Playing Word Games

You will use Python and Visual Studio Code to finish two computer programs for this assignment. Submit the Python program script files with the solutions (file extension is py) in Blackboard by Sunday at 11:59 PM. Programming games might be more interesting than some of the other topics so far. In this assignment, you will create two word-based games: a one-player game similar to Scrabble and a game called Ghost.

Script Requirements

When writing and saving your programming files, ensure that you heed the instructions. Specific things must be placed in the programming code for all programming coursework. The beginning of each script file must include the assignment name, the purpose of the programming code, the date it was written or revised, and who wrote the code. These comments can be written as line comments or a commenting block. Figure 1 depicts an example of what this might look like in your file. Instead of creating script files for this problem set, you will use skeleton files explained further in these instructions. When you have completed and tested your script files, submit the completed script files for this assignment in Blackboard.

All analysis begins with a problem to solve, a question to answer, or something to prove (you might call this a hypothesis). The following information covers the programming requirements and includes hints for completing this work. There are four problems that you need to solve for this assignment.

Scrabble-Like Game Description

In this game, letters are dealt to players. Each valid word is worth points based on the word's length and the letters used. However, unlike Scrabble, there is no game board. However,

Figure 1

Content Needed in Every Script File

```
"""
Assignment 6 Problem 1: Creating a one-player word game similar to
Scrabble

September 22, 2022
John Smith
"""
```

Note. This contains the name of the assignment (Assignment 6 Problem 1), purpose of this assignment (creating a one-player word game similar to Scrabble), date (September 22, 2022), and the name of the developer (John Smith).

unlike Scrabble, there is no game board. Several functions are provided, and some you are required to create. Tests will need to be conducted on functions independent of the other functions and in tandem. As a final note, ensure that your script is riddled with comments explaining what each variable represents, what each function does and returns, and any other personal nuances in your programming code. Proper commenting is probably the easiest points earned in this course. When you submit this programming code, submit the final versions and the previous revisions.

Getting Started

There are a few different files that accompany this assignment. To complete the first game, you'll work with a skeleton file, a testing file, and a word list (covering problems one through five). The file **`problem_6.py`** is a skeleton file. Some unit testing has been created for you in the file **`testing_problem_6.py`**. The second game is accomplished by completing the programming in the skeleton file named **`problem_6_part_2.py`**. You'll also use the word list for this program. While programming, all files must be saved to the same folder. When you submit your work, don't submit the word list.

Run the Code

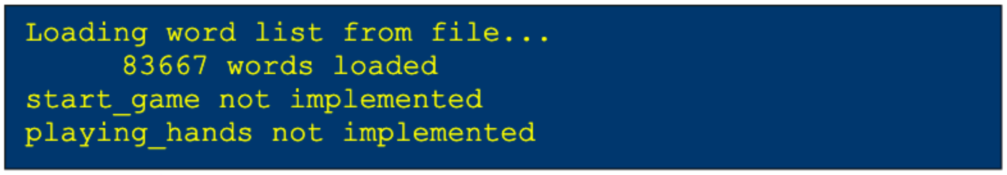
Open the file **`problem_6.py`**. Do not modify this file yet. Look through the code. This code loads the words for **`word.txt`** and calls the **`start_game`** function. More tasks are necessary to make this a game, but this code should run right now. If working correctly, you will see the output shown in Figure 2. If it does not work, ensure all the files are in the same folder. If it still doesn't work, ensure that you completed the steps in the first problem set under the heading *Establish the Working Directory*.

Unit Testing

This assignment requires you to design several modular functions, then glue them together to create the game. Instead of waiting until the entire game is ready, inspect what you expect from each line of code. Test each function as you write them before coding other functions. Unit testing is important whenever you code. There are three tests written for you to get you started. As you progress through this assignment, run **`testing_problem_6.py`**. If your code works as expected, this file will return a message indicating your success. Should your code be insufficient or cause an error, a message indicating your lack of success will be provided. The three pre-written unit test functions are **`test_scoring_words`**, **`test_hand_updates`**, and **`test_word_validity`**. These script files will test the functions **`calc_word_score`**, **`hand_update`**, and **`word_is_valid`**, respectively.

Figure 2

Console Output from Running problem_6.py Skeleton Before Any Changes



```
Loading word list from file...
      83667 words loaded
start_game not implemented
playing_hands not implemented
```

Note. This is an example of what the expected output will look like when you run the program and a user answers the questions. The text is yellow reflects the output from the program. The text in white reflects the user's input.

Problem One: Word Scores

This game requires scores to be calculated from the words played. You need to design a solution to this problem using the function **`calc_word_score`**. This function should accept a string of lowercase letters as input (a word) and the maximum hand size. It will return a score. How a word arrives at this function or how this function fits in the big picture is outside the scope of this objective.

There are several considerations. Some are safe to assume, while other things are unsafe to assume. Lastly, there are specific considerations when calculating the score. It is safe to assume that the word passed to this function is always a lowercase string (although it could be an empty string). Letter values are static; they are defined in the dictionary **`points_by_letter`**. It's not safe to assume that there are always seven letters in a hand. (That's why it's passed as a variable.) As a bonus, if the word length is equivalent to the maximum hand size, the player gets an extra 50 points.

When you have finished solving this problem, use the test file to test your function. Additionally, make sure that you develop tests for this function that satisfy boundaries that are not assessed with the test file you were given. When you have completed testing this function, ensure any testing written in the **`problem_6.py`** file is commented out before submitting your work. If you document your test code elsewhere, ensure that it is included in your submission (even if it's a separate file). Make sure that your comments clarify the purpose of this type of content.

Problem Two: Updating the Hand

Initially, a player is given a quasi-random set of letters. (There is a vowel-to-consonant ratio requirement.) The player's hand may have more than one of a specific letter. The hand is stored in a dictionary to ensure that control can be maintained. For example, if the randomly chosen letters were **`e`**, **`e`**, **`m`**, **`t`**, **`s`**, **`w`**, and **`s`**, the dictionary would group them like terms. The dictionary that represents this hand is **`{e: 2, m: 1, t: 1, s: 2, w:1}`**. The data is represented this way for several reasons. Some of the reasons will be understood as you work through the code to complete different elements of this game. You'll get an error if you query for a key that isn't in a dictionary. To get around this with modular programming, you will use the dictionary method **`.get()`**. However, letters in out. When you need to determine if a letter is in this dictionary, you can use this type of call: **`hand.get('a', 0)`**. If the key is in the dictionary, it returns the value. If it doesn't, it returns **`0`**.

Provided Functions

This skeleton has several predefined functions, **`into_dictionary`**, **`show_hand`**, and **`dealing_hands`**. Near the top of the skeleton file, the function **`into_dictionary`** is provided. If this function is called with a string of letters, it returns a dictionary. The keys are the letters; the values represent that letter's frequency. This is the same type of dictionary used in the game hands. The name sufficiently summarizes the purpose of **`show_hand`**. This function prints the user's hand to the terminal. The hand that a player is dealt is a set of letters chosen quasi-randomly. There is a specific ratio of variables to consonants when letters are assigned.

Removing Letters from a Hand

As the player spells valid words, the letters must be removed from the hand. When the function **`updating_hand`** is called, it returns a hand that no longer includes the letters that were used. Consider the calls and the ways that data is handled in the functions that were provided. Then program the solution to this problem. When you have finished the function, use the testing file's **`test_hand_updates`** to evaluate the function. Ensure that you develop your own tests for any boundaries that aren't checked.

Problem Three: Valid Words

You now have a large portion of this game complete. It's now necessary to develop a method to test whether the word is valid. There are two criteria. The word must be in the word list. Additionally, the letters need to be in the player's hand. Implement the function **`word_is_valid`** to return true if both criteria are met and false if either is not met. When you have finished solving this problem, use the testing file's **`test_word_validity`**. Like the other problems, ensure that develop additional unit tests for any boundaries not assessed.

Problem Four: Playing Hands

The next step is to implement the ability for a user to play with a single hand. Update the function **`playing_hands`** to solve this problem. Don't assume there are seven letters in the player's hand. To see what you might find in the output from this function, see Figures 3, 4, and 5. This function connects several functions together. Read through the information provided in the skeleton to learn more about the requirements of this function. Make sure that you thoroughly test this function and functions interaction.

Problem Five: Playing a Game

A game consists of one-to-many hands. There is one final function needed to complete this game. Delete the code that is present in that function and is not commented. Uncomment the code in the function **`start_game`**. No actual programming is necessary to solve this problem, just some deleting and comment toggling. If you would like, you can try to set the hand size to different values when you play the game.

Ghost: Another Word Game

Ghost is a two-player game. The goal is to implement an interactive Python program that allows two humans to play the game against each other. If you are unfamiliar with this game, take a moment to query Google or Wikipedia. The remaining topics to accomplish this objective include the game rules, where to start, and the program expectations.

Game Rules

Players take turns contributing a letter to a word fragment when playing the game. The player loses if they create a real word unless the word is less than four letters. Another method of losing is to create a fragment that ensures no words can be made (i.e., 'qz,' there are no valid words where these two letters are contiguous).

Getting Started

Use the skeleton file program_6_part_2.py. As stated earlier, this game also requires the word list file. Like the Scrabble project, run the skeleton before making other changes. Like the Scrabble game, the output of this file before any modifications will look like the output shown in Figure 2. If there are errors when you run the unmodified skeleton file, use the troubleshooting tips provided in the Getting Started section of these instructions for the Scrabble game.

Requirements

There are a few additional requirements. The game must be interactive. Each turn should clearly indicate the player responsible for the following letter and provide the word fragment at that point. Additionally, in each turn, the program will ask the player to enter their next letter. The program validates the entry. If the entry is a character and doesn't cause the player to lose, this letter is added to the substring. The game ends if a word is formed that is longer than three letters or if it is no longer feasible to create a word from the word fragment. All code should be within functions. (No lose code.) After you have thoroughly tested your code, comment out any code used in testing. Like any use of strings in programming, pay attention to the casing. If it's possible to ensure all strings are in the same case, it's probably a good idea to handle strings that way. For output examples, see Figure 6. For a programming tip, see Figure 7.

Figure 3

Console Output from Function **`playing_hands`***: All Letters Used in One Turn*

```
Current hand: a a n r d y m

Given the letters in your hand, make a word to earn points or enter '.' to
end your game.
yardman
You earned an additional 50 points for using all of the letters in one
word.
The word yardman got you 63 points. Your total score is 63.
```

Note. This is an example of the output of playing a hand with a word that uses every letter.

Figure 4

Console Output from Function **`playing_hands`***: Multi-Word Game*

```
Current hand: a i l h u j s

Given the letters in your hand, make a word to earn points or enter '.' to
end your game.
jails
The word jails got you 12 points. Your total score is 12.

Current hand: u h

Given the letters in your hand, make a word to earn points or enter ‘.’ To
end your game.
uh
The word uh got you 5 points. Your total score is 17.
```

Note. This is an example of the output from playing a hand with multiple words.

Figure 5

Console Output from Function **`playing_hands`***: Invalid Word*

```
Current hand: a i l h u j s

Given the letters in your hand, make a word to earn points or enter '.' to
end your game.
justice
Invalid word; please try again.
```

Note. This is an example of the output from playing a hand with an invalid word.

Figure 6

Console Output from Playing Ghost

```
Welcome to the Ghost Word Game!

Player 1 will go first. Current word fragment: ''
Player 1 typed the letter: p

Current word fragment: 'p'
Player 2's turn. Player 2 typed the letter: y

Current word fragment: 'py'
Player 1's turn.
Player 1 typed the letter: g

Current word fragment: 'pyg'
Player 2's turn.
Player 2 typed the letter: m

Current word fragment: 'pygm'
Player 1's turn.
Player 1 typed the letter: y

Current word fragment: 'pygmy'
Pygmy is a word! Player 2 wins.
```

Note. This is an example of the output of Ghost from the beginning until game end.

Figure 7

Programming Validation in the Ghost Word Game

```
>>'a' in string.ascii_letters
True
>>'3' in string.ascii_letters
False
```

Note. When creating the Scrabble-type game, letters were validated against the given hand, but in the Ghost game, there is no hand. You still need to ensure that the player used *letters,* not special characters or numbers.