1. Warmups

Put all of these warmup problems in a single Python file called `warmups.py` and demonstrate that each problem works by testing them in a `main` function. You should call your `main` function using the following pattern which will allow you to import this file in Part 2 of the lab.

```python
if __name__ == '__main__':
    main()
```

The `main` function should be the only place where you have `print` statements.

(a) Write a function called `getWords` that takes a string `s` as its only argument. The function should lower-case the words in `s` and return a list of the words in the same order as they appeared in `s`. Note that in this question a "word" is defined as a "space-separated item". For example:

```python
>>> getWords('The cat in the hat ate the rat in the vat.')
['the', 'cat', 'in', 'the', 'hat', 'ate', 'the', 'rat', 'in', 'the', 'vat.'])
```

*Hint:* If you don’t know how to approach this problem, you should read about `str`’s `split` method: https://docs.python.org/2/library/stdtypes.html#str.split

(b) Write a function called `countWords` that takes a list of words as its only argument and returns a dictionary whose keys are the words in the word list and whose values are the frequencies with which each word occurs in `s`. Use the output of the `getWords` function as the input to this function.

```python
>>> getWords('The cat in the hat ate the rat in the vat.')
['the', 'cat', 'in', 'the', 'hat', 'ate', 'the', 'rat', 'in', 'the', 'vat.']
>>> words = getWords('The cat in the hat ate the rat in the vat.')
>>> countWords(words)
{'ate': 1, 'cat': 1, 'rat': 1, 'in': 2, 'the': 4, 'hat': 1, 'vat.': 1}
```

*Hints and notes:*

- If you don’t have experience using dictionaries, read about Python’s `dict` class: https://docs.python.org/2/library/stdtypes.html#mapping-types-dict
- If you do have experience using dictionaries and you may be interested in learning about `defaultdict`, an advanced kind of dictionary that can simplify your code: https://docs.python.org/2/library/collections.html#defaultdict-objects
- There is also a `Counter` class that does all the work for you; however, I wouldn’t recommend using this in your solution unless you’re confident in your ability to use dictionaries as you’ll be using them a lot in this class.
  https://docs.python.org/2/library/collections.html#counter-objects

(c) Write a function called `wordsByFrequency` that takes a list of words as its only argument. The function should return a list of (word, count) tuples sorted by count such that the first item in the list is the most frequent item. The order of items with the same frequency does not matter (but you could try to sort such items alphabetically if you were so inclined). To sort a list of tuples by the second field, use the function `itemgetter` that is part of the `operator` module:

```python
>>> from operator import itemgetter
>>> tupleList = [('rich', 9), ('andy', 5), ('tia', 12)]
>>> tupleList.sort(key=itemgetter(1), reverse=True)
```
>>> tupleList
[('tia', 12), ('rich', 9), ('andy', 5)]

>>> words = getWords('The cat in the hat ate the rat in the vat.')
>>> wordsByFrequency(words)
[('the', 4), ('in', 2), ('ate', 1), ('cat', 1), ('hat', 1), ('rat', 1), ('vat.', 1)]

If you’re interested, you can read more about the itemgetter function here:
http://docs.python.org/library/operator.html

2. Using NLTK

In this part you be using NLTK (http://nltk.org/), an open-source toolkit for natural language processing. Put your code for this part in a file called zipf.py. To use NLTK, you need to add this to the top of your code:

```python
import nltk
```

Important note: NLTK works best under Python 3. You have almost certainly only used Python 2. For the purposes of this class, you will find that there are two big hurdles you will face in transitioning from Python 2 and Python 3:

- You will run python – which is Python 2 – instead of python3.
- print is a function, which impacts the syntax of print statements. You now need to wrap parenthesis around whatever it is you’re printing, just as you would for any other function So:

```python
#python 2
print "Hello, World."
print          # an empty line
print "Put a comma to prevent",
print "going to the next line"

#python 3
print("Hello, World.")
pprint()    # an empty line
print("Add a parameter to prevent", end=' ')  
print("Going to the next line")
```

OK, back to NLTK (running under Python 3, right?):

(a) This is not a question and there’s nothing to turn in, but you shouldn’t skip it. Read/skim chapter 1 and section 2.1 of the NLTK book. While the book does spend a good bit of time teaching basic Python in the first chapter, there’s a lot of useful information there even if you already know Python. The only area of Python you might be wholly unfamiliar with is “list comprehensions” which the books makes use of a lot. Try and follow along with the examples even if you don’t feel comfortable writing these expressions yourself.

(b) (Nothing to turn in here either, just verify that this works before continuing to the next part.) Use the code from Part 1 above to show the five most frequent ‘words’ in Alice in Wonderland:

```python
from warmups import wordsByFrequency, countWords
words = nltk.corpus.gutenberg.words('carroll-alice.txt')
```
Note that although the type of the \texttt{words} object returned is not a list, it can be iterated over using the same \texttt{<for item in list>} pattern. Test that the \texttt{countWords} and \texttt{wordsByFrequency} function work properly using these \texttt{words} as input. Assuming you wrote \texttt{warmups.py} correctly, you should find that the five most frequent ‘words’ in the text are:

\begin{itemize}
\item 1993 ← a comma
\item 1731 ← an apostrophe
\item the 1527
\item and 802
\item . 764 ← a period
\end{itemize}

(c) Iterate over all of the Gutenberg texts included in NLTK and examine the top 5 words in each. Exclude anything that isn’t a word by using Python’s \texttt{isalpha()} method on strings. You can get a list of all the corpora in the Gutenberg collection in NLTK by using the \texttt{fileids()} method:

\begin{verbatim}
fileids = nltk.corpus.gutenberg.fileids()
\end{verbatim}

Is ‘the’ always the most common word? If not, what are some other words that show up as the most frequent word? If you lowercase all the words before you count them, how does this result change, if at all?

(d) In the output above for Alice in Wonderland’s most frequent 5 words, we say that \texttt{comma} has rank of 1, \texttt{apostrophe} has rank of 2, ‘\texttt{the}’ has a rank of 3, and so on. Let $f$ be the relative frequency of the word (e.g. ‘\texttt{the}’ occurs 1527 times out of 34110 word \textit{tokens} so it’s relative frequency is $1527/34110 = 0.0448$) and $r$ be the rank of that word.

To visualize the relationship between rank and frequency, we will create a log-log plot of rank (on the $x$-axis) versus frequency (on the $y$-axis). We will use the \texttt{pylab} library (http://matplotlib.sourceforge.net/):

\begin{verbatim}
import nltk
import pylab
from warmups import wordsByFrequency, countWords

words = nltk.corpus.gutenberg.words('carroll-alice.txt')
counts = wordsByFrequency(words)
n = len(counts)

ranks = range(1, n+1)  #x-axis: the ranks
freqs = [freq for (word, freq) in counts]  #y-axis: the frequencies

pylab.loglog(ranks, freqs, label='alice')  #this plots freq, not relative freq
pylab.xlabel('log(rank)')
pylab.ylabel('log(freq)')
pylab.legend(loc='lower left')
pylab.show()
\end{verbatim}

(e) (Combine your answer to this part with the code above.) In this question we will test how well Zipf’s law works. Read section 1.4.3 in Manning and Schütze\textsuperscript{2} for an explanation of Zipf’s law. In summary, Zipf’s law states that $f \propto \frac{1}{r}$ or, equivalently, that $f = \frac{k}{r}$ for some constant factor $k$, where $f$ is the frequency of the word and $r$ is the rank of the word. Following Zipf’s law, the 50th most common word should occur with three times the frequency of the 150th most common word.

---

\textsuperscript{2}urlhttp://cognet.mit.edu/library/books/mitpress/0262133601/cache/chap1.pdf
Plot the empirical rank vs frequency data (as we just did above) and also plot the expected values using Zipf’s law. For the constant \( k \) in Zipf’s law, you should use \( \frac{1}{H(n)} \) where \( n \) is the number of word types in the corpus and \( H(n) \) is the \( n^{th} \) harmonic number. Use this function\(^3\) to compute harmonic numbers:

```python
import math
def H_approx(n):
    """
    Returns an approximate value of n-th harmonic number.
    http://en.wikipedia.org/wiki/Harmonic_number
    """
    # Euler-Mascheroni constant
gamma = 0.57721566490153286060651209008240243104215933593992
    return gamma + math.log(n) + 0.5/n - 1/(12*n**2) + 1/(120*n**4)
```

To plot a second curve, simply add a second `pylab.loglog(...)` line immediately after the line shown in the example above.

How well does Zipf’s law approximate the empirical data in Alice in Wonderland? How many words (in total, not unique words - also called tokens) does Alice in Wonderland have? Repeat the experiment for a few other texts included in NLTK’s Gutenberg corpus. You can also combine the words from all of the texts in NLTK’s Gutenberg corpus to get a much larger corpus by simply omitting the filename:

```python
words = nltk.corpus.gutenberg.words()
```

How many tokens does this larger corpus have? Repeat the experiment from the previous question using this larger corpus. How does this plot compare with the plots from the smaller corpora? Does Zipf’s law hold for each of these?

(f) Question 23 in Chapter 2 of the NLTK book\(^4\) focuses on Zipf’s law. In the previous two questions you answered 23(a). Implement a solution to 23(b), plotting your answer. Analyze (in a few lines, adding to your comment from the previous question) the resulting plot and what this says about Zipf’s law.

To generate about the same number of tokens as there are in Alice in Wonderland, generate about 300,000 random characters. To generate about the same number of tokens as there are in the whole Gutenberg corpus, generate about 20,000,000 random characters. If you implemented things inefficiently, this could take a while.

3. “i” before “e” except after “c”

In class, we talked about the “i before e” rule and how statistics can help us determine how good a ‘rule’ it really is. Here, we’ll begin by repeating our in-class experiment with the Gutenberg corpus. We’ll then use `pylab` to make some graphs. You should stick with Python to solve this: don’t use command-line tools like `grep` and `wc`. Save your answers in `ibeforee.py`.

In texts, many words will occur multiple times. For example, “friend” occurs 706 times in the Gutenberg corpus. If I ask you to count types, then you should count unique words, and “friend”, despite occuring 706 times, would only count as 1 word. If I ask you to count tokens, then you should count each occurrence of the word, so “friend” would be counted 706 times.

(a) Repeat this for types and then tokens using the Gutenberg text and discuss the results.

i. How frequent is “cie” relative to all “ie” words?

\(^3\)http://stackoverflow.com/questions/484346/python-program-to-calculate-harmonic-series

\(^4\)http://www.nltk.org/book/ch02.html#exercises
ii. How frequent is “cei” relative to all “ei” words?

iii. Compute the fraction of the words that contain “cei” relative to all “cei” and “cie” words by computing \( \frac{\text{count(cei)}}{\text{count(cie)} + \text{count(cei)}} \). This most directly answers the question “after a ‘c’, what’s more likely, ‘ie’ or ‘ei’?”

(b) In class, we suggested omitting words that contained “eigh”, and we suggested removing words ending in “ied”, “ies”, “ier”, and “iest”. Remove those words and redo the previous question. How does this change your results?

(c) Is “i before e except after c” a good rule? Perhaps there’s a better rule, like “i before e except after h”? Repeat the previous question, substituting the letter ‘c’ with all the possible letters. Report the relevant results and discuss.

Use `for letter in "abcdefghijklmnopqrstuvwxyz":` to iterate through all the possible letters.

(d) Experiment! Try something cool that I didn’t mention. This is optional, but it’s fun, so keep going! Or, for fun, try writing it all in a bash script like the `i_before_e.txt` file in the `cs65/class/09-04/` directory. Feel free to use any scripting tools you know (e.g. perl, sed, awk, grep, wc, bc).