1. (0% but required) If it is not blatantly obvious, please indicate where in your source code parts A and B exist.

Parts A and B can be run by running the scripts part-a-runner.py and part-b-runner.py respectively.

The tokenizer process lives on processor.py, although abbreviating is implemented in tokenizer.py to allow unit testing.

The stopper is also implemented in stopper.py.

The stemmer is also implemented in stemmer.py, and it is divided into two methods: stem_a and stem_b, each respectively handling step 1a and 1b.

Overall, the entire work is done on processor object initialization. I did not split tokenzing/stopping/stemming into separate loops/processes, as it made sense to stop and stem the words individually right after tokenizing to save iterations over the resulting tokenized array, stopped array, etc.

Basically, read --> read line --> lowercase --> apostrophe --> split --> (on individual words: abbreviate --> stop --> stem --> add to data).

2. (10%) Description of system, design tradeoffs, etc.

The general system is run from two individual scripts: part-x-runner.py. This was to try and keep most of the code the same for both parts, and be able to handle part-specific data processing separate from the processor script. This can be seen in the fact that part-a-runner.py only writes to file, while part-b-runner.py writes to file, and also utilizes returns from Processor to create a graph.

The stemmer and stopper were written in different classes, while most of tokenizer was not. This was to avoid unnceessary processing time used up for iterating on tokenized words again. For example, if I completely tokenized the text first, I would run the stopper across all tokenized words, then stemmer on the array again. Although the size of the arrays would likely decrease after each step, it's still 3x traversals. Instead, what was done is stopping and stemming individual words after splitting/abbreviating within the tokenization process. This doesn't break the order as all words stopped/stemmed are in the tokenize-->stop-->stem order, and it only requires running through the strings once.

I made heavy use of regexes for their convenience. The tradeoff for this was a slower runtime for stemming, because as shown in the section below, regex takes a lot longer to process strings than manual matching via string indexes, and even functions such as .endswith. Given more time, I probably would try to figure out ways to avoid using regexes, which would make the code a bit messier, but faster.

For time performance reasons, for every regex, I tried to detect some aspects of the string (e.g., ends in eed, eedly, etc) before running the regex, so that I could avoid running the regex on strings that can easily be shown to not match the regex. You can see this in parts of my stemmer.py.

Timing examples

Difference between regex and .endswith

Example on time difference:

```
py -3.9 -mtimeit -s"import re" "re.match('^[a-z]+(at|bl|iz)$','fished')"
500000 loops, best of 5: 504 nsec per loop

λ py -3.9 -mtimeit -s"'fished'.endswith('at') or 'fished'.endswith('bl') or 'fished'.endswith('iz')"
50000000 loops, best of 5: 5.78 nsec per loop
```

Does the same thing, but much faster to do with endswith.

Basically, all the things done with .endswith were things that are easily done that way. Others can likely be done, but it is a TODO for when project is completed for further optimization

Difference between .endswith and direct index comparisons of strings

Example on time difference between using .endswith and string index for last char:

```
λ py -3.9 -mtimeit -s"import re" "s = 'example'" "if s.endswith('s'): s=s[0:-1]"
5000000 loops, best of 5: 85 nsec per loop

λ py -3.9 -mtimeit -s"import re" "s = 'example'" "if s[len(s)-1] == 's': s=s[0:-1]"
5000000 loops, best of 5: 70.1 nsec per loop
```

Example 2

```
λ py -3.9 -mtimeit -s"import re" "token = 'example'" "if token.endswith('ies') or token.endswith('ied'): token = token[0:-3] + ('i' if len(token) > 4 else 'ie')"
2000000 loops, best of 5: 144 nsec per loop

λ py -3.9 -mtimeit -s"import re" "token = 'example'" "d = len(token)" "if token[d-3:d-2] == 'ie' and (token[d-1] == 'd' or token[d-1] == 's'): token = token[0:-3] + ('i' if len(token) > 4 else 'ie')"
5000000 loops, best of 5: 87.8 nsec per loop
```

Example 3

```
λ py -3.9 -mtimeit -s"import re" "token = 'examplified'" "if token.endswith('ies') or token.endswith('ied'): token = token[0:-3] + ('i' if len(token) > 4 else 'ie')"
1000000 loops, best of 5: 259 nsec per loop

λ py -3.9 -mtimeit -s"import re" "token = 'examplified'" "d = len(token)" "if
```

```
token[d-3:d-2] == 'ie' and (token[d-1] == 'd' or token[d-1] == 's'): token =
token[0:-3] + ('i' if len(token) > 4 else 'ie')"
5000000 loops, best of 5: 87.7 nsec per loop
```

3. (5%) List the software libraries you used, and for what purpose.

The libraries used were collection, re, and matplotlib.pyplot.

Collection was used for the Counter data structure, which was used to count up the # of individual vocabulary occurrences in an efficient way, and to be able to easily fetch the top 300 terms.

re was used for regexing to detect if terms matched the condition for stemming.

matplotlib.pyplot was used to graph the total vocab - unique vocab graph.

4. (10%) Based on your experience, list two changes you might make to the tokenization or stemming rules to improve the output.

One change I would make is to remove single letter terms, as they pretty much provide no value. For example, tokenized.txt contains one term "b" originating from b), which should be removed for our purposes. However, I can see that that info may have some significance / use in other, more detailed processing-- in which case, we can do more to take that into account.

Another thing is more specific changes with -ed, -ing words. With our rules, we change "sourced" to "sourc", then do nothing with it as it doesn't fit any of the other conditions we check for further processing. This compares to "source" which stays as it is for terms. This inflates the number of vocabulary and reduces the effectiveness for searching (as idential terms are identified as two separate ones), and we should improve stemming to deal with this.

5. (10%) Figure 4.4. in the textbook (pp. 82) displays a graph of vocabulary growth for the TREC GOV2 collection. Create a similar graph for Moby-Dick.

Graph of Collection Vocabulary - Unique Vocabulary