6.00 Introduction to Computer Science and Programming
Fall 2008

6.00: Introduction to Computer Science and Programming

Problem Set 5: Wordgames

Handed out: Thursday, October 2, 2008
Due: 11:59pm, Thursday October 9, 2008

Introduction

As children, we loved wordgames like Ghost. So, like any good parent or teacher, we'll now force you to do the things that interested us when we were younger. In this problem set, **you'll implement** *two* **wordgames**: first, we'll help you through implementing the 6.00 wordgame, and then you'll implement Ghost on your own.

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 Text Twist, 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 a set of words 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 the words.
- The score for a word is the sum of the points for letters in the word, plus 50 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 8 points (4+1+1+2=8), 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 65 points (4+1+4+3+1+1+1=15, +50 for the 'bingo' bonus of using all seven letters).

Getting Started

1. Download and save

- ps5.py: the skeleton you'll fill in for Problems 1-5
- test_ps5.py: Unit tests for some of your code (more on this later)
- ps5_ghost.py: the skeleton you'll fill in for Problem 6
- words.txt: the list of valid words (all words acceptable in North American Scrabble up to 10 letters long)

Make sure to put all of the files in the same directory!

2. Run the code

Run `ps5.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 implemented.
play_hand not 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. Provided code

The file `ps5.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. Unit testing

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.

We have provided several test functions to get you started. As you make progress on the problem set, run `test_ps5.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_ps5.py` using the provided ps5.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.

I. Word scores

The first step is to implement some code that allows us to calculate the score for a single word.

Problem #1

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 `ps5.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, plus 50 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 (maximum 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 `ps5.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).

Testing:
If this function is implemented properly, and you run `test_ps5.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.

HINT: To iterate through all the characters in a string, try something like this loop, which prints each letter in `word`:

```
for letter in word:
    print letter
```

II. Dealing with hands

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`

A straightforward way to represent a hand in Python is as a `list`:

```
hand = ['a', 'q', 'l', 'm', 'u', 'i', 'l']
```

However, we'll represent the hand in a **different** way, because it simplifies the code we'll need in the `update_hand` and `is_valid_word` functions. (In general, there are many ways to represent, in code, various concepts -- some are better suited to certain operations than others).

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 an error. To avoid this, we can 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. It returns the value found if the key is in the dictionary, and `0` otherwise.

Converting words into dictionary representation

One useful function we've defined for you is `get_frequency_dict`, defined near the top of `ps5.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 the `display_hand` function. Make sure you read through this carefully and understand what it does and how it works.

```python
def display_hand(hand):
    """
    Displays the letters currently in the hand.

    For example:
       display_hand({'a':1, 'x':2, 'l':3, 'e':1})
    Should print out something like:
       a x x l l l e
    The order of the letters is unimportant.

    hand: dictionary (string -> int)
    """
    for letter in hand.keys():
      for j in range(hand[letter]):
         print letter, # the comma ensures everything in the for loop prints on
the same line
      print # this just prints an empty line
```

Generating a random hand

The hand a player is dealt is a set of letters chosen at random. We now need a function that generates this random hand. We have to be careful when randomly picking a hand. We need to ensure that there are enough VOWELS in the hand to allow the player to spell some words.

In our implementation, we use the `randrange` function from the `random` module to generate random numbers. Note that we `import random` at the top of `ps5.py`, and use the syntax `random.randrange()` to call `randrange` from inside module `random`. Make sure you read through our implementation of `deal_hand` carefully, and understand what it does and how it works.

```python
import random
def deal_hand(n):
    """
    Returns a random hand containing n lowercase letters.
    At least n/3 of the letters in the hand should be VOWELS.

    Hands are represented as dictionaries. The keys are
    letters and the values are the number of times the
    particular letter is repeated in that hand.

    n: int >= 0
    returns: dictionary (string -> int)
    """
    hand={}
    num_vowels = n / 3

    for i in range(num_vowels):
        x = VOWELS[random.randrange(0,len(VOWELS))]
        hand[x] = hand.get(x, 0) + 1

    for i in range(num_vowels, n):
        x = CONSONANTS[random.randrange(0,len(CONSANANTS))]
        hand[x] = hand.get(x, 0) + 1

    return hand
```

Notice the access of values using `hand.get(x,0)` because we do not know if the key x is in the dictionary, as discussed in the *Representing hands* section.

Removing letters from a hand

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.)

Problem #2

Implement the `update_hand` function. Make sure this function has no side effects; i.e., it cannot mutate the hand passed in.

```python
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 ...
```

Testing:

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

III. 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.

Problem #3

Implement the `is_valid_word` function.

```python
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 of strings
    """
    # 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?

IV. Playing a hand

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

Problem #4

Implement the `play_hand` function. This function allows the user to play out a single hand.

```python
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. Alternatively, the user may end the
      game by entering a period (.).

    * An invalid word is rejected, and a message is displayed asking
      the user to choose another word.

    * When a valid word is entered, it uses up letters from the hand.

    * After every valid word: the score for that word and the total
      score so far are 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.
      The user may choose to end the hand at any time by inputting
      a single period (the string '.') instead of a word.

    * The final score is displayed.

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

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):

```
Current Hand:  a c i h m m z
Enter word, or a . to indicate that you are finished: him
him earned 8 points. Total: 8 points
Current Hand:  a c m z
Enter word, or a . to indicate that you are finished: cam
cam earned 7 points. Total: 15 points
Current Hand:  z
Enter word, or a . to indicate that you are finished: .
Total score: 15 points.
```

V. Playing a game

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

Problem #5

Uncomment the code that implements the `play_game` function. You should remove the code that is currently uncommented in the `play_game` body. Read through and make sure you understand what this code does and how it works.

There is no coding for this question — the only "work" you have to do here is actually just uncommenting some lines and deleting some other lines of code.

For the game, 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.

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

    * Asks the user to input 'n' or 'r' or 'e'.

    * If the user inputs 'n', let the user play a new (random) hand.
      When done playing the hand, ask the 'n' or 'e' question again.

    * 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.
    """
    # TO DO ...
```

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

VI. Ghost: A Different Wordgame

Ghost is an inanely popular two-player wordgame. Our goal in this problem is to implement an interactive Python program that allows two humans to play a game of Ghost against each other. For those of you who are unfamiliar with the rules, you may read all about it here at Wikipedia. For this problem set, please follow the following rules.

Rules of Ghost

Players form a word by alternating turns saying a letter, which is added on to the end of the word fragment. There are two ways to lose Ghost:

- Forming a word longer than 3 letters ("PEA" is ok, but "PEAR" is not).
- Creating a fragment (of any size) which cannot become a word by adding more letters (for example, "QZ").

Winning Ghost is simply not losing! So, for example, game play proceeds like this:

- Player 1 says a letter. For example, 'P'.
- Player 2 says a letter. For example, 'E'.
- Player 1 says a letter. For example, 'A'. Player 1 has formed the word PEA, but that's okay because it is not longer than 3 letters.
- Player 2 says a letter. For example, 'F'.
- Player 1 says a letter. Player 1 must say 'O', because only one word starts with PEAF. If Player 1 says any other letter, for example 'A', he or she loses, because no word begins with PEAFA.
- Player 2 says a letter, which must be 'W'.
- Player 1 says a letter, which must be 'L'. Player 1 loses, because PEAFOWL is a word.

> **Problem #6**
> Implement a function, `ghost()`, that will start up an interactive Ghost game between two human players.

Getting Started

Download and save ps5_ghost.py into the same directory as your work for this problem set. This file will provide you with the function to load the word list.

Make sure your file runs properly before editing. You should get the following output when running the unmodified version of `ps5_ghost.py`.

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

You will want to do all of your coding for this problem within this file as well because you will be writing a program that depends on each function you write.

Requirements

Here are the requirements for your game:

- The game must be interactive: At each step, it should say who the current player is and what the current word fragment is.
- The player should be asked to input a letter, and the program should make sure that it the input is valid (one alphabetic character only, but it may be uppercase or lowercase).
- The letter should be added to the word fragment and the updated word fragment should be displayed.
- The game should end if the current player has formed a word (longer than 3 characters) or no words can be formed.

Make sure all of your code is within functions: either the main `ghost()` function, or another function that you create.

Hint: To determine if a character is alphabetic, you can see if it is in the `ascii_letters` string (which is part of the `string` module that is imported).

```
>> 'a' in string.ascii_letters
```

```
True
>> 'F' in string.ascii_letters
True
>> '2' in string.ascii_letters
False
```

Update on 7 Oct: Make sure your function accepts both lowercase and uppercase characters. When printing, convert to all uppercase or all lowercase characters (your choice). You may find the functions `string.upper()` and `string.lower()` useful — more information is available in the Python documentation.

The output of an example game may look like this:

```
Welcome to Ghost!
Player 1 goes first.
Current word fragment: ''
Player 1 says letter: P

Current word fragment: 'P'
Player 2's turn.
Player 2 says letter: Y

Current word fragment: 'PY'
Player 1's turn.
Player 1 says letter: T

Current word fragment: 'PYT'
Player 2's turn.
Player 2 says letter: H

Current word fragment: 'PYTH'
Player 1's turn.
Player 1 says letter: O

Current word fragment: 'PYTHO'
Player 2's turn.
Player 2 says letter: N

Current word fragment: 'PYTHON'
Player 2 loses because 'PYTHON' is a word!
Player 1 wins!
```

Here's another example, where a player loses after making an illegal word fragment.

```
Welcome to Ghost!
Player 1 goes first.
Current word fragment: ''
Player 1 says letter: P

Current word fragment: 'P'
Player 2's turn.
Player 2 says letter: Y

Current word fragment: 'PY'
Player 1's turn.
Player 1 says letter: N

Current word fragment: 'PYN'
Player 1 loses because no word begins with 'PYN'!
Player 2 wins!
```

Do not be intimidated by this problem! It's actually easier than it looks. Make sure you break down the problem into logical subtasks. What functions will you need to have in order for this game to work?

This completes the problem set!

Handin Procedure

1. Save
Save your wordgame file as it was provided: `ps5.py`.
Likewise, save your ghost file as it was provided: `ps5_ghost.py`.
Do not ignore this step or save your file(s) with a different name!

2. Time and collaboration info
At the start of the 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 5
# Name: Jane Lee
# Collaborators: John Doe
# Time: 3 hours 30 minutes
#
 .... your code goes here ...
```