6.0001/6.00 Fall 2016

Problem Set 2

Handed out: Tuesday, September 20th, 2016.

Due: Tuesday, September 27th, 2016 at 11:59pm

This problem set will introduce you to the topic of creating functions in Python, as well as looping mechanisms for repeating a computational process until a condition is reached.

<u>Note on Collaboration</u>:
You may work with other students. However, each student should write up and hand in his or her assignment separately. *Be sure to indicate with whom you have worked in the comments of your submission.*

Problem 1: Basic Hangman

You will implement a variation of the classic word game Hangman. If you are unfamiliar with the rules of the game, read http://en.wikipedia.org/wiki/Hangman_(game). Don't be intimidated by this problem - it's actually easier than it looks! We will 'scaffold' this problem, guiding you through the creation of helper functions before you implement the actual game.

A) Getting Started

Download the files "hangman.py" and "words.txt", and **save them both in the same directory**. Run the file hangman.py before writing any code to ensure your files are saved correctly. The code we have given you loads in words from a file. You should see the following output in your shell:

> Loading word list from file...
> 55900 words loaded.

If you see the above text, continue on to Hangman Game Requirements.
If you don't, double check that both files are saved in the same place!

B) Hangman Game Requirements

You will implement a function called `hangman` that will allow the user to play hangman against the computer. The computer picks the word, and the player tries to guess letters in the word.

Here is the general behavior we want to implement. Don't be intimidated! This is just a description; **we will break this down into steps and provide further functional specs later on in the pset so keep reading!**

1. The computer must select a word at random from the list of available words that was provided in words.txt
   Note that words.txt contains words in all lowercase letters.
2. The user is given a certain number of guesses at the beginning.
3. The game is interactive; the user inputs their guess and the computer either:
   a. reveals the letter if it exists in the secret word
   b. penalize the user and updates the number of guesses remaining
4. The game ends when either the user guesses the secret word, or the user runs out of guesses.

Problem 2
Hangman Part 1: Three helper functions

Before we have you write code to organize the hangman game, we are going to break down the problem into logical subtasks, creating three helper functions you will need to have in order for this game to work. This is a common approach to computational problem solving, and one we want you to begin experiencing.

The file hangman.py has a number of already implemented functions you can use while writing up your solution. You can ignore the code in the two functions at the top of the file that have already been implemented for you, though you should understand how to use each helper function by reading the docstrings.

1A) Determine whether the word has been guessed

First, implement the function `is_word_guessed` that takes in two parameters - a string, `secret_word`, and a list of letters (strings), `letters_guessed`. This function returns a boolean - `True` if `secret_word` has been guessed (i.e., all the letters of `secret_word` are in `letters_guessed`), and `False` otherwise. This function will be useful in helping you decide when the hangman game has been successfully completed, and becomes an end-test for any iterative loop that checks letters against the secret word.

For this function, you may assume that all the letters in `secret_word` and `letters_guessed` are lowercase.

Example Usage:

```
>>> secret_word = 'apple'
>>> letters_guessed = ['e', 'i', 'k', 'p', 'r', 's']
>>> print(is_word_guessed(secret_word, letters_guessed))
False
```

1B) Getting the user's guess

Next, implement the function `get_guessed_word` that takes in two parameters - a string, `secret_word`, and a list of letters, `letters_guessed`. This function returns a string that is comprised of letters and underscores, based on what letters in `letters_guessed` are in `secret_word`. This shouldn't be too different from `is_word_guessed`!

We are going to use an underscore followed by a space (_) to represent unknown letters. We could have chosen other symbols, but the combination of underscore and space is visible and easily discerned. Note that the space is super important, as otherwise it hard to distinguish whether _____ is four elements long or three. This is called *usability* - it's very important, when programming, to consider the usability of your program. If users find your program difficult to understand or operate, they won't use it! We encourage you to think about usability when designing your program.

Hint: In designing your function, think about what information you want to return when done, whether you need a place to store that information as you loop over a data structure, and how you want to add information to your accumulated result.

Example Usage:

```
>>> secret_word = 'apple'
>>> letters_guessed = ['e', 'i', 'k', 'p', 'r', 's']
>>> print(get_guessed_word(secret_word, letters_guessed))
'_ pp_ e'
```

1C) Getting all available letters

Next, implement the function `get_available_letters` that takes in one parameter - a list of letters, `letters_guessed`. This function returns a string that is comprised of lowercase English letters - all lowercase English letters that are not in `letters_guessed`.

This function should return the letters in alphabetical order. For this function, you may assume that all the letters in `letters_guessed` are lowercase.

Hint: You might consider using `string.ascii_lowercase`, which is a string comprised of all lowercase letters:

```
>>> import string
>>> print(string.ascii_lowercase)
abcdefghijklmnopqrstuvwxyz
```

Example Usage:

```
>>> letters_guessed = ['e', 'i', 'k', 'p', 'r', 's']
>>> print get_available_letters(letters_guessed)
abcdfghjlmnoqtuvwxyz
```

Problem 3
Hangman Part 2: The Game

Now that you have built some useful functions, you can turn to implementing the function `hangman`, which takes one parameter - the `secret_word` the user is to guess. Initially, you can (and should!) manually set this secret word when you run this function – this will make it easier to test your code. But in the end, you will want the computer to select this secret word at random before inviting you or some other user to play the game by running this function.

Calling the `hangman` function starts up an interactive game of Hangman between the user and the computer. In designing your code, be sure you take advantage of the three helper functions, `is_word_guessed`, `get_guessed_word`, and `get_available_letters`, that you've defined in the previous part!

Below are the game requirements broken down in different categories. Make sure your implementation fits all the requirements!

Game Requirements

A. Game Architecture:

1. The computer must select a word at random from the list of available words that was provided in words.txt. The functions for loading the word list and selecting a random word have already been provided for you in hangman.py.
2. Users start with 6 guesses.
3. At the start of the game, let the user know how many letters the computer's word contains and how many guesses s/he starts with.
4. The computer keeps track of all the letters the user has not guessed so far and before each turn shows the user the "remaining letters"

Example Game Implementation:
```
Loading word list from file...
55900 words loaded.
Welcome to the game Hangman!
I am thinking of a word that is 4 letters long.
-------------
You have 6 guesses left.
Available letters: abcdefghijklmnopqrstuvwxyz
```

B. User-Computer Interaction:

The game must be interactive and flow as follows:
1. Before each guess, you should display to the user:
   a. Remind the user of how many guesses s/he has left after each guess.
   b. all the letters the user has not yet guessed
2. Ask the user to supply one guess at a time. (Look at the user input requirements below to see what types of inputs you can expect from the user)
3. Immediately after each guess, the user should be told whether the letter is in

the computer's word.
4. After each guess, you should also display to the user the computer's word, with guessed letters displayed and unguessed letters replaced with an underscore and space (_)
5. At the end of the guess, print some dashes (-----) to help separate individual guesses from each other

Example Game Implementation:
(The blue color below is only there to show you what the user typed in, as opposed to what the computer output.)

```
You have 6 guesses left.
Available letters: abcdefghijklmnopqrstuvwxyz
Please guess a letter: a
Good guess: _ a_ _
------------
You have 6 guesses left.
Available letters: bcdefghijklmnopqrstuvwxyz
Please guess a letter: b
Oops! That letter is not in my word: _ a_ _
```

C. User Input Requirements:

1. You may assume that the user will only guess one character at a time, but the user can choose any number, symbol or letter. Your code should accept capital and lowercase letters as valid guesses!
2. If the user inputs anything besides an alphabet (symbols, numbers), tell the user that they can only input an alphabet. Because the user might do this by accident, they should get 3 warnings at the beginning of the game. Each time they enter an invalid input, or a letter they have already guessed, they should lose a warning. If the user has no warnings left and enters an invalid input, they should lose a guess.

Hint #1: Use calls to the `input` function to get the user's guess.
   a. Check that the user input is an alphabet
   b. If the user does not input an uppercase or lowercase alphabet letter, subtract one warning or one guess.
Hint #2: you may find the string functions `str.isalpha('your string')` and `str.lower('Your String')` helpful! If you don't know what these functions are you could try typing help(str.isalpha) or help(str.lower) in your Spyder shell to see the documentation for the functions.
Hint #3: Since the words in words.txt are lowercase, it might be easier to convert the user input to lowercase at all times and have your game only handle lowercase.

Example Game Implementation:

```
You have 3 warnings left.
You have 6 guesses left.
Available letters: bcdefghijklmnopqrstuvwxyz
Please guess a letter: s
Oops! That letter is not in my word: _ a_ _
```

```
------------
You have 5 guesses left.
Available letters: bcdefghijklmnopqrtuvwxyz
Please guess a letter: $
Oops! That is not a valid letter. You have 2 warnings left:  _ a_ _
```

D. Game Rules:

1. The user starts with 3 warnings.
2. If the user inputs anything besides an alphabet (symbols, numbers), tell the user that they can only input an alphabet.
   a. If the user has one or more warning left, the user should lose one warning. Tell the user the number of remaining warnings.
   b. If the user has no remaining warnings, they should lose one guess.
3. If the user inputs a letter that has already been guessed, print a message telling the user the letter has already been guessed before.
   a. If the user has one or more warning left, the user should lose one warning. Tell the user the number of remaining warnings.
   b. If the user has no warnings, they should lose one guess.
4. If the user inputs a letter that hasn't been guessed before and the letter is in the secret word, the user loses **no** guesses.
5. **Consonants:** If the user inputs a consonant that hasn't been guessed and the consonant is not in the secret word, the user loses **one** guess if it's a consonant.
6. **Vowels:** If the vowel hasn't been guessed and the vowel is not in the secret word, the user loses **two** guesses. Vowels are *a*, *e*, *i*, *o*, and *u*. *y* does not count as a vowel.

Example Implementation:
```
You have 5 guesses left.
Available letters: bcdefghijklmnopqrtuvwxyz
Please guess a letter: t
Good guess: ta_ t
------------
You have 5 guesses left.
Available letters: bcdefghijklmnopqrtuvwxyz
Please guess a letter: e
Oops! That letter is not in my word: ta_ t
------------
You have 3 guesses left.
Available letters: bcdfghijklmnopqrtuvwxyz
Please guess a letter: e
Oops! You've already guessed that letter. You now have 2 warnings:
ta_ t
```

E. Game Termination:

1. The game should end when the user constructs the full word or runs out of guesses.
2. If the player runs out of guesses before completing the word, tell them they lost and reveal the word to the user when the game ends.

3. If the user wins, print a congratulatory message and tell the user their score.
4. The total score is the number of `guesses_remaining` once the user has guessed the `secret_word` times the number of unique letters in `secret_word`.

Total score = guesses_remaining* number unique letters in secret_word

Example Implementation:
```
You have 3 guesses left.
Available letters: bcdfghijklnopquvwxyz
Please guess a letter: c
Good guess: tact
------------
Congratulations, you won!
Your total score for this game is: 9
```

Example Implementation:
```
You have 3 guesses left.
Available letters: bcdfghijklnopquvwxyz
Please guess a letter: n
Good guess: dolphin
------------
Congratulations, you won!
Your total score for this game is: 21
```

F. General Hints:

1. Consider writing additional helper functions if you need them.
2. There are four important pieces of information you may wish to store:
   a. `secret_word`: The word to guess. This is already used as the parameter name for the `hangman` function.
   b. `letters_guessed`: The letters that have been guessed so far. If they guess a letter that is already in `letters_guessed`, you should print a message telling them they've already guessed that but do not penalize them for it.
   c. `guesses_remaining`: The number of guesses the user has left. Note that in our example game, the penalty for choosing an incorrect vowel is different than the penalty for choosing an incorrect consonant.
   d. `warnings_remaining`: The number of warnings the user has left. Note that a user only loses a warning for inputting either a symbol or a letter that has already been guessed.

G. Example Game:

Look carefully at the examples given above of running `hangman`, as that suggests examples of information you will want to print out after each guess of a letter.

Note: Try to make your print statements as close to the example game as possible!

The output of a **winning** game should look like this. (The blue color below is only there to show you what the user typed in, as opposed to what the computer output.)

```
Loading word list from file...
55900 words loaded.
Welcome to the game Hangman!
I am thinking of a word that is 4 letters long.
You have 3 warnings left.
-------------
You have 6 guesses left.
Available letters: abcdefghijklmnopqrstuvwxyz
Please guess a letter: a
Good guess: _ a_ _
------------
You have 6 guesses left.
Available letters: bcdefghijklmnopqrstuvwxyz
Please guess a letter: a
Oops! You've already guessed that letter. You have 2 warnings left:
 _ a_ _
------------
You have 6 guesses left.
Available letters: bcdefghijklmnopqrstuvwxyz
Please guess a letter: s
Oops! That letter is not in my word.
Please guess a letter: _ a_ _
------------
You have 5 guesses left.
Available letters: bcdefghijklmnopqrtuvwxyz
Please guess a letter: $
Oops! That is not a valid letter. You have 1 warnings left: _ a_ _
------------
You have 5 guesses left.
Available letters: bcdefghijklmnopqrtuvwxyz
Please guess a letter: t
Good guess: ta_ t
------------
You have 5 guesses left.
Available letters: bcdefghijklmnopqrtuvwxyz
Please guess a letter: e
Oops! That letter is not in my word: ta_ t
------------
You have 3 guesses left.
Available letters: bcdfghijklmnopqrtuvwxyz
Please guess a letter: e
Oops! You've already guessed that letter. You have 0 warnings left:
ta_ t
------------
You have 3 guesses left.
Available letters: bcdfghijklmnopqrtuvwxyz
Please guess a letter: e
Oops! You've already guessed that letter. You have no warnings left
so you lose one guess: ta_ t
```

```
------------
You have 2 guesses left.
Available letters: bcdfghijklnopquvwxyz
Please guess a letter: c
Good guess: tact
------------
Congratulations, you won!
Your total score for this game is: 6
```

And the output of a **losing** game should look like this...

```
Loading word list from file...
55900 words loaded.
Welcome to the game Hangman!
I am thinking of a word that is 4 letters long
You have 3 warnings left.
-----------
You have 6 guesses left
Available Letters: abcdefghijklmnopqrstuvwxyz
Please guess a letter: a
Oops! That letter is not in my word: _ _ _ _
-----------
You have 4 guesses left
Available Letters: bcdefghijklmnopqrstuvwxyz
Please guess a letter: b
Oops! That letter is not in my word: _ _ _ _
-----------
You have 3 guesses left
Available Letters: cdefghijklmnopqrstuvwxyz
Please guess a letter: c
Oops! That letter is not in my word: _ _ _ _
-----------
You have 2 guesses left
Available Letters: defghijklmnopqrstuvwxyz
Please guess a letter: 2
Oops! That is not a valid letter. You have 2 warnings left: _ _ _ _
-----------
You have 2 guesses left
Available Letters: defghijklmnopqrstuvwxyz
Please guess a letter: d
Oops! That letter is not in my word: _ _ _ _
-----------
You have 1 guesses left
Available Letters: efghijklmnopqrstuvwxyz
Please guess a letter: e
Good guess: e_ _ e
-----------
You have 1 guesses left
Available Letters: fghijklmnopqrstuvwxyz
Please guess a letter: f
```

```
        Oops! That letter is not in my word: e_ _ e
        -----------
        Sorry, you ran out of guesses. The word was else.
```

Once you have completed and tested your code (where you have manually provided the "secret" word, since knowing it helps you debug your code), you may want to try running against the computer. If you scroll down to the bottom of the file we provided, you will see two commented lines underneath the text `if __name__ ==` "`__main__`":

```
#secret_word = choose_word(wordlist)
#hangman(secret_word)
```

These lines use functions we have provided (near the top of hangman.py), which you may want to examine. Try "uncommenting" these lines, and reloading your code. This will give you a chance to try your skill against the computer, which uses our functions to load a large set of words and then pick one at random.

Problem 4
Hangman Part 3: The Game with Hints

If you have tried playing Hangman against the computer, you may have noticed that it isn't always easy to beat the computer, especially when it selects an esoteric word (like "esoteric"!). It might be nice if you could ask the computer for a hint, such as a list of all the words that match what you have currently guessed.

For example, if the hidden word is "tact", and you have so far guessed the letter "t", so that you know the solution is "t_ _ t", where you need to guess the two missing letters, it might be nice to know that the set of matching words (at least based on what the computer initially loaded) are:

> tact tart taut teat tent test text that tilt tint toot tort tout trot tuft twit

We are going to have you create a variation of Hangman (we call this `hangman_with_hints`, and have provided an initial scaffold for writing it), with the property that if you guess the special character * the computer will find all the words from its loaded list that might match your current guessed word, and print out each of them. Of course, we don't recommend trying this at the first step, since this will print out all 55,900 words that we loaded! But if you are getting close to an answer and are running out of guesses, this might help.

To do this, we are going to ask you to first complete two helper functions:

3A) Matching the current guessed word

`match_with_gaps` takes two parameters: `my_word` and `other_word`. `my_word` is an instance of a guessed word, in other words, it may have some _ 's in places (such as 't_ _ t'). `other_word` is a normal English word.

10

This function should return `True` if the guessed letters of `my_word` match the corresponding letters of `other_word`. It should return `False` if the two words are not of the same length or if a guessed letter in `my_word` does not match the corresponding character in `other_word`.

Remember that when a letter is guessed, your code reveals all the positions at which that letter occurs in the secret word. Therefore, the hidden letter (_) <u>cannot be </u>one of the letters in the word that has already been revealed.

Example Usage:

```
>>> match_with_gaps("te_ t", "tact")
False
>>> match_with_gaps("a_ _ le", "banana")
False
>>> match_with_gaps("a_ _ le", "apple")
True
>>> match_with_gaps("a_ ple", "apple")
False ❓
```

Hint: You may want to use strip() to get rid of the spaces in the word to compare lengths.

3B) Showing all possible matches

`show_possible_matches` takes a single parameter: `my_word` which is an instance of a guessed word, in other words, it may have some _ 's in places (such as 't_ _ t').

This function should print out all words in `wordlist` (notice where we have defined this at the beginning of the file, line 51) that match `my_word`. It should print "No matches found" if there are no matches.

Example Usage:

```
>>> show_possible_matches("t_ _ t")
tact tart taut teat tent test text that tilt tint toot tort tout trot tuft
twit

>>> show_possible_matches("abbbb_ ")
No matches found

>>> show_possible_matches("a_ pl_ ")
ample amply
```

3C) Hangman with hints

Now you should be able to replicate the code you wrote for `hangman` as the body of `hangman_with_hints`, then make a small addition to allow for the case where the user

can guess an asterisk (*), in which case the computer will print out all the words that match that guess.

The user should not lose a guess if the guess is an asterisk.

Comment out the lines of code you used to play the original Hangman game:
```
secret_word = choose_word(wordlist)
hangman(secret_word)
```

And un-comment out these lines of code we've provided at the bottom of the file to play your new game Hangman with Hints:
```
#secret_word = choose_word(wordlist)
#hangman_with_hints(secret_word)
```

Sample Output:
The output from guessing an asterisk should look like the sample output below. All other output should follow the Hangman game described in Part 2 above.

```
Loading word list from file...
    55900 words loaded.
Welcome to the game Hangman!
I am thinking of a word that is 5 letters long.
--------
You have 6 guesses left.
Available letters: abcdefghijklmnopqrstuvwxyz
Please guess a letter: a
Good guess: a_ _ _ _
--------
You have 6 guesses left.
Available letters: bcdefghijklmnopqrstuvwxyz
Please guess a letter: l
Good guess: a_ _ l_
--------
You have 6 guesses left.
Available letters: bcdefghijkmnopqrstuvwxyz
Please guess a letter: *
Possible word matches are:
addle adult agile aisle amble ample amply amyls angle ankle apple
apply aptly arils atilt
--------
You have 6 guesses left.
Available letters: bcdefghijkmnopqrstuvwxyz
Please guess a letter: e
Good guess: a_ _ le
--------
```

This completes the problem set!

6.0001 Introduction to Computer Science and Programming in Python
Fall 2016