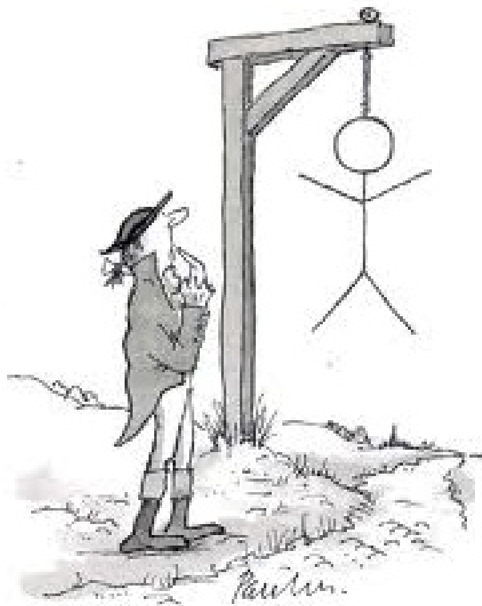
CS51A - Assignment 4
Movie Hangman

Due: Thursday 2/21 at 11:59pm

For this assignment, you will be implementing a text-based version of hangman. There are many variants of the program, so follow the specifications closely.

You may (and I would encourage you to) work with a partner on this assignment. If you do, you must both be there when either of you are working on the project and you should only be coding on one computer (i.e. pair programming). If you would like a partner, but don't have one, talk to Prof Papoutsaki or Dr. Dave in lab and we can try and pair people up.

Make sure that you read through this whole handout first before getting started. I've given an overview of how to proceed and then some hints near the end.



1 Movies Database

For this assignment, we are again going to be utilizing a dataset stored as a list of tuples. This time, the collection is a list of movies along with information about those movies.

To get access to the movies, do the following (which should feel similar to the last assignment!):

- Create a directory called `assignment4` somewhere on your computer.
- Download and the following file *into your assignment4* directory and unzip it by double-clicking on it:

  `http://www.cs.pomona.edu/~dkauchak/classes/cs51a/assignments/assign4-starter.zip`
- Delete the zip file. You should now have a folder called `assignment4` with two files (`movies.py` and `movie_info.csv`).
- Open up Wing and create a new file. Save this file as `assign4.py` *in your assignment4 directory*. Add your name and the assignment number in comments to the top of this file.
- Finally, access the movies: Import the helper functions for this assignment by adding the following into your file below the comments:

  ```
  from movies import *
  ```

  Run your program by clicking the green arrow and then test out to make sure everything loaded correctly. Like last time, there are two functions, `get_practice_movies`, which contains a small, practice set of data for debugging and `get_movies`, which contains the full database of movies.

  Like our quotes data, the movie database is a *list* of *tuples*. In this case, it's a list of triplets: movie title, movie description, and movie release year.

  For example, you can look at the entries in the practice data:

  ```
  >>> practice = get_practice_movies()
  >>> practice[0]
  ('jaws', 'A big shark eats stuff', 1975)
  ```

  Load the real data into a variable and take a look at a few entries to make sure you understand what the data looks like and how to access it.

2 Playing the game

When you "run" the program (press the green button in Wing), the game should immediately start. The program will randomly pick a movie from our movie database and then give the user some information about the movie to help them guess:

```
*** Movie Hangman ***
Year: 1997
A deranged media mogul is staging international
incidents to pit the world's superpowers
against each other. Now 007 must take on
this evil mastermind in an adrenaline-charged
battle to end his reign of terror and prevent
global pandemonium.
```

Specifically, the year the movie was released and a short description.

After this has been displayed, the program should then display the general hangman output:

```
-------------------
Guessed letters:
Movie: -------- ----- ----

Guess a letter:
```

This display has two main components:

1. `Guessed letters:` Shows the letters the player has guessed so far. The letters will be separated by a space and should be *all capitalized.* When the game starts, no letters have been guessed, so this area is blank. The letters should be displayed in alphabetical order.
2. `Movie:` This shows the movie that the player is trying to guess. Each dash represents a letter in the movie (we'll use dashes not underscore since it allows you to see the separate characters). The spaces between words are shown to help the person playing figure out how many words are in the movie. The movie above is three words, with the first with eight letters, the second with five letters, etc. As the user plays, the correctly guessed letters will be filled in.

The program runs, by continually prompting the user for the next letter. When the user enters a letter, there are just two possible cases:

- If the letter does not occur in the movie to be guessed, then that letter should be added to the list of guessed letters and then the display is shown again, prompting the user for another letter.

- If the letter does occur in the movie, then that letter should be added to the list of guessed letters *and* the underscored word is updated with all occurrences of that letter in the appropriate place. Finally, the user is then prompted for another letter.

The following transcript shows these different situations happening as the game is being played:

```
-------------------
Guessed letters:
Movie: -------- ----- ----

Guess a letter: e

-------------------
Guessed letters: E
Movie: -------- -E-E- --E-

Guess a letter: s

-------------------
Guessed letters: E S
Movie: -------- -E-E- --ES

Guess a letter: a

-------------------
Guessed letters: A E S
Movie: -------- -E-E- --ES

Guess a letter:
```

In the first two guesses the letter was in the movie, but in the third it was not.

The game ends when the user fills in all of the dashes in the word. The movie is displayed along with the number of guesses that it took the user. For example, here are the last two guesses and the winning output:

```
-------------------
Guessed letters: A E I M N O R S T V
Movie: TOMORRO- NEVER -IES

Guess a letter: w
-------------------
Guessed letters: A E I M N O R S T V W
Movie: TOMORROW NEVER -IES
```

```
Guess a letter: d
-------------------
You win!
The movie was: tomorrow never dies
You guessed it with 12 guesses
```

3 High-level Implementation

Your program will be keeping track of two key pieces of information.

First, the dashed version of the movie that is partially filled in. For this program, we're going to represent this as a list of strings. For example, in the transcript shown above, the list would start out as:

```
['-', '-', '-', '-', '-', '-', '-', '-', ' ', '-', '-', '-', '-', '-', ' ', '-', '-', '-', '-']
```

corresponding to each letter (including spaces) in the movie *TOMORROW NEVER DIES*.

As the user guesses letters correctly, this will be updated. For example, after the user guesses 'E' and ''A' this list would then be:

```
['-', '-', '-', '-', '-', '-', '-', '-', ' ', '-', 'e', '-', 'e', '-', ' ', '-', '-', 'e', '-']
```

which we see displayed as:

```
Movie: -------- -E-E- --E-
```

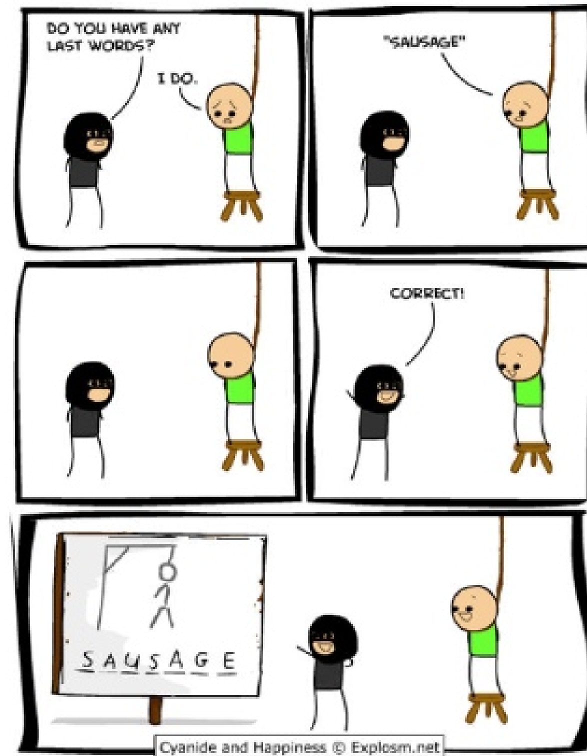
The **second** piece of information that you'll need to keep track of is the list of letters that have been guessed so far. Again, we'll just use a list of strings to keep track of this.

The program then revolves around:

1. Display the current state of the game to the user and get the next letter.
2. Check to see if the letter is in the word. If it is, update the list with the dashed representation.
3. Repeat.

Make sure you understand the basic setup of how we're going to tackle this problem!

4 Implementation requirements



http://www.cravingtech.com/best-cyanide-and-happiness-comic-strips.html

Besides following the game specification described above, your program **MUST** meet the following requirements:

- Your program must contain at least the following four functions:
  - `generate_underscore`: which takes the movie (as a string) as a parameter and returns a list representing the underscored version of the word where each character in the movie is represented as a dash ('-'), except for spaces, which are kept as spaces.

    ```
    >>> generate_underscore("monsters inc")
    ['-', '-', '-', '-', '-', '-', '-', '-', ' ', '-', '-', '-']
    ```

  - `list_to_string`: which takes a list of strings as a parameter and returns the letters in the list as a single string, with each letter capitalized and separated by a space.

    ```
    >>> monster = generate_underscore("monsters inc")
    >>> list_to_string(monster)
    '- - - - - - - -   - - -'
    ```

  - `insert_letter`: which takes three parameters: a letter (as a string), a list of strings representing the current dashed version that the user has filled in so far and, the movie

that the user is trying to guess (as a string). This function should update the current dashed version of the movie assuming the user guessed the letter AND assuming that the letter is in the movie. This function can either be a mutator method and update the dashed list (and then not return anything) or it can return a new list.

Here would be the output of the function if it returned a new list:

```
>>> insert_letter("k", ['-', '-', 'n', 'g', ' ', '-', '-', 'n', 'g'], "king kong")
['k', '-', 'n', 'g', ' ', 'k', '-', 'n', 'g']
```

Hint: Notice that there is a direct, index-wise correspondence between the dashed version of the word and the word itself, e.g. for the example above:

```
['-', '-', 'n', 'g', ' ', '-', '-', 'n', 'g']
  0    1    2    3    4    5    6    7    8
' k    i    n    g         k    o    n    g'
```

(where I've added spaces in "king kong" to make it line up). Your implementation should take advantage of this correspondence.

  – `play_hangman`: which takes no parameters. This function should initiate the hangman game and continue to run until the game finishes. You should include a single call to this at the very end of your file (outside of any functions). This will cause the hangman game to start immediately when the program is run.

- Your program should properly use the above four functions.
- Your program should immediately start playing the game when your program is run.
- Your program should be able to handle upper-case or lower-case letters as input.
- The guessed letters shows should all be capitalized and shown in alphabetical (*hint:*, sorted) order.

5 One path to implementation

There are many different ways to implement your program, but here is one suggestion:

- Write the `generate_underscore` function. The only tricky thing here is that you need to do something different depending on whether the letter is a space or not. Remember, we can iterate over the characters in a string using a `for` loop.
- Write the `list_to_string` function. This should be very similar to list functions you've written previously (or that we've looked at in class). There is a string method called "upper()" that might be useful.
- Write the `insert_letter` function. There are many ways to do this, so think about how you'd like to approach it. The key observation here is that the movie to be guessed and the list representing the current dashed version of the movie are *of the same length* and there is a direct correspondence between the two, as noted above.

- Write `play_hangman` incrementally, testing each step before moving on to the next:
  - Start by having it pick a random movie entry and then display the corresponding information. There is a function called `choice` in the `random` module that might be useful, though there are many ways to do this. Also, there is a method of the string class called `title` that will help your output look nicer :) Note, when you're developing, feel free to print out other information (like the word you're supposed to be guessing) to help you debug your program. Just make sure to remove this information before you hand it in.
  - Add the user input and add the letter that the user guessed into the underscored word appropriately (your functions from above should be useful).
  - Add in functionality to keep track of the letters the user has guessed and to print out the letters guessed in sorted order.
  - Add the functionality to make sure that the user continues to be prompted to enter a new letter as long as the letter input has already been guessed.
  - End the game when the user gets the right answer.
  - Print out the appropriate information for after the user finishes the game.

Here are some things to think about:

- How do the four functions fit into the program? Each of them should play a role in your program and will make your life easier and the program simpler.
- If you're confused about the behavior of your program, put in more print statements to print out variables, etc. For example, it can be very useful to print out the list holding the underscored word as you play the game so you can see what the state looks like. Additionally, you can cheat and print out what movie you're trying to guess so that you don't actually have to play the game when you're debugging. Just make sure to remove all of the extraneous print statements when you're all done.
- This program should not be a lot of code, but some of it can be tricky. Make sure you think about how you want to accomplish each step. Sometimes it helps to write down what you want to do in English (i.e., pseudocode), focusing on the logic, and then translate this into code.

6 Extra credit

You may add any of the following additions to the program for extra credit. Make sure to put in comments at the top of the file indicating any extra credit that you implement.

- **[0.5 points]** Currently, if the user guesses the same letter twice it will appear in the guessed letters twice. Make it so that if the letter has been guessed before, you 1) tell the user this and then 2) don't add it to the guessed letters.

- **[0.5 point]** Keep track of how many letters the player guess that were *NOT* in the movie. Display this as an additional stat when the game finishes.
- **[1 point]** Right now, the user can guess as many times as they want and will therefore always win eventually. Add a feature where you limit the number of wrong guesses that the player can make. Display these as they play, either as a number or some other visual representation (e.g., asterisks for each remaining guess). When the user runs out of guesses, the game should end with a *different* message.

7 When you're done

Make sure that your program is properly commented:

- You should have comments at the very beginning of the file stating your name, course, assignment number and the date.
- Each function should have an appropriate docstring.
- Include other miscellaneous comments to make things clear.

In addition, make sure that you've used good *style*. This includes:

- Following naming conventions, e.g. all variables and functions should be lowercase.
- Using good variable names.
- Proper use of whitespace, including indenting and use of blank lines to separate chunks of code that belong together.
- Make sure that none of the lines are too long, i.e. cross the red line in Wing.

Submit your `assign4.py` file online using the courses submission mechanism. *If you worked with a partner*, only one person should submit the assignment. Make sure that both of your names are at the top of your .py file and when submitting, select your partner during the submission process.

Grading

	points
each time a random movie is picked from the file	1
movie information is displayed appropriately	2
initial underscored word shown correctly	2
each guess is inserted appropriately in underscored movie	3
guessed letters correctly displayed	2
guess letters are updated appropriately	1
game ends correctly when movie is guessed	2
prints out game winning information appropriately	1
handles lower and uppercase letters	1
program starts automatically on run	1
followed function specifications	4
Comments, style	3
extra credit	2
total	23 (+2)

http://xkcd.com/804/
(All the other hangman comics seemed a bit too morbid...)