CSCI 1300 Introduction to Computer Programming
Instructor: Knox
Assignment 9: Re-code in Python
Due Friday, December 1st by 6 pm for bonus
Due Sunday, December 3rd by 6 pm

This is your first assignment in Python. Working with scripting languages is enjoyable and intuitive, thus use this Assignment to develop a good styling code and have fun. This assignment requires you to write one **assignment9.py** file, make sure to write all the functions and any supporting functions in the same file. Please make sure to put your testing code within a main function or within the ""if __name__ ==__main__:" conditional.

1. Once you have your code running on your virtual machine (VM), submit a zip containing *assignment9.py* to the autograder COG .
   https://web-cog-csci1300.cs.colorado.edu

2. You **must also** submit your code to Moodle to get full credit for the assignment.
   - 10 points: Comments at the top of your source files should include your name, recitation TA, and the assignment and problem number.
   - 10 points: Algorithm descriptions comments in your code submission to describe what your code is doing.
   - 10 points: Code in your **`main()`** function that is testing the code in **`assignment9.py`.** 10 points will be deducted if the main function in the file you submit to moodle does not have code testing the functions you have written in **`assignment9.py`**. Remember that COG *does not* run your `main()`, it runs its own main. (Code you write in `main()` will not impact your COG runs).

Assignment 9 Details: We looooooove Python

The goal of this assignment is to translate the work done in previous Assignments to Python and learn to deal with interpreted languages.

Part 1:
In the first part you have to implement two functions:
Note: You may use the logic to implement these functions from your Assignment 2.

1) The first function will compute the U.S. census which provides information about the current U.S. population as well as approximate rates of change.
compute_census(...)
This function calculates the U.S. population in exactly one year (365 days) given three rates and the current US population. compute_census(...) takes 2 parameters: list_of_rates in seconds and current_population, and **returns the projected population in the next year.**

Three rates of change are provided:

- There is a birth every 8 seconds
- There is a death every 12 seconds
- There is a new immigrant every 33 seconds

For example: Making a function call to compute_census()

```
birth_rate = 8
death_rate = 12
new_immigrants_rate = 33
List_of_rates=[birth_rate,death_rate,new_immigrants_rate
]
Curr_population = 1000000

print compute_census(List_of_rates, Curr_population)
>> 3269636
```

2) The second function will convert the given seconds into the format dd:hh:mm:ss.

convert_seconds()
This function does not take any parameters, however it will read input from the user for the seconds. Do not prompt the user for this value, just read from input. Use that number and convert it into days, hours, minutes, seconds. The function should print out the following string concatenated with the computed values:

<user_input> corresponds to: <days> days, <hours> hours, <minutes> minutes, <seconds> seconds.

For Example: Making a function call to convert_seconds()

```
>> Enter the number of seconds you want to convert? <user_input>
>> <user_input> corresponds to: <days> days, <hours> hours, <minutes>
minutes, <seconds> seconds.
```

Part 2:

In this part you will implement a function that generates a text given user inputs.

generate_story(...)

The function takes a list of strings as parameters and returns a generated story by concatenating the list of strings and the user's input for the corresponding prompts.The format of the list that you pass in to your function is the following:
list_to_story = [str1, prompt1, str2, prompt2, str3]

Your list will always start with a story and then followed by prompts and stories in the alternate format as shown above. Your list must atleast have one story and prompt i.e. list_to_story = [str1, prompt1]
Here str1, str2, str3 are sentences that compose the story and prompt1, prompt2 are the questions you should ask to the user in your function. The answers provided by the user must be recorded and stored in your function. Finally you will return a string that is formed after concatenating str1, str2, str3 and prompt1_ans, prompt2_ans in the correct order.

For Example: Making a function call to generate_story(list_to_story)

```
str1 = 'I went skiing to'
prompt1 = 'Enter a location: '
str2 = 'it was really crowded and I stayed in line for'
prompt1 = 'Enter number of hours: '
str3 = 'hours'
list_to_story = [str1, prompt1, str2, prompt2, str3]

print(generate_story(list_to_story))
>> Enter a location: Copper
>> Enter number of hours: 10
>> I went skiing to Copper it was really crowded and I stayed in line for 10 hours
```

Part 3:

In this part you have to implement the similarity score and best match functions from Assignment 4.

1) **similarity_score(...)**

   This function should take two string parameters: seq1 and seq2, and compute the similarity score by using the hamming distance.

   Recall:

   1) *the hamming distance is the number of mismatches between the characters at index i of two strings.*
   2) *The similarity score is computed by using the following formula:* $\frac{lenOfseq - hammingDistance}{lenOfseq}$
   3) *The function should only calculate the similarity score if the two strings are of the same length, otherwise return a 0.*

```
sequence1 = CCGCCGCCGA
            ||*||*||*|
sequence2 = CCTCCTCCTA

Matches = 7, Mismatches = 3
Hamming distance = 3, Length = 10
Similarity = (10 - 3) / 10 = 0.7

>> similarity_score("CCGCCGCCGA", "CCTCCTCCTA");
>> 0.7
```

2) **best_match(...)**

   This function should take two string parameters, a genome and a sequence. The function will return the index i at which we find the best similarity score. To find the best match you will want to compare consecutive subsets of the genome against the sequence. Use similarity_score(...) to implement best_match(...)

Part 4

In this part you will implement a function that computes the average and the median of a given list of values.

calc_stats(...)

This function should take a list of values and compute their average and median. You should return the two values in the form of a list, i.e. stats = [average, median].

```
List_of_numbers = [1, 2.5, 3, 8, 10.5, 5]

>> calc_stats(list_of_numbers)
>> [5.0, 4.0]
```

Part 5

Your task for this part is to read the values off of a file and generate a nested list structure. Each element of the list being returned is itself a list. The second level list consists of a name and a list of ratings.

parse_ratings(...)

This function takes a parameter, file_name, and returns a list of the parsed lines from the file. Your function should read each line of the file, extract the data, and store the usernames and book ratings in the nested list structure you are returning. The file that you have to read the data from holds usernames and book ratings; it has the following format:

David, 1 2 3 0 5
Camilla, 5 3 2 0 0
Monika, 0 0 3
. . .

Notes:

1) User names and book ratings are separated by ‘,’
2) Book ratings are separated by a space.
3) The number of ratings can vary from user to user
4) A rating spans the values 0-5

Each list in the list of lists has a user name and a list of ratings. The following should give you a better idea of the structure of the list of lists following the example above:

```
user1 = 'David'
list_of_rating_user1 = [1, 2, 3, 0, 5]
user2 = 'Camilla'
list_of_rating_user2 = [5, 3, 2, 0, 0]
user3 = 'Monika'
list_of_rating_user3 = [0, 0, 3]

>> parse_ratings(fileName)
>> [
        ['David', [1, 2, 3, 0, 5]],
        ['Camilla', [5, 3, 2, 0, 0]],
        ['Monika',  [0, 0, 3]]
   ]
```