

COSC 130 – Project 02 Instructions

Introduction

The goal of this project is to develop tools for performing basic text analysis tasks, such as processing text read from a text file, tokenizing the text, and performing word counts. You will be asked to write several functions to be used in performing these tasks. You will test your functions by performing text analysis on the novel “A Tale of Two Cities”.

General Instructions

Create a Python script file and a Jupyter notebook file within the same directory. The script file should be named **word_count.py** and the Jupyter notebook file named **Project_02_YourLastName.ipynb**. You will use the script to define your functions and the notebook will be used to load the script and to test your functions.

Please download the files **tale_of_two_cities.txt** and **stopwords.txt**, storing these in the same directory as your script file and notebook file. It is important that these files are all in the same directory. Otherwise, your code will not run correctly when I run it.

Instructions for the Script File

Define functions with the following names: **process_word()**, **process_line()**, **process_file()**, **find_unique()**, **find_frequency()**, **most_common()**, **remove_stop()**, **count_by_length()**, and **count_by_first()**. Descriptions of each of these functions are provided below.

process_word()

This function should accept a single parameter named **word**. This parameter is expected to contain a string representing a word. The function should remove any punctuation from the string and convert it to lowercase. This can be done by performing the following steps.

1. Store the string `' .!?, "\'()*_ :;0123456789'` in a variable named **remove**. This string contains all of the characters to be removed from the beginning and end of **word**, if they are present.
2. Use the **strip()** method for strings to remove punctuation and digits from the beginning and end of **word**. Pass the method the string **remove**. Store the stripped string in a variable.
3. Use the **replace()** method on the string created in Step 1 to replace any single quote characters (likely representing apostrophes) with an empty string. That is, replace occurrences of `"'"` with `""`. Store the result.
4. Use the **lower()** method on the string created in Step 2 to convert it to lower case. Store the result.

The function should return the string created in Step 3.

process_line()

This function should accept a single parameter named **line**. This parameter is expected to contain a string representing a line of text read from a file. The function should perform the following processing steps to the line:

1. Use the **replace()** method to replace any dash characters `" - "` with spaces, storing the result in a variable.
2. Apply the **split()** method to the string created in Step 1 to create a list of individual words contained within the string. Store the resulting list in a variable named **words**.
3. Loop over the elements of **words**. Apply the **process_word()** function to each string in this list. It is possible for the resulting processed word to be an empty string. If the processed word is not empty (in other words, if it has a length greater than 0), then store it in a list named **processed_words**.

The function should return the list **processed_words**.

process_file()

This function should accept a single parameter named **path**. This parameter is expected to contain a string representing the relative location of a text file. The function will create and return a list of processed words contained in the file by performing the following tasks.

1. Use **with** and **open()** to open the file whose location is stored in **path**. Use **readlines()** to read the contents of the file into a list. Each string in this list will represent an entire line of text from the file.
2. Create an empty list named **words**.
3. Loop over the list created in Step 1. Apply the **process_line()** function to each string in this list. The list of words returned by **process_line()** should be concatenated to the end of the list **words**. The combined list should be stored back into **words**. Recall that you can concatenate two lists using the **+** operator.

The function should return the list **words**.

find_unique()

This function should accept a single parameter named **words**. This parameter is expected to contain a list of strings representing words. The function should create a list that contains exactly one copy of any string that appears in **words**.

1. Create an empty list to store the unique words.
2. Loop over the elements of **words**. If a particular element has not already been added to the list of unique words, then append it to that list. Do nothing if the element has already been added to the unique list.

The function should return the list of unique words.

find_frequency()

This function should accept a single parameter named **words**. This parameter is expected to contain a list of strings representing words. The function should create a dictionary recording the number of times each individual word appears in **words**. Each dictionary key should be a string representing a word, and each value should be a count representing the number of times that string appeared in **words**.

1. Create an empty dictionary named **freq_dict** to store the counts.
2. Loop over the elements of **words**. If a particular element has already been added to **freq_dict** as a key then increment the value associated with that key. If the element does not appear as a key in **freq_dict**, then add it as a key with a value of 1.

The function should return the dictionary **freq_dict**.

remove_stop()

Stop words are words that are removed from a collection of words when performing a text analysis. These are typically very common words such as "a" and "the".

This function should accept two parameters named **words** and **stop**. Both parameters are expected to contain a list of strings representing words. The function should return a list obtained by removing from **words** any strings that also appear in **stop**.

1. Create an empty list to store the non-stop words.
2. Loop over the elements of **words**. If a particular element **does not** appear in **stop**, then add it to the list create in Step 1. If the element does appear in **stop**, then do nothing.

The function should return the list of non-stop words.

most_common()

This function should accept two parameters named **freq_dict** and **n**. The parameter **freq** is expected to contain a dictionary recording word counts. The parameter **n** should be an integer. The function should find and display the **n** words with the highest frequency in **freq_dict**. One method of finding the words with the highest frequencies is described below.

1. Create an empty list named **freq_list**. This list will be used to store tuples created from key/value pairs found in **freq_dict**. These tuples will have the form **(value, key)**.
2. Loop over **freq_dict.items()**. For each key/value pair in **freq_dict**, create a tuple of the form **(value, key)** and append this to **freq_list**. It is important that the value (i.e. word count) appears first in the tuple.
3. Use the **sort()** method to sort **freq_list** in descending order. This will sort the list of tuples according to the first element in each tuple, which represents the word count.
4. Print out the first **n** results from **freq_list** in the format shown below. The xxxx symbols should be replaced with words and the ##### symbols should be replaced with word counts. The dashed line should be 16 characters long. Allot 12 characters for the word column and 4 characters for the count column. The word column should be left-aligned and the count column should be right-aligned. The desired alignments can be obtained using f-strings.

Word	Count

xxxx	####
xxxx	####
xxxx	####

This function should not return any value.

count_by_length()

This function should accept a parameter named **words**, which is expected to contain a list of strings representing words. The function should determine the number of strings in **words** of each possible length and display the resulting counts.

1. Create an empty dictionary named **count_dict**.
2. Loop over the elements of **words**. For each element of **words**, calculate the length of the element, storing the result in a variable. If the length found in Step a has been previously added as a key in **count_dict**, then increment the value corresponding to that key. If the length does not appear as a key in **count_dict**, then add it as a key with a value of 1.
3. Create an empty list named **count_list**. Loop over **freq_dict.items()**. For each key/value pair in **count_dict**, create a tuple of the form **(key, value)** and append this to **count_list**.
4. Sort **count_list** in descending order. Note that this will sort the list of tuples according to the first element in tuple, which represents a specific word length.
5. Print the results in **count_list** in the format shown below. The xxxx symbols should be replaced with word lengths and the ##### symbols should be replaced with word counts. The dashed line should be 16 characters long. Allot 12 characters for the word column and 4 characters for the count column. The word column should be left-aligned and the count column should be right-aligned. The desired alignments can be obtained using f-strings.

Length	Count

xxxx	####
xxxx	####
xxxx	####

This function should not return any value.

count_by_first()

This function should accept a parameter named **words**, which is expected to contain a list of strings representing words. The function should determine the number of strings in **words** with each possible starting letter and display the results.

The steps performed by this function are very similar to those described in the **count_by_length()** function. The main difference is that you will be using the first characters of strings in **words** as keys in **count_dict** rather than the length of the string. Note that if **my_string** is a string, then you can access the first character of **my_string** using **my_string[0]**.

Print the results in the format shown below. The x symbols should be replaced with letters and the #### symbols should be replaced with word counts. The dashed line should be 16 characters long. Allot 12 characters for the word column and 4 characters for the count column. The word column should be left-aligned and the count column should be right-aligned. The desired alignments can be obtained using f-strings.

The rows in your output should be arranged so that the letter column is in **increasing** order from a to z.

Letter	Count
x	####
x	####
x	####

This function should not return any value.

Instructions for the Notebook

Any set of instructions you see in this document with an orange bar to the left will indicate a place where you should create a markdown cell. If no instructions are provided regarding formatting, then the text should be unformatted.

Any set of instructions you see with a blue bar to the left will provide instructions for creating a single code cell.

Assignment Header

Create a markdown cell with a level 1 header that reads: "COSC – Project 02". Add your name below that in bold.

Introduction

Create a markdown cell with a level 2 header that reads "Introduction". Add unformatted text explaining the purpose of this project. You may paraphrase the statement in the first paragraph of this document. Explain that you will be using this notebook to test the functions you have created.

We will start by running the script.

Use the Magic command **%run -i word_count.py** to run the contents of your script.

Testing the `process_word()` Function

Create a markdown cell with a level 2 header that reads "**Testing the `process_word()` Function**". Add unformatted text explaining that you will testing the function using a few test strings.

Run the following lines of code:

```
print(process_word('Test!'))
print(process_word('WHAT?!?!?'))
print(process_word('!_(;hEl\`l0?,)123'))
```

This should print the following strings, each on different lines: **test**, **what**, and **hello**.

Testing the `process_line()` Function

Create a markdown cell with a level 2 header that reads "**Testing the `process_line()` Function**". Add unformatted text explaining that you will use a test string to test the **`process_line()`** function.

Create a string named **`test_string`** with the following contents:

```
'This is a "test string". It\'s fifty-eight characters long!'
```

Pass the variable **`test_string`** to the **`process_line()`** function and print the list that is returned by the function. If the function behaves correctly, you should get the following list:

```
['this', 'is', 'a', 'test', 'string', 'its', 'fifty', 'eight', 'characters', 'long']
```

Processing the File

Create a markdown cell with a level 2 header that reads "**Processing the File**". Add unformatted text explaining that you will now use the **`process_file()`** function to read and process the contents of the file **`tale_of_two_cities.txt`**.

Use the **`process_file()`** function to read in the file **`tale_of_two_cities.txt`**, storing the returned list into a variable named **`words`**. Then print the message shown below with the xxxx characters replaced with the number of elements in the list **`words`**. Make sure that the output message is formatted correctly. There should be exactly one space between any two consecutive words or numbers in the sentence.

```
There are xxxx words contained in the file.
```

We will now print the first few words contained in the novel.

Create a new code cell to print the first 20 words of the list **`words`**. Use list slicing to select the first 20 words. Do not use a loop in this code cell.

Unique Words

Create a markdown cell with a level 2 header that reads "**Unique Words**". Add unformatted text explaining that you will now determine the number of unique words in the novel.

Use the `find_unique()` function to create a list of unique words contained in the novel. Store the resulting list in a variable named `unique`. Then print the message shown below with the xxxx characters replaced with the number of elements in the list `unique`. Make sure that the output message is formatted correctly. There should be exactly one space between any two consecutive words or numbers in the sentence.

```
There are xxxx unique words contained in the file.
```

Word Frequency

Create a markdown cell with a level 2 header that reads "**Word Frequency**". Add unformatted text explaining that you will create a dictionary containing word counts for the words in the novel.

Pass the list `words` to the `find_frequency()` function, storing the returned dictionary in a variable.

Then create a list containing four strings, each of which representing a word that appears in the novel at least 100 times, but fewer than 1000 times. It may require a little bit of trial and error and exploration to find four such words. After finding them, loop over the list printing the message shown below for each of the four words. The zzzz characters should be replaced with the actual word, and the xxxx symbols should be replaced with the number of times that particular word appeared in the novel. The double quotes shown below should be included in your output, and there should be exactly one space between any two consecutive words or numbers. Use the dictionary you created to determine the word count for each word.

```
The word "zzzz" appears xxxx times in the file.
```

Most Common Words

Create a markdown cell with a level 2 header that reads "**Most Common Words**". Add unformatted text explaining that you will find and display a list of the 20 most common words found in A Tale of Two Cities.

Use the `most_common()` function along with the `words` list to display the 20 most common words in the novel along with the number of times each word appears.

Stop Words

Create a markdown cell with a level 2 header that reads "**Stop Words**". Add unformatted text explaining that you will create a list of commonly occurring "stop words" that will be removed from the words list.

Use `process_file()` to read the file `stopwords.txt`, storing the result in a variable named `stop`. Then print the message shown below with the xxxx characters replaced with the number of elements in the list `stop`. There should be exactly one space between any two consecutive words or numbers in the sentence.

```
There are xxxx words in our list of stop words.
```

To get a sense as to the sort of words that appear in the list of stop words, we will display the first 50 stop words.

Create a new code cell to print the first 50 words of the list `stop`. Use list slicing to select the first 50 words. Do not use a loop in this code cell.

Counting Non-Stop Words

Create a markdown cell with a level 2 header that reads "**Counting Non-Stop Words**". Add unformatted text explaining that you will determine the number of non-stop words and the number of unique non-stop words found in the novel.

Use the function `remove_stop()` to remove the stop words from the list `words`, storing the result in a variable named `words_ns`. Then use `remove_stop()` to remove the stop words from the list `unique`, storing the result in a variable named `unique_ns`. State the number of elements in each of these new lists by printing the messages shown below with the xxxx characters replaced with the appropriate values.

```
There are xxxx non-stop words contained in the file.  
There are xxxx unique non-stop words contained in the file.
```

To get a sense as to the sort of words that appear in the list of stop words, we will display the first 50 stop words.

Create a new code cell to print the first 50 words of the list `stop`. Use list slicing to select the first 50 words. Do not use a loop in this code cell.

Most Common Non-Stop Words

Create a markdown cell with a level 2 header that reads "**Most Common Non-Stop Words**". Add unformatted text explaining that you will display the 20 most commonly occurring non-stop words.

Use the function `find_frequency()` with the list `words_ns` to create a dictionary of word counts for the non-stop words. Store the result in a variable named `freq_ns`. Then use `most_common()` to display the 20 most commonly-occurring non-stop words in the novel.

Counting Words by Length

Create a markdown cell with a level 2 header that reads "**Counting Words by Length**". Add unformatted text explaining that you will display information concerning the distribution of lengths of unique words found in the novel.

Use the function `count_by_length()` with the list `unique` to display a count of the number of unique words of each length appearing in the novel.

Longest Words

Create a markdown cell with a level 2 header that reads "**Longest Words**". Add unformatted text explaining that you will display the longest several words found in the novel.

If done correctly, the results from the previous code cell should have stated that the novel contains 1 word with 17 characters, 4 words with 16 characters, and 17 words with 15 characters. We will now display these 21 words.

To accomplish this task, we will use the option `key` parameter of the `sorted()` function. Arguments for the `key` parameter should be functions that accept elements of the list being sorted. The list will be sorted according to the values returned by the function when it is applied to each element. For example, the expression shown below will sort the elements of the list `unique` in decreasing order according to their length.

```
sorted(unique, key=len, reverse=True)
```

Use the expression above to sort the words in `unique` according to their length. Then use a loop to print the first 21 elements of the resulting list, with one word per line.

Counting Words by First Letter

Create a markdown cell with a level 2 header that reads "**Counting Words by Length**". Add unformatted text explaining that you will display the number of unique words with each possible first letter.

Use the function `count_by_first()` with the list `unique` to display a count of the number of unique words with each possible first letter appearing in the novel.

Submission Instructions

When you are done, click **Kernel > Restart and Run All**. If any cell produces an error, then manually run every cell after that one, in order. Save your notebook, and then export the notebook as an HTML file. Upload the HTML file to Canvas and upload the IPYNB file to CoCalc, placing the file into the folder **Projects/Project 02**.