STAT535: Statistical Computing **Fall 2020**

Problem Set 2: Testing

P. Flaherty

Objective In this problem set, you'll implement a version of a word game. Don't be intimidated by the length of this problem set. It's a lot of reading, but it is very doable. Let's begin by describing the word game: This game is a lot like Scrabble. Letters are dealt to players, who then construct one or more words using their letters. Each valid word earns the user points, based on the length of the word and the letters in that word.

Collaboration You may work with other students. However, each student must writeup and hand in their assignment individually. Be sure to indicate whom you worked with in the comments of your submission.

Readings

Logistics

Rules

Dealing

- A player is dealt a hand of `HAND_SIZE` letters of the alphabet, chosen at random. This may include multiple instances of a particular letter.
- The player arranges the hand into as many words as they want out of the letters, but using each letter at most once.
- Some letters may remain unused, though the size of the hand when a word is played does affect its score.

Scoring

- The score for the hand is the sum of the score for each word formed.
- The score for a word is the *product* of two components:
  - First component: the sum of the points for letters in the word.
  - Second component: either `7 * word_length - 3 * (n - word_length))` or 1 whichever value is greater, where:
    - `word_length` is the number of letters used in each word
    - `n` is the number of letters available in the current hand

- Letters are scored as in Scrabble; A is worth 1, B is worth 3, C is worth 3, D is worth 2, E is worth 1, and so on. We have defined the dictionary `SCRABBLE_LETTER_VALUES` that maps each lowercase letter to its Scrabble letter value.
- Examples:
  - If `n=6` and the hand includes 1 'w', 2 'e's, and 1 'd' (as well as two other letters), playing the word 'weed' would be worth 176 points: (4+1+1+2) * (7*4 - 3*(6-4)) = 176. The first term is the sum of the values of each letter used; the second term is the special computation that rewards a player for playing a longer word, and penalizes them for any left over letters.
  - If `n =7`, playing the word 'it' would be worth 2 points: (1+1) * (1) = 2. The second component is 1 because 7*2 - 3*(7 - 2) = -1, which is less than 1.

Getting Started

1. Download and save all files on Moodle. This includes the python file `ps2-framework.py`, which should contain all of your code, as it provides a set of initial procedures and templates for new procedures. Those files also include a file for testing your code `test_ps2.py` and a file of legitimate words `words.txt`. **Do not change or delete anything in the file unless specified.**
2. Run `ps2-framework.py` without making any modifications to it, in order to ensure that everything is set up correctly. The code we have given you loads a list of valid words from a file and then calls the `play_game` function. You will implement the functions it needs in order to work. If everything is okay, after a small delay, you should see the following printed out:

   ```
   Loading word list from file...
           83667 words loaded.
   ```

   If you see an `IOError` instead (e.g. No such file or directory), make sure you have `words.txt` in the same directory as `ps2-framework.py`.
3. Now if everything is correct, we should change the name of `ps2-framework.py` to `ps2.py`. We use this name to submit the file to GradeScope. The file `ps2.py` has a number of already-implemented functions you can use while writing up your solution. You can ignore the code between the following comments, though you should read and understand everything else.

   ```
   # -----------------------------------
   # Helper code
   # (you don't need to understand this helper code)
   # (end of helper code)
   .
   .
   .
   # -----------------------------------
   ```

4. This problem set is structured so that you will write a number of modular functions and then glue them together to form the complete game. Instead of waiting until the entire game is ready, you should test each function you write, individually, before moving on. This approach is known as *unit testing*, and it will help you debug your code.

5. We have included some hints about how you might want to implement some of the required functions in the included files. You don't need to remove them in your final submission.

We have provided several test functions to get you started. As you make progress on the problem set, run `test_ps2.py` to check your work so far. If your code passes the unit tests you will see a SUCCESS message; otherwise you will see a FAILURE message. **These tests aren't exhaustive. You may want to test your code in other ways too** (for example, with different test values).

If you run `test_ps2.py` using the initially provided `ps2.py` skeleton, you should see that all the tests fail.

These are the provided test functions:

Problem A:

Word scores The first step is to implement a function that calculates the score for a single word. Fill in the code for `get_word_score` in `ps2.py` according to the function specifications.

As a reminder, here are the rules for scoring a word:

- The score for a word is the *product* of two components:
  - First component: the sum of the points for letters in the word.
  - Second component: either `7 * word_length - 3 * (n - word_length))` or 1 whichever value is greater, where:
    - `word_length` is the number of letters used in each word
    - `n` is the number of letters available in the current hand

You should use the `SCRABBLE_LETTER_VALUES` dictionary defined at the top of `ps2.py` . Do **not** assume that there are always 7 letters in a hand! The parameter `n` is the total number of letters in the hand when the word was entered.

Testing If this function is implemented correctly, and you run `test_ps2.py`, the `test_get_word_score()` tests will pass. You should also test your implementation of `get_word_score` yourself, using some reasonable English words. Note that the wildcard tests will crash due to a `KeyError`. This is fine for now - you will fix this in Problem 4.

Hints You may find the `str.lower` function helpful:

```
s =‘‘My string’’
print(s.lower())
>>>‘‘my string’’
```

If you don't know what this does you could try typing `help(str.lower)` in your Spyder shell to see the documentation for the functions.

Problem B:

Dealing a Hand Please read problem 2 entirely before you begin coding your solution. Most of the functions described below have been implemented for you already.

Representing hands A hand is the set of letters held by a player during the game. The player is initially dealt a set of random letters. For example, the player could start out with the following hand: **a, q, l, m, u, i, l**. In our program, a hand will be represented as a dictionary: the keys are (lowercase) letters and the values are the number of times the particular letter is repeated in that hand. For example, the above hand would be represented as:

```
hand = {'a':1, 'q':1, 'l':2, 'm':1, 'u':1, 'i':1}
```

Notice how the repeated letter '`l`' is represented. With a dictionary representation, the usual way to access a value is `hand['a']`, where '`a`' is the key we want to find. However, this only works if the key is in the dictionary; otherwise, we get a `KeyError`. To avoid this, we can instead use the function call `hand.get('a',0)`. This is the "safe" way to access a value if we are not sure the key is in the dictionary. `d.get(key,default)` returns the value for `key` if `key` is in the dictionary `d`, else it returns `default`. If `default` is not given, it returns `None`, so that this method never raises a `KeyError`.

Converting words into dictionary representation One useful function we've defined for you is `get_frequency_dict`, defined near the top of `ps2.py`. When given a string of letters as an input, it returns a dictionary where the keys are letters and the values are the number of times that letter is represented in the input string. For example:

```
>>> get_frequency_dict(''hello'')
{'h':1, 'e':1, 'l':2, 'o':1}
```

As you can see, this is the same kind of dictionary we use to represent hands.

Displaying a hand Given a hand represented as a dictionary, we want to display it in a user-friendly way. We have provided the implementation for this in the `display_hand` function. Take a few minutes right now to read through this function carefully and understand what it does and how it works.

Generating a random hand The hand a player is dealt is a set of letters chosen at random. We provide you with a function that generates a random hand, `deal_hand`. The function takes as input a positive integer `n`, and returns a new dictionary representing a hand of `n` lowercase letters. Again, take a few minutes to read through this function carefully and understand what it does and how it works.

Removing letters from a hand (you implement this) The player starts with a full hand of `n` letters. As the player spells out words, letters from the set are used up. For example, the player could start with the following hand: **a, q, l, m, u, i, l**. The player could choose to play the word **quail**. This would leave the following letters in the player's hand: **l, m**.

You will now write a function that takes a hand and a word as inputs, uses letters from that hand to spell the word, and returns a **new** hand containing only the remaining letters. Your function should **not** modify the input hand. For example:

```
>> hand = {'a':1, 'q':1, 'l':2, 'm':1, 'u':1, 'i':1}
>> display_hand(hand)
```

```
a q l l m u i
>> new_hand = update_hand(hand, 'quail')
>> new_hand
{'l': 1, 'm': 1}
>> display_hand(new_hand)
l m
>> display_hand(hand)
a q l l m u i
```

(**NOTE:** Alternatively, in the above example, after the call to `update_hand` the value of `new_hand` could be the dictionary `{'a':0, 'q':0, 'l':1, 'm':1, 'u':0, 'i':0}`. The exact value depends on your implementation; but the output of `display_hand()` should be the same in either case.)

IMPORTANT: If the player guesses a word that is invalid, either because it is not a real word or because they used letters that they don't actually have in their hand, they still lose the letters from their hand that they did guess as a penalty. Make sure that your implementation accounts for this! Do not assume that the word you are given only uses letters that actually exist in the hand. For example:

```
>> hand = {'j':2, 'o':1, 'l':1, 'w':1, 'n':2}
>> display_hand(hand)
j j o l w n n
>> hand = update_hand(hand, 'jolly')
>> hand
{'j':1, w':1, 'n':2}
>> display_hand(hand)
j w n n
```

Note that one 'j', one 'o', and one 'l' (despite that facts that the player tried to use two, because only one existed in the hand) were used up. The 'y' guess has no effect on the hand, because 'y' was not in the hand to begin with. Also, the same note from above about alternate representations of the hand applies here.

Implement the `update_hand` function according to the specifications in the skeleton code.

Testing Make sure the `test_update_hand` tests pass. You may also want to test your implementation of `update_hand` with some reasonable inputs.

Hints You may wish to review the documentation for the ".copy" method of Python dictionaries.

Problem C:

Valid Words At this point, we have not written any code to verify that a word given by a player obeys the rules of the game. A *valid* word is in the word list (we ignore the case of words here) **and** it is composed entirely of letters from the current hand.

Implement the `is_valid_word` function according to its specifications.

Testing Make sure the `test_is_valid_word` tests pass. You may also want to test your implementation with some reasonable inputs. Try testing it by calling it multiple times on the same hand. What shold the correct behavior be?

Problem D:

Playing a hand We are now ready to begin writing the code that interacts with the player.

Implement the `play_hand` function. This function allows the user to play out a single hand. You'll first need to implement the helper function `calculate_handlen`, which can be done in under five lines of code.

To end the hand early, the player must type "!!" (two exclamation points). Note that after the line *`# BEGIN PSEUDOCODE`* there is a bunch of, well, pseudocode. This is to help guide you in writing your function. Check out the https://stellar.mit.edu/S/course/6/fa14/6.0001/courseMaterial/topics/topic1/resource/WhyPseudocode/WhyPseudocode.pdfWhy Pseudocode? [**whypseudocode**] resource to learn more about the What and Why of Pseudocode before you start this problem.

Your output should match the examples below. You should not print extraneous `None` messages.

```
Current Hand: a j e f o r x
Enter word, or "!!" to indicate that you are finished: jar
"jar"  earned 90 points. Total: 90 points

Current Hand: o f x e
Enter word, or "!!" to indicate that you are finished: fox
"fox"  earned 234 points. Total: 324 points

Current Hand: e
Enter word, or "!!" to indicate that you are finished: !!
Total score: 324 points

Current Hand: a c f i i t x
Enter word, or "!!" to indicate that you are finished: fix
"fix"  earned 117 points. Total: 117 points

Current Hand: a c t i
Enter word, or "!!" to indicate that you are finished: ac
This is not a valid word. Please choose another word.

Current Hand: t i
Enter word, or "!!" to indicate that you are finished: it
"it"  earned 28 points. Total: 145 points

Ran out of letters. Total score: 145 points.
```

Testing Try out your implementation as if you were playing the game: run your program and call the `play_hand` function from your shell with a hand and the `word_list`.