COMP 348: ASSIGNMENT 2

Important Info:

1. *Feel free to talk to other students about the assignment. That's not a problem. However, when you write the code, you must do this yourself. Source code cannot be shared under any circumstance. This is considered to be plagiarism.*

2. *Assignments must be submitted on time in order to received full value. Appropriate late penalties (< 1 hour = 10%, < 24 hours = 25%) will be applied, as appropriate.*

3. *The graders will be using a Python 3.x interpreter to test your code. Make sure that your code is written and tested in a version 3.x environment, not a 2.x environment.*

DESCRIPTION: In this assignment, you will have a chance to gain experience with the Python programming language. Python is an easy to use, dynamically type Object Oriented language. It is syntactically similar to C-style languages like C++, C# and Java, but is somewhat simpler to understand.

In short, you will be working with basic objects and various Python data structures in order to develop a simple command-line game that allows people to guess English words. The idea is as follows:

The user will be asked to guess 4-letter words. To make this realistic, we must utilize a large list of possible words. For this purpose, we have provided a text file containing over 4000 4-letter English words. Your Python application will read the file from disk and load all strings into a data structure of your choice.

Your program will then randomly select a string from the database and ask the user to guess that string. Initially, the string will be listed as "----". The user will have the choice to try to guess the word directly or to guess a letter at a time. The initial screen might look like this:

```
++
++ The great guessing game
++

Current Guess:  ----
Letters guessed:

g = guess, t = tell me, l for a letter, and q to quit

Enter Option:
```

Note the following features:

1. When the game starts, the screen/display will be cleared and the menu will appear at the top of the display
2. The game title is displayed first (The great guessing game)
3. Next, the current guess is shown. Initially, this is '----' since no letters have been requested.
4. "Letters guessed" will show the letters already guessed by the user. This is important since it is easy to lose track of the letters you have requested earlier in the game. Initially, of course, this list is empty.
5. The four 'options' follow:
    a. Guess: attempt to guess the word
    b. Tell me: You give up and ask the game to simply show you the correct word.
    c. Letter: select an individual letter that might be in the word
    d. Quit: End the game session and display a final report (more on this below)
6. Finally, the cursor waits for you to select an option.

We will look at the various game options below. However, before we do that, note that the game can be played in two *modes*. The first of these is *play* mode, and it corresponds to the screen depicted above. The second is *test* mode and it is the mode that will be used by both the graders and by you when you are developing the application. Both modes work in exactly the same way. However, *test* mode will also display the word that has been randomly chosen for you. This is useful, as it allows you (and the grader) to see if the code is working properly. From this point onwards, we will actually depict the game in *test* mode.

Note that the mode will be provided as a command line option. So if we have a python app/file called `words.py`, we would invoke the program from the command line either as `python3 words.py play` or `python3 words.py test`

Before we go any further, please keep in mind that proper error checking for input values must be provided in this application. Specifically, you must check the command line parameter to ensure that a valid mode is being used, and the menu option must also be validated, as indicated below (note that we are in *test* mode, and the current word – `dime` – is displayed for us):

```
++
++ The great guessing game
++

Current Word:  dime
Current Guess:  ----
Letters guessed:

g = guess, t = tell me, l for a letter, and q to quit

Enter Option: a

Invalid option. Please re-enter:
```

Okay, so let's assume that the *letter* option has been chosen. You will now enter a single letter when prompted. There are just two possibilities: it's in the word or it's not. If it's not, you will receive a message to that effect, as shown below:

```
++
++ The great guessing game
++

Current Word:  vino
Current Guess:  ----
Letters guessed:

g = guess, t = tell me, l for a letter, and q to quit

Enter Option: l

Enter a letter: j

@@
@@ FEEDBACK: Not a single match, genius
@@


Press any key to continue...
```

We have guessed the letter 'j', which is not in the word 'vino' (note that some of the words in our list probably aren't even real words). In any case, we get a (rather rude) message that the letter is not there (feel free to be a little more pleasant with your feedback).

At this point, you will be asked to "Press any key to continue". As the prompt suggests, you can press anything to proceed, followed by the `Enter` key. In fact, you can simply press `Enter/return` by itself and the game will move on.

Note that every time we press a key to continue, the screen will be cleared, and the menu will be re-displayed at the top. In other words, the game content does not keep scrolling down the screen.

So what do we see now? The menu is shown again, but this time we see the letter that we have guessed, as in:

```
++
++ The great guessing game
++

Current Word:  vino
Current Guess:  ----
Letters guessed: j

g = guess, t = tell me, l for a letter, and q to quit

Enter Option:
```

So the `Letters guessed` line now shows us that we have used a 'j'...and probably should not pick this letter again.

Of course, with a little luck, we will pick a letter that is actually in the word. In that case, we get a slightly more encouraging message:

```
++
++ The great guessing game
++

Current Word:  vino
Current Guess:  ----
Letters guessed: e a q j

g = guess, t = tell me, l for a letter, and q to quit

Enter Option: l

Enter a letter: i

@@
@@ FEEDBACK: Woo hoo, you found 1 letters
@@


Press any key to continue...
```

In this case, we requested 'i', which happens to be in our word. When we continue, the relevant letter will now be displayed in the appropriate position in the `Current Guess`. This is illustrated below (from a different game, after selecting an 'o' for the word 'torn').

```
++
++ The great guessing game
++

Current Word:  torn
Current Guess:  -o--
Letters guessed: p o u e

g = guess, t = tell me, l for a letter, and q to quit

Enter Option:
```

Here, we can see that the 'o' is in the second position. We can also see that we have already tried three other invalid letters ('p', 'u', and 'e').

Note that if the letter is found multiple times in the current word (e.g., 's' in 'saws'), the game will confirm that two letters have been found and it will update the `Current Guess` accordingly ('s--s').

Okay, so the player has made some progress and wants to make a guess for the word. Using the 'g' option, the game will prompt the user for an entry and then check for a match. A good guess would of course confirm the selection, as follows:

```
++
++ The great guessing game
++

Current Word:  vino
Current Guess:  vi--
Letters guessed: q i a j e v

g = guess, t = tell me, l for a letter, and q to quit

Enter Option: g

Make your guess: vino

@@
@@ FEEDBACK: You're right, Einstein!
@@


Press any key to continue...
```

A bad guess would not be so kind:

```
++
++ The great guessing game
++

Current Word:  vino
Current Guess:  vi--
Letters guessed: q i a j e v

g = guess, t = tell me, l for a letter, and q to quit

Enter Option: g

Make your guess: vine

@@
@@ FEEDBACK: Try again, Loser!
@@


Press any key to continue...
```

Note that the game should be case insensitive. So it doesn't matter if the user enters 'vino' or 'Vino'; the match should still be found. In fact, the same is true for letter: an 'M' is the same as an 'm'.

While the game looks quite easy when run in *test* mode, it can be surprisingly tedious in *play* mode. As a result, the user may want to simply give up occasionally. If so, we select the 'tell me' option, and the game will mock you as follows:

```
++
++ The great guessing game
++

Current Word:  mots
Current Guess:  ----
Letters guessed: r y q

g = guess, t = tell me, l for a letter, and q to quit

Enter Option: t

@@
@@ FEEDBACK: You really should have guessed this...'mots'
@@


Press any key to continue...
```

Okay, so that's the basic idea. You will try letters for a given word until you either guess the word correctly or you give up. In either case, the game will then automatically select a new word for you and start the process again. You can play as many games as you like until you finally decide to quit.

When you select the 'quit' option the application will of course terminate. However, before it does so, it will give you a report on your game experience. Specifically, it will provide a summary of each game that you have played, highlighting the key successes/failures of each game, along with an associated score. After all, what is a game without a score.

How does scoring work? To begin, note that each letter in the English language is used with a certain frequency ("e" is the most common letter, "z" the least). A table of letter frequencies is given later in this document.

The letters that are still blank at the time of a correct guess will be summed together to give a total. Basically, the point value for a given letter is equal to its frequency. Note that it is easiest to guess a word if you first uncover the most common letters. However, this leaves the least points in the remaining word. Of course, the number of letters that you request also affects your score. So you should divide the sum of the frequencies by the number of times you requested a letter. Finally, an incorrect guess costs you 10% of your final score for the current word (2 incorrect guess are 20% of the total and so on).

What happens if you give up? Then you should lose points. Here, the total points lost should simply be the sum the letters that have not yet been guessed/uncovered.

In case it's not obvious, the scores for games that you win (with a correct guess) will always be positive. The scores for games that you lose (you give up) will always be negative. Moreover, the highest scores will be for games in which you guess the right word immediately, while that the lowest negative scores will be for games in which you give up immediately. In practice, most winning games will produce a fairly small score since you will select many letters before you finally make a correct guess.

Now back to the final report. Once the user has finished playing, they will select `quit`. At that point, you will print a short report (after clearing the screen) that summarizes the info. Assuming the user was a language wizard, a great report might look like this:

```
++
++ Game Report
++

Game       Word       Status     Bad Guesses     Missed Letters  Score
----       ----       ------     -----------     --------------  -----
1          ohed       Success    0               1               23.04
2          dyes       Success    0               1               10.58
3          zarf       Success    0               1               16.39

Final Score: 50.01
```

Here, all three games were won quickly, with only one missed letter per game and no bad word guesses. All three games received high scores as a result. You will also note that a Final Score is calculated for the session – this is simply the total of all games played.

On the other hand, most players will not be wizards, so the report is not going to look this nice. A more realistic report might be as follows:

```
++
++ Game Report
++

Game       Word       Status     Bad Guesses     Missed Letters  Score
----       ----       ------     -----------     --------------  -----
1          tace       Gave up    0               0               -32.71
2          glue       Success    1               1               4.30
3          dark       Success    2               3               0.21
4          cube       Gave up    2               3               -16.97
5          fumy       Success    0               1               4.38
6          yeld       Success    1               3               2.48

Final Score: -38.31
```

In this case, there were four good games and two bad ones. The bad ones were really bad, resulting in large negative scores. The good games weren't actually all that good because a lot of letters were requested and multiple bad guesses were made. As a result, the overall final score is actually negative.

In terms of the report itself, note the following. Each line displays the word, final status, the number of bad word guesses, the number of missed letters and the final score. Moreover, each column is nicely formatted so that all columns "line up" in a consistent way. Finally, floating point values are always displayed with 2 digits of precision after the decimal point.

As a final note, if the user quits before completing any games, you should still display the report, but you will have with no rows and the final score will be zero.

That's it for the game. Who needs Wordle…

STRUCTURE: In this assignment, you will write your code using Python's Object Oriented syntax. You really only need three classes. The first will be called **Guess** and it will represent the core application logic itself (menu display, user input, scoring logic, etc). This class will be stored in Guess.py.

The second will be called **Game** and it will just maintain information about a specific game (needed for the final report). It will be stored in Game.py.

The third class will be called **StringDatabase** and it will be responsible for the disk I/O (i.e., the word file) and random selection of the word for a new game. It will be stored in StringDatabase.py.

Finally, you will also have a very simply non object oriented file called **words.py**. It simply parses the command line parameter and starts the guessing game.

FREQUENCIES: The letter frequencies are listed below. You can represent this data however you like in your application.

Letter	Frequency
a	8.17%
b	1.49%
c	2.78%
d	4.25%
e	12.70%
f	2.23%
g	2.02%
h	6.09%
i	6.97%
j	0.15%
k	0.77%
l	4.03%
m	2.41%
n	6.75%
o	7.51%
p	1.93%
q	0.10%
r	5.99%
s	6.33%
t	9.06%
u	2.76%
v	0.98%
w	2.36%
x	0.15%
y	1.97%
z	0.07%

So that's the basic idea. If you haven't used Python before, you will find that it is a very accessible language with a lot of documentation and supporting materials online. In fact, we are not providing any specific hints or suggestions for any component of the code since a simple web search (including the main online python docs) will produce multiple examples and/or suggestions within a matter of seconds. If you like to code, this assignment should actually be fun.

DELIVERABLES: Your submission will have exactly 4 source files: `words.py`, `Guess.py`, `Game.py`, and `StringDatabase.py`. The program itself will be run from `words.py`. Do not include the `four_letters.txt` file since the marker will already have this.

Note that you should not organize your code into a package(s). Instead, all four files - plus the four_letters.txt file - will simply be placed inside the current folder and will be run from there. Please ensure that this works since this is exactly what the graders will be doing with your submission.

Once you are ready to submit, combine the four source files into a zip file. The name of the zip file will consist of "a2" + last name + first name + student ID + ".zip", using the underscore character "_" as the separator. For example, if your name is John Smith and your ID is "123456", then your zip file would be combined into a file called a1_Smith_John_123456.zip". The final zip file will be submitted through the course web site on Moodle. You simply upload the file using the link on the assignment web page.

As always, it is your responsibility to verify the contents of your submission. You will be graded solely on what you submit at the deadline.

Good Luck