COMP1551: Core Programming
Coursework I (Provisional Version)*
Anagram Antics
Due: 10am Monday 24th November

Brandon Bennett

This coursework is worth 10% of your final module grade.

Your submissions will be marked during the lab sessions on Monday 24th and Tuesday 25th November (both at 2-4pm).

Be sure to attend the lab session allocated on your timetable on one of these days.

Prelimiaries

In this coursework you will be implementing a program that has a complex functionality built up from a number of small and relatively simple functions. The specification takes you through the programming function by function. You should code up all the functions in a file `anagrams.py`. It will be easiest to create the program by coding and testing the functions in the order they are specified. Each function can be coded independently, without knowing what the other functions do. However, it may be useful to read ahead to see where you are going. In particular, make sure you read the hints carefully.

Program Specification (Function by Function)

1. Define a function `anagram(` *str1*`,` *str2* `)`, which takes two strings as arguments and returns `True` if the words have exactly the same letters in them, otherwise it returns `False`. More precisely any letter that occurs in one of the strings should occur exactly the same number of times in the other. [6 marks]

   Hints: For example, if you implement `anagram` correctly, you should be able to carry out the following Python interaction:

   ```
   >>> anagram( "sidebar", "seabird" )
   True
   >>> anagram( "cheese", "pancakes" )
   False
   ```

   What should `anagram( "rat", "rat" )` return? The specification makes it clear that it should return `True`, though you might not normally call a word an anagram of itself.

   There are various ways to code the `anagram` function. One fairly straightforward way involves turning the strings into lists of characters using the built-in `list(` *str* `)` function. Alternatively, if you look at what value is returned when you apply the built-in function `sorted` to a string, you may be able to define `anagram` very concisely.

*Some small clarifications may be made in the final version, but it will be essentially the same as this.

2. Define a procedure `test_anagram()`. This should call your `anagram` function with different string arguments. It should do this for at least three positive and three negative cases. For each case the string inputs and the result obtained should be printed out in an easily understandable format. [2 marks]

3. Define a function `get_dictionary_word_list()`. This should return a list of words (i.e. strings) obtained by reading from the file `dictionary.txt`, which you should download from the VLE. Ensure that the strings in the returned list do not have newline characters at the end. [4 marks]

   Hints: Make sure the file `dictionary.txt` is in the same directory as your Python code file.

   This is nothing to do with Python's 'dictionary' data type. Here, we are just dealing with a file that contains a huge list of all but the rarest words in the English language (i.e. the words that would be listed in an ordinary dictionary.)

   Make sure you `return` the list of anagrams from your function. Do not `print` it out. That's a very different thing, although when using the command prompt they may seem to have the same effect.

4. Define a procedure `test_get_dictionary_word_list()`.

   This should call your function `get_dictionary_word_list` to obtain the word list from the dictionary file. It should then print out the number of words in the list and the first 10 words in the list [2 marks]

   Specifically, `test_get_dictionary_word_list()` should print output similar to the following:[1]

```
Number of words in list: 178691
First 10 words are:
aa
aah
aahed
aahing
aahs
aal
aalii
aaliis
aals
aardvark
```

5. Define a function `find_anagrams_in_word_list(` *str*, *str_list* `)`, where the input parameters should be a string and a list of strings. The value returned should be a list of all those strings in *str_list* that are anagrams of *str*. You should code this making use of your previously defined function `anagram`. [4 marks]

   Hints: Remember that the function should `return` the list of anagrams not `print` it out. You don't need to write a separate test procedure for this (although you may if you wish). This will get tested by the `test_find_anagrams` procedure specified below.

6. Define a function `find_anagrams(` *str* `)`, that returns a list of all those words in the dictionary file `dictionary.txt` that are anagrams of the input string *str*. You should code this making use of your previously defined functions `get_dictionary_word_list` and `find_anagrams_in_word_list`. [2 marks]

[1] If you get additional line spaces, these are probably due to newline characters at the end of the strings in the list.

Hint: This should be fairly easy. You just need to use `get_dictionary_word_list` to get the word list. Then you can send the string and the word list to `find_anagrams_in_ word_list` to get the desired result.

7. Define a procedure `test_find_anagrams`, which calls the `find_anagrams` function for a variety of input strings. This procedure should test at least 6 strings and illustrate cases of strings having different numbers of anagrams. [2 marks]

8. Define a function `partial_anagram(` *str1*`,` *str2* `)`, which returns `True` if every letter that occurs in *str1* occurs at least as many times in *str2*, otherwise it returns `False`. (Note: there may be additional letters in *str2* which do not occur in *str1*, which is why *str1* is only a partial anagram of *str2*.) [4 marks]

   Hint: If you convert *str2* into a list, you can then loop over the characters in *str1* checking that each is present in *str2*. If it is not present then *str1* cannot be a partial anagram of *str2* (so return `False`). If it is present you need to remove that character from the list (since it has been used up) and then carry on looping. If you loop through all the characters in *str1* and always find a match in *str2*, then you can return `True`.

9. Define a function `find_partial_anagrams_in_word_list(` *str*`,` *str_list* `)`, where the input parameters should be a string and a list of strings. The output should be a list of all those strings in *str_list* that are partial anagrams of *str*. You should code this making use of your previously defined function `partial_anagram`. [2 marks]

   Hints: You can define this in much the same way as you did for `find_anagrams_in_ word_list`, except you are looking for partial anagrams rather than complete anagrams.

   If you define it correctly you should be able to carry out the following Python interaction:

```
>>> wordlist = get_dictionary_wordlist()
>>> find_partial_anagrams_in_wordlist( "brandon", wordlist )
['ab', 'abo', 'ad', 'ado', 'adorn', 'an', 'and', 'andro', 'anon',
 'ar', 'arb', 'ba', 'bad', 'ban', 'band', 'bar', 'bard', 'barn',
 'baron', 'bo', 'boa', 'boar', 'board', 'bod', 'bond', 'bora',
 'born', 'bra', 'brad', 'bran', 'brand', 'bro', 'broad', 'dab',
 'dan', 'darb', 'darn', 'do', 'dobra', 'don', 'dona', 'donna',
 'dor', 'drab', 'na', 'nab', 'nan', 'nard', 'no', 'nob', 'nod',
 'nona', 'nor', 'oar', 'oba', 'od', 'oda', 'on', 'or', 'ora',
 'orad', 'orb', 'rad', 'radon', 'ran', 'rand', 'road', 'roan',
 'rob', 'roband', 'rod']
```

10. Define a procedure `test_find_partial_anagrams_in_word_list`(). Write a suitable test procedure that displays the results of calling `find_partial_anagrams_in_wordlist` for at least 4 different words, using the dictionary word list. [2 marks]

11. Define a function `remove_letters(` *str1*`,` *str2* `)`, which returns the string obtained by removing from *str2* every letter occurring in *str1*. [4 marks]

    Hints: This can be done with 4 or fewer lines of code. There are several ways of doing it. If you convert *str2* into a list, you can then use the `remove` method to take letters out of the list. Alternatively, you can operate on the string directly using the `replace` method. Note that `remove` changes the actual list it operates on (e.g. try `L = ['a', 'b', 'c']` followed by `L.remove('b')` then look at the value of `L`); whereas, the `replace` method returns a new string as its value (so you would do something like `newstring = "this string".replace("tri", "o")`).

Note that `"this string".replace("s", "ck", 1)` will replace just the first occurrence of `"s"` in the string, whereas if you leave of the final argument, all occurrences will be replaced.

Python has a rather odd quick way to join a list of strings together to form a single string: `joined_string = ''.join(['a','b','c'])`. This could be useful.

You may assume that all letters in *str1* do in fact occur in *str2*. This will be the case in the context we will be using it. However, it would be safer to check this and give and signal an error or warning.

12. Define a function `test_remove_letters()`, performs a number of test calls of `remove_letters` to check it is working correctly. [2 marks]

13. `find_two_word_anagrams_in_word_list(` *str*, *str_list* `)`, where the input parameters should be a string and a list of strings. The output should be a list of all strings made up of two words separated by a space, such that both of these words are in *str_list* and the combination of the two words is an anagram of *str*. [6 marks]

    Hints: The following Python interaction illustrates the correct operation of this function:

```
>>> wordlist = get_dictionary_wordlist()
>>> find_two_word_anagrams_in_wordlist( "brandon", wordlist )
['and born', 'band nor', 'barn don', 'barn nod', 'bond ran',
 'born and', 'born dan', 'bran don', 'bran nod', 'brand no',
 'brand on', 'dan born', 'darn nob', 'don barn', 'don bran',
 'nard nob', 'no brand', 'nob darn', 'nob nard', 'nob rand',
 'nod barn', 'nod bran', 'nor band', 'on brand', 'ran bond',
 'rand nob']
```

    This may seem a bit tricky, but now we have all the supporting functions in place it should be relatively straightforward. Anyway, I will explain how I did it, so you can follow that method if you wish. Or you can try your own way. Here is an informal, but structured description of my algorithm:

    - Initialise a list `two_word_anagrams` to the empty list, `[]`.
    - Call `find_partial_anagrams_in_word_list` to get a list of all the partial anagrams of *str* that can be found in the word list.
    - Then, do a loop that runs over all these partial anagrams. For each partial anagram *part_anag* do the following:
      - Remove the letters of *part_anag* from the input string, *str*, to get a string, *rem* that is the remaining letters after taking away the partial anagram;
      - Call `find_anagrams_in_word_list` on *rem* (and the input word list) to get a list of anagrams of the remaining letters;
      - For each anagram *rem_anag* of the remaining letters, form the string *part_anag* `+ " " +` *rem_anag* and add it to the list `two_word_anagrams`.
    - At this point the list `two_word_anagrams` should have all the two word anagrams in it, so return that list from your function.

14. Define a procedure `test_find_two_word_anagrams()`. This should call `find_two_word_anagrams_in_word_list` for various words, using the word list obtained from the dictionary file. At least 6 words of varying lengths should be tried. [2 marks]

15. This last part will not be so prescriptively specified, so you will need to work out yourself how to break it down into manageable functions (I used 3 functions to implement it). If you have had enough, you could skip this part since it has comparatively low marks for the amount of work involved. But now you have come this far, it should not take to much longer to finish this last part. And if you finish you will definitely get some pay-back in terms of entertainment as well as marks.

    To do this part, you will need to download the file `students.txt` from the VLE.

    Your aim is to find all one or two word anagrams that can be formed the name of each student in the Core Programming class. For each student you should take the letters contained in their first and last names and compute the list of all one and two word anagrams of these letters.

    More specifically, your procedure should run through all students in alphabetical order of their surname. For each student it should print:

    - the full name of the student, [1 mark]
    - the first and last names of the student, [1 mark]
    - a string containing of all letters in the first and last names of the student, but without any spaces, hyphens or apostrophes and with all letters in lower case, [1 mark]
    - a list of all one word anagrams of the letter string specified in the previous item, [1 mark]
    - a list of all two word anagrams of the specified letter string. [2 marks]

    Hints: You need to read the names in from the file `students.txt`. There is various other information in the file, so you need to strip that out. The slice operator and `strip` method will be useful for this. The `split` method will be useful for dividing the name into its separate parts. You always get the surname followed by the first name, so you can just take the first two. To get rid of hyphens and apostrophes you can use the `replace` method.

    If you include middle names you will not get much joy from the program. The problem is that you will have too many letters to make two words. The program will find very large numbers of possibilities for the first word (i.e. partial anagrams of the name) but will then not be able to find a second anagram to use up the rest of the letters. This means that as well as finding hardly any anagrams, the system will take an extremely long time to run.

    You will find hardly any single word anagrams of any student names. I think there is only 1. This is mainly because nearly all English words are shorter than the number of letters occurring in most people's names. But there are lots of two word anagrams.

[50 marks total]

Submission Instructions

Submit your solution via the VLE. It should consist of a single file, which should start with comments giving your name and student ID. You will have to rename your file `anagrams.py` to `anagrams.txt` before uploading it to the VLE.

Any questions about this assessment should be posted in the coursework forum for Core Programming on the VLE.