>

These new functions can then accept y as their argument to finish the computation:

In [9]: for i in range(1, 100):
   for j in range(1, 100):
      assert(myFun(i, j) == myFunCurry(i)(j))

We can generalize this to functions of 3 arguments:

In [10]: def myFun3(x, y, z):
   return x ** 2 + 3 * y + z
   print (myFun3(3, 4, 5))

26

In [12]: def myFun3Curry(x):
   def inner1(y):
      def inner2(z):
         return x ** 2 + 3 * y + z
      return inner2
   return inner1

   print (myFun3Curry(3)(4)(5))

26

In [13]: for i in range(1, 100):
   for j in range(1, 100):
      for k in range(1, 100):
         assert(myFun3(i, j, k) == myFun3Curry(i)(j)(k))

We call this process (moving from a function that takes k arguments to a series of functions that each take 1 argument) Currying. “Currying” is named after Haskell Curry – and so is the Haskell programming language!
0.2 Map and Reduce

Two of the most common higher order functions are map and reduce (sometimes called fold). map takes an input list and a function \( f \) of one argument, and returns a new list. The \( i \)th element of the output list is \( f \) applied to the \( i \)th element of the input list.

reduce takes a list and a function of two arguments, and returns a single value. That value is computed by applying the function \( f \) to the first two elements of the list, then applying the result of that to the third element, then the result of that to the fourth element, and so on: \( f(f(inp[0], inp[1]), inp[2]), inp[3]) \ldots \)

These are built in functions in Python, but we’ll write our own versions using higher order functions:

```python
In [16]: import numpy as np
data = np.loadtxt('math_scores.txt')
def myMap(f, inp):
    res = []
    for i in inp:
        res.append(f(i))
    return res

def myReduce(f, inp, init=None):
    if (init == None):
        res = inp[0]
    else:
        res = f(init, inp[0])
    for i in range(1, len(inp)):
        res = f(res, inp[i])
    return res

Let's use myReduce to compute the average of our input data:

```py
In [17]: total = myReduce(lambda x, y : x + y, data)
count = myReduce(lambda x, y : x + 1, data, 0)
avg = total / count

print (total, count, avg)
```

```text
2850713.8999999915 50000 57.014277999999834
```

We can then use myMap and myReduce together to compute the variance:

```python
In [19]: sqerr = myMap(lambda x : (x - avg) ** 2, data)
var = myReduce(lambda x, y : y + x, sqerr) / count

print (var)
```

```text
250.58829593871462
```
We can compare these to the average and variance computed by the NumPy functions:

```python
In [20]: print (avg, np.mean(data))
    print (var, np.var(data))
```

```
57.01427799999834 57.014278
250.58829593871462 250.58829593871602
```

1 Data Structures

We have already seen two basic data structures in python. First, we saw lists:

```python
In [21]: list1 = [0, 2, 4, 6, 8]
    print (type(list1))
    print (list1)
    print (list1[2:4])
    list2 = list1 + [10]
    print (list2)
```

```
<class 'list'>
[0, 2, 4, 6, 8]
[4, 6]
[0, 2, 4, 6, 8, 10]
```

Wait, two data structures? Yes! Strings in Python are a data structure too. In fact, like lists, strings are a sequence data structure, that supports several of the same operations as lists:

```python
In [22]: string1 = 'Hello'
    print (type(string1))
    print (len(string1))
    print (string1[1:4])
    string2 = string1 + '!!'
    print (string2)
    for s in string2 :
        print (s)
```

```
<class 'str'>
5
ell
Hello!
H
e
l
l
o
!```
1.1 Tuples

Another sequence type in Python is the tuple. These look a lot like lists, with a few exceptions. First, you define them with ( ) instead of [ ]. Second, tuples are immutable. Once you define them, you cannot add or remove items from them. Think of tuples as a way of defining structures. You can get at the elements of tuples by indexing into them, just like lists or strings:

```
In [23]: tuple1 = ('Hello', 3.14, 2)
   print ("{} {}".format(tuple1, type(tuple1)))
   print ("{} {}".format(tuple1[1], type(tuple1[1])))

('Hello', 3.14, 2) <class 'tuple'>
3.14 <class 'float'>
```

And you can get at elements of a tuple by iterating over them (again, just like lists or strings)

```
In [24]: for t in tuple1 :
   print ("{} {}".format(t, type(t)))

Hello <class 'str'>
3.14 <class 'float'>
2 <class 'int'>
```

Here’s a fancier way to iterate over a tuple:

```
In [25]: for i, t in enumerate(tuple1) :
   print ("{} {}".format(t, type(t)))
   print ("{} {}".format(tuple1[i], type(tuple1[i])))

Hello <class 'str'>
Hello <class 'str'>
3.14 <class 'float'>
3.14 <class 'float'>
2 <class 'int'>
2 <class 'int'>
```

What’s going on with enumerate up there? That’s a special function for iterating through sequence types (meaning you can use it on strings and lists, too) that emits *tuples* as its output. The tuples it emits are of the form (index, value). The looping code takes advantage of a handy Python trick called *unpacking* that lets you get at the elements of a tuple without having to index them.

```
In [27]: s, f, i = tuple1
   print (s)
   print (f)
   print (i)
```

5
Hello 3.14 2

In [28]: for packed in enumerate(tuple1):
   print (packed)

(0, 'Hello')
(1, 3.14)
(2, 2)

Using tuples as your replacement for C-like structs can be tricky, if the tuples get complicated (think about how hard it might be to remember the organization of the tuple). Python provides named tuples as a way around this, which we will get to in a later lecture when we talk about objects.

1.2 Sets

Python includes sets as a built-in data type. They operate just like Java sets or STL sets: unordered groups of elements that maintain a uniqueness property, where each value only appears once in the set

In [29]: set1 = {'a', 'b', 'c'}
   print (set1) #note the ordering!

{'b', 'c', 'a'}

In [30]: set2 = {'a', 'b', 'c', 'a'}
   print (set2)

{'b', 'c', 'a'}

In [31]: set2.add('d')
   print (set2)
   set2.remove('a')
   print (set2)

{'b', 'c', 'a', 'd'}
{'b', 'c', 'd'}

In [32]: set3 = set()  #empty set initialization
   print (set3)
   set3.add('a')
   set3.add('b')
   set3.add('a')
   print (set3)
set()
{'b', 'a'}

In [33]: for d in set2 :
    print (d)
b
c
d

2 Dictionaries

The final “basic” data structure in Python is the dictionary. (Other languages call them “associative arrays.” You probably know them as “maps”): data structures that let you map keys to values. Each key in a Python dictionary is unique, and that key maps to a certain value.

In [34]: dict1 = {'a': 0, 'b': 1, 'c': 3}
    print (dict1['a'], dict1['c'])
0 3

In [35]: dict1['a'] = 10
    print (dict1['a'], dict1['c'])
10 3

When iterating over a dictionary, you iterate over the keys. If you want to iterate over both the keys and the values, use items

In [36]: for k in dict1 :
    print (k, dict1[k])

    for k, v in dict1.items() :
        print (k, v)
a 10
b 1
c 3

Wait, what’s going on with items? We’re not calling it like we do other functions like len or min or max. iteritems is a method of the dict class. dict1 in the above example (like all Python data) is an object. (We saw similar ways of calling methods when we append items to lists, or add items to sets.)