Problem Set 3: The 6.0001 Wordgame

Handed out: Monday, September 28, 2015

Due: Wednesday, October 7, 2015 at 11:59 PM

Introduction

In this problem set, you'll implement a version of the 6.0001 wordgame!

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 6.0001 wordgame: This game is a lot like Scrabble or Words With Friends, if you've played those. Letters are dealt to players, who then construct one or more words out of their letters. Each **valid** word receives a score, based on the length of the word and the letters in that word.

The rules of the game are as follows:

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 these will affect the score of the hand.

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 [5 * *wordlen* - 2 * (*n*-*wordlen*)] or 1, whichever value is greater, where:
        - *wordlen* is the number of letters used in the 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.
- For example, 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 128 points: (4+1+1+2) * (5*4 - 2*2) = 128. 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.

- As another example, if *n*=7, playing the word 'it' would be worth 2 points: (1+1) * (1) = 2. The second component is 1 because 5*2 - 2*(7 - 2) = 0, which is less than 1.

Workload

Please let us know how long you spend on each problem. We want to be careful not to overload you by giving out problems that take longer than we anticipated.

Getting Started

1. Download and save Problem Set 3. This includes the python file `ps3.py`, which will be used to create your code, as it provides a set of initial procedures and templates for new procedures. It also includes a file for testing your code, and a file of legitimate words. **Do not change or delete anything in the file unless specified**.
2. Run `ps3.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.
          play_game not yet implemented.
   ```

   If you see an `IOError` instead (e.g., *No such file or directory*), you should change the value of the `WORDLIST_FILENAME` constant (defined near the top of the file) to the **complete** pathname for the file `words.txt` (This will vary based on where you saved the file).

3. The file `ps3.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 word playing 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_ps3.py` as you go.

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.**

If you run `test_ps3.py` using the initially provided `ps3.py` skeleton, you should see that all the tests fail.

These are the provided test functions:

`test_get_word_score()`

Test the `get_word_score()` implementation.

`test_update_hand()`

Test the `update_hand()` implementation.

`test_is_valid_word()`

Test the `is_valid_word()` implementation.

`test_wildcard()`

Test the modifications made to support wildcards.

Problem 1: Word scores

The first step is to implement some code that allows us to calculate the score for a single word. Fill in the code for `get_word_score` in `ps3.py` according to the function specifications.

You will want to use the `SCRABBLE_LETTER_VALUES` dictionary defined at the top of `ps3.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 played. Finally, you may want to review the documentation for the string module's `lower()` function.

Testing: If this function is implemented properly, and you run `test_ps3a.py`, you should see that the `test_get_word_score()` tests pass. 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.

Problem 2: Dealing with hands

****Please read problem 2 entirely before you begin coding your solution****

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. Notice that 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 use the 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 `ps3.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 the implementation of a function that generates this 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 hand, a set of letters. As the player spells out words, letters from this set are used up. For example, the player could start out with the following hand: **a, q, l, m, u, i, l** The player could choose to spell 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 the remaining letters in the 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
>> hand = update_hand(hand, 'quail')
>> hand
{'l': 1, 'm': 1}
>> display_hand(hand)
l m
```

(**NOTE:** Alternatively, in the above example, after the call to `update_hand` the value of 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.

HINT: You may wish to review the "`.copy`" method of Python dictionaries.

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

Problem 3. 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 **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. In particular, you may want to test your implementation by calling it multiple times on the same hand - what should the correct behavior be?

Problem 4. Wildcards

We want hands to contain wildcard letters, which will be denoted by an asterisk (*). Each hand dealt should initially contain exactly one wildcard as one of its letters. The player does not receive any points for using the wildcard (unlike all the other letters), though it **does** count as a used or unused letter when scoring.

During the game, a player wishing to use a wildcard should enter "*" (without quotes) instead of the intended letter. The word-validation code should not make any assumptions about what the intended letter should be, but should verify that at least one valid word can be made with the wildcard in the desired position.

Case #1

```
Current Hand:  c o w s * z
Enter word, or "&&&" to indicate that you are finished: cows
"cows" earned 144 points. Total: 144 points

Current Hand:  * z
Enter word, or "&&&" to indicate that you are finished: &&&
Total score: 144 points
```

Case #2

```
Current Hand:  c o w s * z
Enter word, or "&&&" to indicate that you are finished: c*ws
"c*ws" earned 128 points. Total: 128 points

Current Hand:  o z
Enter word, or "&&&" to indicate that you are finished: &&&
```

```
Total score: 128 points
```

Case #3

```
Current Hand:  c o w s * z
Enter word, or "&&&" to indicate that you are finished: c*wz
That is not a valid word. Please choose another word.

Current Hand: o s
Enter word, or "&&&" to indicate that you are finished: &&&
Total score: 0 points
```

Modify the `deal_hand` function to support always giving one wildcard in each hand. Note that `deal_hand` currently ensures that one third of the letters are vowels and the rest are consonants. Leave the vowel count intact, and replace one of the consonant slots with the wildcard. You will also need to modify one or more of the constants defined at the top of the file to account for wildcards.

Then modify the `is_valid_word` function to support wildcards. **Hint:** Check to see what possible words can be formed by replacing the wildcard with other letters. You may want to review the documentation for string module's `index()` function.

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

Problem 5. 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 "`&&&`" (three ampersands).

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 Why Pseudocode? resource to learn more about the What and Why of Pseudocode before you start this problem.

Testing: Try out your implementation as if you were playing the game.

Note: Your output may differ depending on what messages you print out, but it should conform to the same behavior. **You should not print extraneous "`None`" messages.**

Case #1

```
Current Hand: a r * e j j m
Enter word, or "&&&" to indicate that you are finished: jar
"jar" earned 70 points. Total: 70 points
```

```
Current Hand: e * m j
Enter word, or "&&&" to indicate that you are finished: e*m
"e*m" earned 52 points. Total: 122 points

Current Hand: j
Enter word, or "&&&" to indicate that you are finished: &&&
Total score: 122 points
```

Case #2

```
Current Hand: a t f i x l *
Enter word, or "&&&" to indicate that you are finished: fix
"fix" earned 91 points. Total: 91 points

Current Hand: a t l *
Enter word, or "&&&" to indicate that you are finished: tl
That is not a valid word. Please choose another word.

Current Hand: a *
Enter word, or "&&&" to indicate that you are finished: a *
"a*" earned 10 points. Total: 101 points

Ran out of letters. Total score: 101 points
```

Problem 6. Playing a game

A game consists of playing multiple hands. We need to implement one final function to complete our wordgame.

Implements the `play_game` function according to its specifications. For the game, you should use the `HAND_SIZE` constant to determine the number of cards in a hand.

Do **not** assume that there will always be 7 letters in a hand! The global variable `HAND_SIZE` represents this value. Our goal is to keep the code modular - if you want to try playing your word game with 10 letters or 4 letters you will be able to do it by simply changing the value of `HAND_SIZE`!

To do this, you will also need to implement `substitute_hand`. Read the description in `ps3.py`, then implement the described behavior. You will want to check the methods associated with dictionaries, such as `d.keys()` and `d.pop(some_key)`, where `d` is a dictionary, and `some_key` is a key that exists in the dictionary. You may also want to look at the code for `deal_hand` to see how `random.choice` can be used to select an element at random from a set of elements.

Note that we are not providing you with pseudocode for this problem. However, as you are deciding how to implement these functions, you may want to write your own as a guideline.

Testing: Try out this implementation as if you were playing the game. Try out different values for `HAND_SIZE` with your program, and be sure that you can play the wordgame with different hand sizes by modifying *only* the variable `HAND_SIZE`.

This completes the problem set!

Hand-in Procedure

1. Save
Save your solution file with the name that was provided – `ps3.py`.

Do not ignore this step or save your file with a different name!

2. Time and collaboration info
At the start of each file, in a comment, write down the number of hours (roughly) you spent on this problem set, and the names of whomever you collaborated with. For example:

```
# Problem Set 3
# Name:
# Collaborators:
# Time:
#
... your code goes here ...
```

3. Submit

To submit a file, upload it to the Problem Set 3 submission page on Gradebook. **Be sure to name your submission when you upload it on Gradebook - we will not be able to grade it otherwise!** You may upload new versions of the file until the 11:59pm deadline, but anything uploaded after that time will be counted towards your late days, if you have any remaining. If you have no remaining late days, you will receive no credit for a late submission. Please do not have more than one submission per file. **If you wish to resubmit a file you've previously submitted, delete the old file and then submit the revised copy.**

After you submit, please be sure to open up your submitted file and double-check that you can download it and you have submitted the right file.