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 [1]: 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 3) 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 [2]: def myFunCurry(x):
    #note that this only takes one argument!
    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:

```python
In [3]: inter = myFunCurry(3)
print inter
<function inner at 0x103ba9938>
```
This function is the same as if we had written a function that substituted in 3 for $x$:

In [4]:
```python
def inter2(y):
    return 3 * 3 + y

print inter2
```

<function inter2 at 0x103ba96e0>

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

In [5]:
```python
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 [6]:
```python
def myFun3(x, y, z):
    return x ** 2 + 3 * y + z

print myFun3(3, 4, 5)
```

26

In [7]:
```python
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 [8]:
```python
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!
Data structures

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

```python
In [9]:
    list1 = [0, 2, 4, 6, 8]
    print type(list1)
    print list1
    print list1[2:4]
    list2 = list1 + [10]
    print list2

<type '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 [10]:
    string1 = 'Hello'
    print type(string1)
    print len(string1)
    print string1[1:4]
    string2 = string1 + '!
    print string2
    for s in string2 :
        print s

<type 'str'>
5
ell
Hello!
H
e
l
l
o
!
```

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 [11]:
    tuple1 = ('Hello', 3.14, 2)
    print "{} {}".format(tuple1, type(tuple1))
    print "{} {}".format(tuple1[1], type(tuple1[1]))

    ('Hello', 3.14, 2) <type 'tuple'>
    3.14 <type 'float'>

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

In [12]:
    for t in tuple1 :
        print "{} {}".format(t, type(t))

    Hello <type 'str'>
    3.14 <type 'float'>
    2 <type 'int'>

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

In [13]:
    for i, t in enumerate(tuple1) :
        print "{} {}".format(t, type(t))
        print "{} {}".format(tuple1[i], type(tuple1[i]))

    Hello <type 'str'>
    Hello <type 'str'>
    3.14 <type 'float'>
    3.14 <type 'float'>
    2 <type 'int'>
    2 <type '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 [14]:
    s, f, i = tuple1
    print s
    print f
    print i

    Hello
    3.14
    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 when we talk about objects.

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 [16]:
```python
set1 = {'a', 'b', 'c'}
print set1  #note the ordering!
set(['a', 'c', 'b'])
```

In [17]:
```python
set2 = {'a', 'b', 'c', 'a'}
print set2
set(['a', 'c', 'b'])
```

In [18]:
```python
set2.add('d')
print set2
set2.remove('a')
print set2
set(['a', 'c', 'b', 'd'])
set(['c', 'b', 'd'])
```

In [19]:
```python
set3 = set()  #empty set initialization
print set3
set3.add('a')
set3.add('b')
set3.add('c')
print set3
set([])
set(['a', 'b'])
```
In [20]:

```python
for d in set2:
    print d
```
cbd

Comprehensions

Python provides set and list *comprehensions*, which are efficient ways of processing sets and lists to produce new sets and lists (think mathematical set notation)

In [21]:

```python
import numpy as np
data = list(np.random.randint(-4, 4, 25))
print data
```

```
[-1, 3, 2, -2, 3, 2, 0, 3, -2, 2, -2, -3, 3, -2, -1, -2, 0, -4, -2, 0, -4, 3, -2, -4, 3]
```

In [22]:

```python
absdata = [abs(d) for d in data]
print absdata
```

```
[1, 3, 2, 2, 3, 2, 0, 3, 2, 2, 2, 3, 3, 2, 1, 2, 0, 4, 2, 0, 4, 3, 2, 4, 3]
```

In [23]:

```python
absset = {abs(d) for d in data}
print absset
```

```
set([0, 1, 2, 3, 4])
```

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 [24]:

```python
dict1 = {'a': 0, 'b': 1, 'c': 3}
print dict1['a'], dict1['c']
```

```
0 3
```

In [25]:

```python
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 `iteritems`

```python
In [26]:
for k in dict1 :
    print k, dict1[k]

for k, v in dict1.iteritems() :
    print k, v
```

```
a 10
c 3
b 1
a 10
c 3
b 1
```

Wait, what's going on with `iteritems`? 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.)

### Classes and Objects

This is not a particularly formal introduction to the Python data model and object model. For that, please refer to documentation on the Python data model ([https://docs.python.org/2/reference/datamodel.html](https://docs.python.org/2/reference/datamodel.html)) and Python classes ([https://docs.python.org/2/tutorial/classes.html](https://docs.python.org/2/tutorial/classes.html)).

Python, like C++ and Java, is **object oriented**. The basic data model in Python is that everything is an object of some sort. An object combines data and methods. *Everything in Python is an object*, including "simple" data like integers and floats.

A **class** in python defines a set of **attributes**: these can be variables or methods. This defines a set of properties that you want all objects of a certain type to have. An **object** in Python is an **instance** of a class: it shares attributes with all other classes, but can also have attributes (think: member data) that is different from other instances. This lets you have objects with their own "local" data.

Methods for a class take an extra `self` argument. When you invoke a method on an object (think `myList.append(x)`), this `self` argument refers to the object you invoked the method on (in the example, `myList`).
In [27]:
    class Counter (object):
        totalCount = 0 #shared number across all instances

        def __init__(self): #constructor for the class.
            self.count = 0 #local count for each instance

        def incr(self):
            Counter.totalCount += 1
            self.count += 1

        def __str__(self):
            #special function like "toString" in Java
            return "Total count: {}, Local count: {}".
            format(Counter.totalCount, self.count)

In [28]:
    c1 = Counter()
    c2 = Counter()
    print c1
    print c2

Total count: 0, Local count: 0
Total count: 0, Local count: 0

In [29]:
    for i in range(0,5):
        c1.incr()
        c2.incr()

    print c1
    print c2

Total count: 10, Local count: 5
Total count: 10, Local count: 5

Classes themselves, like functions, are just objects, as are the methods inside them:

In [30]:
    print type(Counter)
    print type(Counter.incr)

<type 'type'>
<type 'instancemethod'>

Unsurprisingly, like with functions, Python lets you create new classes dynamically and return them. This gives us a handy way to create things that behave like structures, using the namedtuple method:

In [31]:
    import collections
    Point = collections.namedtuple('Point',['x', 'y', 'color'])
Pandas

The place where you will probably be using classes the most is when manipulating pandas *dataframes*: this is the key class provided by pandas (in addition to *series*), and it provides a number of instance methods for manipulating data. We will not spend a lot of time deconstructing pandas dataframes in class -- we will explain as much as is needed in relevant homeworks. You can also look at the docs (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html)

```python
In [32]: p = Point(2.4, 3.7, 'red')
   ...: print p
   ...:
Point(x=2.4, y=3.7, color='red')

In [33]: print p.x, p.y, p.color
   ...:
2.4 3.7 red
```

```python
In [ ]: import pandas as pd
   ...: data = pd.read_csv('hw02_problem3.csv', header=None, skipinitialspace=True)
   ...: print type(data)
   ...: data
   
   ...: data1 = data[(data[2] == 'white')]
   ...: print type(data1)
   ...: data1
   
   ...: data2 = data[(data[2] == 'white')][[0, 1]]
   ...: data2
```