Homework: Monty Hall Simulation

Table of Contents

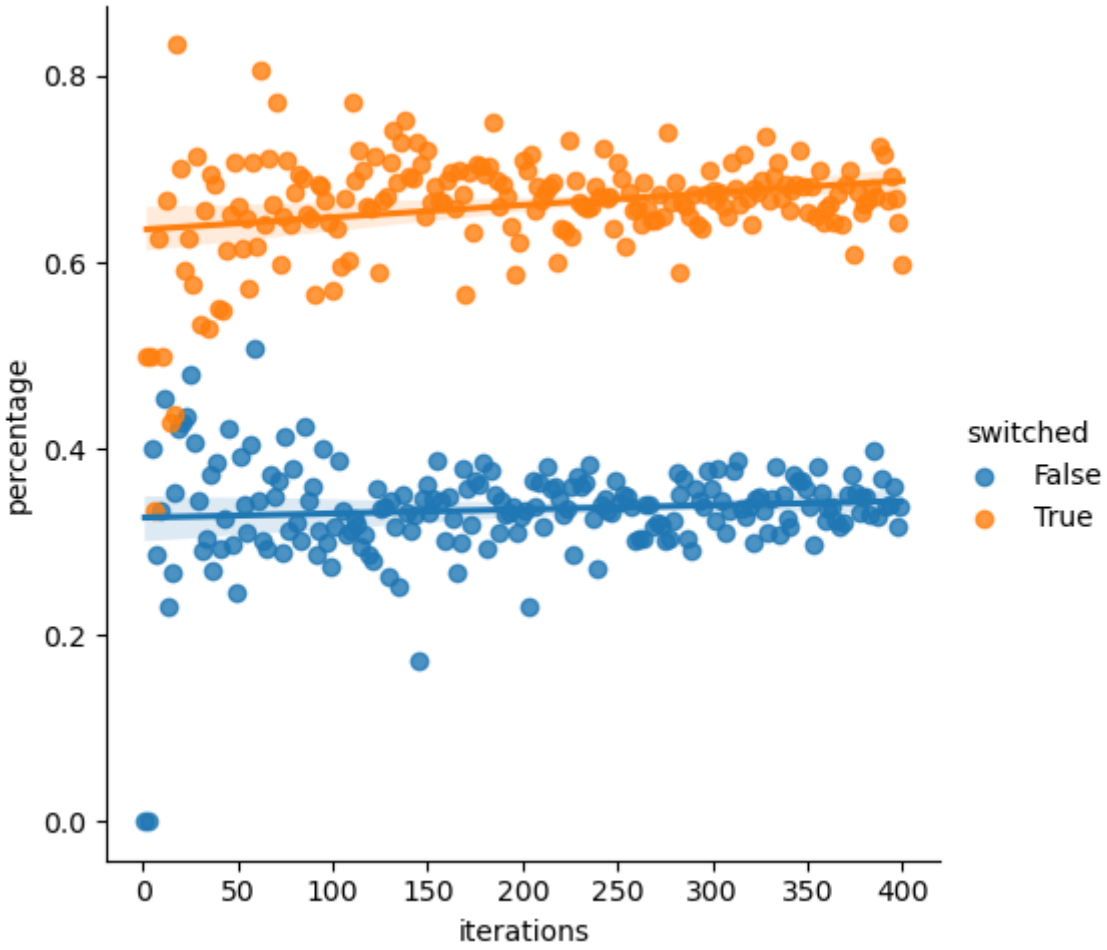
Background

For this homework we will be creating a simulation of the famous Monty Hall Problem. For context on this problem watch this video which explains the math behind the Monty Hall problem.

We will be writing two scripts, one script (named `monty_hall.py`) which simulates the Monty Hall Problem given different conditions that we determine (number of times that we play the game, whether or not we switch doors and how many doors there are). The other script (named `visualization.py`) will run simulations and use the conditions of the simulations that we run, as well as the percentage of times that we won in order to create a visualization where the x axis is how many times we played the game, and the y axis is the win percentage.

What we should be seeing is that as we play the game more and more times, the win percentage starts to approach the statistical probability of a certain condition being met. Essentially, through this exercise we will be demonstrating a proof of the “Central Limit Theorem”. For more information on the central limit theorem please watch this video.

You will know that you performed the homework correctly when the output of the `visualization.py` is something similar to the image below. In this case, the Monty Hall game is being simulated 400 times where half of those times the simulation is switching doors and half of those times it is not switching doors. Each simulation is using 3 doors to play the game. We can see that as the amount of times that we play the game increases, the values cluster around the specific probabilities, where the probability of winning if we switch doors is 66 % and the probability of winning if we don’t switch doors is 33 %.



NOTE

You will need to use the Seaborn package to complete this assignment. Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics. We will discuss this library in class, however you may need to reference the documentation for more information.

Instructions

- Your scripts should be named `monty_hall.py` and `visualization.py` accordingly.
- There should be an `if __name__ == "__main__":` statement at the end of your scripts.
- You may write additional classes, methods, and/or functions if you wish.
- The name of your files should consist exclusively of lower-case letters, numbers, and underscores, and the file extension .py. Your filename should not start with a number.
- Your script MUST contain docstrings.
- Any created files should export to the same directory as your python script.

monty_hall.py

- This script should import random, and create a class named Simulation.

Simulation class

Functionality

- Simulation should take an attribute, an integer, which represents the amount of doors that the simulation will use to play the game.

Attributes

- numdoors - An integer; the number of doors that will be used to play the game.

Methods

- __init__()
  - **Parameters**
    - Self
    - doornum - An integer; the number of doors that will be used to play the game.
  - **Functionality**
    - Set an attribute named numdoors that will be the number of doors that will be used to play the game.
- set_random_doors()
  - **Parameters**
    - self
  - **Functionality**
    - Will use numdoors number to create a list containing “zonk” strings. This list should be as long as the number. Meaning, if the number passed in is 2, it should create a list containing two “zonks” (ex. [“zonk”,”zonk”]). It should then replace one of those items in the list to the string “car” at random. It should return this list.
- choose_doors()
  - **Parameters**
    - self
  - **Functionality**
    - Call `set_random_doors()` and save the list to a variable. Pick and remove a random item from the this list which represents the door that the user/contestant has chosen. It should then remove a “zonk” from the list (the order of the zonk doesn’t matter for this part, it can be the first zonk, it can be the last zonk or it can be a random zonk). It should then pick and remove a random door from the list as the alternate door. Return the contestant door and the alternate door in that order as a tuple.
- play_game()
  - **Parameters**
    - self
    - switch - A boolean; Default value of False; Determines whether a contestant decides to

switch their door when playing the game and given the option to do so. This value will not change over time, if the value is `False` it will remain so through all iterations of game played.

- iterations - An integer; Default value of 1; The number of times that a person will play the game. Each time they play a game the doors should be random, and the choices should be random.

◦ **Functionality**

- The purpose of this method is to return the percentage of the amount of times that the game was won as a decimal (float). So for example, if my iterations is 3 (this means I play the game 3 times) and I win 2 of those 3 times, then my returned value is .66. If my iterations is 1 and I win 1 time then my returned value is 1. If my iterations is 4 and I don't win any games then my returned value is 0.
- To do so, you will be playing the game as many times as the value of `iterations`. You will use `choose_doors()` to pick doors, the user door and an alternate each time you play.
- To determine if a game has been won is to look at whether the door that the player chose is 1) the "car" and the switch parameter is False or 2) if the alternate door is the "car" and the switch parameter is True. If either of those conditions are met then it counts as a win.
- We will want to keep track of how many times we win and lose given how many iterations we are running and calculate the final win percentage. Return this percentage.

NOTE You may run into an error when calculating the win percentage. This will require some error handling on your part. You may lose points for not correctly identifying the proper exception to handle.

`if __name__ == "__main__":`

- Test out your script by creating an instance of Simulation and running a simulation by playing a game where the player switches doors and the iterations is 1_000.

NOTE You may be so inclined to test your code using more iterations. Keep in mind, doing this might take significant computational power. Be weary of adding too many 0s into your script. A sweet spot for most machines is 1000. You should notice that your returned value is somewhere near .66 for this instance.

visualization.py

- This script should import monty_hall.py, pandas and seaborn.
- This script will create a class named Plot.

`Plot class`

Functionality

- This class stores the data for a particular instance of a simulation of the monty hall problem. It contains functionality to export a visualization of the win percentages.

Attributes

- doors: An integer; The number of doors that the simulation will be based on.
- iterations: An integer; The number of iterations that a simulation will be based on.
- sequence: A list; Starts empty, is later populated by dictionaries each containing: num of iterations a game was played, percentage won for that simulation, doors used in that simulation, whether the door was switched or not for that simulation.

Methods

- __init__()
  - **Parameters**
    - self
    - doors - An integer; Defaults to 3; The number of doors that the simulation will be based on.
    - iterations - iterations: An integer; Defaults to 200; The number of iterations that a simulation will be based on.
  - **Functionality**
    - Create an attribute named sequence that will be a list that will eventually contain dictionaries that we will use to create a data frame. Write a loop that will run for as many times as the value of iterations. Starting at 1, the loop will determine if the current value of iteration is odd or even. If even: the loop should create an instance of simulation from the monty_hall namespace and invoke the play_game method. It will pass in the number of doors when created, then pass in switched as True and iterations will be passed in using the current iteration that we are on. If the current iteration number is odd then it will do the same thing but it will create an instance of simulation where the switched value is False. Using the returned win percentage, the loop will append a dictionary to the sequence attribute where the keys are “iterations” (what our current iteration is), “percentage” (our returned percentage value), “doors” (how many doors our simulation uses) and “switched” (our boolean as a string representing whether we switched the door or not. After the loop is completed we will call the make_plot method.
      - For example: Lets say that my simulations parameter is 3. I will start at 1 and create that simulation instance where my iterations argument for the play_game method is 1 and append that dictionary to the list. I will then go to 2 and do the same where the iterations argument is 2. I will then go to 3 and do the same where the iterations argument is 3 and stop.
- make_plot()
  - **Parameters**
    - self
  - **Funcionality**

    - This method will use the sequence attribute to create a pandas Dataframe. We will then use the lmplot method of seaborn in order to create a plot object where x is “iterations”, y is “percentage”, data is the pandas dataframe that we created and “hue” is “switched”. You should then use the savefig method of the plot object in order to export a png of the visualization. The name of your png file should dynamically include the iterations attribute and the doors attribute.

```
if __name__ == "__main__":
```

- Test out your script by calling an instance of Plot by passing in doors as 5 and iterations as 100.

Running your program

Your program is designed to run from the terminal. To run it, open a terminal and ensure you are in the directory where your script is saved.

To properly test your scripts you may have to run each script independently.