Yatzy Maxi Game

About Game

This is a game called "Yatzy" with the modification "Maxi". In this game you have to roll the dice and then choose the appropriate combinations. The game ends when there are no combinations left, you can read more about this game and its modifications here.

About project

The project includes all the features of a board game (except that you can not throw a die at a friend if he got you), namely the ability to reroll the dice, select the number of players (including the game with one player), as well as a table of player scores, which will be shown at the end of the game.

Conception of project

> Note: we expect to get an A grade, and we welcome your feedback on how we can improve the code and get us closer to that grade.

Since we were not able to use classes and objects, we had to use other methods.

Main problem: multiple checks (branching) Difficulties are caused by the fact that there are a total of 20 combinations in the game. First of all: to display active combinations to the user we need to make 20 if else expressions, which complicates the readability of the code significantly, but in addition, the combinations have a different algorithm for scoring points.

> For example, one of the combinations in the upper section, fours, adds up all the fours in the set of dice if there is at least one four, while the combination "Maxi Yatzi" gives exactly 100 points if all the dice have the same values, no matter what those values are.

This complicates the situation considerably, because in addition to 20 if else statements to check the user's choice, we need to do the same number of them for each combination. As a result, it takes 80 lines of code, and it does not simplify it in any way.

Besides, another problem was that each player should have their own combination table with values filled in. Moreover, each player's values are filled in depending on their choice, as well as on the dice they rolled (randomness), respectively, each player should have their own combination table. Classes and objects seem like ideal solutions here, but… We were forbidden to use them… Accordingly, we developed our own structure using the permitted tools. Dictionaries.

It all starts with one large dictionary `players`. In this dictionary there are several (depending on how many the player specifies) dictionaries called `player`, each player has a dictionary `combinations` and a key value pair `"Current Dices":[]` (which will later store specific dices rolled during the player's turn). In the dictionary "combinations" there are two other dictionaries: `upper_section` and `lower_section`, they contain all the combinations that are in the game. These combinations are also dictionaries that contain the points function (?), as well as the number of points that they currently contain (initially it is zero). Let's talk about the points function in more detail.

Each combination will have its own scoring function, it will return a unique number of points corresponding to the scoring rules for this combination if the combination is applicable to the current set of dice and 0 points if it is not applicable. This will solve two problems at once: branching and counting, since to check the applicability of a combination, you just need to enter the set of dice into the function, and if the function returns zero, make sure that the combination is not applicable, and in order to count the points, we just need to do it again. As a result, instead of 40 checks, we will only need to make a loop. However, now another problem appears.

Lots of functions We will need to create 20 functions, which significantly complicates the readability of the code. However, if you look at the combinations in the top section, you can see a pattern: all they do is find the number of dice with the value (1, 2, 3, 4, 5, 6), and then multiply this number by this very value. As a result, we will need to create 6 one-line functions, the algorithm of which differs by only one digit. Python has a tool that allows you to solve this problem - wrapper. Using this tool is not prohibited, since in essence it is a function, which in turn returns other functions. That is, wrapper is a function generator. Now we can create 6 functions with just one cycle, and then enter them into an array, which will significantly simplify the understanding of the code and facilitate the solution of the problem.

As a result, we implemented a system of classes and objects using only dictionaries, arrays and functions. Which in turn does not violate the condition of the task.

Documentation

The project will have the following functions:

- `get_dices()` - returns dice that will be generated randomly (with the ability to re-roll at the user's discretion)
- `generate_players(amount)` - creates players, returns the `players` dictionary (more details in the diagram above), where the number of players is specified by the user
- `get_all_combinations(player)` - gets all combinations template for more easier access
- `render_box(string)` - renders one string in the box (for better view)

- `render_scoresheet(players, player)` - renders all combinations of player
- `render_players_table(players)` - renders table with player name and player scores
- `safe_in_input()` - ensures that the code will not be broken if the user specifies an invalid value (for example, a string or a negative number), returns the number entered by the user

In addition to these functions, there are internal ones (for example, a function generator and functions for combinations), but they will not be used relative to the programmer who writes the code, so there is no point in including them in this documentation. Internal functions are used to implement and ensure the correct operation of the functions described above.

Note: you can find all sources in the `code` directory.