Introduction

In this program, the objective was to create a list of anagrams of a specific word, which was attained from user input. The program then read a list of words off a downloaded text and stored them into a set. There were two methods to implement in order to optimize the code. The first, was to create a method to dismiss any anagram duplicates. More specifically it would disregard the second character if it presented itself twice. The second method was created to store the prefixes of all the words from the word set into a second set. This would speed up the process of creating the anagrams because it would first check to see if the partial word was a prefix of any word in the word set. Both of these methods were called in the method that created the anagrams. If the word did not contain a duplicate and it was found in the set containing all the prefixes then the anagrams was created.

Design and Implementation

The first step that I took was to create the method that investigated whether or not the user’s word contained any duplicate characters. I established the obvious base case which stated that if the length of the word was 1 it would return false, false being there was no duplicate character detected. In the else, I nested in another if and else statement. Before going on to the next if statement I assigned the word of its index of 1 and so forth ([1:]) to the variable named r\_letters (remaining letters). This was done in order for it to be possible to compare the remainder of the characters of the word – besides the character at index 0 – to the character at index 0. If this comparison was correct the statement would return true, else it returned the method’s name passing the user’s word with its index at 1 and so forth ([1:]) in its parameters.

For the next method, which included creating the set containing the prefixes for all the words from the download, I initially attempted to split the string up into sections. First, I included a for loop that would iterate through the whole first set that was created to hold all the words available. Within the loop, I split the string and stored it into a variable named split\_word. I then created an empty set that was meant to hold all the prefixes. After more thought put in, I realized how to successfully insert the prefixes into a second set. First, I sent in the set that contained all the words (my\_set) through the parameters and immediately created a prefix variable (prefix) that contained an empty string. A for loop was then necessary to iterate through the range of the length of the word set. A temporary prefix set was created (temp\_prefix) which stored the ith index of my\_set and this was done to save each character. Then the variable prefix and temp\_prefix were added together and stored into the variable prefix. Essentially the previous character and the next character were being combined. And finally, the prefix set was added to the empty string that was created earlier with the variable name “prefix”.

The third and final method was created to produce the anagrams. But, before I could define the method I inserted a variable that would start the time the method to create the anagrams took to run. I printed the time outcome in the main. Within this method I passed the user input which was named r\_letters (remainding letters) and an empty string named s\_letters (scrambled letters). I inserted the first if statement which again declared the obvious base case that stated if the length of the r\_letters equals to 0 then it would print the s\_letters. The else, began with a for loop which was to iterate through r\_letters. Within the for loop I created a second variable to store the scrambled letters (scrambled\_letters) at index i, i being what is being iterated. Another variable was created named remaining\_letters which was set to r\_letters at the ith index to the first index available ([:i]) plus r\_letters at index ith + 1 ([i:]) to the last index available. This was to remove the letter from the remaining list in order to allow for any word scrambling. And finally, the method calls on itself taking in remaining\_letters and s\_letters + scramble\_letter as its parameters.

I knew then that after all the methods were created, the two methods that contained the prefix set and the investigation of duplicates, had to be implemented within the method that created the anagrams. This was another difficult task. My attempt was to include an if statement after the method was defined which stated if the check\_duplicate method returned true, again meaning the word contained a duplicate element, then it would exit the system. If it returned false then the anagram would be composed. The prefix method was also meant to return a Boolean, true if the word was found within the prefix set and false if it was not. Therefore, if the checking for duplicates method returned False and the prefix method returned true then the anagrams for the word would successfully print.

The design choices created themselves. I knew the very first thing that was necessary in order to set the code into motion was to hard code the file that was downloaded into a set. Then from there the user was asked for a word and immediately the word was tested to see if it was even eligible, meaning if it was even in the list of words downloaded. All this was stored in the main. Then next came the three methods.

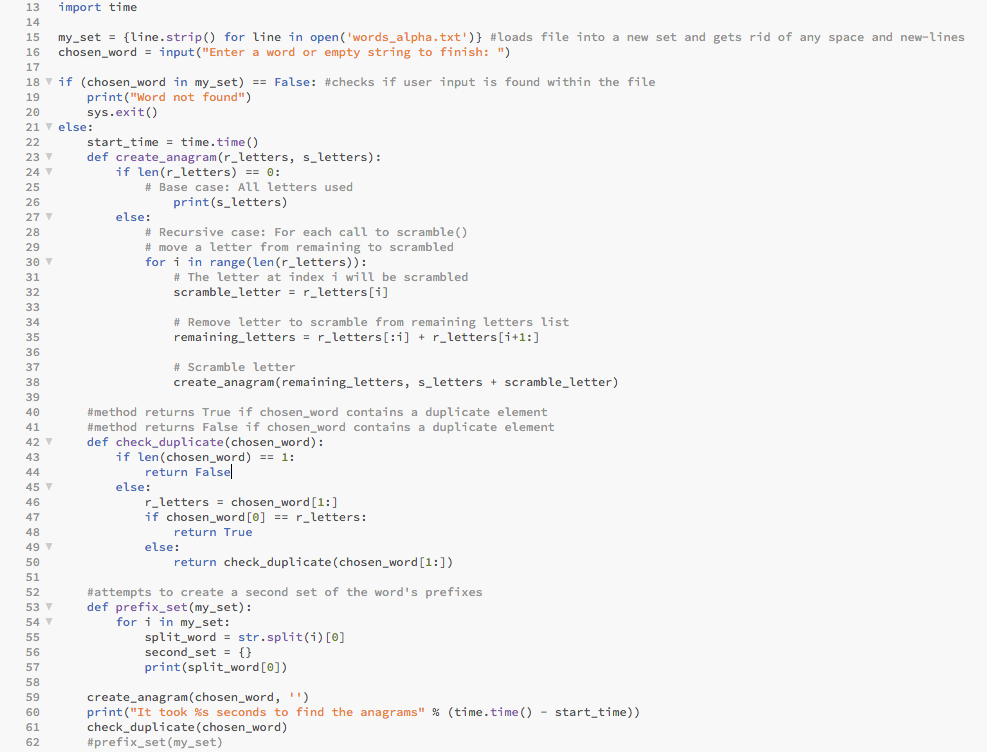
Experimental Results

The experiment that was conducted was inputting a word that was not found within the list of downloaded words. The name *Anakin* was inserted and the anagram was run. The only problem with was that name, *Anakin*, was not found within the list but instead various similar words such as *canakin*, *hyperanakinesia*, and many more where they all contained the name *Anakin*. With this in mind, the user’s input could simply not be checked to see if it was found within the list by using the set of prefixes. If this were to be done then it would return true, meaning the word is eligible to use even though it is not since the word *Anakin* alone is not part of the downloaded words. As to avoid this, as soon as the user input was set to a variable, the word was then sent to an if statement that stated if the user’s word was found within the list of downloaded words it would continue in its objective. If not, then the system would exit. With this implemented, the word *Anakin* was no longer eligible, but the word *canakin* was, and rightfully so.

Conclusions

From this lab, a more in-depth understanding of recursion was gained. Tracing was a necessity for figuring out how to implement recursion and in that there was trial and error. This repeated practice allowed me to adapt the best way to trace these recursion problems which was drawing out boxes with arrows clearly indicating the order of what was being returned. A visual representation of recursion made it so I was able to understand it more clearly.

Appendix



Academic Honesty Certification

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.

Sofia Gutierrez September 9, 2019