Problem Set 3: The 6.00 Wordgame

Handed out: Thursday, February 28, 2013
Due: Tuesday, March 12, 2013 at 11:59 PM

Introduction

In this problem set, you'll implement *two* versions of the 6.00 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 6.00 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 *n* letters chosen at random (assume *n=7* for now).
- 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 (these won't be scored).

Scoring

- The score for the hand is the sum of the score for each word formed.
- The score for a word is the sum of the points for letters in the word, multiplied by the length of the word, plus 30 points if all *n* letters are used on the first go.
- 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, 'weed' would be worth 32 points ((4+1+1+2)*4=32), as long as the hand actually has 1 'w', 2 'e's, and 1 'd'.
- As another example, if *n*=7 and you get 'waybill' on the first go, it would be worth 135 points (the base score for 'waybill' is (4+1+4+3+1+1+1)*7=105, plus an additional 30 point bonus for using all *n* letters).

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, save, and extract `ps3.zip`
2. Run `ps3a.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 `ps3a.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 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_ps3a.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_ps3a.py` using the provided `ps3a.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.

Part A

Problem 1. Word scores

The first step is to implement some code that allows us to calculate the score for a single word. The function `get_word_score` should accept a string of lowercase letters as input (a *word*) and return the integer score for that word, using the game's scoring rules. Fill in the code for `get_word_score` in `ps3a.py`:

```
def get_word_score(word, n):
  """
  Returns the score for a word. Assumes the word is a
  valid word.

  The score for a word is the sum of the points for letters in the
  word, multiplied by the length of the word, plus 30 points if all n
  letters are used on the first go.

  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.

  word: string (lowercase letters)
  n: integer (HAND_SIZE; i.e., hand size required for additional points)
  returns: int >= 0
  """
  # TO DO ...
```

You may assume that the input word is always either a string of lowercase letters, or the empty string "". You will want to use the `SCRABBLE_LETTER_VALUES` dictionary defined at the top of `ps3a.py`. You should not change its value. Do **not** assume that there are always 7 letters in a hand! The parameter n is the number of letters required for a bonus score (the maximum number of letters in the hand). Our goal is to keep the code modular - if you want to try playing your word game with *n=10* or *n=4*, you will be able to do it by simply changing the value of `HAND_SIZE`!
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`, using some reasonable English words.

Problem 2. Dealing with hands

****Please read problem 2 entirely before you begin coding the solution to**

problem 2**

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 `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 `ps3a.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 object, hand containing n lowercase letters. Again, take a few minutes (right now!) 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 d`isplay_hand()` should be the same in either case.)
Implement the `update_hand` function. Make sure this function has no side effects; i.e., it cannot mutate the hand passed in.

```
def update_hand(hand, word):
  """
  Assumes that 'hand' has all the letters in word.
  In other words, assumes that however many times
  a letter appears in 'word', 'hand' has at least as
  many instances of that letter in it.

  Updates the hand: uses up the letters in the given word
  and returns the new hand, without those letters in it.

  Has no side effects: does not modify hand.

  word: string
  hand: dictionary (string -> int)
  returns: dictionary (string -> int)
  """
  # TO DO ...
```

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 written code to generate a random hand and display that hand to the user. We can also ask the user for a word (Python's `raw_input`) and score the word (using your `get_word_score`). However, 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.

```
def is_valid_word(word, hand, word_list):
  """
  Returns True if word is in the word_list and is entirely
  composed of letters in the hand. Otherwise, returns False.

  Does not mutate hand or word_list.

  word: string
  hand: dictionary (string -> int)
  word_list: list (string)
  """
  # TO DO ...
```

Testing: Make sure the `test_is_valid_word` tests pass. 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. 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 `calculate_handlen` function, which can be done in under five lines of code.

```
def calculate_handlen(hand):
    """
    Returns the length (number of letters) in the current hand.

    hand: dictionary (string-> int)
    returns: integer
    """
    # TO DO ....


def play_hand(hand, word_list):

    """
    Allows the user to play the given hand, as follows:

    * The hand is displayed.
    * The user may input a word or a single period (the string ".")
      to indicate they're done playing
    * Invalid words are rejected, and a message is displayed asking
      the user to choose another word until they enter a valid word or "."
    * When a valid word is entered, it uses up letters from the hand.
    * After every valid word: the score for that word is displayed,
      the remaining letters in the hand are displayed, and the user
      is asked to input another word.
    * The sum of the word scores is displayed when the hand finishes.
    * The hand finishes when there are no more unused letters or the user
      inputs a "."

      hand: dictionary (string -> int)
      word_list: list of lowercase strings
    """
    # BEGIN PSEUDOCODE
    ....
```

Note that after the line `# BEGIN PSEUDOCODE` there is a bunch of, well, pseudocode! This is to help guide you in writing your function.

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

Note: Do **not** assume that there will always be 7 letters in a hand! The global variable `HAND_SIZE` represents this value. Here is some example output of `play_hand` (your output may differ, depending on what messages you print out):

Case #1

```
Current Hand:  a c i h m m z
  Enter word, or a "." to indicate that you are finished: him
  "him" earned 24 points. Total: 24 points

  Current Hand:  a c m z
  Enter word, or a "." to indicate that you are finished: cam
  "cam" earned 21 points. Total: 45 points

  Current Hand:  z
  Enter word, or a "." to indicate that you are finished: .
  Total score: 45 points.
```

Case #2

```
Current Hand:  a s t t w f o
  Enter word, or a "." to indicate that you are finished: tow
  "tow" earned 18 points. Total: 18 points

  Current Hand:  a s t f
  Enter word, or a "." to indicate that you are finished: tasf
  Invalid word, please try again.

  Current Hand:  a s t f
  Enter word, or a "." to indicate that you are finished: fast
  "fast" earned 28 points. Total: 46 points.

  Total score: 46 points.
```

Problem 5. Playing a game

A game consists of playing multiple hands. We need to implement one final function to complete our word-game program. Write the code that implements the `play_game` function. You should remove the code that is currently uncommented in the `play_game` body. Read through the specification and make sure you understand what this function accomplishes. For the game, you should use the `HAND_SIZE` constant to determine the number of cards in a hand.

```
def play_game(word_list):
  """
  Allow the user to play an arbitrary number of hands.

  1) Asks the user to input 'n' or 'r' or 'e'.
    * If the user inputs 'n', let the user play a new (random) hand.
    * If the user inputs 'r', let the user play the last hand again.
    * If the user inputs 'e', exit the game.
    * If the user inputs anything else, ask them again.

  2) When done playing the hand, repeat from step 1
  """
  # TO DO ...
```

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`.

Part B

****Part B is dependent on your functions from `ps3a.py`, so be sure to complete `ps3a.py` before working on `ps3b.py`****

You decide to teach your computer (SkyNet) to play the game you just built so that you can prove once and for all that computers are inferior to human intellect. In Part B you will make a modification to the `play_hand` function from part A.

Problem 6. Computer Chooses a Word

First we must create a function that allows the computer to choose a word. We have provided the function `get_perms(hand, n)` (defined in *perm.py*, but usable by simply calling `get_perms(hand, n)`). The specification is as follows:

```
def get_perms(hand, n):
  """
  Takes in the current hand and a number.  It returns all
  possible permutations of size n given the letters in the hand.

  If n > len(hand), returns an empty list.

  hand: dictionary (string -> int)
  n: int
  returns list (string)
  """
  #IMPLEMENTED...
```

You are not required to know how `get_perms` works.
It is your responsibility to create the function `comp_choose_word(hand, word_list)`:

```
def comp_choose_word(hand, word_list):
  """
  Given a hand and a word_dict, find the word that gives
  the maximum value score, and return it.

  This word should be calculated by considering all possible
  permutations of lengths 1 to HAND_SIZE.

  If all possible permutations are not in word_list, return None.

  hand: dictionary (string -> int)
  word_list: list (string)
  returns: string or None
  """
  # BEGIN PSEUDOCODE
  ....
```

Note: Again, we're providing you with some pseudocode for this function. If you follow the pseudocode, you'll create a computer player that is legal, but not always the best. Feel free to delete the provided pseudocode and try your own approach if you'd prefer!

Problem 7. Computer's turn to play a hand

Now you need to write a function that's very similar to Part A's `play_hand` (hint hint...)
Implement the `comp_play_hand` function. This function should allow the computer to play the game through completion.

```
def comp_play_hand(hand, word_list):
  """
  Allows the computer to play the given hand, following the same procedure
  as play_hand, except instead of the user choosing a word, the computer
  chooses it.

  1) The hand is displayed.
  2) The computer chooses a word.
  3) After every valid word: the word and the score for that word is
  displayed, the remaining letters in the hand are displayed, and the
  computer chooses another word.
  4)  The sum of the word scores is displayed when the hand finishes.
  5)  The hand finishes when the computer has exhausted its possible
  choices (i.e. comp_choose_word returns None).

  hand: dictionary (string -> int)
  word_list: list (string)
  """
  # TO DO …
```

Problem 8. You & Your Computer

Now that your computer can choose a word, you need to give the computer the option to play. Write the code that re-implements the `play_game` function. You will modify the function to behave as described below in the function's comments. As before, you should use the `HAND_SIZE` constant to determine the number of cards in a hand. If you like, you can try out different values for `HAND_SIZE` with your program.

```
def play_game(word_list):
  """
  Allow the user to play an arbitrary number of hands.

  1) Asks the user to input 'n' or 'r' or 'e'.
      * If the user inputs 'e', immediately exit the game.
      * If the user inputs anything that's not 'n', 'r', or 'e', keep asking them again.

  2) Asks the user to input a 'u' or a 'c'.
     * If the user inputs anything that's not 'c' or 'u', keep asking them again.


  3) Switch functionality based on the above choices:
  * If the user inputted 'n', play a new (random) hand.
  * Else, if the user inputted 'r', play the last hand again.

  * If the user inputted 'u', let the user play the game
    with the selected hand, using play_hand.
  * If the user inputted 'c', let the computer play the
    game with the selected hand, using comp_play_hand.

  4) After the computer or user has played the hand, repeat from step 1


  word_list: list (string)
  """
  # TO DO ...
```

Note: You may notice that things run slowly when the computer plays. This is to be expected. If you want (totally optional!), feel free to investigate ways of making the computer's turn go faster - one way is to preprocess the word list into a dictionary (string -> int) so looking up the score of a word becomes much faster in the `comp_choose_word` function. Be careful though - you only want to do this preprocessing *one time* - probably right after we generate the `word_list` for you (at the bottom of the file). If you choose to do this, you'll have to modify what inputs your functions take (they'll probably take a word dictionary instead of a word list, for example).

This completes the problem set!

Hand-in Procedure

1. Save
Save your solution files with the names they were provided: `ps3a.py` and `ps3b.py`. You do *not* need to submit anything besides these two files.
Do not ignore this step or save your file(s) 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: Jane Lee
# Collaborators (Discussion): John Doe
# Time: 1:30
#
.... your code goes here ...
```

3. Submit

To submit a file, upload it to your Stellar workspace. You may upload new versions of each 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.
To submit a pset with multiple files, you may do one of two things:

1. You may submit a single .zip file that contains all of the requested files.
2. You may submit each file individually through the same Stellar submission page. Be sure that the top of each code file contains a comment with the title of the file you are submitting, e.g.

```
# Problem Set 3A
```

Also, after you submit, please be sure to open up your submitted file and double-check you submitted the right file. Please do not have more that one submission per file. If you wish to resubmit a file you've previously submitted, delete the old file (using the Stellar "delete" link) and then submit the revised copy.