Homework 2

MPCS 51042 - Python Programming - Fall 2021

Deadline: October 12 at 5:00 pm CDT

Write each problem's solutions in a separate source file: `problem1.py` and `problem2.py`. Upload the 2 source files to Gradescope before the deadline.

- The source files and functions **must** be named as specified.
- You may define additional functions, if your solution requires them.
- You **must** comment each function definition with the following info. Place each function's comments immediately below its `def` statement.
  - The purpose of the function
  - The meaning of the function's parameters, if any
  - The meaning of the function's return value, if any

Problem 1

This problem has many opportunities to use functional programming. You will not be graded on your usage of functional programming, but you are encouraged to practice.

Background

The video game *Super Mario Party* has a game board where players roll a 6-sided dice and move the resultant number of spaces. Each character can use either a standard 6-sided dice (with numbers 1 through 6) or a character-specific 6-sided dice. For example, Mario's dice has the numbers 1, 3, 3, 3, 5, and 6; and Luigi's dice has the numbers 1, 1, 1, 5, 6, and 7.

The following Python `dict` describes all the characters' dice, in addition to a standard dice.[1].

```
dice = {
    'Boo' : (0, 0, 5, 5, 7, 7),
    'Bowser' : (0, 0, 1, 8, 9, 10),
    'BowserJr' : (1, 1, 1, 4, 4, 9),
    'Daisy' : (3, 3, 3, 3, 4, 4),
    'DiddyKong' : (0, 0, 0, 7, 7, 7),
    'DonkeyKong' : (0, 0, 0, 0, 10, 10),
    'DryBones' : (1, 1, 1, 6, 6, 6),
    'Goomba' : (0, 0, 3, 4, 5, 6),
    'HammerBro' : (0, 1, 1, 5, 5, 5),
    'Koopa' : (1, 1, 2, 3, 3, 10),
    'Luigi' : (1, 1, 1, 5, 6, 7),
    'Mario' : (1, 3, 3, 3, 5, 6),
    'MontyMole' : (0, 2, 3, 4, 5, 6),
    'Peach' : (0, 2, 4, 4, 4, 6),
    'PomPom' : (0, 3, 3, 3, 3, 8),
    'Rosalina' : (0, 0, 2, 3, 4, 8),
    'ShyGuy' : (0, 4, 4, 4, 4, 4),
    'Standard' : (1, 2, 3, 4, 5, 6),
    'Waluigi' : (0, 1, 3, 5, 5, 7),
    'Wario' : (6, 6, 6, 6, 0, 0),
    'Yoshi' : (0, 1, 3, 3, 5, 7)
}
```

Part A: Mean and Standard Deviation from Stdlib

Write a function, `mean_stddev_stdlib` that:

- Takes a `dict` as an argument. Assume it is formatted like `dice`, above.
- Returns a new `dict`. Each key is the name of a character. Each value is a sequence of 2 floats: the population mean and population standard deviation of the character's dice roll.
- The sequence can be either a `list` or a `tuple`, your choice.
- Use the `mean` and `pstdev` functions from the standard library `statistics` module.

For example, the first entries of the result should be:

```
{'Boo': [4, 2.943920288775949],
 'Bowser': [4.666666666666667, 4.384315479321969], ... }
```

[1]The value 0 represents either an actual 0 or a special event that does not move the character, such as gaining coins.

Part B: Mean and Standard Deviation from User Functions

Write a function, `mean_stddev_no_stdlib` that:

- Takes a `dict` as an argument. Assume it is formatted like `dice`, above.
- Returns a new `dict` that is formatted like the result of Part A.
- Do **not** use any built-in or standard library functions that directly return the mean and standard deviation. Other functions, such as `sum`, are fine. You may also define your own functions for mean and standard deviation, if you like.

The results should be the same as Part A (though you may see small numerical errors due to different implementations).

Part C: Sorted Results

Write a function `mean_stddev_sorted` that:

- Takes a `dict` as an argument. Assume it is formatted like `dice`, above.
- Returns a **sorted** sequence such that each element is a nested sequence of two items: character name and another nested sequence of 2 items. The items of the innermost nested sequence are mean and standard deviation.
- The outermost sequence must be **sorted** by mean, then standard deviation.
- Each nested sequence can be either a `list` or a `tuple`, your choice.

For example, the first few entries of the results are:

```
[('HammerBro', [2.8333333333333335, 2.1921577396609844]),
 ('Rosalina', [2.8333333333333335, 2.733536577809454]),
 ('Goomba', [3, 2.309401076758503]),
 ('Yoshi', [3.1666666666666665, 2.3392781412697]),
 ('Daisy', [3.3333333333333335, 0.4714045207910317]),
 ('ShyGuy', [3.3333333333333335, 1.4907119849998598]). ... ]
```

Part D: Filtered Results

Write a function `mean_stddev_filtered` that:

- Takes a `dict` as an argument. Assume it is formatted like `dice`, above.
- Returns a nested sequence formatted and sorted like the results of Part C.
- Return **only** the entries such that mean ≥ 3.5.

Considering this and/or other criteria, who would you choose to play *Super Mario Party*?

Problem 2

Nowadays, we take word completion for granted. Phones, word processors, and search engines suggest completions for words as we type. You will implement two functions that could be used for word completion. These will take advantage of dynamic programming techniques, though you don't need to know dynamic programming to do it!

The first function, `fill_completions`, constructs a completions dictionary to lookup possible word completions. This dictionary is indexed by a tuple of (letter, letter position). It is created from a vocabulary of words, which is provided in "articles.txt"

The second function, `find_completions`, returns the set of possible completions for a given prefix. It will use a completion dictionary and set operations to quickly retrieve the results.

Finally, you will implement a simple, interactive main program for running your functions.

Specifications

- `fill_completions(filename)` takes a filename (a string) as input. It uses the words from this file to create and return a "completion dictionary" (a dict) where:
  - Each key of the dict is a tuple of the form `(i, x)`, where `i` is a non-negative int and `x` is a lowercase letter. These represent a letter position and a letter in a given word.
  - For each key `(i, x)`, the associated value is the set of words that contain the letter `x` at position `i`. For example, suppose the file contains the word "Python" and `c_dict` is the completion dictionary. Then the sets `c_dict[0, "p"]`, `c_dict[1, "y"]`, `c_dict[2, "t"]`, `c_dict[3, "h"]`, `c_dict[4, "o"]`, and `c_dict[5, "n"]` will all contain the word `"python"`.

  When parsing words from the input file, follow these guidelines:
  - Assume words are separated by whitespace.
  - Keep only words for which `isalpha()` is true.
  - Convert words to lowercase before adding them to the completion dictionary.
- `find_completions(prefix, c_dict)` takes a prefix (a string) and a completion dictionary (a dict). It returns the set of words (a set) that complete the prefix, based on the provided dictionary. If there are no possible completions, the function returns an empty set.
- `main()`, the test driver:
  - Calls `fill_completions` to create a completions dictionary from articles.txt. This file contains the text of recent articles pulled from BBC.
  - Repeatedly prompts the user for a prefix to complete.
  - For each prefix, call `find_completions` to get the set of possible completions.
  - Print the **sorted** set of completions, with one word per line. If no completions are possible, print "No completions".
  - If the user enters "quit", end the program.
  - Note that this `main` function is not used in the automated unit tests, so exact formatting of the output is not a concern.

To run the `main()` function, be sure to put a block at the end of your script with the follow lines:

```
if __name__ == '__main__':
    main()
```