Problem Set 3: The 6.00/6.0001 Word Game

Handed out: Thursday, September 21, 2017
Due: Friday, September 29, 2017 at 5:00 PM

Introduction

In this problem set, you'll implement a version of the 6.00/6.0001 word game! Don't be intimidated by the length of this document; the highlighted sections indicates the bulk of what you need to code and how you test it. **You may want to review the textbook readings on dictionaries before starting this pset (Section 5.6 will be most relevant).**

Let's begin by describing the word game: This game is a lot like Scrabble or Words With Friends. 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.

The rules of the game are as follows. **Do not start coding yet – as in Pset 2, we will break this down into steps below!**

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 [5 * *word_length* + 8 * (*word_length*-*n*)] or 2, whichever value is greater, where:
        - *word_length* 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.
- Examples:
    - 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 32 points: (4+1+1+2) * (5*4 + 8*(4-6)) = 32. 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 4 points: (1+1) * (2) = 4. The second component is 2 because 5*2 + 8*(2 - 7) = -30, which is less than 2.

Getting Started

1. Download and save `ps3.zip`. This includes the python file `ps3.py`, which should contain all of your code, as it provides a set of initial procedures and templates for new procedures. `ps3.zip` also includes a file for testing your code `test_ps3.py`, and a file of legitimate words `words.txt`. **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 `words.txt` and then calls the `play_game` function. You will implement the functions it needs in order to work later in the pset. If everything works correctly, the following should be 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*), make sure you have saved `words.txt` in the same directory as `ps3.py`!

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 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.

If your code passes a unit test you will see "ok" next to the test; otherwise you will see a FAIL message. For each failed test case, there will then be a section with the test name describing how the test case failed. **We will grade your pset using additional test cases, so you may want to test your code in other ways too** (for example, with different test values)**.**

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

Your output will look similar to this:

```
test_play_game_2_hands (__main__.TestPlayHandAndGame) ... FAIL
...
Traceback (most recent call last):
  File "/Users/user/Desktop/ps3/debugging/test_play_hand_game.py", line 313, in
test_play_game_2_hands
    self.fail(NOT_IMPLEMENTED)
AssertionError: You have not implemented this function yet.
```

For each test case that fails you will see a section that says either FAIL (or ERROR if your code throws an error) followed by the test name (in blue above) with a stack trace of the error message/failure. The most important line of the message is the last one (highlighted in yellow above) which will contain a message telling you why this test failed.

When tests pass, the output will look something like this:

```
test_play_hand_2 (__main__.TestPlayHandAndGame) ... ok
```

Problem 1: 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 `ps3.py` according to the function's scoring specifications.

You should 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 entered.

Finally, you may find the `str.lower` function helpful:

```
>>> s = "My String"
>>> print(s.lower())
"my string"
>>> print(s)
"My String"
```

NOTE: the `str.lower` function does **not** mutate the original string.

Testing: If this function is implemented correctly, and you run `test_ps3.py`, the following test will pass: **`test_get_word_score`**. You should also test your implementation of `get_word_score` yourself, using some reasonable English words. Note that the wildcard tests may crash due to a KeyError if you choose not to use the dictionary .get() method. 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**** 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 helper 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 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 (TO IMPLEMENT!)

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.

HINT: You may wish to review the documentation for the "`.copy`" method of Python dictionaries. You also might want to check the methods associated with dictionaries, such as `.keys`, or review the `del` keyword.

Testing: Run `test_ps3.py`. The following tests will now pass if your implementation is correct: **`test_update_hand_1, test_update_hand_2, test_update_hand_3`**. 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 (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: Running `test_ps3.py` the following tests will now pass: **`test_is_valid_word_hello_valid, test_is_valid_word_rapture_invalid, test_is_valid_word_honey_valid, test_is_valid_word_honey_invalid, test_is_valid_word_evil_valid, test_is_valid_word_even_invalid, test_is_valid_word_hello_invalid`**. You should also 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 to allow hands to contain wildcard letters, which will be denoted by an at sign (**@**). **Wildcards can only replace consonants.** 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 consonant should be, but should verify that at least one valid word can be made with the wildcard as a consonant in the desired position.

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 vowels count intact, and replace one of the consonants slots with the wildcard.

Testing: After modifying deal_hand to account for wildcards, the **`test_deal_hand_wildcard`** test should pass.

You will also need to modify one or more of the constants defined at the top of the file to account for wildcards with respect to scoring. Depending on your implementation, you may also need to modify `get_word_score` to account for the wildcard.

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 consonants. You may want to review the documentation for string module's `.find() function` and make note of its behavior when a character is not found. You might also want to look at the `.replace()`

`function`. The constant `CONSONANTS` defined for you at the top of the file may be helpful as well.

Testing: Make sure the following `test_ps3.py` tests pass: **`test_wildcard_1`, `test_wildcard_2`, `test_wildcard_3`, `test_wildcard_4`, `test_wilcard_score`.** You may also want to test your implementation with some reasonable inputs.

The examples below show how wildcards should behave in the context of playing a hand, which you will implement in Problem 5 below. Don't worry about that part yet - just pay attention to how the wildcard is handled.

Example #1: A valid word made without the wildcard

```
Current Hand:
c o w s @ z
Enter word, or "*END*" to indicate that you are finished: cows
"cows" earned 36 points. Total: 36 points

Current Hand:
@ z
Enter word, or "*END*" to indicate that you are finished: *END*
Total score for this hand: 36 points
```

Example #2: A valid word made using the wildcard

```
Current Hand:
c o w s @ z
Enter word, or "*END*" to indicate that you are finished: @ows
"@ows" earned 24 points. Total: 24 points

Current Hand:
c z
Enter word, or "*END*" to indicate that you are finished: *END*
Total score for this hand: 24 points
```

Example #3: An invalid word with a wildcard

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

Current Hand:
w s
Enter word, or "*END*" to indicate that you are finished: *END*
Total score for this hand: 0 points
```

Example #4: Another invalid word with a wildcard

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

Current Hand:
o z
```

```
Enter word, or "*END*" to indicate that you are finished: *END*
Total score for this hand: 0 points
```

Problem 5. Playing a hand

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

First implement the helper function `calculate_handlen`, which can be done in under five lines of code. After you are done, implement the `play_hand` function. This function allows the user to play out a single hand.

To end the hand early, the player **must** type "`*END*`".

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: run your program and call the `play_hand` function from your shell with a hand and the word_list. The following test cases from `test_ps3.py` will now pass: **`test_play_hand_basic`, `test_play_hand_1`, `test_play_hand_2`, `test_play_hand_with_invalid_word`, `test_play_hand_correct_handlen`, `test_play_hand_wildcard`.**

Note: Your output **should** match the examples below. **You should not print extraneous "`None`" messages. The grader requires that you ask the user for these inputs in the exact same order, and do not prompt the user for any additional information.**

Example #1

```
Current Hand:
a j e f @ r x d
Enter word, or "*END*" to indicate that you are finished: jar
"jar" earned 20 points. Total: 20 points

Current Hand:
@ f d e x
Enter word, or "*END*" to indicate that you are finished: fe@
"fe@" earned 10 points. Total: 30 points

Current Hand:
d x
Enter word, or "*END*" to indicate that you are finished: *END*
Total score for this hand: 30 points
```

Example #2

```
Current Hand:
a c f i @ t x
Enter word, or "*END*" to indicate that you are finished: fix
"fix" earned 26 points. Total: 26 points
```

```
Current Hand:
a c t @
Enter word, or "*END*" to indicate that you are finished: tc
That is not a valid word. Please choose another word.

Current Hand:
a @
Enter word, or "*END*" to indicate that you are finished: a@
"a@" earned 10 points. Total: 36 points

Total score for this hand: 36 points
```

Problem 6. Playing a game

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

Implement the `substitute_hand` and `play_game` functions according to their specifications. For the game, you should use the `HAND_SIZE` constant to determine the number of letters in a hand.

Do **not** assume that there will always be 7 letters in a hand! 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`!

When implementing substitution, you might want to check the methods associated with dictionaries, such as `.keys`, or review the `del` keyword. 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 (such as a string). Also make sure in substitute_hand that if the user has more than one of the letter they are substituting, they should receive that number of the randomly chosen letter. Assume the user is substituting a letter in their hand. Also make sure that consonants are only substituted for consonants and vowels for vowels. You can not substitute the wildcard (@).

Testing: After implementing `substitute_hand`, the following test cases should pass: `test_substitute_hand_vowel`, `test_substitute_hand_not_same_letter`, `test_substitute_hand_constonant`.

When implementing replay, make sure that you use the maximum of the two scores of the hand for the final score calculation.

Also, make sure that a use can only use replay and substitution once each over the course of an entire game. After they use replay/substitution your code should no longer prompt them asking them if they would like to replay a hand or substitute a letter.

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 word game with different hand sizes by modifying *only* the variable `HAND_SIZE`.

After completing this section all test cases starting with test_play_game should pass (`test_play_game_2_hands`, `test_play_game_basic`, `test_play_game_with_replay`, `test_play_game_with_substitution_and_replay`). Also, if you have implemented all other parts of this pset, all of the test cases should now pass!

You should try to test your code using a variety of possible inputs, i.e. replacing, substituting, substituting with replacing, etc.

***HINT* If you are having trouble passing some of the unit tests for play game, we have written a function `test_play_game` that takes in word_list, a list of hands, and optionally the letter that you want to be "chosen" by substitution. This function will run your code using these hands as the output of deal hand, and chose that letter in substitute hand. You can find more on this function in its DOCSTRING. It is at the bottom of the helper code.**

Example: (Strings printed to not have to match exactly, only points)

```
Enter total number of hands: 2
Current hand:
a c i @ p r t
Would you like to substitute a letter? no

Current hand:
a c i @ p r t
Please enter a word or '*END*' to indicate you are done: part
"part" earned 12 points. Total: 12 points

Current hand:
c i @
Please enter a word or '*END*' to indicate you are done: @ic
"@ic" earned 60 points. Total: 72 points

Total score for this hand: 72
----------
Would you like to replay the hand? no
Current hand:
d d @ e o u t

Would you like to substitute a letter? yes
Which letter would you like to replace: e

Current hand:
d d @ a o u t
Please enter a word or '*END*' to indicate you are done: out
"out" earned 6 points. Total: 6 points

Current hand:
d d @ a
Please enter a word or '*END*' to indicate you are done: *END*
Total score for this hand: 6
----------
Would you like to replay the hand? yes
Current hand:
d d @ a o u t
```

```
Please enter a word or '*END*' to indicate you are done: dad
"dad" earned 10 points. Total: 10 points

Current hand:
@ o u t
Please enter a word or '*END*' to indicate you are done: out
"out" earned 21 points. Total: 31 points

Current hand:
@
Please enter a word or '*END*' to indicate you are done: *END*
Total score for this hand: 31
----------
Total score over all hands: 103
```

Workload

Please let us know how long you spend on this problem set in the comment at the top of the file. We try to not overload you by giving out problems that take longer than we anticipated.

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: Alyssa P. Hacker
# Collaborators: Ben Bitdiddle
# Time: 3.14 hours
#
... your code goes here ...
```

3. Submit

To submit a file, upload it to the Problem Set 3 submission page on Stellar. You may upload new versions of the file until the 5:00pm 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.