**Regex – Normalization**

(HW1)

**Harsha Vardhan Khurdula**

**Purdue Fort Wayne**

**CS 59000-08 Natural Language Processing**

**Instructor – Dr. Jonathan Rusert**

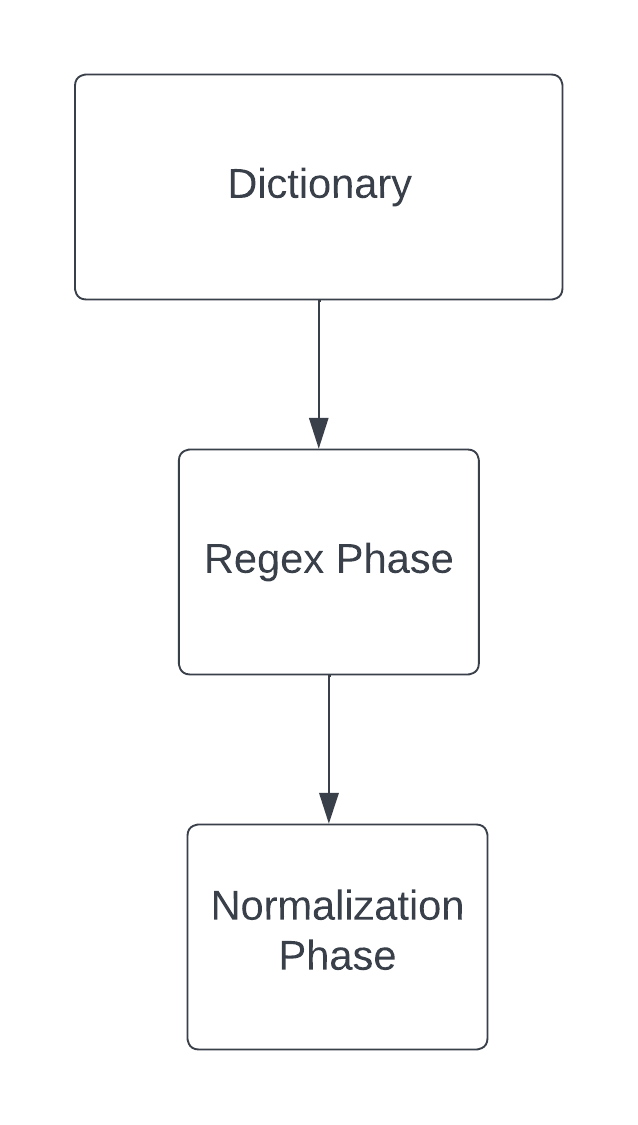
**January 17Th , 2023.**

**Approach and Observations for implementing the task:**

My primary approach to the given task of creating a Dictionary (a lexicographical set of words.), is to divide the entire process into two phases:

* Regex Phase &
* Normalization Phase.

While using Object Oriented approach, in order to be able to create a module named ‘*Dictionary*’, instead of just writing a simple python script.



***Flowchart i. Dictionary() breakdown.***

**Regex Phase**

The Regex Phase is the initial filter, that aims to capture certain words in English that are spelled differently based on Geography (British spelt words in this context, that are spelled differently in United States.) and replace them with their equivalents used in United States. It is also responsible for replacing abbreviated titles, which are used to address other people such as Mr. or Dr. or Ms. Etc.

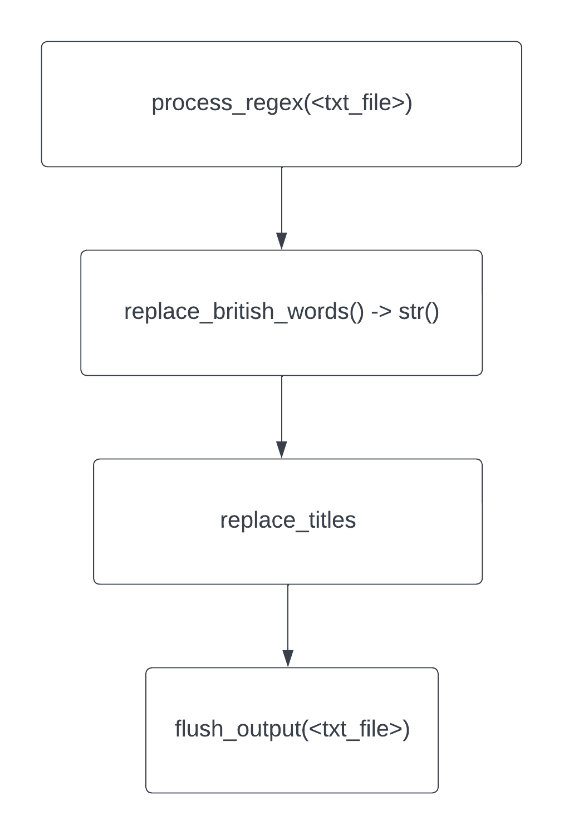
*Quantification:*

The idea would be creating a set of methods that support the two main functionalities within the module.

***Dictionary().process\_regex(self, <filename>):***

The first task would be to replace all British words, my initial approach was creating a finite pair of words, as a dictionary, for faster lookup of the pattern in loop, will print every match before replacing the words, by force debug/printing. Which is a totally wrong approach considering the criteria mentioned, “Do not use Data Structures to store and replace words, use a regular expression to capture and substitute those words.” So I divided the task into methods, and work on them as I figure out the regular expression which resulted in the following methods that process\_regex(self, <filename>) uses:

* replace\_british\_words(self) -> str()
* replace\_titles(self) -> None
* flush\_output(self, filename) -> None



***Flowchart ii. process\_regex breakdown.***

* ***replace\_british\_words(self) -> str()****:*

I went back to my IDE, and started messing around with regular expressions, to replace words like Colour with Color, Flavour with Flavour.

The initial regex that helped replace all these words was, **/.\*our/**

However, the pattern would also cause replacement of the word, our in general. Which seemed pretty undesirable. After some pensive trial and error, **"/(<=.)our)/g**" is the pattern that I came up with, and with this as a parameter to *re.sub(<this\_pattern>, "or", <text>),* I was able to replace most of the basic British spelled words such as:

- Colour, Neighbour, Humour, and Flavour.

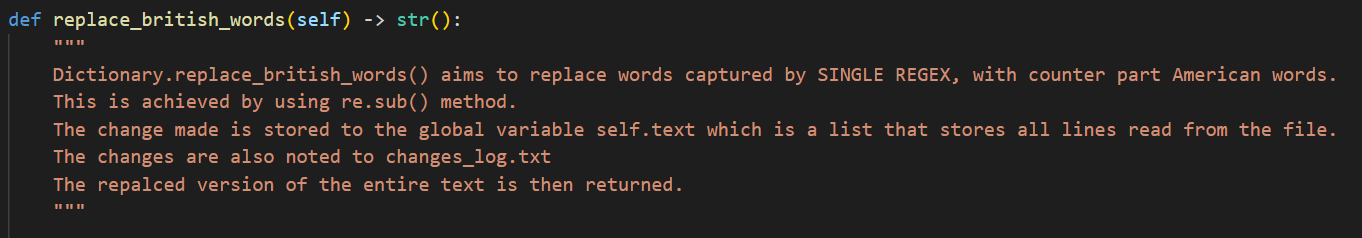
but not – ‘Our’.

I also realized that there will be words that will be spelt the same way in both of these vocabularies. Upon some browsing for words that are spelt the same way in both these languages, this expression captures the following words which are spelled the same in both vocabularies.

Examples:

'Contour', 'Velour', 'Paramour' and 'Troubadour'

However, these words are not used with in the specific book, "The war of the worlds." And it is far better than having to replace a simple word like ‘our’.

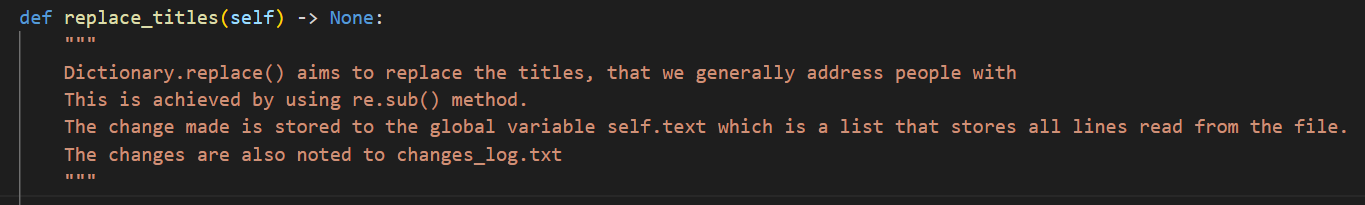


***Img i. Method Prototype and Description for replacing British words***.

* ***replace\_titles(self) -> None***:

Replacing abbreviated titles, was much simpler as we could simply use python dictionary to store <key> : <value> pairs in the form of <title> : <abbreviation>. And then substitute all the occurrences with the pattern/title/key with its value/abbreviation while iterating through the data structure that stores these titles. The global attribute that stores these key : value pairs is,

* + Dictionary().titles = dict()

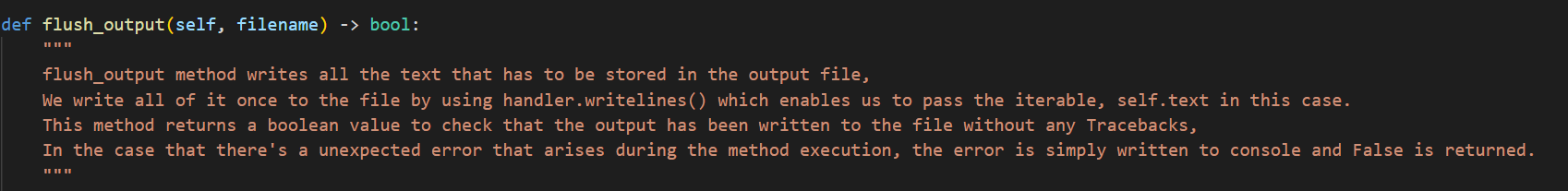


***Img ii. Method Prototype and Description for replacing abbreviated Titles.***

* ***Flush\_output(self, filename) -> None:***

Now that we’ve replaced the necessary words, we are at the end of phase I, all we have to do is store the changes made to a text file named, ‘regex.txt’. The changes made are stored in a str() attribute of Dictionary

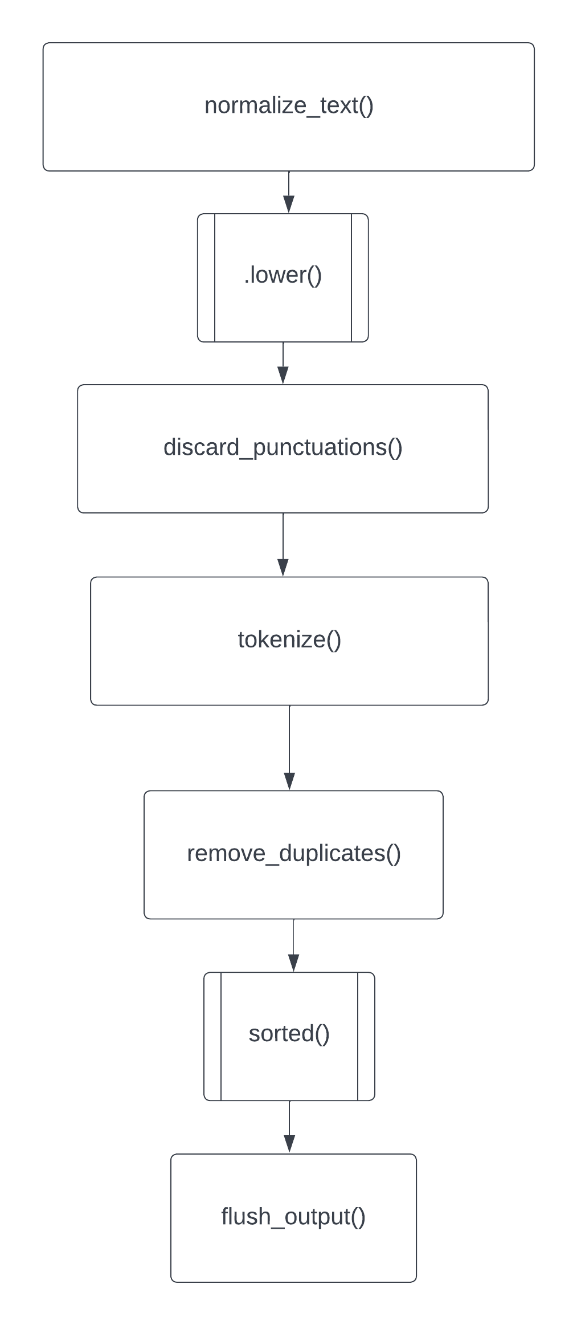
* + Dicitonary().text =str()



***Img iii. Method Prototype and Description for storing output to ‘regex.txt’***

**Normalization Phase**

The normalization phase mainly aims at capturing and storing words that are actually contextual and can be processed later. This phase has been divided into sub methods as well to ease to use of it programmatically. The functionality however, is driven by our method *Dictionary.normalize\_text(self, filename) -> None*

******

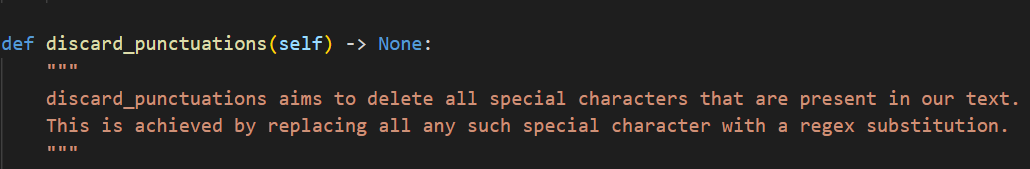
***Flowchart iii. Normalization breakdown.***

I took a linear approach to normalize the previously stored text, from Phase I. The steps involved are:

1. Make all words of same CASE (lower) used basic lower method by reading all the text at once, this is simply achieved by using inbuilt method for strings, string.lower() method.

2. Strip away all the punctuations:

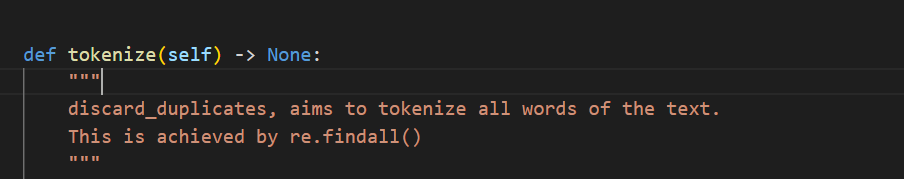
The basic idea here is to remove all the punctuations that are used in writing. For this a predefined string is used for storing a string of all the possible punctuations that can be used while writing text. And this string is used to replace each punctuation with an empty string thus replacing a special character with “” by using re.sub(). The method that enables this is ***Dictionary.discard\_punctuations(self) -> None***:



***Img iv. Method Prototype and Description for removing punctuations from text.***

3. Tokenize:

The objective of this step is to capture all the words that are present within the text. This is achieved by using a method Dictionary.tokenize(self) -> None: which implements re.findall() that returns a list of words, matching the pattern “/[a-z]+/” which are non empty strings of individual words.

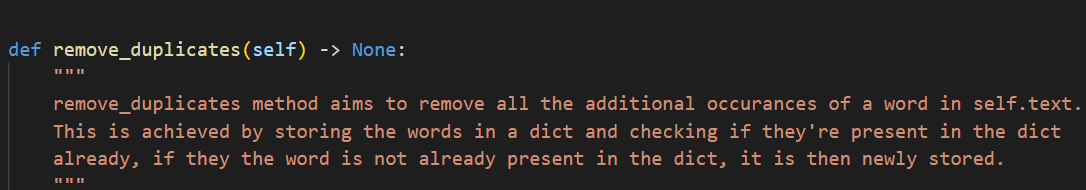


***Img v. Method Prototype and Description for tokenizing the text.***

4. Discard duplicates:

This step involves removing the reoccurring words within the list that we acquired from the previous step i.e. Tokenize. This is done, by using a python dict which maps every word to self, and we iterate through the list of words, while storing a word within dictionary if it was not stored previously, on the other hand if the word is already present on the dictionary, we simply ignore the current occurrence and move on to next the word. This is achieved by a method,

***Dictionary.remove\_duplicates(self) -> None:***



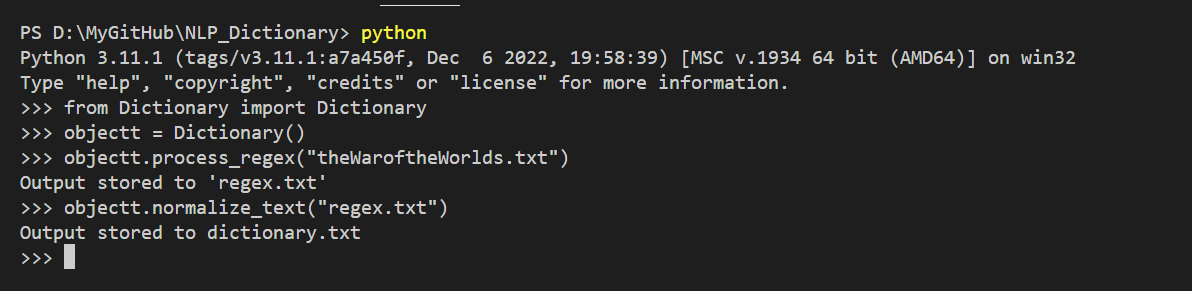
***Img vi. Method Prototype and Description for removing duplicate occurrences of words.***

5. Sort the words, in a lexicographical fashion by using inbuilt python method, sorted() which sorts the elements of an iterable. In this case, the list of non-duplicate words are lexicographically arranged.

5. Flush the output to dictionary.txt, this is achieved by using the same flush\_output() method which is used within regex phase of this module.

**Sample Use Case:**

Assuming that you are in the Directory with the module and .txt file the following would be the commands to run the script.



* from Dictionary import Dictionary() #we are importing the class() from the module.
* Object = Dictionary() #this is redundant but it eases the use.
* Object.process\_regex(“filename.txt”) #executes the process\_regex() method.
* Object.normalize\_text(“filename.txt”) #executes the normalize\_text() method.

**References:**

(2019, October 15). *String.Punctuation in Python*. Geeksforgeeks. Retrieved January 16, 2023, from <https://www.geeksforgeeks.org/string-punctuation-in-python/>

Python.org (2022, October 11). *re - Regular*. The Python Standard Library. Retrieved January 16, 2023, from <https://docs.python.org/3/library/re.html>

Khurdula, H. V. (2023, January 14). *NLP\_Dictionary*. Github. <https://github.com/Khurdhula-Harshavardhan/NLP_Dictionary.git>