Lab #10

Physics 91SI Spring 2013

Objective: In this lab, you will use Python's support for functional programming features to write fast, powerful, and compact code.

As usual, log on to `corn` and clone over the starter repository:

```
hg clone /afs/ir.stanford.edu/class/physics91si/src/lab10 lab10
```

Remember to `hg commit` often to save your changes, and submit your code at the end of class.

Part 1: 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.

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.
- Extend the `plot_seq()` function to label all of the peaks with their *y*-values. This is easiest to do with a `lambda` function, mapped across your set of peaks.
- 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. Your main method should then plot these as side-by-side scatterplots (use `plt.subplot()`). What pattern do you see? Try a very large *N*, such as 10000.

All of these tasks can be done using the usual procedural constructs, but are much easier using comprehensions and anonymous functions (lambdas). If you're having trouble, think about the problem "procedurally," see if you can find a correspondence between this procedure and functional code.

Part 2: Function-fu

This part of the lab is designed to be done interactively, using `ipython --pylab` 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:

- Find and replace, which replaces all matches of a regular expression with the provided string (see function prototype)
- Dictionary, which takes a string and returns the set of unique words, case-insensitive
- Factorial function, recursively defined using a lambda expression *(ask us if you don't know about recursion)*
- Jumble, which randomizes the order of a sequence
- Vector outer product (`outer()`), which generates a matrix with elements $A_{ij} = u_i * v_j$ (rows are the vector *v* multiplied by elements of the vector *u* – don't worry if you mess up and get the transpose)
- ***Bonus:*** harmonic filter, which takes the Fourier transform of a data sequence, finds the strongest frequency, and either suppresses all harmonics and subharmonics (integral multiples and divisors) of this frequency, or retains them and suppresses all other frequencies - a handy tool for processing periodic signals. Think of this as the composition of mathematical functions, and try to write it using only lambdas and Python/numpy builtins.

For some of these, you may need to define short helper functions or use conventional loops, but 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.

Function prototypes and further documentation can be found in `functions.py`, and you should write your finished code in this file for submission. However, we encourage you to prototype in IPython - either by importing your module (`import functions`) or by directly entering the code. Remember that the up-arrow key lets you browse the command history, and typing a prefix lets you easily recall commands that started with those characters – handy if you want to edit a lambda expression.

Part 3: The Next Step

If you have time, go back and look at some of your past Python assignments, particularly `language.py` and `filter.py` from lab 4, `contacts.py` from lab 6, or `data_analysis.py` from lab 7 and 8. If you haven't finished one of these, try writing it using functional tools – many functions can be expressed in two or three lines using lambdas and comprehensions!