Computer Science I — CSCI-141
Calculating Text Statistics — Project

v1.11 11/07/2014

1 Cheat Checking

This assignment is far more significant than an individual lab and is worth 10% of your total grade in the course. It will take an average student an estimated 10+ hours to complete this assignment. Due to the scope and importance of this project, all students' code across all sections will be run with a cheat checker.

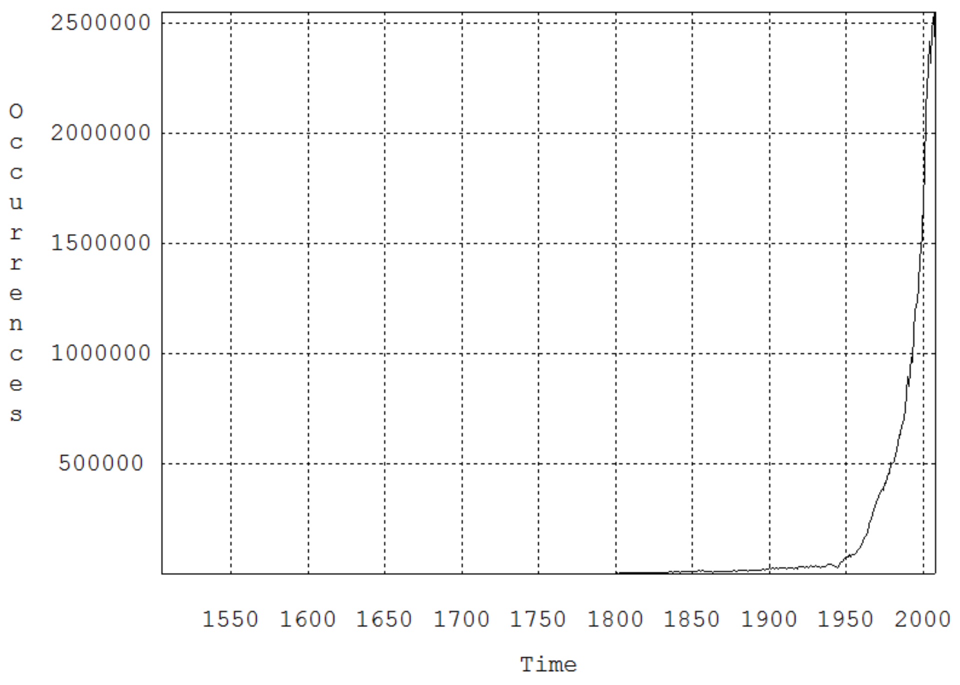
Do not supply any code you write to any other students, and likewise do not accept code from any other student. Failure to do so will result in an automatic zero on this assignment and potentially further action.

2 Problem Description

In this assignment, you will extract additional high level information from a dataset that contains the frequencies of words from various written sources. The dataset that you will use can be downloaded from `http://www.cs.rit.edu/~csci141/pub/Projects/unigram/unigram.zip`.

This dataset contains data from the year 1800 to the year 2008. It is a subset of approximately 3 terabytes of data that was originally compiled by Google[1]. Here is an example of the historical information that it contains on the word "project"[2]:

1. See `http://storage.googleapis.com/books/ngrams/books/datasetsv2.html`.
2. Note that this information is already provided by the dataset.



Also included in the zip file are **test programs** you will use during the development of your solution. *Do not change these files, or you may make your programs fail testing.*

3 Getting Started

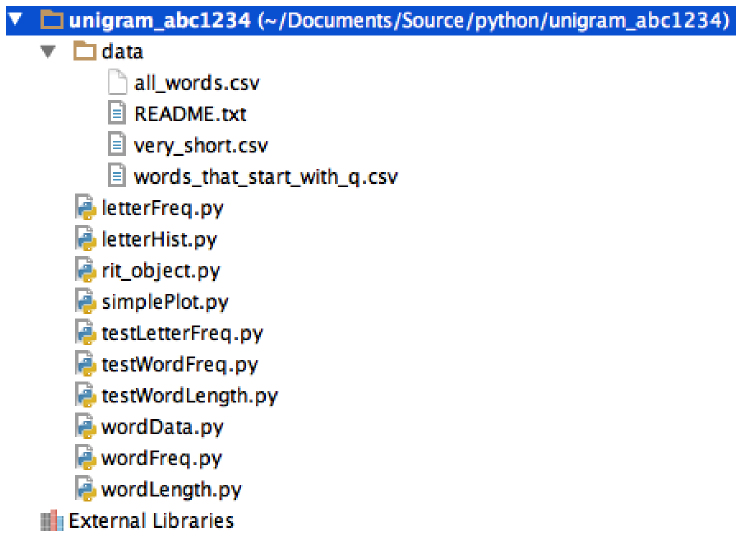
You should create a new project in PyCharm using the provided zip file. There should be a `data/` subdirectory with the data files, along with five source files. *D*O NOT CHANGE THE CONTENTS OF ANY SUPPLIED FILE!:

1. `rit_object.py`: For classes you will be defining in `wordData.py`.
2. `simplePlot.py`: A plotter for the second and third tasks.
3. `testLetterFreq.py`: A test program for the first task.
4. `testWordFreq.py`: A test program for the second task.
5. `testWordLength.py`: A test program for the third task.

In addition to the supplied files, you will be writing the following, from scratch:

1. `letterFreq.py`: The main program for the first task.
2. `letterHist.py`: A supporting turtle plotter program for the first task.
3. `wordData.py`: A supporting module used in all three tasks.
4. `wordFreq.py`: The main program for the second task.
5. `wordLength.py`: The main program for the third task.

The layout of your project should eventually look like the following:



```
unigram_abc1234 (~/Documents/Source/python/unigram_abc1234)
    data
        all_words.csv
        README.txt
        very_short.csv
        words_that_start_with_q.csv
    letterFreq.py
    letterHist.py
    rit_object.py
    simplePlot.py
    testLetterFreq.py
    testWordFreq.py
    testWordLength.py
    wordData.py
    wordFreq.py
    wordLength.py
External Libraries
```

4 Main Programs

You will be writing three main programs for this project. It is important that each of your main programs include the following, at the bottom, as a way to run your `main` function. Failure to do this will cause the test programs to run your main programs by accident.

```
def main():
    """Your main function here"

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

5 Reading the Data File

There are three files that are provided: `very_short.csv`, `words_that_start_with_q.csv`, and `all_words.csv`. You should move progressively to larger files as you test your programs. Each line of the file has these, comma-separated fields:

- The word (`string`);
- The year (`int`);
- The number of times that word appeared in any book for that year (`int`);

For a given year, counts of the number of occurrences of a single word will be stored in an object of type `YearCount` that will have two slots: count and year, both of which are of the type `int`. `YearCount` should inherit from the `rit_object`. The `YearCount` class must be implemented in `wordData.py` and imported by our main program.

In order to get started, a function called `readWordFile(fileName)` that will read in the entire unigram dataset is needed. This function should be implemented in `wordData.py`.

`readWordFile(fileName)`

Inputs:

`fileName`: A string giving the name of a unigram csv file.

Output:

A dictionary mapping words to lists of `YearCount` objects. For every word, there is exactly one list of `YearCount` objects. Each `YearCount` object contains a year in which a word appeared and the count of the number of times the word appeared that year. This is probably easier to understand from the example below.

Example:

```
>>> import wordData
>>> words = wordData.readWordFile('very_short.csv')
>>> print(words)
{'airport': [YearCount( year=2007, count=175702 ), YearCount( year=2008,
 count=173294 )], 'wandered': [YearCount( year=2005, count=83769 ),
YearCount( year=2006, count=87688 ), YearCount( year=2007, count=108634 ),
YearCount( year=2008, count=171015 )], 'request': [YearCount( year=2005,
count=646179 ), YearCount( year=2006, count=677820 ), YearCount( year=2007,
count=697645 ), YearCount( year=2008, count=795265 )]}
```

Look carefully! Getting the format exactly right is very important. For instance, observe that words maps the key `'wandered'` to a list of `YearCount` objects. This list of `YearCount` objects is 4 entries long, and every `YearCount` object contains exactly two items, the year and the number of occurrences of `'wandered'` in that year.

The next helper function will give you some good practice with our new data structures. It should again be implemented in `wordData.py`.

`totalOccurrences(word, words)`

Inputs:

`word`: The word for which to calculate the count. Not guaranteed to exist in `words`.

`words`: A dictionary mapping words to lists of `YearCount` objects.

Output:

The total number of times that a word has appeared in a book in the entire dataset.

Example:

```
>>> import wordData
>>> words = wordData.readWordFile('very_short.csv')
>>> print(wordData.totalOccurrences('wandered',words))
```

```
451106
>>> print(wordData.totalOccurrences('quetzalcoatl',words))
0
```

While somewhat interesting on its own, this function is primarily intended as a building block for implementing other functions in what follows. You will have to define `YearCount`, `readWordFile` and `totalOccurrences` in a separate, shared module `wordData.py`, and you will need it in the rest of the project.

6 Tasks

There are several things that one could do with this dataset such as comparing the relative importance of words. See Google's own demo that can be accessed at `http://books.google.com/ngrams`. The goal of this project will be to try to obtain some new information from the available dataset.

You will develop Python modules to calculate and plot for analysis:

- `letterFreq.py`: the relative letter frequencies of letters in the English language,
- `wordFreq.py`: the aggregate word counts and the distribution of word frequencies, and
- `wordLength.py`: the length of the average written word over the periods of time for which data are available.

You also will use the `turtle` module to draw your own histogram for the relative letter frequencies to visualize your results (for the first task, in `letterHist.py`).

6.1 Plotting Letter Frequencies

In this task, you'll create a plot of relative frequencies of letters in the English language. This should be written in a main program called `letterFreq.py`. It will use what you have already written in `wordData.py`, and eventually `letterHist.py` for plotting.

6.1.1 Calculating Letter Frequency Values

`letterFreq(words)`

Inputs:

`words`: A dictionary mapping words to lists of `YearCount` objects.

Output:

A list containing the relative frequency of letters scaled by the total

letter count in alphabetical order.

Example:

```
>>> import wordData
>>> import letterFreq
>>> words = wordData.readWordFile('very_short.csv')
>>> letterFreq.letterFreq(words)
[0.03104758705050717, 0.0, 0.0, 0.03500991824543893, 0.2536276129665047, 0.0, 0.
0, 0.0, 0.013542627927787708, 0.0, 0.0, 0.0, 0.0, 0.017504959122719464, 0.013542
627927787708, 0.013542627927787708, 0.10930884736053291, 0.15389906233882777, 0.
10930884736053291, 0.12285147528832062, 0.10930884736053291, 0.0, 0.017504959122
719464, 0.0, 0.0, 0.0]
```

In the example above for `very_short.csv`, we get `0.03` by first counting the total number of `a`'s occurring in all the words in the input data set. This number is then divided by the total number of letters in all the words. Note that `'wandered'` and `'airport'` each contain exactly one `a`, and these words occur a total of `(83769 + 87688 + 108634 + 171015 + 175702 + 173294)` times (see Appendix A for the data file content). To get the total number of letters, we observe that `'wandered'` is of length `8`, `'airport'` is of length `7`, and `'request'` is of length `7`, and multiply each of these numbers by the total number of occurrences of each, e.g. `8 * (83769 + 87688 + 108634 + 171015)`.

6.1.2 Making Your Own Histogram Plot Function

Finally, the data will be visualized using a histogram. The function that does the plotting should have the following format:

`letterFreqPlot(freqList)`

Inputs:

> `freqList`: A list of floating points values between 0.0 and 1.0. The first entry corresponds to the letter `'a'`, the second entry to the letter `'b'`, and so on.

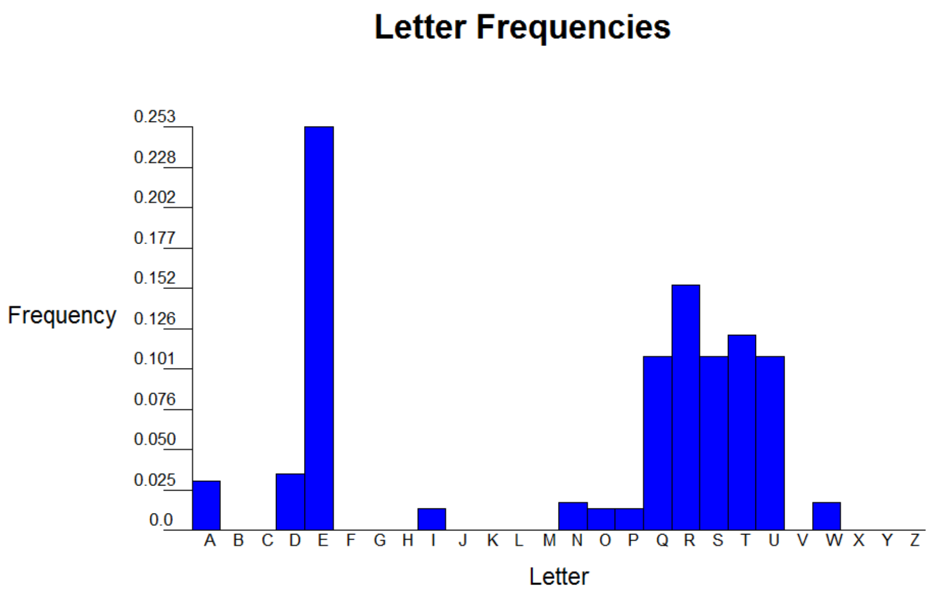
Output:

> The function returns None, draws a histogram of letter frequencies.

Example:

```
>>> import wordData
>>> import letterFreq
>>> import letterHist
>>> words = wordData.readWordFile('very_short.csv')
>>> freqList = letterFreq.letterFreq(words)
>>> letterHist.letterFreqPlot(freqList)
```

The corresponding plot is shown below. Your task will be to implement it by using the `turtle` module to draw the graphics. Do not hard code the height of the bars; let the maximum data value determine the maximum value of the vertical axis. Your implementation of `letterFreqPlot` should work for all data files. At the same time, you can rely on the fact that there are going to be exactly 26 bins. Properly laying out the bars and labels of a histogram in the general case would be too difficult.



Your plot should have a title, and labels on the x-axis for each letter (i.e. bar of the histogram) and on the y-axis showing frequencies. Place your code into a separate `letterHist.py` module.

6.2 Plotting Aggregate Word Counts

Here, we will define the class `WordCount` that will have two slots: `word` and `count`, which are of the types `str` and `int` respectively. `WordCount` should inherit from the `rit_object` and must be defined in `wordData.py`.

You will make another main program for this task, `wordFreq.py`. It will use `wordData.py` and `simplePlot.py`.

`wordFrequencies(words)`

Inputs:

`words`: The word data, which is a dictionary (`dict`) that maps words (`strings`) to a list of `YearCount`'s.

Output:

A list of `WordCount` objects in decreasing order from most to least frequent.

The module `simplePlot.py` provides the function `wordFreqPlot(freqList)` that will scale the data using the logarithmic scale for both dimensions. This will result in a `log-log` plot. You don't have to write your own plots for the remaining tasks.

`wordFreqPlot(freqList)`

Inputs:

`freqList`: A `list` of `WordCount` objects.

Output:

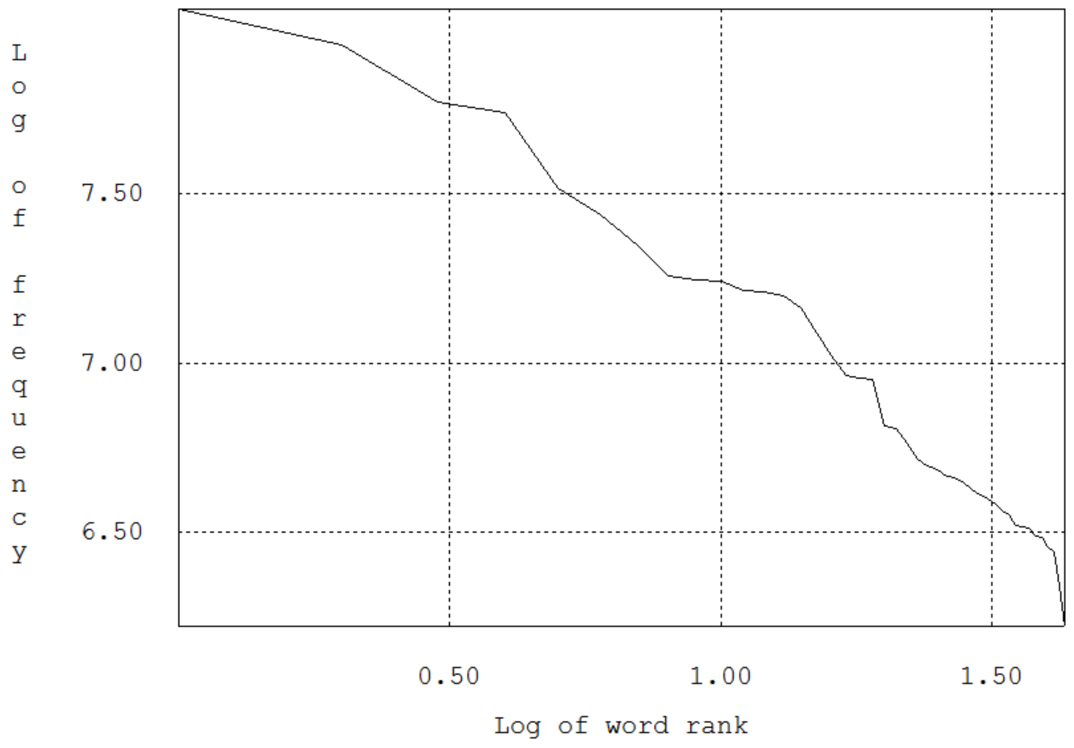
Returns None. Plots a `log-log` plot.

Example:

```
>>> import wordData
>>> import wordFreq
>>> import simplePlot

>>> words = wordData.readWordFile('words_that_start_with_q.csv')
>>> freqList = wordFreq.wordFrequencies(words)
>>> simplePlot.wordFreqPlot(freqList)
```

Conveniently, `totalOccurrences` works perfectly for this purpose. The x-axis is the rank of the word, where rank 1 is the most common word in English, rank 50 is the 50th most common word, and so forth. The y-axis is the total number of occurrences in all books in the database. However, the scale in which they are displayed is logarithmic; i.e. $x = log_{10}$(rank of the word) and $y = log_{10}$(number of occurrences of the word).



Since the word rank vs. total number of occurrences looks roughly as a straight line with negative slope on a `log-log` plot, it must follow a power law. This observation between word occurrence and word rank is known as Zipf's law. Intriguingly, nobody really knows why it holds. The choice of the `log-log` plot allows us to see the relationship; otherwise the data points would scale badly.

6.3 Plotting Average Word Length over a Period of Time

You will complete the project by calculating another interesting statistical value of the underlying data. How did the average length of words change over time? One could expect this to be roughly correlated with the tendency of authors to use of an extended vocabulary. However, more importantly, it will provide you with an opportunity to hone your Python skills. The challenge is writing code to calculate something like that.

You will write a third main program, `wordLength.py`. It will use the functions form `wordData.py` and `simplePlot.py`.

To accomplish this goal, you will need to write the following functions, all in your main program:

`occurrencesInYear(word, year, words)`

Inputs:

> `word`: The word in question passed in as a `str`.
>
> `year`: The year in question. This is an `int`.
>
> `words`: The word data which is a dictionary (`dict`) that maps words (`str`).
>
> to a list of `YearCount`'s.

Output:

> The number of occurrences of the word in the year (`int`).

`averageWordLength(year, words)`

Inputs:

> `year`: The year in question (`int`).
>
> `words`: The words data which is a dictionary (`dict`) that maps words (`strings`) to a
>
> list of `YearCount`'s.

Output:

> The average word length for the year in question (`float`).

`averageWordLengthYears(startYear, endYear, words)`

Inputs:

`startYear` The start year, an (`int`).

`endYear` The end year, likewise an (`int`).

`words` The words data which is a dictionary (`dict`) that maps words (`strings`) to a list of `YearCount`'s.

Output:

The list of `float`'s that contains the average word lengths by year in the increasing order for years between `startYear` and `endYear` – both inclusive.

Example:

```
>>> import wordData
>>> import wordLength
>>> words = wordData.readWordFile('very_short.csv')
>>> print(words)
{'request': [YearCount( year=2005, count=646179 ), YearCount( year=2006, count=6
77820 ), YearCount( year=2007, count=697645 ), YearCount( year=2008, count=79526
5 )], 'airport': [YearCount( year=2007, count=175702 ), YearCount( year=2008, co
unt=173294 )], 'wandered': [YearCount( year=2005, count=83769 ), YearCount( year
=2006, count=87688 ), YearCount( year=2007, count=108634 ), YearCount( year=2008
, count=171015 )]}
>>> print(wordLength.occurrencesInYear('wandered', 2007, words))
108634
>>> print(wordLength.averageWordLength(2006, words))
7.114548770228398
>>> print(wordLength.averageWordLengthYears(2005, 2008, words))
[7.1147602294958, 7.114548770228398, 7.110627395031065, 7.150069236398865]
```

Your implementation of `averageWordLengthYears` should be based on `averageWordLength`, which, in turn, should use `occurrencesInYear` as a building block. To arrive at the average word length in 2006 in the above example, we observe that the word `'wandered'` appears `87,688` times and is of length 8, the word `'airport'` appears `0` times, and the word `'request'` appears `677,820` times and is of length 7. This results in an average length of `7.1` letters.

In order to create the corresponding plot, use the function `averageWordLengthPlot` that can be found in `simplePlot.py`.

`averageWordLengthPlot(startYear, endYear, lengthsList)`

Inputs:

`startYear` The start year, an (`int`).

`endYear` The end year, likewise an (`int`).

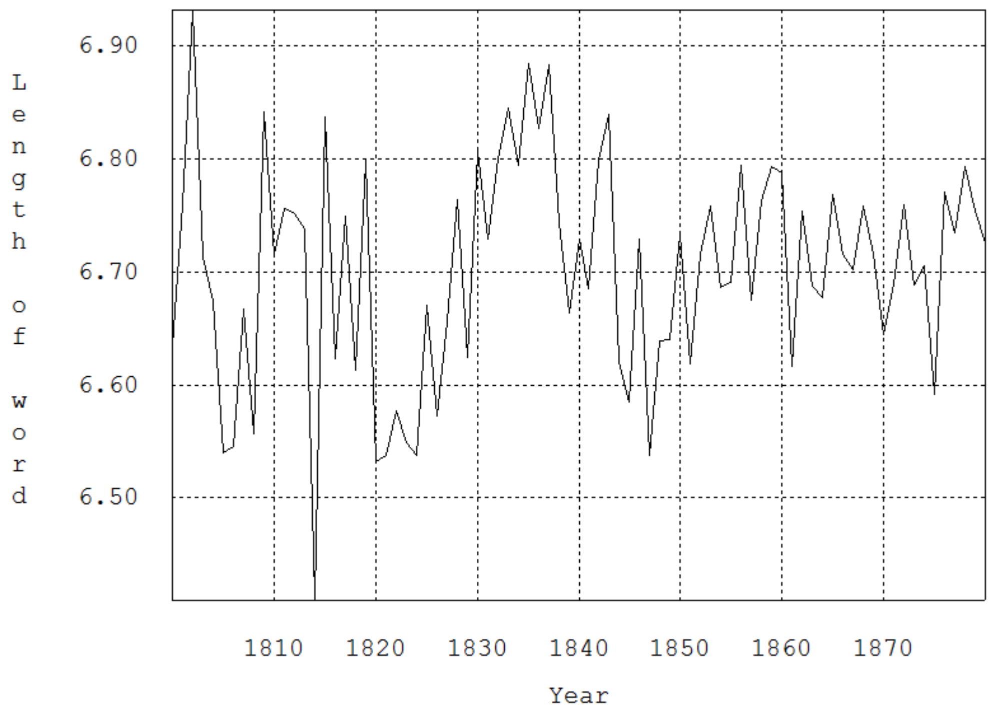
`lengthsList` The list of `float`'s that contains the average word lengths by year in the increasing order for years between `startYear` and `endYear` – both inclusive.

Output:

Returns None. Draws the plot of the average word length for the given period.

Example:

```
>>> simplePlot.averageWordLengthPlot(1800, 1880, lengthsList)
```



This range and restricted dataset (`words_that_start_with_q.csv`) was used intentionally so that you can discover the results for yourself using the full data set and date range. Note that the commands needed to obtain `lengthsList` are completely analogous to the previous example and have been omitted. Create the same plot using the dataset `all_words.csv` for the period 1800-2008. What trend can you observe in recent years?

6.4 Constraints

These are the major restrictions placed on the implementation of your modules.

- Your implementation must use Python3. Using Python version 2 is prohibited.
- All classes you design must inherit from `rit_object`.
- You must draw the diagram with the Python `turtle` module.
- Do not hard code the height of the histogram display to be the maximum value. Instead, design your plotter so that the vertical scale of the display has a maximum value equal to the height of the biggest histogram bar.

7 Testing

Please make sure that your program modules provide the interface required by the test programs. These files have been provided for you to develop and test your modules:

- `simplePlot.py`,
- `testLetterFreq.py`,
- `testWordFreq.py` and
- `testWordLength.py`

Do not change the interface of any of these functions because the test programs will be used to verify your code.

Your program will be tested against different, previously unseen, test cases, which are based on the same data files as those provided.

You should make sure that your code scales well on the large dataset `all_words.csv`. Note that running the test programs on this large data set can take several minutes to finish.

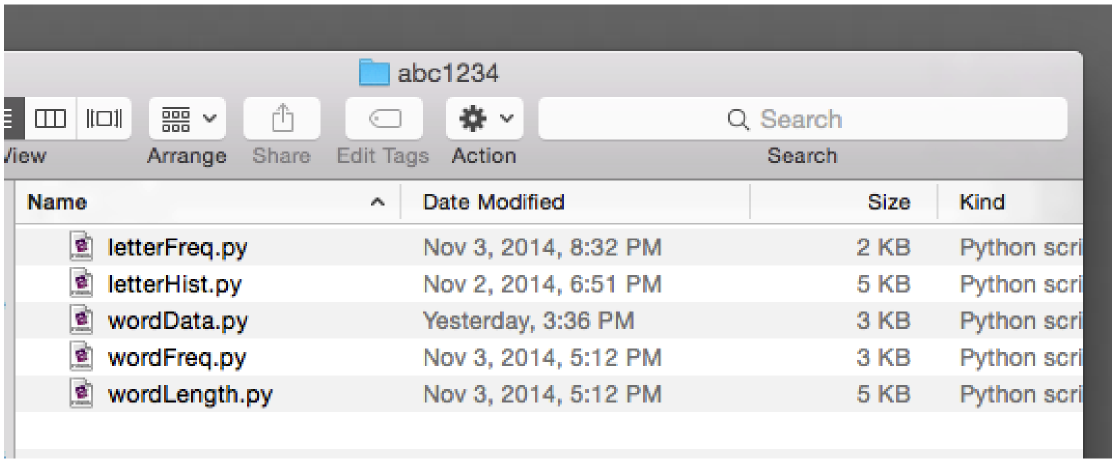
8 Submission

8.1 Submission Instructions

Please submit your code in a file called `abc1234.zip`, where `abc1234` is your RIT username. This file must contain *only* the files listed in the *Files To Submit* section.

The **zip file structure must be "flat"**; that means there must be *no folders within the zip file*. Your zip must not contain any nested folders, nor may any nested folders contain your `.py` files.

Create a separate folder with your username, and then copy only the files you wrote into it, to be zipped up:



Name	Date Modified	Size	Kind
letterFreq.py	Nov 3, 2014, 8:32 PM	2 KB	Python scri
letterHist.py	Nov 2, 2014, 6:51 PM	5 KB	Python scri
wordData.py	Yesterday, 3:36 PM	3 KB	Python scri
wordFreq.py	Nov 3, 2014, 5:12 PM	3 KB	Python scri
wordLength.py	Nov 3, 2014, 5:12 PM	5 KB	Python scri

Upload `abc1234.zip` to the MyCourses Project dropbox.

Do not use compression mechanisms other than zip; that is, do not use .rar, .7z, .gz, or .tar formats.

8.2 Files To Submit

These are the file modules you must submit; do not submit anything else.

- `wordData.py` – implements the functions `readWordFile` and `totalOccurrences`, as well as the classes `YearCount` and `WordCount`;
- `letterFreq.py` – implements `letterFreq`, draws the letter histogram plot;
- `letterHist.py` – implements your plotter, `letterFreqPlot`;
- `wordFreq.py` – implements `wordFrequencies`, draws the Zipf's law based on the ranking and occurrences of the words in the provided dataset on a log-log plot;
- `wordLength.py` – implements `averageWordLength`, `averageWordLengthYears` and `occurrencesInYear`, draws average word length for a specified period of time based on the words in the provided dataset;

9 Grading

9.1 Grading Breakdown

Here is the grading outline for your implementation(100%):

- 20% The classes and functions in `wordData.py` and `letterFreq.py`;
- 20% The histogram plotting in `letterHist.py` (see the example on page 5);
- 25% The classes and functions in `wordFreq.py`;
- 25% The classes and functions in `wordLength.py`;
- 10% Documentation and Style:
  Each file must have a *header docstring* containing the file name and your name. (Remember, docstrings have 'triple quote' syntax).
  Each function must have its own, *function docstring* with the following:
  - a purpose statement;
  - description of input parameters and types; and
  - description of the function's return value(s) and types.

  Finally for style, the code should be well organized on the page. There should be at least one blank line between each component (i.e. between class definitions and between functions).

9.2 Grading Deductions

It is possible that you got everything working but cheated to succeed, or you failed to follow directions and made testing, cheat-checking and grading of your work very difficult and time-consuming. This leads to these possible deductions over and above the grading breakdown described earlier.

- 100% If **cheat-checking** detects that your modules are strongly similar to those of others, you will be subject to RIT's Academic Honesty Policy as detailed in `http://www.cs.rit.edu/~csci141/syllabus.html`. That means you will receive a 0 grade, and all parties to the cheating incident will receive that treatment. If the infraction is more serious, there may be further penalties (see the syllabus).

- 100% If your modules do not work with the provided test programs, you will receive a 0 grade. If the test programs cannot execute your functions, or if your functions cause the test programs to crash, then you will fail this assignment. You must name the specified functions *exactly as used in the testers*. If you fail to name the functions exactly as stated, you have broken the interface specification. Use the provided testing code to test your code and ensure that you follow the interface with your implementations.
- 10% If you fail to follow the zip submission directions exactly, you will recieve a 10% penalty off the top of your earned grade. You must not include anything in your zip file that was not specified in the submission section. Also, your zip must not contain any nested folders, nor may any nested folders contain your `.py` files.

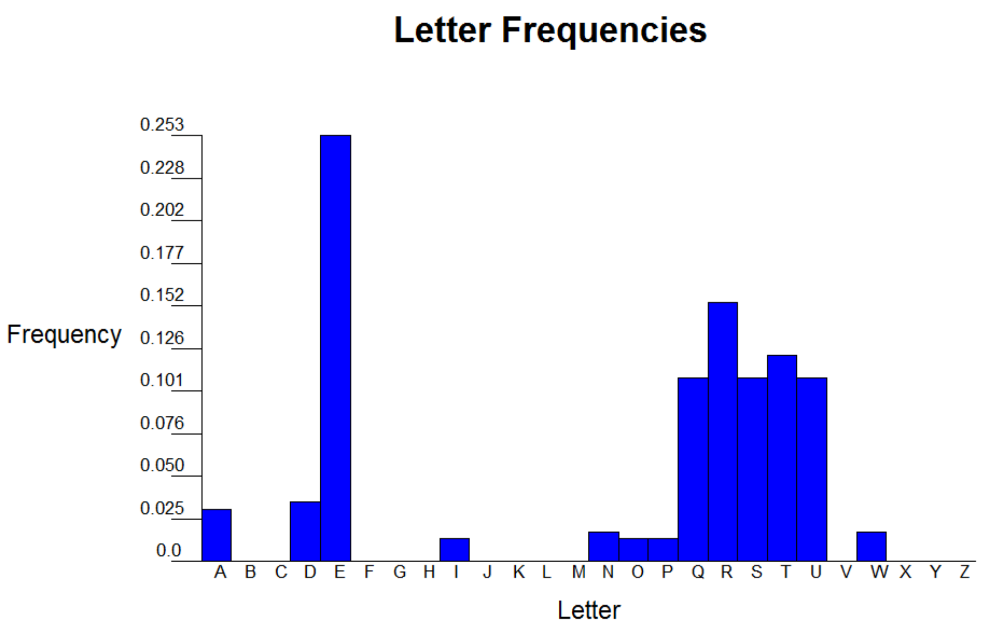
Appendix A: The `very_short.csv` Dataset

```
airport, 2007, 175702
airport, 2008, 173294
request, 2005, 646179
request, 2006, 677820
request, 2007, 697645
request, 2008, 795265
wandered, 2005, 83769
wandered, 2006, 87688
wandered, 2007, 108634
wandered, 2008, 171015
```

Appendix B: Sample Runs

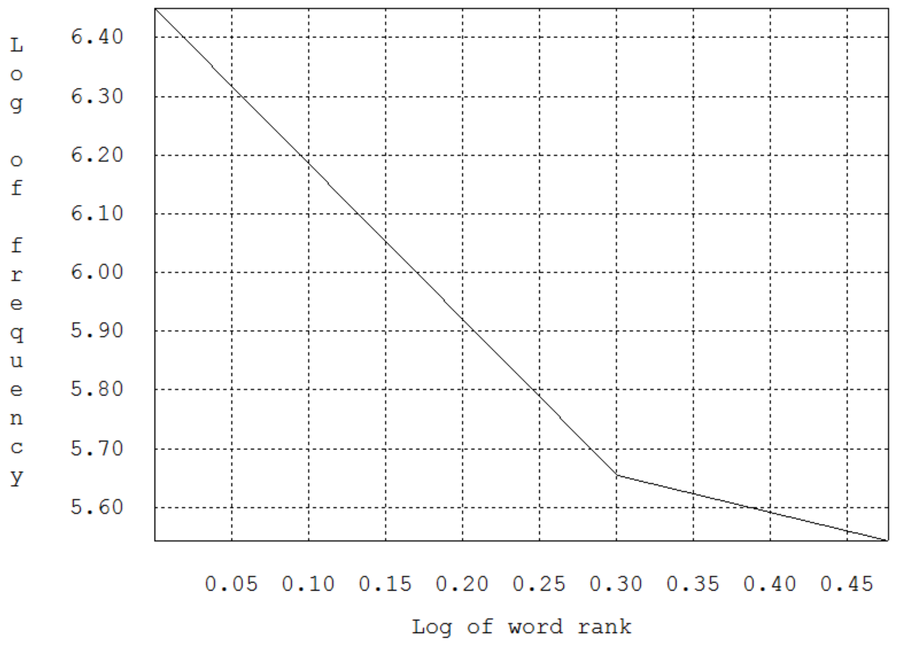
The three main programs that you have to write should provide for user input/output. For example, for letter frequencies one would have the following:

```
$ python3 letterFreq.py
Enter word file: very_short.csv
Enter word: airport
Total occurrences of airport: 348996
Letter frequencies: [0.03104758705050717, 0.0, 0.0, 0.03500991824543893,
0.2536276129665047, 0.0, 0.0, 0.0, 0.013542627927787708, 0.0, 0.0, 0.0, 0.0,
0.017504959122719464, 0.013542627927787708, 0.013542627927787708,
0.10930884736053291, 0.15389906233882777, 0.10930884736053291,
0.12285147528832062, 0.10930884736053291, 0.0, 0.017504959122719464,
0.0, 0.0, 0.0]
# plot is displayed
```



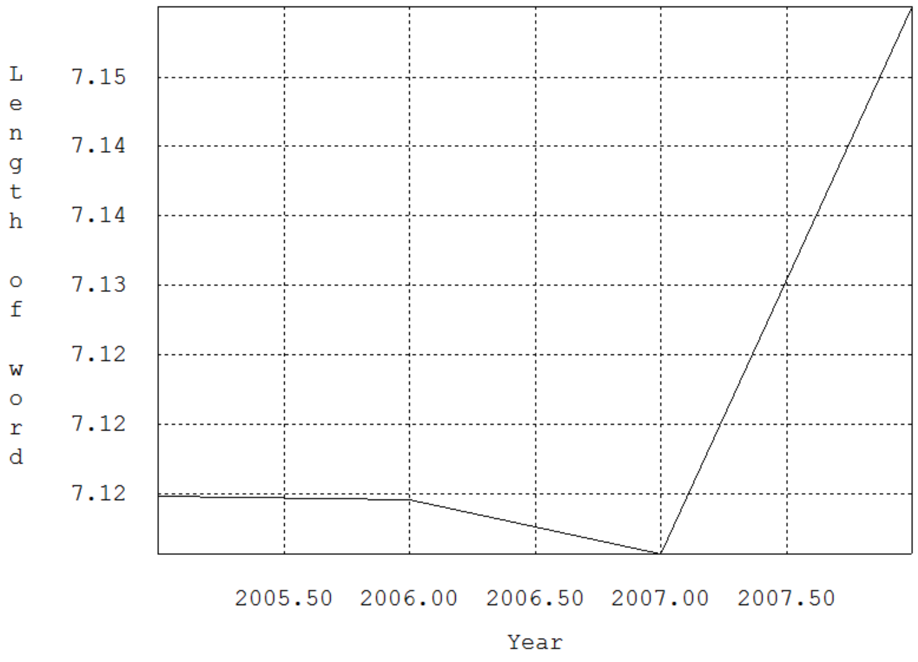
For word frequency, the user should be allowed to query for one ranked word:

```
$ python3 wordFreq.py
Enter word file: very_short.csv
Enter rank (1-3): 1
Rank 1: WordCount( word: request, count: 2816909 )
# plot is displayed
```



For average word length:

```
$ python3 wordLength.py
Enter word file: very_short.csv
Enter a word: airport
Enter a year: 2007
The word "airport " occurred 175702 times in the year 2007
Enter a year: 2005
The average word length for the year 2005 is 7.1147602294958 letters
Enter a start year: 2005
Enter an end year: 2008
# plot is displayed
```


Length of word
Year