COSC 101 Homework 4 Spring 2016

Due date: **Monday, April 4 , 11:00 p.m.**

Introduction

The goal of this homework is to create a *well-designed* game to play Hangperson (yes, that's a gender-neutral version of "Hangman"). If you're unsure how to play, please talk to your instructor or a fellow student and play a practice game or two. It's pretty important that you have a clear image in mind for how the game works before diving in to the assignment.

Please write your code in the template file `hw5.py` included with this homework. Note that the file includes a function (already written!) for drawing the gallows, and the start of a `main` function for your program.

Required functionality

Here are the basic guidelines for how your program should behave:

- The "gallows" that you draw will be in low-tech ASCII (just letter, numbers and/or symbols). (We won't be using Turtle or another graphics package.) A function called `draw_gallows` is provided in the `hw5.py` template file. Given an integer `misses` representing the number of missed guesses, it draws the gallows and `misses` body parts.

  There should be exactly **seven** missed guesses allowed before declaring that a player has lost. Think to use constants to make your program flexible and adaptable to change.

- Each time you ask for a new letter to guess, you should display the letters that have been previously guessed, and the current "state" of guessing. For the current guess state, you can display the actual letters that have been correctly guessed, and underscores for letters that are yet to be guessed correctly. For example, if the secret word is `SUNNY` and the player has correctly guessed the `N`, you should print something like:

  ```
  __ __ N N __
  ```

  Note that the blanks are formed with a couple underscore characters. Note also that the output is rather nicely formatted. You should not print out a "raw" Python list.

- When a player wins or loses (i.e., when a game is over), you should ask the player if he or she would like to play again. The player should be able to simply type `y` or `n` to respond (or `Y` or `N`). After getting a keyboard response from the player, you should either end the program or initiate a new game based on the response.

- Your program should be able to handle the case when a player enters a non-letter, more than one letter, or some other kind of bogus guess.
  The program should not crash, an appropriate error message should be shown, and the game should continue. Any invalid guesses (non-letters) should *not* count against a player.

- Any duplicate guesses should not count against a player. For example, if a player has already guessed `A` and they guess it again, you should print a message like `You've already guessed A`, but otherwise do nothing.

- Make your output in all parts of the game look nice. For example, if you have some information stored in a Python list that you want to display, don't just print the list verbatim. Instead, construct some nicer looking output. Example output is described below.

- Once that you're sure about how the game is played and you understand the basic requirements listed above, you'll need to think carefully about how to design implement the game in Python. Ensure that all your functions are SOFA: that they are short, do one thing, take few arguments, and are pitched at a consistent level of abstraction. Moreover, make sure that your functions contain well-formed docstrings and that you include comments next to portions of code that need some further explanation.
- Once your game is working and fully tested with a hard-coded `wordlist` to choose the secret from, i.e. the list of strings provided and the alternative ones you used you have completed the `EASY` version of the game. The last requirement is to implement an `HARD` version of the game, that uses a complex set of strings to play the game.
- Extend your program so that at the start the player may choose to play the `HARD` version.
  - When the `HARD` version is selected the player is prompted to first enter an integer, which defines the `length` of the word to guess.
  - The program reads the wordlist file, `wordlist.txt` and automatically generates the corresponding filtered list of words to randomly choose the secret from: all the words in your list should be of the length the user gave.
  - The game is then played as before but at the end the secret is also printed (especially in the lose condition) as is the word list used.

Example output

The following web pages demonstrate what a correctly working hangperson game should look like. Study these examples closely! Your code must exhibit the exact same logic, though you are free to revise the particular language used in the prompts to the user.

- Invalid input: http://cs.colgate.edu/cosc101/a/hw/web/invalid_input.html
- Playing again: http://cs.colgate.edu/cosc101/a/hw/web/playing_again.html
- Losing a game: http://cs.colgate.edu/cosc101/a/hw/web/too_many_guesses.html

Tips and suggestions

One of the major challenges for this homework is to develop a well-designed program, where the problem is broken down appropriately into functions. As you consider the design of your program, think about the major tasks involved in game play and how you might break these tasks into separate functions. It helps to create a diagram that charts out the various subtasks to handle in the main part of game play (for example, get the next letter guess, check whether that letter has already been guessed, update the "guess state", etc.) Some of these subtasks are probably good candidates for putting in separate functions (for example, getting valid input from the user is a great candidate for a separate function). Thinking about how to develop a good design as you work through writing the program is the first key challenge for this homework. A "working" program is not sufficient for full credit for this homework: it also needs to be well designed. You should also look closely at the examples of game-playing programs that were presented in class.

The second key challenge for this problem is to decide how to manage the *guessing state* and to keep track of letters that have been guessed. Think about how you want to store this state and what kinds of things you will have to know each time a letter is guessed and each time you need to print out what letters have been correctly guessed in the secret word. Specific suggestions about keeping track of the game state:

- You need to keep track of the unique letters guessed by a player. (First, make life easier and convert everything to upper case.) You could store these previous guesses in a string or in a list, using concatenation in either case. To test whether a letter has been previously guessed, you could use the `in` relational operator.

- You also need to keep track of the number of wrong guesses in order to draw the gallows correctly. A simple integer counter will work.
- You need to keep track of the word a user is trying to guess and which letters have been correctly guessed. A list could work well here. Consider keeping a list that is the same length as the word the player is trying to guess. Let's call it `guess_state`. Initially, you might store a series of strings like `__` (two underscores) in each element of the `guess_state` list. That way, you can print out the guess state of the word quite easily by iterating through the list and printing each element. For example, if a player is trying to guess the letters of a 5 letter word, at the beginning of the game the guess state revealed to the player should be:

  ```
  __ __ __ __ __
  ```

  Conveniently, this is exactly the initial contents of the `guess_state` list.

  When a player makes a new guess, you could iterate through the word they're trying to guess and test for each letter in the word whether their guess is equal to that letter. For every match, you could then update the corresponding location in the `guess_state` list with the actual letter that belongs there. Thus, you would still be able to walk through the `guess_state` list and print out what the player should see before their next guess.

One final suggestion: as you're starting to write the program, you might want to print out the secret word before the game begins so that you know exactly what should happen with each guessed letter. That way, you'll be able to test when your program is behaving correctly (e.g., revealing letters that should be revealed) or not.

Grading

Your assignment will be graded on two criteria:

1. Correctness (75%): your hangperson game must be implemented correctly. Refer to the example output above to help ensure that your program works the way it should. Don't forget to implement the `HARD` version.
2. Program design and style (25%): program design is particularly important for this assignment. Organize your functions so that each one is relatively short does *one* thing and takes few arguments. Think about how to break down the problem of "playing hangperson" into smaller subtasks. And as usual:

   - Functions should have meaningful docstrings
   - Variable names should be meaningful
   - Programs should contain at least a few descriptive comments. Do *not* comment every line of code with low level explanations of what each line does. Focus on high level ideas.
   - All code should be structured so that the logic is clear and easy to follow.

Submission

Just upload your `hw5.py` file to Moodle.

Challenge problem: Evil hangperson

If you complete the hangperson game as described, you can try to complete this optional challenge problem. Modify the hangperson to have an "evil" option. Prompt the user to play evil or regular hangperson. If the user chooses to play evil hangperson, then the computer does something really sneaky: it avoids committing to a secret word for as long as possible.

Here's how it might work. The computer has a list of words to choose from. Each time the user guesses a letter, the computer can tell the user that the letter is not in the word, provided that the computer still has some words left in its list that don't have that letter. If eventually every word in the evil computer player's list has the guessed letter, then the computer can finally commit to a word (by choosing one at random). To get full credit, your evil player can cheat in this way but must ultimately still play a game that appears to be fair. For instance, at the end of the game, win or lose, the computer player must expose its secret word. So, if the player guesses 'a' then 'e' then 'o' then 'u' then 'i' then 'y' and the computer says that none of those letters are in the word, then the computer player is in trouble unless it can come up with a word that has no vowels in it!