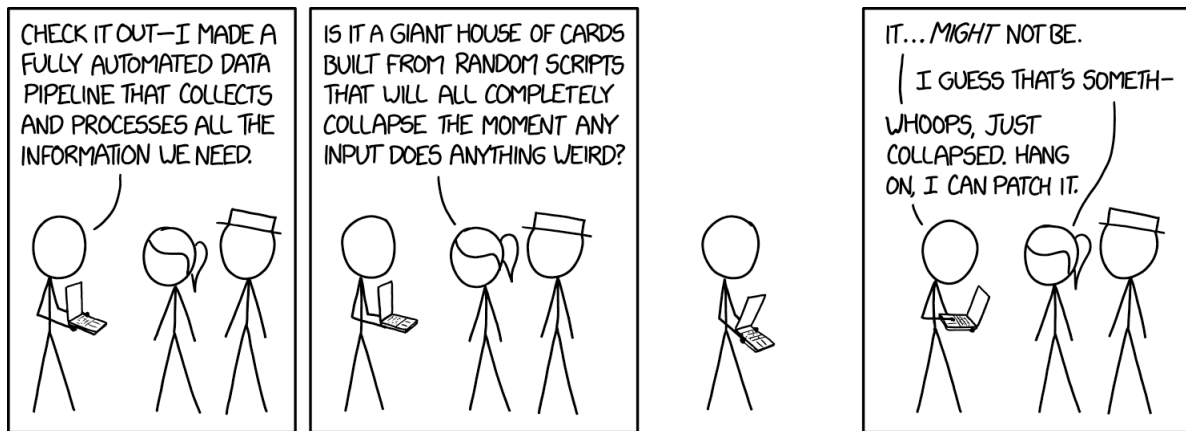


CS50—Assignment 6

Corpus Analysis

Due: Monday, October 12 at 11:59am



<https://xkcd.com/2054/>

For this assignment, we're going to be doing some basic corpus analysis. A corpus is a collection of documents/texts. By looking at different characteristics of those texts, in aggregate, we can often start to understand more about the language used in the corpus.

Data (Group)

To give us a bit of variety (and allow us to make comparisons) we're going to be looking at text from a few different sources. Follow the same procedure you've done for the last two assignments for downloading the starter:

- Move the **assignment6** folder containing this PDF into the **cs50** folder you've been using for all your cs50 assignments; open the **assignment6** folder in VS Code. You can find the zip file again here: <https://cs.pomona.edu/classes/cs50/assignment6.zip>
- Make a new Python file called **assign6.py**. Add your name and the assignment number in comments to the top of this file.
- Your **assignment6** folder should now have five files in it:
 - **assign6.py**: The empty Python file you just made.
 - **test.txt**: A simple test file for you to test your functions on before trying them on the full data.

- `sam.txt`: Text from Dr. Seuss's "Green Eggs and Ham". Mostly just for fun :)
- `normal.txt`: 100K sentences from English Wikipedia¹.
- `simple.txt`: 100K sentences from Simple English Wikipedia², which contains similar content as English Wikipedia, but written to be more broadly accessible.

Open up a couple of them and take a look at what's inside. Each of these files is a text file with one sentence per line. From a terminal (at least on Mac or Linux), you could try e.g. `head -n 10 simple.txt` to read the first ten lines.

File and Dictionary Practice (Group)

We're going to get some practice working with files and dictionaries. Like last time, we'll do some exercises. Implement the following functions, that should work on a file called `menu.txt` with exactly these contents (do not submit this file to Gradescope):

```
spider donut|7
spider cider|18
nice cream|15
bisicle|15
temmie flakes|3
junk food|17
glamburger|120
```

Note that there shouldn't be any blank line at the end of the file.

1. Write a single function that prints out the contents of `sam.txt` twice in a row.
2. Write a function that takes two file names as input, and prints out alternating lines of both files. E.g., given `menu.txt` and `test.txt`, the first four lines of output should be the following:

```
spider donut|7
This is a sentence
spider cider|18
This is another sentence
```

If one file runs out, just finish printing the rest of the other file.

3. `read_until(filename, stop)`: Read the file at the given path line by line until finding the string `stop`, at which point the lines read so far should be returned in a list.
4. `split_at_bar(string)` should use `find` and slicing to split a line of text like the ones above into the menu item and the price, returning both in a tuple.

¹<https://en.wikipedia.org>

²<https://simple.wikipedia.org>

5. `build_menu()` should read `menu.txt` and return the menu as a dictionary from product names to prices.
6. `total_cost(menu)` should take a menu dictionary as input and compute the total cost of buying everything on the menu.
7. `most_expensive(menu)` should take a menu dictionary as input and return the name of the most expensive item.
8. `buy(menu, inventory, item_name)` should be a function that takes a menu dictionary, an *inventory* dictionary, and the name of an item and returns a boolean if the purchase was successful. The inventory should initially be defined as `{'gold':200}`. Buying an item (if we have enough money and the store sells the item) should reduce the quantity of gold and increase the quantity of the purchased item.
9. Write, in a multi-line string, series of calls starting from `build_menu` and calling `buy` successively on the same inventory, which should bring the money in the inventory down as close to zero as possible.

Overview (Individual)

Our goal for this assignment is to write a program that takes in a file and outputs some statistics about the text in the file. Specifically, the following would be printed out if we ran our program on the test data:

```
-----
Total sentences:      4
Longest sentence:    6
Shortest sentence:   4
Ave. sentence length: 4.75
-----
Top ten most frequent words
1: this      3
2: sentence  3
3: is        2
4: a         2
5: does      2
6: another  1
7: count     1
8: as        1
9: i         1
10: think    1
```

Sentence Stats (Individual)

1. **[3 points]** The first step when dealing with text is often to break the sentence down into words (aka tokens). As a first step towards that, we want to split the sentence based on whitespace. Write a function called `my_split` that takes as input a string (representing a sentence) and returns a list of all of the strings in the sentence that are separated by spaces. **You may not use the split method.** (We are trying to implement this function ourselves).

For example,

```
>>> my_split("I would not, could not, in a car.")
['I', 'would', 'not,', 'could', 'not,', 'in', 'a', 'car.']
```

There are a few ways you can write this function. One route is to loop over the sentence a character at a time to identify the words, adding them to a list once you find them.

Hint: The string module (<https://docs.python.org/3/library/string.html>) has pre-defined strings with names such as `whitespace` that might be helpful.

2. **[2 points]** If we just split on whitespace, the tokens that we get still contain punctuation characters (e.g., `'car.'` in the example above). Write a function called `tokenize` that takes as input a string (representing a sentence) and returns a list of the “words” in the sentence, lowercased. We’ll take a simple approach to this that captures most of the common cases:
 - First split the sentence into “basic words” (using your `my_split` function).
 - Then, for each of these basic words, if it ends in one of a punctuation mark (e.g., `.`, `,`, `;`, `?`), remove that character from the end of the word.

For example:

```
>>> tokenize("I would not, could not, in a car.")
['i', 'would', 'not', 'could', 'not', 'in', 'a', 'car']
```

Hint: The string module (<https://docs.python.org/3/library/string.html>) has pre-defined strings with names such as `punctuation` that might be helpful.

3. **[2 points]** Write a function called `get_sentence_lengths` that takes as input a *filename* and returns a list of the number of words in each line/sentence in the file after tokenizing (i.e. calling `tokenize`). For example:

```
>>> get_sentence_lengths("test.txt")
[4, 4, 6, 5]
```

These correspond to the lengths of the tokenized lists returned by calls for each line in the file, i.e.

```
>>> tokenize("This is a sentence.")
['this', 'is', 'a', 'sentence']
>>> tokenize("This is another sentence.")
['this', 'is', 'another', 'sentence']
>>> tokenize("Does this count as a sentence?")
['does', 'this', 'count', 'as', 'a', 'sentence']
>>> tokenize("I think, yes, it does.")
['i', 'think', 'yes', 'it', 'does']
```

4. [2 points] Write a function called `print_sentence_stats` that takes a *filename* (as a string) as a parameter and *prints* out four statistics about the sentence in the file: 1) the total number of sentences, 2) the length of the longest sentence, 3) the length of the shortest sentence, and 4) the average sentence length. Your results be formatted with the text label, followed by a tab character (`\t`) and then the number, i.e.:

```
Total sentences:      4
Longest sentence:     6
Shortest sentence:    4
Ave. sentence length: 4.75
```

(This is the output from `test.txt`)

Hint: Use your `get_sentence_lengths` function!

Word Stats (Individual)

The second part of the analysis looks at the frequency of words, specifically, finding the top ten most frequent words in the corpus. We're going to accomplish this in a few steps:

- First, we'll create a dictionary with all of the word frequencies, specifically, where the key is the word and the value is the frequency of that word in the corpus.
 - Given this dictionary, we can traverse all of the entries and find the word with the highest frequency. This will be the first entry in our top ten list.
 - To get the next entry, we will *delete* the entry with the highest frequency from the dictionary. If we then search again for the word with the highest frequency, we'll get the second most frequent word. We'll repeat this process until we have the top 10 words.
4. [3 points] Write a function called `add_words_to_dict` that takes two parameters as input: 1) a dictionary where the key is a string (i.e., a word) and the value is an int representing the current count of that word and 2) a list of strings representing words. The function should increment the count in the dictionary of each of the words in the list by 1. Specifically, if the word is not in the dictionary yet, it should add it with the value 1. If it is, it should update the value to be one larger. Note, this function mutates the dictionary and does *not* return a value.

```

>>> test_dict = {}
>>> add_words_to_dict(test_dict, ["a", "b", "b", "c"])
>>> test_dict
{'a': 1, 'b': 2, 'c': 1}
>>> add_words_to_dict(test_dict, ["a", "c", "c"])
>>> test_dict
{'a': 2, 'b': 2, 'c': 3}
>>> add_words_to_dict(test_dict, ["a"])
>>> test_dict
{'a': 3, 'b': 2, 'c': 3}

```

5. **[3 points]** Write a function called `get_word_counts` that takes a filename (as a string) as input and creates a dictionary of the words frequencies in the file. You should tokenize each line of the file. *Hint:* Don't work too hard in this function. Most of the work has already been done. Specifically, think about how both `tokenize` and `add_words_to_dict` can be used in this function.

```

>>> get_word_counts("test.txt")
{'this': 3, 'is': 2, 'a': 2, 'sentence': 3, 'another': 1, 'does': 2, 'count': 1,
 'as': 1, 'i': 1, 'think': 1, 'yes': 1, 'it': 1}

```

6. **[3 points]** Write a function called `dict_count_max` that takes a dictionary of word frequencies as input and returns the word in the dictionary with the highest frequency.
7. **[3 points]** Write a function called `print_top_ten` that takes a filename (as a string) as input and prints the top ten most frequent words, along with their frequencies (see formatting in the example above).

Hints:

- Use your `get_word_counts` function.
- To print the top 10, find the most frequent word using `dict_count_max`, print it out, and then delete it. Repeat this 10 times.
- To remove an item from a dictionary, use the `pop` method, which takes the key to remove as input. Here's an example of using `pop` in the python shell:

```

>>> test = {"a":1, "b":10, "c":5}
>>> test
{'a': 1, 'b': 10, 'c': 5}
>>> test.pop("b")
10
>>> test
{'a': 1, 'c': 5}

```

8. **[1 points]** Finally, write a function called `print_all_stats` that takes a filename (as a string) as input and prints the sentence level and word level stats formatted as seen in the example above (in the Overview section)

Analysis (Individual)

Now we can use our functions to analyze a few corpora. At the end of your file, add the following delimiter:

```
# -----  
# Analysis section
```

9. **[2 points]** Run your program on both `normal.txt` and `simple.txt` and include their output as a triple-quoted string. Additionally, include 2-3 sentences analyzing the differences, i.e. what does the data imply about the different sources? Does it make sense?
10. **[3 points]** Do one additional analysis of your own choosing that compares the two corpora. You must write at least one function to help you with this analysis. Include the quantitative results for both corpora along with 1-2 sentences talking about the results. Here are some suggestions, though feel free to get creative:
 - Vocabulary size, i.e. the total number of *unique* words.
 - Average word length.
 - Use of punctuation, e.g. the number of commas per sentence.

When you're done (Individual)

Make sure that your program is properly commented:

- You should have comments at the very beginning of the file stating your name, course, assignment number and the date.
- You should have comments delimiting the two sections.
- Each function should have an appropriate docstring.
- Include other miscellaneous comments to make things clear.

In addition, make sure that you've used good *style*. This includes:

- Following naming conventions, e.g. all variables and functions should be lowercase.
- Using good variable names.
- Good use of booleans. You should NOT have anything like:)

```
if boolean_expression == True:
```

or

```
if boolean_expression == False:
```

instead use:

```
if boolean_expression:
```

or

```
if not (boolean_expression): # or some other way of negating the expression
```

- Proper use of whitespace, including indenting and use of blank lines to separate chunks of code that belong together.
- Make sure that none of the lines are too long.

Submitting

You will need to submit your `assign6.py` file and `group.py` file (and not any of the starter code/data that you downloaded) on Gradescope.

Grading

	points
Sentence Stats	9
Word Stats	13
Analysis	5
Comments, style	3
total	30