

## 6.00: Introduction to Computer Science and Programming

# Problem Set 3: Word Game

Handed out: Wednesday, September 26, 2007

DUE:

- **Problem #1: 11:00am Friday, September 28, 2007.**
- **Problems #2-6: 11:00am Tuesday, October 2nd, 2007.**

## Introduction

As children, we loved word games like Hangman. 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 word games**: first, we'll help you through implementing the 6.00 word game, and then you'll implement Hangman on your own.

Don't be intimidated by the length of this problem set. It's a lot of reading, but should all be doable.

Let's begin by describing the 6.00 word game: This game is a lot like Boggle 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 multiplied by the length of the word.
- Letters are scored as follows: each vowel ( $a, e, i, o, u$ ) is worth one point; each consonant (all other letters) is worth two points.
- So, 'weed', for example, would be worth 24 points  $(2+1+1+2)\times 4$ , as long as the hand actually has a 'w', two 'e's, and a 'd'.

### 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.

### Collaboration

You may work with other students. However, each student should hand in the assignment separately. Also, be sure to indicate with whom you have worked. This is the collaboration policy for all the problem sets in this course.

## Getting Started

### 1. Download and save

- `ps3.py`: the skeleton you'll fill in
- `test_ps3.py`: Unit tests for some of your code (more on this later)
- `words.txt`: the list of valid words

### 2. Run the code

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...
55900 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 `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. 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_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` after downloading it, 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 `ps3.py`:

```
def get_word_score(word):  
    """  
    Returns the score for a word. Assumes the word is a  
    valid word.  
  
    The score for a word is the sum of the scores of the  
    individual letters, multiplied by the length of the  
    word.  
  
    Letters that are VOWELS get a score of 1. Letters that  
    are CONSONANTS get a score of 2.  
  
    word: string  
    returns: int >= 0  
    """  
    # TO DO ...
```

You may assume that the input is always either a string of lowercase letters, or the empty string `""`. You may want to use the `VOWELS` and `CONSONANTS` constants defined at the top of `ps3.py`. You should not change their values.

### Testing:

If this function is implemented properly, and you run `test_ps3.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.

## 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.

### 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. Make sure you read through this carefully and understand what it does and how it works.

```
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 to generate random numbers. Make sure you read through our implementation of `deal_hand` carefully, and understand what it does and how it works.

```
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[randrange(0,len(VOWELS))]
    hand[x] = hand.get(x, 0) + 1

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

return hand
```

## 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.

```
def update_hand(hand, word):
    """
    Updates the hand: uses up the letters in the given word
    and returns the new hand, without those letters in it.

    Assumes that 'word' is a valid word. Therefore,
    all the letters in 'word' are in the given 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.

```
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.

    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?

**HINT:** You may find the `get_frequency_dict` function useful (defined near the top of `ps3.py`). When given a string of letters as an input, it returns a dictionary where the keys are the letters and the values are the number of times that letter is repeated 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 have been using to represent hands.

### 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.

```
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.
    * 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 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.
      The user can also finish playing the hand by inputting a single
      period (the string '.') instead of a word.

    hand: dictionary (string -> int)
    word_list: list of strings
```

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

### Testing:

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

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 15 points. Total: 15 points
Current Hand: a c m z
Enter word, or a . to indicate that you are finished: cam
cam earned 15 points. Total: 30 points
Current Hand: z
Enter word, or a . to indicate that you are finished: .
Total score: 30 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.

```
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. Hangman: A Different Wordgame

For this problem, you will implement a variation of the classic wordgame, Hangman. For those of you who are unfamiliar with the rules, you may read all about it [here](#). In this problem, the second player will always be the computer, who will be picking a word at random.

### Problem #6

Implement a function, `hangman()`, that will start up an interactive Hangman game between a player and the computer.

## Getting Started

Download and save `ps3_hangman.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 `ps3_hangman.py`.

```
Loading word list from file...
 55900 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 computer must select a word at random from the list of available words that was used in the word game. As in the word game, the function for loading this list has already been provided for you.
- The game must be interactive: it should let a player know how many letters the word the computer has picked contains and ask the user to supply guesses. The user should receive feedback immediately after each guess. You should also display to the user the partially guessed word so far, as well as either the letters that the player has already guessed or letters that the player has yet to guess.
- A player is allowed some number of guesses. Once you understand how the game works, pick a number that seems reasonable to you.
- A player loses a guess only when s/he guesses incorrectly.
- The game should end when the player constructs the full word or runs out of guesses. If the player runs out of guesses (s/he "loses"), reveal the word to the player when the game ends.

The output of an example game may look like this:

```
Welcome to the game, Hangman!
I am thinking of a word that is 4 letters long.
```

```
-----
You have 8 guesses left.
Available letters: abcdefghijklmnopqrstuvwxyz
```

```
Please guess a letter: a
Good guess: _a__
```

```
-----
You have 8 guesses left.
Available letters: bcdefghijklmnopqrstuvwxyz
```

```
Please guess a letter: s
Oops! That letter is not in my word: _a__
```

```
-----
You have 7 guesses left.
Available letters: bcdefghijklmnopqrtuvwxyz
```

```
Please guess a letter: t
Good guess: ta_t
```

```
-----
You have 7 guesses left.
Available letters: bcdefghijklmnopqruvwxyz
```

```
Please guess a letter: r
Oops! That letter is not in my word: ta_t
```

```
-----
You have 6 guesses left.
Available letters: bcdefghijklmnopquvwxyz
```

```
Please guess a letter: m
Oops! That letter is not in my word: ta_t
```

```
-----
You have 5 guesses left.
Available letters: bdefghijklmnopquvwxyz
```

```
Please guess a letter: c
Good guess: tact
```

```
-----
Congratulations, you won!
```

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 word game file as it was provided: `ps3.py`.

Likewise, save your hangman file as it was provided: `ps3_hangman.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 3
# Name: Jane Lee
# Collaborators: John Doe
# Time: 1:30
#
.... your code goes here ...
```

### 3. Submit

Upload the file to your workspace. If there is some error uploading to your workspace, email the file to the 6.00 staff.

You may upload (or email) new versions of the problem set until the **11am deadline**, but anything uploaded after that will be ignored.