Final Project

CONTENTS

Executive Summary

In this project you **write**, **package**, and **publish** a Python module and accompanying files. The project will implement a simple **Monte Carlo simulator** using a set of related classes.

The project is designed to integrate what you have learned in this class by calling upon the following areas of knowledge:

- Basic syntax, expressions, and statements in Python
- Python Classes with initialization methods
- Data manipulation with Pandas
- Literate programming with `docstring` s and documentation
- Unit testing with Unittest
- Simple plotting with Pandas
- Program modularization and packaging

Requirements

You write **three classes** that will work in tandem to generate various outcomes.

- A **Die** class
- A **Game** class
- An **Analyzer** class

In addition, you will write **unit tests**, a **scenario** script, and **documentation** for these classes.

Note: You will put all three classes in a single file called `montecarlo.py` and the unit tests in a file called `montecarlo_tests.py`. The scenario script will be put into a Jupyter Notebook called `montecarlo_demo.ipynb`.

The **Die** class

General Definition

A die has N sides, or “faces”, and W weights, and can be rolled to select a face.

- W defaults to 1.0 for each face but can be changed after the object is created.
- Note that the weights are just numbers, not a normalized probability distribution.
- The die has one behavior, which is to be rolled one or more times.
- Note that what we are calling a “die” here can be any discrete random variable associated with a stochastic process, such as using a deck of cards or flipping a coin or speaking a language. Our probability model for such variable is, however, very simple – since our weights apply to only to single events, we are assuming that the events are independent. This makes sense for coin tosses but not for language use.

Specific Methods and Attributes

- An **initializer**
  - Takes **an array of faces** as an argument. The array's data type (`dtype`) may be strings or numbers.
  - Internally iInitializes the weights to `1.0` for each face.
  - Saves both faces and weights into a **private dataframe** that is to be shared by the other methods**.**
- A method to **change the weight** of a single side.
  - Takes two arguments: the face value to be changed and the new weight.
  - Checks to see if the face passed is valid; is it in the array of weights?
  - Checks to see if the weight is valid; is it a float? Can it be converted to one?
- A method to **roll** the die one or more times.
  - Takes a parameter of how many times the die is to be rolled; defaults to `1`.
  - This is essentially a random sample from the vector of faces according to the weights.
  - Returns a list of outcomes.
  - Does not store internally these results.
- A method to **show** the user the die’s **current set of faces and weights** (since the latter can be changed).
  - Returns the dataframe created in the initializer.

The **Game** class

General Definition

A game consists of rolling of one or more dice of the same kind one or more times**.**

- Each game is initialized with one or more of similarly defined dice (Die objects).
- By “same kind” and “similarly defined” we mean that each die in a given game has the same number of sides and associated faces, but each die object may have its own weights.
- The class has a behavior to play a game, i.e. to rolls all of the dice a given number of times.
- The class keeps the results of its most recent play.

Specific Methods and Attributes

- An **initializer**
  - Takes a single parameter, a list of already instantiated similar Die objects.
- A **play** method
  - Takes a parameter to specify how many times the dice should be rolled.
  - Saves the result of the play to a **private dataframe** of shape N rolls by M dice.
  - The private dataframe should have the roll number is a named index.
  - This results in a table of data with columns for roll number, the die number (its list index), and the face rolled in that instance.
- A method to **show** the user the results of the most recent play.
  - This method just passes the private dataframe to the user.

    - Takes a parameter to return the dataframe in narrow or wide form.
        - This parameter defaults to wide form.
        - This parameter should raise an exception of the user passes an invalid option.
    - The narrow form of the dataframe will have a two-column index with the roll number and the die number, and a column for the face rolled.
    - The wide form of the dataframe will a single column index with the roll number, and each die number as a column.

The **Analyzer** class

General Definition

An analyzer takes the results of a single game and computes various descriptive statistical properties about it. These properties results are available as attributes of an Analyzer object. Attributes (and associated methods) include:

- A **face counts per roll**, i.e. the number of times a given face appeared in each roll. For example, if a roll of five dice has all sixes, then the counts for this roll would be 6 for the face value '6' and 0 for the other faces.
- A **jackpot** count, i.e. how many times a roll resulted in all faces being the same, e.g. all one for a six-sided die.
- A **combo** count, i.e. how many *combination types* of faces were rolled and their counts.

Specific Methods and Attributes

- An **initializer**
    - Takes a game object as its input parameter.
    - At initialization time, it also infers the data type of the die faces used.
- A **jackpot** method to compute how many times the game resulted in all faces being identical.
    - Returns an integer for the number times to the user.
    - Stores the results as a dataframe of jackpot results in a public attribute.
    - The dataframe should have the roll number as a named index.
- A **combo** method to compute the distinct combinations of faces rolled, along with their counts.
    - Combinations should be sorted and saved as a multi-columned index.
    - Stores the results as a dataframe in a public attribute.
- A **face counts per roll** method to compute how many times a given face is rolled in each event.
    - Stores the results as a dataframe in a public attribute.
    - The dataframe has an index of the roll number and face values as columns (i.e. it is in wide format).

Unit Test

Each method in each class should have unit tests to test if the methods receive the correct inputs and return valid outputs.

General Requirements for Classes

This applies to both the main module and the unit test.

- All classes must have appropriate docstrings. For these, you may use language included in this document.
- All methods must have appropriate docstrings. Again, you may use language included in this document.
- Methods must be named consistently.

Scenarios

To demonstrate the use of your simulator, write a script using a Jupyter Notebook that performs the following tasks:

Scenario 1: A 2-headed coin

1. Create a fair coin (with faces H and T) and one unfair coin, in which one of the faces has a weight of 5 and the others 1.
2. Play a game of 1000 flips with all fair dice.
3. Play a game of 1000 flips with two unfair dice and one fair die.
4. For each game, use an Analyzer object to determine the relative frequency of jackpots – getting either all Hs or all Ts.
5. Compute relative frequency as the number of jackpots over the total number of rolls.
6. Show your results, comparing the two relative frequencies, in a simple bar chart.

Scenario 2: A 6-sided die

1. Create a fair die and two unfair dice, all of six sides with the faces 1 through 6. One unfair die (Type 1) will weight 6 five times more than the others (i.e. it has weight of 5 and the others a weight of 1 each). The other unfair die (Type 2) will weight 1 five times more than the others.
2. Play a game of 10000 rolls with 5 fair dice.
3. Play a game of 10000 rolls with 2 unfair dice of type 1, 1 unfair die of type 2, and the rest fair dice.
4. For each game, use an Analyzer object to determine the relative frequency of jackpots and show your results, comparing the two relative frequencies, in a simple bar chart.
5. Also compute 10 most frequent combinations of faces for each game. Plot each of these as bar charts.

Scendario 3: Letters of the Roman Alpha

1. Create a "die" of letters from a to z with weights based on their frequency of usage.
2. Play a game involving 5 of these dice with 1000 rolls.
3. How many combos can you that look like actual English words? **NOTE**: "combo" here just means resulting sequence, not literally combination as opposed to permutation.
4. Based on your eye count, what is the relative frequency of these words versus the rest?
5. Note: the Notebook will supply the weights for the letters.

Note: The method to compute faces per roll may be useful in computing jackpots.

Deliverables

A GitHub repo with the following elements

1. A minimal directory structure and file content of a Python library that can be installed by other users.
2. A single .py file (the module) containing all the classes.
3. A single .py file containing unit tests of all the methods in the preceding files using Unittest.
4. A single .txt file showing the results of unit testing all methods.
5. A single .ipynb file that demonstrates the module with the scenarios described above.
6. A README.md file (see below).
7. A license file, e.g. the MIT License.

Details about what to be submitted to Gradescope will be forthcoming.

About the README file

The README file will be your the main source of documentation for your users, in addition to your use of docstrings in your code.

Will consist of the following:

- Metadata
    - Specify your name and the project name (i.e. Monte Carlo Simulator).
- Synopsis
    - Show demo code of how the classes are used, i.e.

        - installing
        - importing
        - Creating dice
        - Playing games
        - Analyzing games.
- API description
    - A list of all classes with their public methods and attributes.
    - Each item should show their docstrings.
    - All paramters (with data types and defaults) should be described.
    - All return values should be described.
    - Do not describe private methods and attributes.
- Manifest
    - A list of all the files in the repo.

Rubric

Grading will be as follows

Points (weight)	Criterion
10	All classes and methods created and put into a single file.
10	All classes and methods well named and documented.
10	All methods tested. That is, unit tests have been created and passed.
10	All classes imported into notebook with all scenarios completed as specified.
10	All code is packaged properly, hosted on GitHub, and installable.
50	**TOTAL**

Collaboration

You may work in groups to discuss idea and approaches, but all deliverables must be yours.

Appendix

The Frequency of Letters in the English Language. (Source.)

```
A       8.4966
B       2.0720
C       4.5388
D       3.3844
E       11.1607
F       1.8121
G       2.4705
H       3.0034
I       7.5448
J       0.1965
K       1.1016
L       5.4893
M       3.0129
N       6.6544
O       7.1635
P       3.1671
Q       0.1962
R       7.5809
S       5.7351
T       6.9509
U       3.6308
```

```
V       1.0074
W       1.2899
X       0.2902
Y       1.7779
Z       0.2722
```