Assignment 1

Logistics

You must create one Jupyter notebook containing your answers for all problems. Please create markdown cells to clearly separate your solutions to the three problems. For each problem, your answer should contain (a) functions with associated docstrings, (b) one sample execution of your code. Please make sure you don't replicate code; if you find yourself using the same code over and over again, put it in a function and call that function.

The assignment is due on **July 21, before the beginning of class**.

1 Rock, Paper, Scissors (20 points)

This is a popular game (see here), and you must create a computer player.

Game setup. Each game round consists two turns, the first by the computer and the second by a human. The computer continues playing rounds until the human chooses to quit.

- The computer chooses one of Rock, Paper, and Scissors, but keeps its choice secret.
- The computer asks for the human's input.
- The human chooses one of Rock, Paper, and Scissors, or Quit.
- Unless the human quits, the computer figures out the result of the game, as follows:
  - Rock smashes Scissors, so Rock beats Scissors.
  - Scissors can cut up paper, so Scissors beat Paper.
  - Paper covers Rock, so Paper beats Rock.

  If both players chose the same, it is a draw. The computer reports the result of this round.

- If the human chooses to quit, the computer reports:
  - the number of games played, and
  - the number of times the human won.

Computer's brains. The computer must be able to exploit some human biases. If the human has played Rock most often, the computer should assume that he or she will play Rock in the next round, so the computer should play Paper. If the human has played Rock and Paper equally often, and Scissors less often, the computer should assume that the human is going to play either Rock or Paper (both equally likely) in the next round. (What should the computer play?)

Hence, **your program should remember how many Rock, Paper, or Scissors were played by the human.** Note that we don't need to remember the order in which the human chooses these; the total counts so far for each choice will be enough.

Gotchas.

- *User input:* How you want to receive the user's input is up to you, but you *must* check the user's input to make sure it is valid (you can assume that the user input is of the correct type). If it isn't, request the user for input again.

2 Voters in Florida (20 points)

The file `FloridaVoters.html` contains a Web Table of republican and democratic voters in various counties in Florida. Write code that reads in this file **as a standard text file** and prints out the counties, along with the number of republican and democratic voters in those counties, *sorted by the number of democratic voters.* The output should look like this:

```
LAFAYETTE 1373 2672
GLADES 2190 3110
LIBERTY 720 3372
...
MIAMI-DADE 362161 539367
BROWARD 249762 566185
Total 4377713 4637026
```

Note that the numbers in the HTML file contain commas, but we got rid of them in order to do the sorting. Also, while we should technically ignore the data for `Total`, let's not worry about it here.

Please note: There are many libraries for automatically reading and parsing HTML. **You should not use them.** You must read in the HTML file as a plain text file, and figure out the correct regular expressions to extract the county names and the numbers of voters.

Hint: You may want to read in the file and create a list of tuples of the form `[('ALACHUA', 47329, 77996), ('BAKER', 6963, 5813), ...]` and then do the sorting and printing.

3 Near-duplicate detection (20 points)

Tweets on a subject are often nearly (but not exactly) duplicates of each other. The file `Santa.txt` contains a few tweets about Santa Claus, one tweet per line. We will try to detect tweets that we have already seen before.

(a) Convert each tweet into a dictionary of phrases (6 points). Write a function called `moving_window` that takes as input a tweet (that is, a string), and converts it into a bunch of phrases. Each phrase is three consecutive words in the tweet. For example, the tweet ``This is an awesome tweet, dude'' consists of the phrases `` This is an'', ``is an awesome'', ``an awesome tweet,'', and ``awesome tweet, dude''. Return a dictionary whose keys are these phrases, and just set the corresponding values to 1. This dictionary contains all the unique 3-word phrases in the tweet.

(b) Calculate similarity between two tweets (6 points). To check if one tweet is a near-duplicate of another, we compute their *cosine similarity*:

$$\textit{cosine(tweet1, tweet2)} = \frac{\textit{matches}}{\sqrt{n_1 * n_2}},$$

where *matches* is the number of 3-word phrases in common between the two tweets, and n_1 and n_2 are the number of phrases in the two tweets respectively.

Write a function called `cosine` that takes as input two dictionaries. Each dictionary contains the 3-word phrases from one tweet. Return the cosine similarity between the phrases in the two dictionaries.

Hint: You can use the function `math.sqrt(x)` to calculate the square-root of any floating point number x, but you must import the math module.

(c) Read in tweets, and output near-duplicates (8 points). Read in the tweets in the file `Santa.txt`. For each tweet, figure out if it is a near-duplicate of any of the previously-seen tweets. We say that the two tweets are near-duplicates if their cosine similarity is greater than 0.5.

You should call the functions `moving_window` and `cosine` here.

4 The Google of Quotes (40 points)

The file "quotes.txt" contains pairs of lines, with the first line being a quote and the following line being the person who said it. We want to build a search engine that, given a word or words, finds the best matching quotes.

(a) Build a list of full quotes (5 points). Read in the file, and create a list of *full quotes* of the form "quote - speaker". For example, *"The heart has its reasons, of which the mind knows nothing. - Blaise Pascal"*.

(b) Words from full quotes (5 points). Write a function that takes a full quote as argument and outputs a list of the words in the it. The words should all be lower-case, and should contain only characters, digits, or underscore.

Hint: Use the `lower()` function of strings, and `re.split()` to split into words, but you must figure out the regular expression for `re.split()`.

(c) Build the postings-list dictionary (6 points). A **postings-list** is a dictionary whose keys are full quotes, and whose values are themselves dictionaries with key being a word, and value being the number of times the word occurs in the full quote. So, for the key *"The heart has its reasons, of which the mind knows nothing. - Blaise Pascal"*, the value will itself be the following dictionary: {`'the':2, 'heart':1, 'has':1, 'its':1, 'reasons':1, 'of':1, 'which':1, 'mind':1, 'knows':1, 'nothing':1, 'blaise':1, 'pascal':1`}.

(d) Build the reverse postings-list dictionary (6 points). A **reverse postings-list** is a dictionary whose keys are the words, and the values are themselves dictionaries with the key being a full quote, and the value being the number of times the word appeared in the full quote.

So, for the key *"entertainer"*, the value is the dictionary `{'An actor is at most a poet and at least an entertainer. - Marlon Brando': 1}` (only this quote contains the word "entertainer").

(e) Write a TF-IDF function (8 points). To measure how much a full quote is *about* a particular word, one typically uses the TF-IDF measure.

- TF stands for "term frequency"; the term frequency of a word w in a full quote q is the number of times w occurs in q, divided by the maximum number of times *any* word occurs in q.
- IDF stands for "inverse document frequency": the IDF of a word w is the logarithm of the ratio of the total number N of full quotes to the number of full quotes in that contain the word w.
- TF-IDF of a word w for a full quote q is just the product of the TF and IDF.

So, for the word *"entertainer"* in the Marlon Brando quote of part (d):

- The TF is 0.5 (it occurs once, while the most frequent word in that quote is "at", which occurs twice, so the TF ratio is 0.5)
- The IDF is $\log(886/1)$, since there are 886 documents and the word *"entertainer "* occurs in only one full quote.

Write a function to compute the TF-IDF of any word in any full quote, using the postings and reverse-postings.

Hint: Do *import math* and use *math.log()* to get logarithms.

(f) Quote search using a single word (5 points). Write a function that takes a word as argument, and returns a dictionary whose keys are full quotes containing that word, and whose values are the TF-IDF score of that word for that full quote.

(g) Quote search using multiple words (5 points). Write a function that takes a *list* of words as argument, and returns a dictionary whose keys are full quotes containing one or more of the words in that list, and whose values are the *sum of TF-IDF scores* of the words in that list for that full quote.

For example, if you called the function with the list of words `[heart, mind, disease]`, and you have a full quote *"The heart has its reasons, of which the mind knows nothing - Blaise Pascal"*, then you want to compute the sum of TF-IDF scores of `heart` and `mind` for this full quote.