CS5001
Spring 2019
Homework 5
Assigned: February 14, 2019
Deadline: February 21, 2019 at 9:00am

To submit your solution, compress all files together into one .zip file. Login to `handins.ccs.neu.edu` with your CCIS account, click on the appropriate assignment, and upload that single zip file. You may submit multiple times right up until the deadline; we will grade only the most recent submission.

Your solution will be evaluated according to our CS5001 grading rubric. This is a programming-only assignment! Part #1 counts for 35% of your HW5 score, and part #2 counts for the remaining 65%.

If you apply your late token, your new deadline is **February 25, 2019 at 9:00am**. Email laneys@northeastern.edu to notify us you'll be cashing it in.

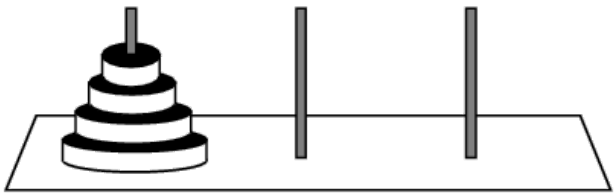
Programming #1

- Filename: **`hanoi.py`**
- Starter code (don't modify this part; we'll test using the same file): **`hanoi_viz.py`**

Once upon a time, a demon named Glory wanted to bring about the end of the world. Spike, who doesn't necessarily want an apocalypse but enjoys mischief, gives Glory a puzzle to solve that will end the world if she does it correctly.

Spike's puzzle: There is a tower (Tower A) with several disks stacked on it, and Glory has to move them to a second tower (Tower B). She has to follow the following rules:

1. Only one disk may be moved at a time.
2. A larger disk may never be placed on a smaller disk.
3. She can use a third tower, Tower C, to temporarily hold some disks.



When Glory completes the puzzle (successfully moves all the disks from A to B), the world ends. There is nothing Buffy can do to stop her, and we're all doomed.

For this assignment, you'll help Glory in her apocalyptic quest. You'll write:

- A driver, which prompts the user for the number of disks and kicks off the process.

- A recursive function to move disks from one tower to another.

We've provided starter code, hanoi_viz.py, to do a little ascii illustration of the towers. You'll need to call functions defined in this module for the assignment.

Requirements. Your program must:

- Prompt the user to give you the number of disks. Prompt them repeatedly until they give you a valid number.
    - Use a conditional to make sure that the user has provided an integer, not a float or a string or anything else.
    - The integer they give you must be between 1 and 8 (inclusive).
- Choose names for the towers. These can be anything you like, but keep them short or the illustrations won't look right.
- Call the given `initialize_towers` function before calling your own, recursive function.
- Define a recursive function that moves disks from one tower to another. **<==== this is really the heart of the assignment.**
    - This function's name is **`move_tower`** (exact name required; this'll help us test it).
    - In addition to calling itself, this function should call the `move_disk` function in the `hanoi_viz` module.

Helpful hints:

- Even though I obviously put my own fangirl spin on this story, it's still the legend of the Towers of Hanoi, a famous computer science problem. Read up on it before you start!
- You don't need a lot of code to make your recursive function work. Put your recursive hat on: To move *n* disks, you need to recursively move *n-1* disks and then move that last disk.
- Your recursive function will need to call itself more than once in the recursive case.
- The `hanoi_viz` functions work with a dictionary named towers, but once it's initialized, you don't need to modify it.
- Take a look at the isdigit function for making sure the user has provided an integer.

How do you get the AMAZING points for this program? (In addition to meeting the specs, writing good comments, and having good variable names, function structure, etc.):

- Apart from comments, write your `move_tower` function in 5 lines or fewer.
- Use constants instead of literals for the names of the towers, max height, and min height.
- You can be creative with tower names, but make them short so that the illustrations look nice.

Example of running the program (one disk moves at a time; we want to get both disks from A to B, and we use C as a helper).

```
What is the number of disks (2-5)?
2.5
Oops, that wasn't a number, please try again!
What is the number of disks (2-5)?
two
Oops, that wasn't a number, please try again!
What is the number of disks (2-5)?
2
 A       B       C
 X
XXX


Moving disk from A to C
 A       B       C

XXX              X


Moving disk from A to B
 A       B       C

        XXX      X


Moving disk from C to B
 A       B       C
         X
        XXX


>>>
```

Programming #2

- Starter code:
  - `wordlist.txt` (download this into the same directory as your .py files, but don't submit it with your solution)
  - `wordlist.py` (to process the wordlist)
  - `test_scrabble.py` (to test the functions you're required to write as part of hw5)
- Your files:
  - `wordgame.py` (the driver)
  - `scrabble_points.py` (functions)

This is a word game that's sort of a cross between Scrabble and Countdown. It's a one-player game. The player attempts to create words from 7 randomly-chosen letters, and they are awarded points according to the value of each letter.

Points	**Letters**
1	A, E, I, O, U, L, N, S, T, R
2	D, G
3	B, C, M, P
4	F, H, V, W, Y
5	K
8	J, X
10	Q, Z

Seven letters are in play at a time, and they're chosen at random from a big bag of 100 letters. Most letters have more than one instance in the bag, following these frequencies:

Frequency	**Letters**
1	J, K, Q, X, Z
2	B, C, F, H, M, P, V, W, Y, blank
3	G
4	D, L, S, U
6	N, R, T
8	O
9	A, I
12	E

Play proceeds as follows:

- We begin with a bag of letters, following the table above -- there are 12 “E” letters in the bag, 9 “A” letters, 9 “I” letters, and so on.
- The user is repeatedly presented with a menu of options:
  - `D` - Draw 7 letters from the bag
  - `W` - Make a word from the letters in play
  - `P` - Print all words played so far
  - `Q` - Quit
- The game is over when the player enters Q to quit, or there are no more letters in the bag or in play.

Player Chooses D to Draw

Every time the user chooses D to draw, 7 letters are removed at random from the bag and put into play.

When the user chooses D, if any letters are already in play, they are discarded. They don’t go back into the bag; they’re just gone.

Player Chooses W to Make a Word from the Letters in Play

Every time the user chooses W to make a word, they are prompted to enter a word that can be formed with the 7 letters currently in play.

- If they enter a word that is not in the wordlist, it is invalid and they get no points. The letters in play remain the same.
- If they enter a word that cannot be made from the letters currently in play, it is invalid and they get no points. The letters in play remain the same.
- If they enter a word that they’ve played it already, it is invalid and they get no points. The letters in play remain the same.
- If they enter a valid word, they are awarded points for it based on the letters used. The letters used for the word are no longer in play and replaced with new letters.

For a valid word, count up the total points based on the letter values described above. Add the word’s value to the player’s overall points.

Player Chooses P to Print

Every time the user chooses P to print, they see all the valid words they’ve played in the game so far, along with the points awarded for each one, something like this:

```
You have a total of 12 points so far.
HEN -- 6 points
BLUE -- 6 points
```

Player Chooses Q to Quit

The game is over when the user enters Q to quit, or there are no more letters in the bag or in play.

Note that the D-draw option discards all the letters currently in play and replaces them with 7 new ones. But when the user plays a valid word, only the letters used in the word are replaced. The letters-in-play get kind of "refreshed", if you will, but not completely replaced unless the word actually used all 7 letters.

Starter Code

- We've written code that reads from the file **`wordlist.txt`**, and returns a list of strings to represent the dictionary. The wordlist.txt file, which you need to download into the same directory as your Python files, is the built-in Unix wordlist. We do some clean-up, such as converting the words to all uppercase and removing spaces. There are still some words with punctuation, and some words I'm surprised are words ('snee'?) but we'll consider it a valid wordlist and let those eccentricities be Alex Trebek's problem.
- We've also provided a test suite to test two functions that you are required to write. Beyond these, any functions you choose to define are up to you as long as they meet our usual standards of "good" functions. You must write functions that pass the tests included for:
    - **`bag_of_letters`**
        - Params: dictionary where *key* = letter and *value* = frequency of that letter.
        - Returns: a list of letters, with each one repeated according to its frequency.
        - Example Input: **`{'A':1, 'B':2}`**. Expected output: **`['A', 'B', 'B']`**
    - **`get_word_value`**.
        - Params: a word to evaluate (string), and a dictionary where *key* = letter and *value* = points that letter is worth.
        - Returns: the total point value of the word (an int). If any letter in the word does not appear in the letter-value mapping, then return 0 points.
        - Example Inputs: **`'HI', {'H':4, 'I':1}`**. Expected output: **`5`**
        - Example Inputs: **`'HELLO', {'H':4, 'I':1}`**. Expected output: **`0`**

Requirements:

- Write the two functions as described above such that all the tests in our provided test suite pass.
- For additional functions you write, you are not required to submit a test suite, but you need to be confident that they work for any reasonable input.
- Point values and letter frequencies must be allocated using the distributions above.
- Allow the user to enter upper or lowercase letters, both for their choice from the menu and the words they want to play in the game.
- If the user enters an invalid menu option, repeatedly re-prompt them until they enter a good one.
- When the user plays a valid word, they are awarded points, which is added to their total.
- When the user attempts to play an **invalid** word (not in the dictionary from wordlist.txt, not possible from the letters in play, and/or they already got points for the word), they get no points and the letters in play stay the same.
- If the user chooses D for draw, the 7 letters currently in play are discarded and 7 new letters are drawn.
- If a blank tile is in play, it can take the place of any single letter.
- Unlike in scrabble, we count the blank tiles as whatever the letters are worth.

    - For example, suppose the letters in play are `'A', 'E', 'C', '_', 'W', 'Z',` and `'L'`. The user can play the word **`CAKE`**, using the blank tile as a K. The total points for this point are: 3(C) + 1(A) + 5(K) + 1(E) = 10.
    - In "real" scrabble, the blank would count as zero points. We're nicer.
    - The user could *not*, however, make the word **`CABLE`**, because the blank tile can count for one letter but not two.
- The game ends when the user enters Q or there are no more letters in the bag or in play.

Example of game play (as always, your wording can vary as long as your program is friendly and informative!). We thought there would be too many screenshots to see it working, so instead we made a silly little video during Snowmageddon: https://youtu.be/fE6HwVy8HTE

AMAZING points:

- Write a really short main. 20 lines, tops, not including comments.
- Use constants for any variable whose value doesn't change once it's defined.
- Write a test suite for every function you add. Maybe there's one function that you don't test because it prompts the user, but it's really tiny. Everything else is tested, and you and I can both be confident that it works great.