Project 1: Counts

Due: Wednesday Sep. 16th, 11:59am (Eastern Time)

Deliverables

- Three python files: `tokenizer.py`, `zipf.py`, and `ibeforee.py`
- `Writeup.md` with answers to all 12 questions from all 4 sections.

The answers to each of the questions in the lab writeup should be placed in a markdown file called `Writeup.md`. Read about **markdown syntax** if you're not familiar. You may want to use **atom** or **VS Code** to edit your markdown files since there is a built-in markdown previewer built into both of these editors.

Note though, that while your answers should be written to `Writeup.md`, you're code needs to print to `stdout` as well so that your writeup file is easily verifiable.

Python3

In this class, we're going to use Python 3 and not Python 2. Depending on when/where you took intro CS, it's possible you only know Python 2.

For the purposes of this class, you will find that there are a couple habits you will need to break or get into when transitioning from Python 2 and Python 3:

- You will type `python3` at the command line. Typing `python` on the department machines will run Python 2.
- In Python 3, `print` is a function. This means that the syntax of print statements now requires that you use parenthesis just as you would for any other function.

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

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

1 Start up Functions

Put all of your code in a single Python file called `tokenizer.py` and demonstrate that your functions work by writing 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.

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

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

1.1 get_words

Write a function called `get_words` that takes a string *s* as its only argument. The function should 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:

```
>>> get_words('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, read about **str.split()**.

1.2 count_words

Write a function called `count_words` that takes a list of words as its only argument and returns a dictionary that maps a word to the frequency that it occurred in the source string *s*. Use the output of the `get_words` function as the input to this function.

```
>>> s = 'The cat in the hat ate the rat in the vat'
>>> words = get_words(s)
>>> count_words(words)
{'The': 1, 'cat': 1, 'in': 2, 'the': 3, 'hat': 1, 'ate': 1, 'rat': 1, 'vat': 1}
```

Notice that this is somewhat unsatisfying because `the` is counted separately from `The`. We can easily fix this by lower-casing all of the words before counting them using **str.lower()**

```
>>> words = get_words(s.lower())
>>> count_words(words)
{'the': 4, 'cat': 1, 'in': 2, 'hat': 1, 'ate': 1, 'rat': 1, 'vat': 1}
```

Note: Be sure that you are confident in your ability to use dictionaries in Python since you'll be using them a lot in this class. If you don't have experience using dictionaries, read about Python's **dict** data structure. If you do feel comfortable with Python dicts, try using **collections.defaultdict**, a slightly more advanced kind of dictionary that can simplify your code.

1.3 words_by_frequency

Write a function called `words_by_frequency` 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!).

```
>>> words_by_frequency(words)
[('the', 4), ('in', 2), ('cat', 1), ('hat', 1), ('ate', 1), ('rat', 1), ('vat', 1)
    ]
```

Hint: To sort a list of tuples by the second field, use the **operator.itemgetter** function.

```
>>> import operator
>>> tupleList = [('cat', 9), ('hat', 5), ('rat', 12)]
>>> tupleList.sort(key=operator.itemgetter(1), reverse=True)
>>> tupleList
[('rat', 12), ('cat', 9), ('hat', 5)]
```

2 Tokenization

In this part of the lab, you will explore some files from **Project Gutenberg** and improve on the code you just wrote by adding to and/or modifying the tokenizer.py program you began in the previous section.

2.1 Alice in Wonderland

In the `/data/cs65-F20/gutenberg` directory there are a number of `.txt` files containing texts found in the Project Gutenberg collection. Read in Lewis Carroll's *"Alice's Adventures in Wonderland"*, which is stored in the file `/data/cs65-F20/gutenberg/carroll-alice.txt`. Use your `words_by_frequency` and `count_words` functions from Part 1 to explore the text. Assuming that you wrote your code properly in Part 1, and assuming that you lower-cased all of the words in the text, you should find that the five most frequent words in the text are:

```
the      1603
and       766
to        706
a         614
she       518
```

Questions

1. Use your `count_words` function to find out how many times the word `alice` occurs in the text.
2. The word `alice` actually appears 398 times in the text, though this is not the answer you got for the previous question. Why?

Examine the data to see if you can figure the answer to Q.2 before continuing.

2.2 Splitting Punctuation

We now see that there is a deficiency in how we implemented the `get_words` function. When we are counting words, we probably don't care whether the word was adjacent to a punctuation mark. For example, the word `hatter` appears in the text 57 times, but if we queried the `get_words` dictionary, we would see it only appeared 24 times. However, it also appeared numerous times adjacent to a punctuation mark so those instances got counted separately:

```
>>> word_freq = words_by_frequency(words)
>>> for (word, freq) in word_freq:
...     if 'hatter' in word:
...         print('%-10s %3d' % (word, freq))
...
hatter      24
hatter.     13
hatter,     10
hatter:      6
hatters      1
hatter's     1
hatter;      1
hatter.'     1
```

Our `get_words` function would be better if it separated punctuation from words. We can accomplish this by using the **re.split** function. (We will talk more about regular expressions in class.) Be sure to **import re** at the top of your file to make `re.split()` work. Below is a small example that demonstrates how `str.split` works on a small text and compares it to using `re.split`:

```
>>> text = '"Oh no, no," said the little Fly, "to ask me is in vain."'
>>> text.split()
['"Oh', 'no,', 'no,"', 'said', 'the', 'little', 'Fly,', '"to', 'ask', 'me', 'is',
 'in', 'vain."']
>>> re.split(r'(\W)', text)
['', '"', 'Oh', ' ', 'no', ',', '', ' ', 'no', ',', '', '"', '', ' ', 'said', ' ',
    'the',
 ' ', 'little', ' ', 'Fly', ',', '', ' ', '', '"', 'to', ' ', 'ask', ' ', 'me', '
   ', 'is',
 ' ', 'in', ' ', 'vain', '.', '', '"', '']
```

Note that this is not exactly what we want, but it is a lot closer. In the resulting list, we find empty strings and spaces, but we have also successfully separated the punctuation from the words. Using the above example as a guide, write and test a function called `tokenize` that takes a string as an input and returns a list of words and punctuation, but not extraneous spaces and empty strings. You don't need to modify the `re.split()` line: just process the resulting list to remove spaces and empty strings.

```
>>> tokenize(text.lower())
['"', 'oh', 'no', ',', 'no', ',', '"', 'said', 'the', 'little',
 'fly', ',', '"', 'to', 'ask', 'me', 'is', 'in', 'vain', '.', '"']
>>> print(' '.join(tokenize(text.lower())))
```

```
" oh no , no , " said the little fly , " to ask me is in vain . "
```

2.3 Testing new tokenize

Use your `tokenize` function in conjunction with your `count_words` function to list the top 5 most frequent words in `carroll-alice.txt`. You should get this:

```
’         2871      <-- single quote
,         2418      <-- comma
the       1642
.          988      <-- period
and        872
```

2.4 filter_nonwords

Write a function called `filter_nonwords` that takes a list of strings as input and returns a new list of strings that excludes anything that isn't entirely alphabetic. Use the **str.isalpha()** method to determine if a string is comprised of only alphabetic characters.

```
>>> text = ‘"Oh no, no," said the little Fly, "to ask me is in vain."’
>>> tokens = tokenize(text)
>>> filter_nonwords(tokens)
[‘Oh’, ‘no’, ‘no’, ‘said’, ‘the’, ‘little’, ‘Fly’, ‘to’, ‘ask’, ‘me’,
 ‘is’, ‘in’, ‘vain’]
```

Use this function to list the top 5 most frequent (actual) *words* in `carroll-alice.txt`:

```
the       1642
and        872
to         729
a          632
it         595
```

2.5 Counting within a corpus

Iterate through all of the files in the `/data/cs65-F20/gutenberg` directory and print out the top 5 words for each. To get a list of all the files in a directory, use the **os.listdir** function:

```
>>> import os
>>> directory = '/data/cs65-F20/gutenberg/'
>>> files = os.listdir(directory)
>>> files
['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', 'bible-kjv.txt',
 'blake-poems.txt', 'bryant-stories.txt', 'burgess-busterbrown.txt',
 'carroll-alice.txt', 'chesterton-ball.txt', 'chesterton-brown.txt',
 'chesterton-thursday.txt', 'edgeworth-parents.txt', 'melville-moby_dick.txt',
 'milton-paradise.txt', 'README', 'shakespeare-caesar.txt',
 'shakespeare-hamlet.txt', 'shakespeare-macbeth.txt', 'whitman-leaves.txt']
>>> infile = open(os.path.join(directory, files[0]), 'r', encoding='latin1')
```

This example also uses the function **os.path.join** that you might want to read about.

A note about encodings: This `open` function above uses the optional encoding argument to tell Python that the source file is encoded as `latin1`. We will talk more about encodings later in class, but be sure to use this `encoding` flag to read the files in the Gutenberg corpus.

Questions

3. Loop through all the files the `/data/cs65-F20/gutenberg` directory that end in .txt. Is 'the' always the most common word? If not, what are some other words that show up as the most frequent word?
4. If you don't lowercase all the words before you count them, how does this result change, if at all?

3 Zipf's Law

In this part of the lab, we will test Zipf's law by using **matplotlib**, a 2D plotting library. Put your code for this part in a new file called `zipf.py`.

3.1 Tabulating the Frequency v. Rank relationship

In this part of the lab, we will be exploring the relationship between the frequency of a token and the *rank* of that token. If you count all the tokens in `carroll-alice.txt` — like we did in Part 2.3 above — we would say that single quote has rank of 1, comma has rank of 2, `the` has a rank of 3, and so on, since this is the order that they appeared when we listed the tokens from most frequent to least frequent.

Be sure that you are using your `tokenize` function from above and that you **are not** calling your `filter_nonwords` function but that you **are lowercasing** the text. (If you don't lowercase, the results will be slightly different, but you will reach the same conclusions.)

Let f be the relative frequency of the word (e.g. assuming you lowercased the text, `the` occurs 1642 times out of 35652 tokens so its relative frequency is $1642/35652 = 0.04606$) and r be the rank of that word.

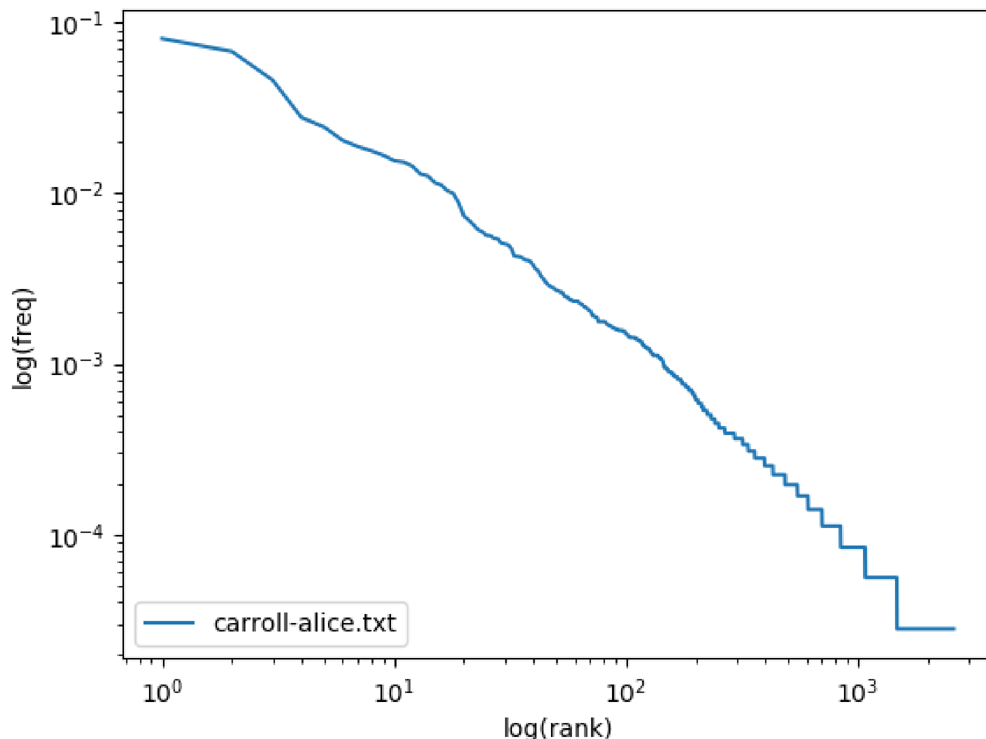
To visualize the relationship between rank and frequency, we will create a **log-log plot** of rank (on the x-axis) versus frequency (on the y-axis). We will use the `pylab` library, part of `matplotlib`:

```
from matplotlib import pylab
from tokenizer import tokenize, words_by_frequency, count_words

text = open('/data/cs65-F20/gutenberg/carroll-alice.txt', 'r', encoding='latin1').
    read()
words = tokenize(text.lower())
```

```
counts = words_by_frequency(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 frequency, not relative
    frequency
pylab.xlabel('log(rank)')
pylab.ylabel('log(freq)')
pylab.legend(loc='lower left')
pylab.show()
```

If all goes according to plan your output should look like this:



3.2 Testing Zipf's Law

Now we can test how well Zipf's law works. Read the **Wikipedia's article on Zipf's law**. In summary, Zipf's law states that $f = \frac{k}{r}$ for some constant factor k, where f is the frequency of a word and r is its rank. Following Zipf's law, the 50th most common word should occur with three times the frequency of the 150th most common word. 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 the formulation of Zipf's law above, you should use $\frac{T}{H(n)}$ where T is the number of word *tokens*

in the corpus, n is the total number of word *types* in the corpus, and $H(n)$ is the nth **harmonic number**.

Use **this function to compute harmonic numbers**:

```
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.

Questions

5. How well does Zipf's law approximate the empirical data in `carroll-alice.txt`? How many words (tokens, not types) does `carroll-alice.txt` have?
6. Repeat the last question but for two other texts in the `/data/cs65-F20/gutenberg` directory — which texts you pick is up to you.
7. In your `tokenizer.py` file, add a function called `all_files()` that takes a directory as its only parameter and returns a string containing all of the `.txt` files in that directory concatenated together. You should use code like you wrote in Part 2.5 to iterate through all of files in order to form one large string.
   (a) Use the `all_files()` function to read in all of the texts and repeat the Zipf's law experiment with this larger corpus.
   (b) How many tokens are in this new corpus?
   (c) How does this plot compare with the plots from the smaller corpora?
8. Does Zipf's law hold for each of the plots you made? What intuitions have you formed?
9. Take a look Python's **random.choice()** function. Use it to generate random 'English' text (e.g. a call like `random.choice(‘‘abcdefg...  ’’)` should work simply, but be careful to include the space character). Use the string concatenation operator to accumulate characters into a (very) long string. Then tokenize this string, and generate a Zipf plot like before. Compare the output of actual English and Random 'English'. What do you make of Zipf's Law in light of this?

4 "i before e"... sort of

There is a common adage about English spelling that states that if you don't know whether a word is spelled with 'ie' or 'ei,' that you should use 'ie' unless the immediately preceding letter is 'c.' (If you didn't grow up in a majority English speaking country, you may not have heard this before. In which case you can read about the **'i' before 'e' except after 'c' rule here**). Collecting statistics from text corpora can help us determine how good a 'rule' it really is. We'll begin by experimenting with the files in `/data/cs65-F20/gutenberg` and use `matplotlib` to make some graphs. Save the code you write to explore these things in `ibeforee.py`.

Read in all of the texts from `/data/cs65-F20/gutenberg` into a single corpus and we'll call this combination of files the Gutenberg corpus.

Questions

10. Repeat these four parts for types and then for tokens using the Gutenberg corpus.
    (a) How frequent is 'cie' relative to all 'ie' words?
    (b) How frequent is 'cei' relative to all 'ei' words?
11. The 'i before e' rule has some exceptions. Recompute your answers to the question above, but don't count any word that has 'eigh', 'ied', 'ier', 'ies', or 'iest'. How does this change your result?
12. Is *"i before e except after c"* a good rule in practice? Perhaps there's a better rule, like *"i before e except after h"*? Try substituting 'c' with all possible letters. Report the relevant results and discuss.

5 Bonus Ideas?

Did you think of any cool extensions or new questions while working on this lab?

- Different ways of deriving Zipf's law? Implement a statistical rather than visual evaluation of Zipf's law?
- Another way to visualize the text we're working with?
- Other spelling conventions or prescriptions that you might test or evaluate?
- What does the distribution of word lengths look like?
- Could you implement some or all of the lab (minus the plots) purely in a scripting language like Perl or Bash?

Be creative and have fun: if you think of something you find interesting, go ahead and implement it and include it as a bonus section at the end of `Writeup.md`. If I think it's interesting too then you can get up to 10% extra credit on the Lab.