Problem Set 2

Handed out: Monday, September 21, 2015.

Due: Tuesday, September 29, 2015

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.

Note on Collaboration:
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 instead see an IOError (e.g., "No such file or directory"), you should change the value of the WORDLIST_FILENAME constant (defined near the top of the file, at line 15) to the complete pathname for the file words.txt. Windows users should change the backslashes in the following example to forward slashes.

For example, if you saved hangman.py and words.txt in the directory "C:/Users/Ana/" (on Windows, this would be "C:\Users\Ana\") change the line:

```
WORDLIST_FILENAME = "words.txt"
```

to

WORDLIST_FILENAME = "C:/Users/Ana/words.txt"

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. The functions for loading the word list and selecting a random word have already been provided for you in hangman.py.
2. The game must be interactive and flow as follows:
    a. 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.
    b. Before each guess, you should display to the user:
        i. how many guesses s/he has left
        ii. all the letters the user has not yet guessed
    c. Ask the user to supply one guess (i.e., letter) at a time.
    d. Immediately after each guess, the user should be told whether the letter is in the computer's word.
    e. After each guess, you should also display to the user the computer's word, with guessed letters displayed and unguessed letters replaced with an asterisk (*)
    f. At the end of the guess, print some dashes (-----) to help separate individual guesses from each other
3. Additional rules:
    a. Users start with 6 guesses. Remind the user of how many guesses s/he has left after each guess.
    b. If the letter has already been guessed, the user loses one guess. Print a message telling the user the letter was a duplicate.
    c. If the letter hasn't been guessed and the letter is in the secret word, the user loses no guesses.
    d. If the letter hasn't been guessed and the letter is not in the secret word, the user loses one guess if it's a consonant.
    e. If the letter hasn't been guessed and the letter is not in the secret word, the user loses two guesses if it's a vowel. Vowels are *a*, *e*, *i*, *o*, and *u*. *y* does not count as a vowel.
    f. The game should end when the user constructs the full word or runs out of guesses. If the player runs out of guesses before completing the word, reveal the word to the user when the game ends. If the user wins, print a congratulatory message.

Note: For the purposes of this pset, **you may assume all letters will be lowercase**. You may also assume that players will only guess one character at a time (a-z), and you may assume that the player will not submit invalid guesses such as numbers, symbols, or capital letters.

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 decided 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 asterisks, 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 asterisk (*) to represent unknown letters. We could have chosen other symbols, but the asterisk is visible and easily discerned.

For example, we might have chosen the underscore (_), but one challenge with using that symbol is that is can be hard to determine the number of consecutive unguessed letters. It's difficult to tell whether ____ is four elements long or three elements long, whereas it is easy to see that **** is four elements long. 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.

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 the 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
```

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!

As mentioned in the game requirements, you may assume that players only guess one character at a time (a-z) and that the player will not submit invalid guesses such as numbers, symbols, or capital letters.

Hints:

- Use calls to the `raw_input` function to get the user's guess.
- Consider writing additional helper functions if you need them.
- There are four important pieces of information you may wish to store:
    1. `secret_word`: The word to guess. This is already used as the parameter name for the `hangman` function.
    2. `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, and count that as a mistake.
    3. `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.
    4. 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.

Example Usage:
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 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: *a**
------------
You have 5 guesses left.
Available letters: bcdefghijklmnopqrstuvwxyz
Please guess a letter: s
Oops! That letter is not in my word: *a**
------------
You have 4 guesses left.
Available letters: bcdefghijklmnopqrtuvwxyz
Please guess a letter: t
```

```
Good guess: ta*t
------------
You have 4 guesses left.
Available letters: bcdefghijklmnopqrtuvwxyz
Please guess a letter: e
Oops! That letter is not in my word: ta*t
------------
You have 2 guesses left.
Available letters: bcdefghijklmnopqrtuvwxyz
Please guess a letter: e
Oops! You've already guessed that letter: ta*t
------------
You have 1 guesses left.
Available letters: bcdfghijklnopquvwxyz
Please guess a letter: c
Good guess: tact
------------
Congratulations, you won!
```

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 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: 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. In the file we provided, you will see two commented lines:

```
#secret_word = choose_word(wordlist).lower()
#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.

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.

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`.

Example Usage:

```
>>> match_with_gaps("te*t", "tact")
False
>>> match_with_gaps("a**le", "banana")
False
>>> match_with_gaps("a**le", "apple")
True
```

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 48) that match `my_word`. It should print a blank line 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*")

>>> show_possible_matches("a*pl*")
ample amply apple apply
```

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).lower()
hangman(secret_word)
```

And 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).lower()
#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 atoll
--------
You have 6 guesses left.
Available letters: bcdefghijkmnopqrstuvwxyz
Please guess a letter: e
Good guess: a**le
--------
```

This completes the problem set!

Hand-in Procedure

1. Naming Files

Save your solutions with the original file name: hangman.py. **Do not ignore this step or save your files with a different name!**

2. Time and Collaboration Info

At the start of your file, in a comment, write down the number of hours (roughly) you spent on the problems in that part, and the names of the people with whom you collaborated.

For example:

```
# Problem Set 2
# Name: Jane Lee
# Collaborators: John Doe
# Time Spent: 3:30
# Late Days Used: 1 (only if you are using any)
# … 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 11:59 PM 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 the following:

1. Submit a single .zip file that contains all the files.
2. 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 1A

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