Exercises

Level 6: Monte Carlo in Python

6.1: Random Number Generation

1) Write code that generates a list of 200,000 uniform random numbers, ranging from 1 to 20. Additionally, generate 200,000 normally distributed random numbers (mu=10, sigma=7) and 200,000 lognormally distributed random numbers (mu=1, sigma=0.5). Export these lists of numbers to a single CSV file (should have 200,000 rows and three columns):

- **a.** Open this CSV file in Excel.
- **b.** Create three additional Excel Worksheets (named 'Uniform', 'Normal', and 'Lognormal'). Name the original Worksheet 'Input'.
- **c.** In each Worksheet, create a Histogram corresponding to its input data column (Insert->Histogram).
- **d.** Notice how the Uniform graph appears almost flat, the normal graph appears to be a bell curve, and the lognormal graph appears to be a bell curve with a large tail.

This is an example of *convergence* due to the Law of Large Numbers: If you generate enough random numbers using a certain distribution, it will always start to converge to the distribution.

Note that you will need to save the result as an Excel spreadsheet (.xlsx) instead of .csv for submission.

2) Do the same thing as Exercise 1, but this time we are going to print the distribution in the Python shell output. You can do this using horizontal dashes (-). The naïve approach would be to print one dash for each time a certain number appears (each number getting its own row of dashes). However, the frequencies will be in the thousands, so using one dash per frequency will overflow the screen. Therefore, to do this properly, you need to scale all the frequencies down (this is called *normalizing* the curve). We wish to scale each frequency in the histogram to have a min of 1 dash and a max of 100 dashes. We can do so using the following formula:

$$\frac{max_{new} - min_{new}}{max_{old} - min_{old}} * (freq - max_{old}) + max_{new}$$

$$Where\ max_{new} = 100, min_{new} = 1, max_{old}\ is\ the\ original\ maximum\ and\ min_{old}\ is\ the\ original\ minimum.$$

Also, note that you will need to round each decimal value to the nearest integer (we don't wish to have a separate frequency entry for every decimal). Example output:

1: ---
2: --------
4: ----------
7: -------
8: --
11: -

3) The following is a famous problem in probability based on a game show. It has generated much controversy due to the non-intuitive nature of the solution. The objective of this exercise is to write a simulation of the game that empirically demonstrates the accepted solution is indeed the correct solution. The game/problem is as follows:

You are presented with three doors. Behind one of the doors is a brand-new Lamborghini. Behind the other two doors are goats. You do not know which door contains the Lamborghini.

*The game show host asks you to choose one door; you choose a door. The host then opens one of the two doors that you did **not** choose and behind that door is a goat. The host then gives you the following options: Either stay with your originally chosen door or switch doors to the remaining, closed door.*

Should you stay, switch, or it makes no difference?

To solve this problem, do the following:

- **a)** Note down your hypothesis (you may have heard the solution to this famous problem already – no problem!).
- **b)** Model the game/player using Python OOP. The player should be a class that defines the switch strategy and has functions to choose a door and whether or not to switch. The game class should consist of a player object and the game logic.
- **c)** The game class should have a **playGame** function which does the following:
    - **a.** Randomly places the prize behind one of three doors.
    - **b.** Asks its player object to choose a door.
    - **c.** Figure out which door to 'open' (it should be one of the two non-selected doors, but it must not have the prize behind it). Query the player object whether or not to switch to the remaining door (the player object should either always switch or always not switch).
    - **d.** Check if the final chosen door is a winner or loser and return the Boolean result.
- **d)** Play the game one time from **main** to verify that it works.
- **e)** Play the game in a loop of 10,000,000 times from **main** and store the results of each play in a list. The average of this list should be the approximate probability of winning with your chosen strategy (stay or switch). Time this function (it may take some time).
- **f)** Was your hypothesis correct?

Congrats on your first Monte Carlo simulation in Python! The subsequent exercises will look to speed thus up by using multi-processing in Python.

6.2: Concurrency

1) In this exercise, we will look to make the Monty Hall simulation achieve true multi-processing. This is a good segue to financial Monte Carlo as the concepts and approaches are the same.

- **a)** Create and initialize five processes. Note that starting processes takes some time, and is the upfront cost of using multi-processing.
- **b)** Execute all five processes. Give each process 1/5 of the total simulations (2,000,000 each).
- **c)** Combine the five returned results lists and take the average, to get the overall result.
- **d)** Time all of the above (starting from b). Does total runtime improve from the previous level?
- **e)** Try decreasing/increasing the number of processes to determine the optimal runtime.

BONUS: Try the above using multithreading and compare/contrast the performance vs. multiprocessing.