Lab 11

Learning Objectives:

I. **What characters do I type next?** *How do I write comprehensions in Python?* This lab will give you the opportunity to practice all the tidbits of list, dict, and set comprehensions.

E. **Don't Reinvent the Wheel** *Don't write code that you don't need to.* Functional tools exist that allow you to write code quickly and easily—recognize when you can use them, and then use them!

Part 1 will help you put all the different parts of a list comprehension together, one by one.
Part 2 will have you work with small examples showcasing concepts from functional programming.
Part 3 is for you to test out advantages of listcomps and functional programming in a larger context.

Part 1: Building a List Comprehension from the Ground Up

Comprehensions have a lot of pieces of syntax. This part will help you use them all together, implementing one at a time. When you're done, `%save` your solutions in the file `list_comp.py`

Write a list comprehension that:

- **Returns a list of the first 10 letters of the alphabet.** Hint: here's an alphabet: (import the string module first) `string.ascii_lowercase`.
- Returns a list of the first 10 letters of the alphabet **except the sixth one.**
- (Harder!) Returns a list of the first *10* letters of the alphabet except the sixth one, **each repeated each 1 , 2, and 3 times:** `['a', 'aa', 'aaa', 'b', ..., 'c', ...]`
- Returns a list like the above, but in a grid:
  `[['a', 'aa', 'aaa'], ['b', ...], ['c', ...], ...]`
- Returns a list like the above, but if the number matches the index of the character *mod 3* (e.g. `'c'` and `3`, instead print a single capitalized version of that character:
  `[['A', 'aa', 'aaa'], ['b', 'B', 'bbb'], ['c', 'cc', 'C'], ['D', ...], ...]`

Part 2: Function-fu

This part of the lab is designed to be done interactively, using IPython and the NumPy library functions. You are now familiar with nearly all the major features in Python, and now is your chance to see how powerful the language can be. Your task is to write functions for the following tasks, in as compact and elegant a form as possible. When you're done, `%save` your solutions in the file `functions.py`

- Squares: Create a list of the first 10 square numbers. First use a list comprehension, then use a `lambda` function and `map`
- Products: Use a `lambda` function with `reduce` to multiply up the numbers from 1 to 5.
- Filenames: Say that you have a string with a list of file names `"test1.py test2.py`

`test3.py ..."` and you want to output a list of just the filenames without the `.py` extension. Do this with a list comprehension.

- Dictionary, which takes a string (ex: `"Spam and eggs should not have legs!"`) and returns a `set` of the unique words, case-insensitive.
- A graph of the square function, which is a `dict` that has the first ten integers as its keys and their squares as its items: `{1: 1, 2: 4, 3: 9, ...}`.
- (BONUS) Factorial function, recursively defined using a `lambda` expression *(ask us if you don't know about recursion). (Note you can use if/else inside your lambda expression!)*

The bulk of the work can be done with maps, comprehensions, and lambdas. Strive for your code to be well-decomposed and readable – in Python, these qualities also lead to fast and bug-free code.

Part 3: If you have time – Hailstone Sequence

The Hailstone Sequence is a sequence of integers formed by a simple rule: if *n* is even, divide by 2, and if n is odd, multiply by 3 and add 1. It is conjectured (the Collatz Conjecture: http://xkcd.com/710/), though neither proven nor disproven, that for any starting *n* this sequence will converge to 1 in a finite number of iterations. We want you to try and do these tasks below efficiently using functional programming techniques.

The starter code in `hailstone.py` contains a function, `hailstone`, that will iteratively compute the sequence starting at *n*. Your job is to extend the functionality of this code in the following ways:

- Write the `get_peaks` function that finds all the peaks (local maxima) of the sequence, returning them as a set of tuples `(i, y)` where i is the index in the original sequence. You might find the built-in `enumerate` function or its numpy counterpart `np.ndenumerate` helpful here.
- Write the `-range` mode in the main method, which will compute hailstone sequences for *n=1,2,...,N* and, for each, calculate the length of the sequence and number of peaks.

If you're having trouble, think about the problem "procedurally" ("the regular way"), and see if you can find a correspondence between this procedure and functional code.