CS1110 Spring 2015

Assignment 5: Due Thursday March 26 at 11pm

You must work either on your own or with one partner. If you work with a partner, you and your partner must first register as a group in CMS and then submit your work as a group.

You may discuss background issues and general solution strategies with others, but the programs you submit must be the work of just you (and your partner). We assume that you are thoroughly familiar with the discussion of academic integrity that is on the course website. Any doubts that you have about "crossing the line" should be discussed with a member of the teaching staff before the deadline.

Objectives. Lists, functions with list parameters, search. The assignment is based on Lectures 14, 15, and 16 and Labs 6,7.

1 Set-Up

Start by setting up a folder called `A5`. Into this folder download `simpleGraphicsE.py` (an expanded version) and the template modules `ShowPF.py`, `ShowLS.py` and `ShowTF.py` from the assignments page on the course website. LET A5 BE THE CURRENT WORKING DIRECTORY WHENEVER YOU ARE WORKING ON THIS ASSIGNMENT.

2 The Perfect Shuffle

Suppose we have a deck of 8 cards numbered 0 through 7:

0	1	2	3	4	5	6	7

We split the deck into its top and bottom halves:

0	1	2	3

4	5	6	7

We reassemble the deck by alternately taking cards from the top and bottom halves:

0	4	1	5	2	6	3	7

This is an example of the *perfect shuffle*. Let's perfect shuffle the deck again:

0	4	1	5

2	6	3	7

0	2	4	6	1	3	5	7

And again:

0	2	4	6

1	3	5	7

0	1	2	3	4	5	6	7

Yikes! We are back to where we started from! Apparently, if you perfect shuffle a deck of 8 cards 3 times, then you return to the original deck.

In this problem you will explore this "cycle back" phenomena by developing a module `ShowPF.py` that has two functions:

```
def PF(x):
    """ Perfect shuffles the values in x.
    PreC: x is a list of ints with even length,

def numPF(n):
    """ Returns an int that is the number of perfect shuffles that
     have to be applied to a length-n list of distinct integers in
     order for it to be restored to its original form.

    PreC: n is an even int.
    """
```

Some advice. Study the $n = 8$ example above. To implement `PS` you will have to understand where the values in the "top half" of `x` end up and where the values in the"bottom half" end up. Submit your final implementation of `ShowPS.py` to CMS. We have provided a template on the course website that includes a modest test script.

3 Fitting a Line to Data

Summation of the values in a list is fundamental. That is why Python provides a `sum` function. The 1-liner

```
 s = sum(x)
```

is equivalent to

```
s = 0
for k in range(len(x))
   s+=x[k]
```

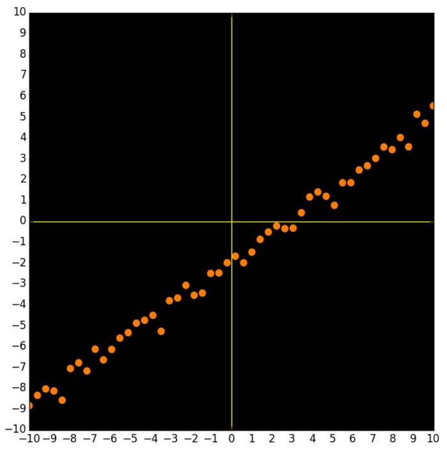
assuming that `x` is an initialized list of numbers.

Throughout the sciences and engineering, the "sigma notation" is used to indicate summation. The notation

$$s = \sum_{k=0}^{n-1} x_k$$

is how we say "sum the n values in the list x." Instead of `x[k]` we write x_k and instead of "`for`" we write "$\sum$."

To be good at computer-based problem solving you have to be good with the summation notation because it is ubiquitous and because it signals the presence of loops and lists (our current preoccupation). So let's get some practice by fitting a line $y = a + bx$ to a cloud of data that "looks linear", e.g.:



The recipes for "the best" a and b make heavy use of the sigma notation. In particular, a and b are given by

$$b \;=\; \sigma_{xy}/\sigma_{xx} \qquad a \;=\; \bar{y} - b\bar{x}\ ,$$

where

$$\bar{x} \;=\; \frac{1}{n}\sum_{k=0}^{n-1} x_k \qquad \sigma_{xx} \;=\; \frac{1}{n}\sum_{k=0}^{n-1}(x_k - \bar{x})(x_k - \bar{x})$$

$$\bar{y} \;=\; \frac{1}{n}\sum_{k=0}^{n-1} y_k \qquad \sigma_{xy} \;=\; \frac{1}{n}\sum_{k=0}^{n-1}(x_k - \bar{x})(y_k - \bar{y}).$$

Here is another example that should clarify what the Σ-notation means:

$$\sum_{k=0}^{2}(x_k - \bar{x})(y_k - \bar{y}) \;=\; (x_0 - \bar{x})(y_0 - \bar{y}) \;+\; (x_1 - \bar{x})(y_1 - \bar{y}) \;+\; (x_2 - \bar{x})(y_2 - \bar{y})$$

Note that $\bar{x}$ and $\bar{y}$ are easily obtained via `sum`:

```
xbar = sum(x)/n
ybar = sum(y)/n
```

In this problem you are to implement a function `LSFit(x,y)` that returns a and b as defined by the above formulae. The parameters `x` and `y` should be float lists having the same length. The list `x` should have the property that its entries are distinct.

The template module `ShowLS.py` comes equipped with some graphics so that you can see the quality of the line that is produced by `LSFit(x,y)`. Submit `ShowLS.py` (with your finished implementation of `LSFit(x,y)`) to CMS.

4 The Traveling Fanatic Problem

The template module `ShowTF.py` contains a function `MLB()` that returns three lists: `City`, `x`, and `y`. `City` is a list of strings that that name North American cities that have a major league baseball franchise, i.e., Anaheim, Arlington, Atlanta, Baltimore, Boston, Chicago, Cincinnati, Cleveland, Denver, Detroit, Houston, Kansas City, Los Angeles, Miami, Milwaukee, Minneapolis, New York, Oakland, Philadelphia, Phoenix, Pittsburgh, San Diego, San Francisco, Seattle, St.Louis, Tampa, Toronto, and Washington. `MLB` also returns a pair of lists `x` and `y` that contain the (x, y) coordinates of the cities. (Assuming flat earth etc.) Everything is indexed from 0 to 27.

Starting in city i, a baseball Fanatic sets out on a Grand Tour in which each city is visited exactly once before returning to city i. In an effort to minimize the total distance traveled, the Fanatic chooses the next stop according to a simple rule:

Go to the nearest unvisited city

In this problem you are to implement a function `Route(x,y,i)` that returns the itinerary (an integer list that indicates the order of the stops) and the total distance traveled. A small $n = 5$ example will clarify all.

We start out with a triplet of length-n lists:

`idx`	:	0	1	2	3	4
`x`	:	100	200	300	250	150
`y`	:	200	100	400	100	300

Suppose the Fanatic starts out in City 3. This means that the Fanatic is currently located at (250,100) and the itinerary list (so far) is `[3]`. The odometer is set to zero. We now use the list method `pop` to remove "City 3 data" from the three lists leaving us only unvisited city data:

`idx`	:	0	1	2	4
`x`	:	100	200	300	150
`y`	:	200	100	400	300

Given the “current” Fanatic location (250,100), it is determined that the Fanatic is closer to (200,100) than it is to (100,200), (300,400), or (150,300). Thus, City 1 is the next stop. The itinerary is updated to `[3,1]`, the odometer is updated, and the three lists are updated so that they only house unvisited city data:

`idx`	:	0	2	4
`x`	:	100	300	150
`y`	:	200	400	300

Given that the Fanatic is located at (200,100), it is determined that the Fanatic is closer to (100,200) than it is to (300,400), or (150,300). Thus, City 0 is the next stop. The itinerary is updated to `[3,1,0]`, the odometer is updated, and the three lists are updated so that they only house unvisited city data:

`idx`	:	2	4
`x`	:	300	150
`y`	:	400	300

Given that the Fanatic is located at (100,200), it is determined that the Fanatic is closer to (150,300) than it is to (300,400). Thus, city 4 is the next stop. The itinerary is updated to `[3,1,0,4]`, the odometer is updated, and the three lists are updated so that they only house unvisited city data:

`idx`	:	2
`x`	:	300
`y`	:	400

It follows that the last stop is city 2 so the final itinerary is `[3,1,0,4,2]`. The odometer is updated. At this point the Fanatic returns to the starting city and that requires one final update of the odometer.

When the “Grand Tour” is over, `Route(x,y,i)` returns the itinerary list, e.g., `[3,1,0,4]`, and the final odometer reading.

A template module `ShowTF.py` is provided on the website. It includes some graphics so that you can see a schematic of the journey. You are also required to print out the list of cities in the order that they are visited. Details in template module. Submit `ShowTF.py`.