Intro. to Computing (810:051) **Lab 10** **Name:_____**

Objectives:

- Practice using keywork arguments, a pseudo-arguments of the form *args and **kwargs
- Practice writing higher-order functions

For today's lab you'll need several files, so start the lab by downloading the following file to your desktop:
http://www.cs.uni.edu/~fienup/cs051s10/labs/lab10.zip
Extract this file to the Desktop (or your flash drive) by right-clicking on lab10.zip icon and selecting Extract All.

Part A: Yesterday in lecture we reviewed the syntax of a function definition and the type of formal parameters:

- formal parameter names which are required parameters
- keyword arguments of the form: id = value which assigns the formal parameter id a specified default value. Note: keyword arguments can only appear as the last parameters in a parameter list
- a pseudo-argument of the form *args that captures all of the remaining non-keyword arguments in a tuple.
- a pseudo-argment keyword argument of the form **kwargs which captures all of the remaining keyword arguments into a dictionary

For example, the function definition "`def foo(x, y, *args, **kwargs):`" called with "`foo(1, 2, 3, 4, a=5, b=6)`" will result with *formal parameter* x containing 1, y containing 2, args containing (3, 4), and kwargs containing {'a':5, 'b':6}. Python uses pass-by-value parameter passing, which copies the value of the *actual parameters* to the *formal parameters*. Since variables associated with built-in collections (e.g., lists, dictionaries, etc.) and objects contain references, *actual parameters* to these only copy their reference values to corresponding formal parameters.

For each of the following, predict the output of the Python code segment.

```
def foo(a, b, *args, **kwargs):
      print 'a=',a,'b=',b ,'args=',args,\
            'kwargs=',kwargs
      a = "dog"
      b['two']=2

myString = "cat"
myDict = { 'four':4, 'one':1 }
foo(myString, myDict, 10, 'bye', 11, s=11, t=12)
print 'myString =', myString
print 'myDict =', myDict
```

```
def bar(a, b = 8, *args, **kwargs):
    print 'a=',a,'b=',b ,'args=',args,'kwargs=',kwargs
    a = "dog"
    b=2

s=1
t=2
myTuple = ('pi', 3.14)
bar(myTuple, s=11, t=12)
bar(s=11, t=12, a=4, b=6)
bar(myTuple, s, t, d=4 )
print 'myTuple =', myTuple
```

After you have predicted the output and compared it with the actual output, raise your hand and explain your results.

Part B: Higher-order functions

In Python, functions are *first-class data objects*, so we can write *higher-order functions* that expect a function and a set of data values as arguments. Python has the following predefined higher-order functions with are often useful:

Function	General Syntax	Example	Description
`apply`	apply(object[, args[, kwargs]])	apply(pow, (2, 3)) 8	Call a callable object with positional arguments taken from the tuple args, and keyword arguments taken from the optional dictionary kwargs
`map`	map(fn, sequence[, sequence, ...])	map(len, ['cat', 'i', 'at']) [3, 1, 2]	Return a list of the results of applying the function to the items of the argument sequence(s).
`filter`	filter(fn or None, sequence)	def odd(n): return n % 2 == 1 filter(odd, range(9)) [1, 3, 5, 7]	Return those items of sequence for which function(item) is true. If function is None, return the items that are true
`reduce`	reduce(fn, sequence[, initial])	def add(x, y): return x + y reduce(add, [2, 3, 4, 5]) 14	Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value.

A *lambda* is an anonymous function with no name that can be used to avoid defining a function and then passing it as a parameter to the higher-order functions. The general syntax of a lambda is:

lambda <argname-1, argname-2, ..., argname-n> **:** <expression using argname's>

The `reduce` example above would be: `reduce(lambda x, y: x + y, [2, 3, 4, 5])`

a) Rewrite the filter example above which filters out all the non-odd values from a list using a **lambda** instead of the function `odd`.

b) Write the code for a filtering that generates a list of the positive numbers in a list named `numbers`. You should first define You should use a lambda to create the auxiliary function.

c) Write the code for a reducing that creates a single string from a list of strings named `words`.

d) If more than one sequence is given to the `map`, the function is called with an argument list consisting of the corresponding item of each sequence, substituting None for missing values when not all sequences have the same length. If the function is None, return a list of the items of the sequence (or a list of tuples if more than one sequence). Suppose we had a list of strings for state abbreviations (e.g., statesList = ['IA', 'OR', ...]) and a parallel list of corresponding state counts (e.g., stateCounts = [40, 35, ...]). Write the code for a mapping that takes both lists as arguments and builds a new list of tuples with corresponding state count and abbreviation (e.g., [(40, 'IA'), (35, 'OR'),...]).

After you have implemented all of the higher-order functions above, raise your hand and demonstrate your code.

If you do not get done today, then show me the completed lab in next lab period. Make sure that you log off the computer before you leave.