Object-Oriented Programming
50:198:113 (Fall 2018)

Homework:	**2**	**Professor:**	**Suneeta Ramaswami**
Due Date:	**10/8/18**	**E-mail:**	`rsuneeta@camden.rutgers.edu`
Office:	**321 BSB**	**URL:**	`http://crab.rutgers.edu/~rsuneeta`
		Phone:	**(856)-225-6439**

Homework Assignment 2

The assignment is due by 11:59PM of the due date. The point value is indicated in square braces next to each problem. Each solution must be the student's own work. Assistance should only be sought or accepted from the course instructor. Any violation of this rule will be dealt with harshly.

This assignment requires you to use lists and dictionaries, and carry out some file processing (read from and write into files). In this and all future assignments, you are graded not only on the correctness of the code, but also on clarity and readability. Hence, I will deduct points for poor indentation, poor choice of object names, and lack of documentation. For documentation, use a common sense approach. While I do not expect every line of code to be explained, all code blocks that carry out a significant task should be documented *briefly* in clear English.

Please read the submission guidelines at the end of this document before you start your work.

Important note: When writing each of the following programs, it is important that you name the functions exactly as described because I will assume you are doing so when testing your programs. If your program produces errors because the functions do not satisfy the stated prototype, points will be deducted.

1. I will type `python3 problem1.py` at the command line to test `problem1.py`, and
2. After importing the `problem2` module into the Python shell, I will call the `freq_distribution` and `ordered_freq_distribution` functions to test `problem2.py`.

For each problem, put all your function definitions in one file. *Do not create a separate file for each function.* The files `problem1.py` and `problem2.py` should contain *all* the function definitions for Problems #1 and #2, respectively.

Problem 1 [25 points] Numbering lines in a text file. An instructor for a writing-intensive course evaluates students on the basis of written essays. Students submit essays electronically as text files. The instructor then provides comments on specific lines or specific paragraphs of the essay, so that the students may revise and improve them. In order to provide this kind of feedback, it would be extremely useful to have an automated way to number the lines and the paragraphs of a text file. In this problem, you are asked to write a program to accomplish this task. Further details are described below.

1. Write a function called `insert_line_para_nums` with two parameters, `infilename` and `outfilename`. This function reads from a file called `infilename` (this is the file containing an essay) and writes into a file called `outfilename`. Each line of `infilename` is written exactly as it is into `outfilename`, but with the difference that a pair of numbers,

namely the paragraph number and line number, are inserted at the start of each line. The two numbers are separated by a comma, followed by a space, and then the line of text. Remember to open and close all files as necessary.

Note the following points:

- Blank lines separate paragraphs. There may be more than one blank line between paragraphs.
- Line numbers start at 1 and increase sequentially until the end of the file. Even blank lines have a line number.
- Paragraph numbers also start at 1 and increase sequentially until the end of the file. All the lines in a given paragraph have the same paragraph number. However, blank lines do not belong to any paragraph number. Therefore, all blank lines should indicate the paragraph number as 0 (zero).
- Each paragraph number and the line number is right-justified in a field width of 4 (we'll assume that none of the essays have more than 9999 lines).

For example, suppose the essay is as follows:

```
My essay is kind of short. It
is only going to have a few inarticulate
lines and even
fewer paragraphs.



The second paragraph
has arrived, and you can see
it's not much.

The third paragraph now arrives
and departs as hastily.
```

The output file should look like this:

```
0 ,    1
1 ,    2 My essay is kind of short. It
1 ,    3 is only going to have a few inarticulate
1 ,    4 lines and even
1 ,    5 fewer paragraphs.
0 ,    6
0 ,    7
0 ,    8
2 ,    9 The second paragraph
2 ,   10 has arrived, and you can see
2 ,   11 it's not much.
0 ,   12
3 ,   13 The third paragraph now arrives
3 ,   14 and departs as hastily.
```

2. Write a function called `print_lines` with three parameters: `lpnumfilename`, `start`, and `finish`. The first parameter `lpnumfilename` is the name of a file that contains paragraph and line numbers inserted at the beginning of each line (as described above).

The second and third parameters are integers with `start` $\leq$ `finish`. The function should print all lines of the file starting at line number `start` up to (and including) line number `finish`. Note that all lines, including blank lines, should be printed. The lines may also span one or more paragraphs. If `start` is larger than the last line number in the file, nothing is printed. The function is responsible for opening and closing the file.

3. Write a function called `print_paragraph` that has two parameters: `lpnumfilename` and `paranum`. As above, the first parameter `lpnumfilename` is the name of a file that contains paragraph and line numbers inserted at the beginning of each line. The function should print all lines (and only those lines) of the file that belong to paragraph number `paranum`. If there is no paragraph numbered `paranum`, nothing is printed.

Please download `problem1.py` from the Sakai site. It contains the Python code that I will use to test your program. Your job is to insert implementations for the three functions listed above *into that file*. Three sample essay files (`essay1.txt`, `essay2.txt`, and `essay3.txt`) are included as well. Feel free to also use your own text files to test your program.

Problem 2 [25 points] Frequency distribution of characters in a file. Several problems in computer science require the frequency distribution of characters in a file. For example, a famous and widely used method called *Huffman coding* is used to compress data by using fewer bits to encode more frequently occurring characters. In order to do this, the frequency distribution of the characters in the file must be determined first. Compression methods used to transmit digital images and video over the internet use some form of Huffman coding.

In this problem, you are asked to find the frequency distribution of the characters in a given text file, i.e., the number of times each character occurs in the file. The character could be any printable non-whitespace character on the keyboard (for example, upper/lower case letters, digits, punctuation marks etc.). Note that we ignore whitespace characters (space, tab, and newline characters). **You will use a dictionary to carry out this task.** In particular, you are asked to implement the following functions:

1. **(7 points)** A function called `freq_distribution` with two parameters: `infile` and `distfile`. The first parameter, `infile`, is the name of the file for which we want to compute the frequency distribution. The second parameter, `distfile` is the name of the file into which the frequency distribution is to be written in *sorted order of the characters*. That is, the characters should be listed in alphabetically sorted order in the output file. Each line of the file should contain the character and its frequency. Use string formatting to make sure the output is *neatly formatted*.

   In order to carry out the above task, you must first create a dictionary to store the frequency distribution. You will do this by *calling the helper function* `freq_dictionary` described in #(3) below. Remember to close `distfile` when you are done writing into it.

2. **(10 points)** A function called `ordered_freq_distribution` with two parameters: `infile` and `ordered_distfile`. The first parameter, `infile`, is the name of the file for which we want to compute the frequency distribution. The second parameter, `ordered_distfile` is the name of the file into which the frequency distribution is to be written in *sorted order of the frequency*. That is, the characters should be printed into the file in *decreasing* order of frequency. If several characters have the same frequency, they should be

printed in alphabetical order. Use string formatting to make sure that the output in `ordered_distfile` is neatly formatted.

Once again, in order to carry out this task, you must first create a dictionary to store the frequency distribution by calling the helper function `freq_dictionary` described in #(3) below. Remember to close `ordered_distfile` when you are done writing into it.

3. **(8 points)** A function called `freq_dictionary` with a single parameter `infile`, the name of the text file for which we want to compute the frequency distribution. The function should first open `infile` for reading. It should then create a dictionary with key:value pairs, where the key is a character occurring in `infile` and its associated value is the frequency of that character in the file (i.e., the number of times it occurs in the file). Note that every non-whitespace character (characters other than newline, space, and tab) occurring in the file should be included in the dictionary. Upper and lower case letters should be kept distinct. Characters that do not occur in the file should not be included in the dictionary. The function must **return** the dictionary. Remember to close the file *prior to* the return statement.

For example, suppose `essay.txt` is the name of a file containing the following (uninteresting) essay:

```
how now brown cow
```

Then, after the function call `freq_distribution("essay.txt", "freq.txt")`, a file called `freq.txt` should contain the following:

```
b   1
c   1
h   1
n   2
o   4
r   1
w   4
```

After the function call `ordered_freq_distribution("essay.txt", "ordfreq.txt")`, a file called `ordfreq.txt` should contain the following:

```
o   4
w   4
n   2
b   1
c   1
h   1
r   1
```

Submission Guidelines

Implement the first problem in a file called `problem1.py` and the second one in a file called `problem2.py`. **Your name and RUID should appear as a comment at the very top of each file.**

Submit your homework files via Sakai as follows:

1. Use your web browser to go to the website `sakai.rutgers.edu`.
2. Log in by using your Rutgers login id and password, and click on the `OBJECT-ORIENTED PROG F18` tab.
3. Click on the 'Assignments 2' link on the left and go to 'Programming Assignment #2' to find the homework file (`hw2.pdf`) and the module stub `problem1.py` for Problem 1. In addition, you will find sample test files to be used for both problems.
4. Use this same link to upload your two homework files (`problem1.py` and `problem2.py`) when you are ready to submit.

You must submit your assignment at or before 11:59PM on October 8, 2018.