Before you begin

- Make a new project for this lab
- Create `sentiment.py`
- Grab the four text files available on Canvas:
  - sentiment.txt – A file with a bunch of words and their positivity score
  - test_positive.txt – A file which should be evaluated as positive
  - test_negative.txt – A file which should be evaluated as negative
  - test_neutral.txt – A file which should be evaluated as neutral
- Look up the following string functions: `split, join, strip`
  - String functions page in Python
  - Types page in Python
  - Dealing with punctuation and white space
- Go through the command line and systems arguments page

Learning Objectives

You should be able to do the following:

- Read text from files
- Deal with large data sets
- Be able to use dictionaries/hash functions

Notes

Come back to these after you have read through the problems

- If you find that you're doing the same thing two or more times, consider separating that part of the code out into its own function. **If you find yourself copy and pasting large blocks of code, there's likely a better way to do it** (and we will deduct points on code quality if we see this).
- Pay attention to the definition of a word that we give you. For the purpose of this lab, words do not have to be dictionary words, e.g. "egg", "xyzzy", "123", and "mcdonaldscorporation" are all valid words.
- When writing the code for this lab, try to do it in as few lines of code as possible. Our reference solution has about 40 Functional Lines of Code (FLOC). A FLOC is a line that does something (i.e. not blank, and not a comment). You don't have to make your code this short to get it to work but, if you're writing substantially more code than this, then you should take a break, and see if there's a simpler way of doing things.
- While the `sentiment` part of this code is based off the provided text file, your code should be able to take read in any dictionary we want. Code should be usable for as many use cases as possible.
- There are many ways within PyCharm to run with command line arguments.
  - For debugging in the debugger, you can hard code the system argument commands by setting them directly. Make sure you take this out before turning your code in.
  - You can set the command arguments in the Preferences page
  - You can run directly from the terminal (`python sentiment.py test_positive.txt`)

Grading Rubric

In regards each item, please see associated footnotes for each item * .

- Problem 1
  - [1 points] `load_score_dict` reads in a file and properly outputs a correct dictionary. *
  - [1 points] `load_score_dict` handles files with blank lines and comment lines starting with #. *
  - [1 points] `load_score_dict` can be run with either 0 or 1 arguments, defaulting to sentiment.txt if no argument is passed in. *
- Problem 2
  - [2 points] `get_words` returns the correct set of words *
  - [1 points] `get_words` return has punctuation stripped from it. *
  -
  - [1 points] `get_words` returns lowercase. *
- Problem 3
  - [2 points] `score_sentence` returns the correct score. *
- Problem 4
  - [2 points] Command line interface (CLI) responds to a text file input. *
  - [1 points] Result of CLI input is the correct output. *
  - [1 points] Prints a useful message if the user neglects to pass a text file.
- Standard
  - [2 points] Code passes PEP8 checks with 10 errors max.
  - [2 points] Code passes TA-reviewed style checks for cleanliness, layout, readability, etc.

Problem 1

For this assignment, create a new file called `sentiment.py`. All code should be defined in this file.

Write a function called `load_score_dict` which takes in one optional argument representing the file name of the score dictionary. It should default to sentiment.txt so when we call the function with no arguments, we should get this dictionary. It should return a dictionary where the keys are the words and the values are the corresponding sentiment scores as floats.

`Note:` You should assume the following formatting specifications: There may be blank lines in the file which should be ignored. There may be lines which start with #; these lines should be ignored. Otherwise, you may assume the line is of the format `word [value]`, separated by a single space.

Problem 2

Write a function called `get_words` which takes in a single argument, a string representing a sentence. It should return an iterable of the unique words in the string. Words must be lowercase and contain at least 1 alphanumeric character (i.e. the empty string is not a word). Words should have all punctuation characters stripped. A punctuation character is any one defined by punctuation in Python's string module. Punctuation should be completely removed (i.e. not replaced by spaces). Note that because of this, you may get some "words" which don't look like English words; this is OK. You may assume that the sentence contains only a mix of alphanumeric characters, spaces, tabs, and newlines as defined by `string.punctuation`. Hint: Python has a built-in data structure for unique sets of words, which you may find useful. **Example:**

```
my_sentence = "Grocery list:     3 boxes Land-o-Lakes butter, Aunt Jemima's butter pancake mix"
words = get_words(my_sentence)
# Should output something like: ['grocery', 'list', '3', 'boxes', 'landolakes', 'butter', 'aunt', 'jemimas
# Can be any appropriate iterable, notice butter only appears onces
```

*Indicates that the autograder will tell you if this problem is correct on our set of test cases and assign you credit.

Problem 3

Create a new function called `score_sentence` which takes in two arguments: a string representing a sentence, and a dictionary containing the scores. It should then take the unique words in the sentence and output a score for the sentence by summing the corresponding score from the dictionary for each word. If the word is not in the dictionary, the corresponding value should be 0. **Example:**

```
my_sentence = "Welcome, welcome to my house!"
scores = {'welcome': 0.5, 'house': -0.25}
final_score = score_sentence(my_sentence, scores)  # Should output 0.25
```

Problem 4

In your `if __name__ == '__main__'` section, write an interface which allows in a user to pass in the name of a file through the command line like this: `python sentiment.py some_file.txt`

If the user fails to pass in a file name, your code should handle the situation without throwing an `Exception` by printing a message to the console instructing the user to input a file name.

Otherwise, your code should load in the text from the file, extract the list of unique words, and compute the sentiment score using the dictionary from sentiment.txt. It should then print out the word `Positive` to the console if the file score is > 0, `Negative` if the file score is < 0, and `Neutral` otherwise. There should be no other output to the console.

We suggest using the provided text files to confirm whether your code is working or not.