GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

Project Description

My term project idea came from an assignment I had in another course of mine this semester. Due to my major, I am currently in another class teaching me basic Python programming for the purpose of preparing us for interpreting results from psychological studies, and one of our most recent exercises was solving the probability-based Monty Hall Problem using the basics we had been taught in that class.

While I was working on the assignment, I was explaining my work to my roommate, and I realized that he was having a hard time understanding the reasoning behind the proper solution because he couldn't visualize it. Thus, my project idea was to take my knowledge of the Monty Hall Problem's solution and use it to create a program that can visually show why the puzzle's seemingly irrational answer is the way it is, using a tree data structure.

Therefore, the purpose of my term project is to solve the Monty Hall Problem:

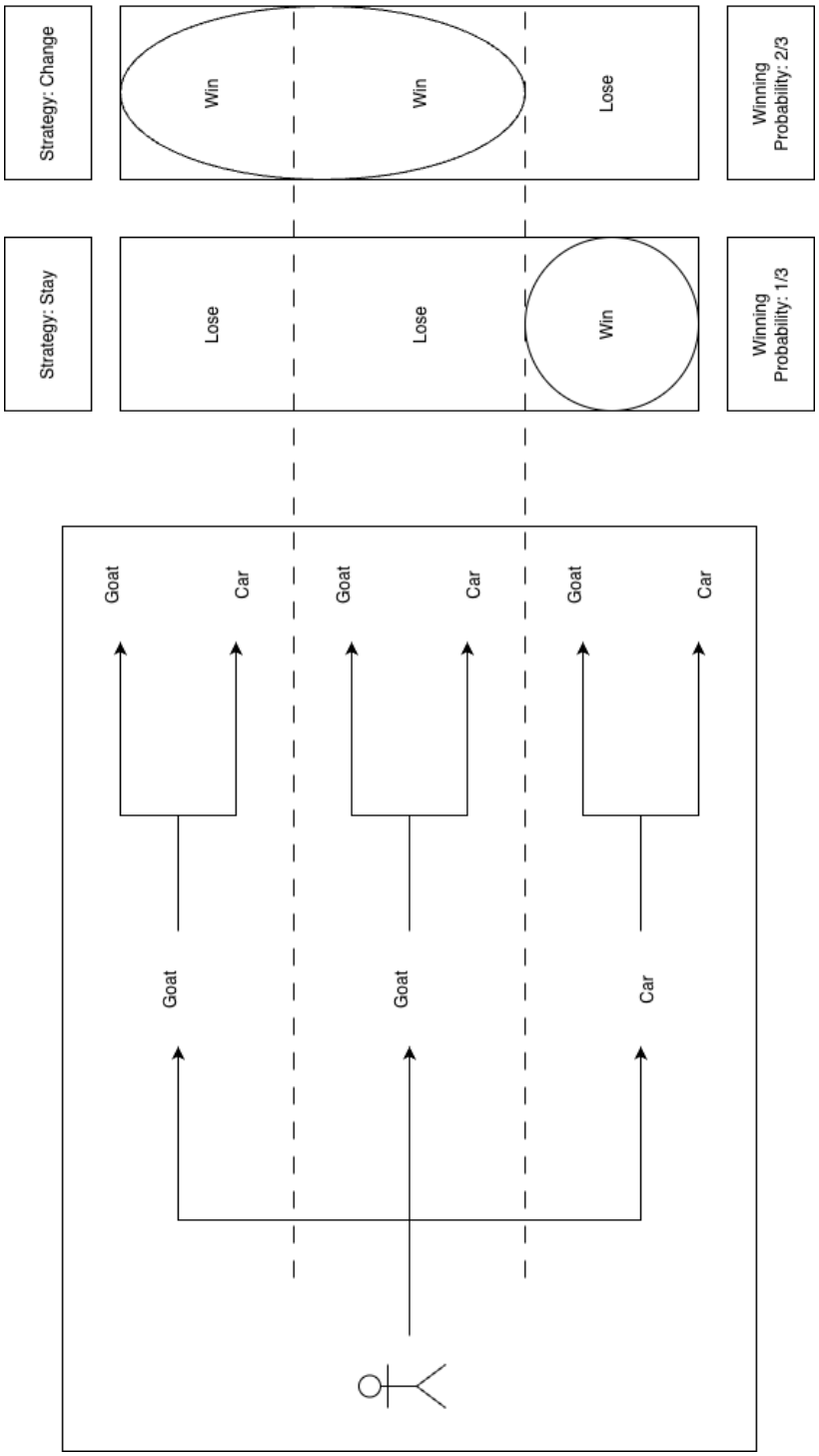
> The task is to choose from one of three doors. Behind two of the doors are goats, and behind the one remaining door is a car. The goal is to try to choose the door with the car behind it.
>
> Once a door has been chosen, the game show host reveals what's behind one of the two doors remaining that wasn't picked—and he always only reveals a door with a goat behind it.
>
> Now there are two doors left to choose from to find the one with the car behind it (since you know that the one the game show host opened has a goat behind it).

Should you stick with the door you originally guessed, switch your guess to the other unopened door, or does making a choice at this stage even really matter in terms of your chances of selecting the door with the car behind it?

GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

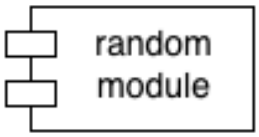
Program Model

The following diagram outlines the conceptual basis of the Monty Hall Problem and my vision for what my program will help solve and visualize:



GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

The following table diagrams are meant to model the flow of my planned program:



Classes

Door
+ num: int numerical label + behind: str prize of goat or car
+ print_door(): prints label and prize + open_door(): returns prize

Doors
+ door1: Door first door + door2: Door second door + door3: Door third door + door_list: lst list of all three doors
+ set_up_doors(): puts prizes behind doors and adds doors to list + remove_door(Door): removes specified door from list

DecisionNode
+ count: int num of times decision was made + next_goat: DecisionNode choosing goat next + next_car: DecisionNode choosing car next
+ decide(): increments count by 1 + print_node(int, str): recursively prints node and subtree

DecisionTree
+ root: DecisionNode
+ set_up_tree(): sets up decision levels for 1st and 2nd guess + make_guesses(Door, Door): increments total decisions count and moves to first guess DecisionNode + make_guesses_recursive(DecisionNode, Door): recursively increments decision level counts + print_tree(): prints decision tree

GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

Functions

montyhall_gameplay(strategy)	
objects	**functionality**
strategy	str passed to function that determines guess2 selection
doors	Doors object to play game
goat_door_list	list to track Door objects hiding goats
guess1	Door object randomly chosen from doors; returned
door_revealed	Door object hiding a goat that isn't guess1; removed from doors
guess2	Door object chosen from doors using strategy; returned along with "win" or "lose" str depending on if guess2 hiding goat or car

montyhall_simulator(strategy, n)	
objects	**functionality**
strategy	str passed to function to pass to montyhall_gameplay()
n	int passed to function to specify number of times montyhall_gameplay() is run
my_tree	DecisionTree to track decision counts; returned to main
my_wins	int to track number of times game is won
first_guess	Door object returned by montyhall_gameplay() and used to populate first decision level of my_tree
second_guess	Door object returned by montyhall_gameplay() and used to populate second decision level of my_tree
result	str describing whether game was won or lost
my_wins_percent	int initialized through calculation of my_wins / n * 100; returned to main

Main Workflow

main()	
objects	**functionality**
ktree	DecisionTree returned by montyhall_simulator() when "keep" strategy is passed to it; printed for visualization
k_win_rate	float returned by montyhall_simulator() when "keep" strategy is passed to it; added to win_rates as value
ctree	DecisionTree returned by montyhall_simulator() when "change" strategy is passed to it; printed for visualization
c_win_rate	float returned by montyhall_simulator() when "change" strategy is passed to it; added to win_rates as value
rtree	DecisionTree returned by montyhall_simulator() when "random" strategy is passed to it; printed for visualization
r_win_rate	float returned by montyhall_simulator() when "random" strategy is passed to it; added to win_rates as value
win_rates	dict with strategy names as keys and strategy win rates as values
best_strat	dict with "percent_wins", "strategy" as keys and highest strategy win rate, corresponding strategy name from win_rates as values

GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

Program/Model Description

To solve this problem and help visualize its seemingly counterintuitive answer, I created four different classes: a *Door* class, a *Doors* class, a *DecisionNode* class, and a *DecisionTree* class.

The *Door* class was meant to help create objects that a) labelled the doors from the player's point of view and b) identified what was behind the doors for the game show host (my program). The *Doors* class created lists of three doors each since there needed to be three doors per game, and it also allowed for easy access to each of the three doors during each round.

The *DecisionNode* class helped create nodes for the *DecisionTree* class by creating an object that holds the *count* of how many times that particular decision has been made and *two links* to the next two possible choices that can be made from that point on: picking a door with either a goat behind it or a car behind it.

Thus, the *DecisionTree* class is used to map out the *results* of making a particular strategy choice after your first guess: after a door with a goat behind it has been revealed, what do you get if you stick with the door you originally guessed, switch your guess to the other remaining door, or just flip a coin and pick between the two remaining doors randomly?

To populate the *DecisionTree* objects for each of the three strategy choices, the *montyhall_gameplay()* function plays *one* round of the game before returning the first door the player guesses (always random), the second door the player guesses (based on the selected strategy), and whether they won the car or not.

The *montyhall_simulator()* function then runs the *montyhall_gameplay()* function a specified, large number of times to see if a pattern emerges before returning the percentage of times the player actually won the game. It also creates and populates a *DecisionTree* object for each of the three strategy types before returning that as well.

The *DecisionNode* class has a recursive method for printing a node's subtree, and the *DecisionTree* class has a recursive method for populating the tree each time a new game is played and new decisions are made, fulfilling the requirement for algorithm implementation.

The *random* library is used to randomly populate what's behind the three doors each time the game is played, and it is also used during the game if the strategy chosen is to randomize the second guess.

The data structures used primarily include lists, dictionaries, and trees.

My current expectations for my program are decent since it appears that it is implementable and will work well based on what I have run so far. If there are any weaknesses, I expect them to occur in the setup and population of the *DecisionTree* objects since I haven't coded non-BSTs before, and I tend to struggle with recursive algorithms.

GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

Program Implementation

The full file is available in my GitHub Repository.

```
"""
File: mh_simulator.py

Description: see write-up project/program description
"""

import random

class Door:
    """
    Creates Door objects with numerical labels that hide a prize.
    """
    def __init__(self, num, obj):
        self.num = num
        self.behind = obj

    def print_door(self):
        """
        Prints door's label and hidden prize.
        """
        print(f"Door {self.num}: {self.behind}")

    def open_door(self):
        """
        Returns door's hidden prize.
        """
        return self.behind


class Doors:
    """
    Creates lists of three doors each.
    Allows for easy access to each of the three doors during each round.
    """
    def __init__(self):
        self.door1 = Door(1, None)
        self.door2 = Door(2, None)
        self.door3 = Door(3, None)
        # requirement 1: data structure - list
        self.door_list = [self.door1, self.door2, self.door3]

    def set_up_doors(self):
        """
        Sets up list of three doors for each round of the game.
        """
```

GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

```
        self.door1.behind = random.choice(["goat", "car"])

        if self.door1.open_door() == "car":
            self.door2.behind = "goat"
            self.door3.behind = "goat"
        else:
            self.door2.behind = random.choice(["goat", "car"])

            if self.door2.open_door() == "car":
                self.door3.behind = "goat"
            else:
                self.door3.behind = "car"

        self.door_list = [self.door1, self.door2, self.door3]

    def remove_door(self, revealed_door):
        """
        Removes a door revealed to have a goat behind it.
        """
        for door in self.door_list:
            if door == revealed_door:
                self.door_list.remove(door)
                break


class DecisionNode:
    """
    Represents how many times a particular decision has been made
    and links to the next two possible choices that can be made.
    """
    def __init__(self):
        self.count = 0
        self.next_goat = None
        self.next_car = None

    def decide(self):
        """
        Adds to the count of the number of times a decision has been made.
        """
        self.count += 1

    # requirement 2: algorithm - recursion
    def print_node(self, level = 0, label = ""):
        """
        Recursively prints a DecisionNode, its label,
        and its children with indentation.
        """
        # Recursively call for goat child, passing next level and goat label
        if self.next_goat is not None:
```

GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

```
            self.next_goat.print_node((level + 1) * 2, label = "Goat")

        # Print the current node's label and count
        if label:
            print("    " * level, f"---> {label}: {self.count}")
        else:
            # For the root node, which has no label
            print(f"Start: {self.count}")

        # Recursively call for car child, passing next level and car label
        if self.next_car is not None:
            self.next_car.print_node((level + 1) * 2, label = "Car")


# requirement 1: data structure - tree
class DecisionTree:
    """
    Maps out results of making particular strategy choice after first guess.
    """
    def __init__(self):
        self.root = DecisionNode()

    def set_up_tree(self):
        """
        Sets up decision levels and nodes of decision tree.
        """
        self.root.next_goat = DecisionNode()
        self.root.next_car = DecisionNode()
        first_level = [self.root.next_goat, self.root.next_car]

        for node in first_level:
            node.next_goat = DecisionNode()
            node.next_car = DecisionNode()

    def make_guesses(self, first_door, second_door):
        """
        First step of recursive decision-making process.
        """
        self.root.decide()

        first_node = None
        if first_door.open_door() == "goat":
            first_node = self.root.next_goat
        elif first_door.open_door() == "car":
            first_node = self.root.next_car

        self.make_guesses_recursive(first_node, second_door)

    # requirement 2: algorithm - recursion
```

GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

```
    def make_guesses_recursive(self, node, second_door):
        """
        Rest of the steps in the recursive decision-making process.
        """
        if node.next_goat is not None and node.next_car is not None:
            if second_door.open_door() == "goat":
                self.make_guesses_recursive(node.next_goat, second_door)
            elif second_door.open_door() == "car":
                self.make_guesses_recursive(node.next_car, second_door)

        node.decide()

    def print_tree(self):
        """
        Prints the entire decision tree structure, starting from the root.
        """
        self.root.print_node()



def montyhall_gameplay(strategy):
    """
    This function plays through one round of the game.
    """
    # set up doors
    doors = Doors()
    doors.set_up_doors()

    # know what's behind doors
    goat_door_list = []
    for door in doors.door_list:
        if door.open_door() == "goat":
            goat_door_list.append(door)

    # make first guess and return result later to simulator
    guess1 = random.choice(doors.door_list)

    # reveal a door with a goat and whittle down to 2 options
    if guess1.open_door() == "goat":
        door_revealed = None
        for goat_door in goat_door_list:
            if goat_door != guess1:
                door_revealed = goat_door
                break
    else:
        door_revealed = random.choice(goat_door_list)

    doors.remove_door(door_revealed)
```

GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

```
    # make second guess and later return result to simulator
    # this is where strategy argument comes in
        # either keep first guess door or change to other remaining door
    if strategy == "keep":
        guess2 = guess1
    elif strategy == "change":
        guess2 = None
        for door in doors.door_list:
            if door != guess1:
                guess2 = door
                break
    else:
        guess2 = random.choice(doors.door_list)

    # return win or loss
    if guess2.open_door() == "car":
        return guess1, guess2, "win"
    return guess1, guess2, "lose"


def montyhall_simulator(strategy, n):
    """
    This function simulates the results of playing through n rounds of the
game.
    """
    my_tree = DecisionTree()
    my_tree.set_up_tree()
    my_wins = 0

    for _ in range(n):
        first_guess, second_guess, result = montyhall_gameplay(strategy)
        my_tree.make_guesses(first_guess, second_guess)
        if result == "win":
            my_wins += 1

    my_wins_percent = my_wins  / n * 100

    return my_tree, my_wins_percent



def main():
    """
    Main function of the program. Combines functionality of all test cases.
    """
    # run the simulator and save results for later
    ktree, k_win_rate = montyhall_simulator("keep", 10000)
    ctree, c_win_rate = montyhall_simulator("change", 10000)
    rtree, r_win_rate = montyhall_simulator("random", 10000)
```

GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

```
    # requirement 1: data structure - dictionary
    # get the percentage of times you win when you keep vs change vs randomize
    win_rates = {"keep": k_win_rate,
                 "change": c_win_rate,
                 "randomize": r_win_rate}
    best_strat = {"strategy": "", "percent_wins": 0}

    # choose the best strategy from between the 3
    for strat, win_rate in win_rates.items():
        if win_rate > best_strat["percent_wins"]:
            best_strat["strategy"] = strat
            best_strat["percent_wins"] = win_rate

    print()
    print(f"The best strategy is to {best_strat['strategy']} your", end = " ")
    print(f"guess due to its {best_strat['percent_wins']:.2f}% win rate.")
    print()

    # print the decision trees
    print("Tree of Counts when Keeping Original Guess:")
    ktree.print_tree()
    print()

    print("Tree of Counts when Changing Guess:")
    ctree.print_tree()
    print()

    print("Tree of Counts when Randomizing Guess:")
    rtree.print_tree()


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

GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

Test Cases

The following test cases output results nearly identical to the program implementation's main function, but with slightly greater detail in order to check for accuracy at all levels. The full files are available in my GitHub Repository.

```
"""
File: testcase_randomstrat.py

Description: Tests whether the simulator can successfully randomize
the second guess and whether the resulting DecisionTree object
is populated correctly.
"""

import mh_simulator as mhs

# run the simulator using a randomization strategy and save its results
rtree, r_win_rate = mhs.montyhall_simulator("random", 10000)

# output the results of randomizing the second guess
print()
print(f"The result of randomizing all guesses "
     f"is winning {r_win_rate:.2f}% of the time.")
print()

# print the decision tree that develops when the second guess is randomized
print("Tree of Counts when Randomizing All Guesses:")
rtree.print_tree()
```

```
"""
File: testcase_keepstrat.py

Description: Tests whether the simulator can successfully keep
the second guess the same as the first and whether
the resulting DecisionTree object is populated correctly.
"""

import mh_simulator as mhs

# run the simulator using a keep-original-guess strategy and save its results
ktree, k_win_rate = mhs.montyhall_simulator("keep", 10000)

# output the results of keeping the original guess
print()
print(f"The result of keeping the original guess "
     f"is winning {k_win_rate:.2f}% of the time.")
print()
```

GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

```
# print the decision tree that develops when the original guess is kept
print("Tree of Counts when Keeping Original Guess:")
ktree.print_tree()
```

```
"""
File: testcase_changestrat.py

Description: Tests whether the simulator can successfully change
the second guess from the first guess and whether
the resulting DecisionTree object is populated correctly.
"""

import mh_simulator as mhs

# run the simulator using a change-second-guess strategy and save its results
ctree, c_win_rate = mhs.montyhall_simulator("change", 10000)

# output the results of changing the second guess
print()
print(f"The result of always changing the second guess "
     f"is winning {c_win_rate:.2f}% of the time.")
print()

# print the decision tree that develops when the second guess is changed
print("Tree of Counts when Changing Next Guess:")
ctree.print_tree()
```

```
"""
File: testcase_beststrat.py

Description: Tests whether the program can help settle on
the correct solution to the Monty Hall Problem or not.
"""

import mh_simulator as mhs

# run the simulator and save results for later
ktree, k_win_rate = mhs.montyhall_simulator("keep", 10000)
ctree, c_win_rate = mhs.montyhall_simulator("change", 10000)
rtree, r_win_rate = mhs.montyhall_simulator("random", 10000)

# requirement 1: data structure - dictionary
# get the percentage of times you win when you keep vs change vs randomize
win_rates = {"keep": k_win_rate,
             "change": c_win_rate,
             "randomize": r_win_rate}
best_strat = {"strategy": "", "percent_wins": 0}
```

GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

```
# choose the best strategy from between the 3
for strat, win_rate in win_rates.items():
   if win_rate > best_strat["percent_wins"]:
       best_strat["strategy"] = strat
       best_strat["percent_wins"] = win_rate

print()
print(f"The best strategy is to {best_strat['strategy']} your", end = " ")
print(f"guess due to its {best_strat['percent_wins']:.2f}% win rate.")
```

GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

Main/Test Case Output

```
/usr/local/bin/python3.13 /Users/azizmutabanna/Documents/CS 313E/Term Project/testcase_randomstrat.py

The result of randomizing all guesses is winning 50.16% of the time.

Tree of Counts when Randomizing All Guesses:
                        ---> Goat: 3330
         ---> Goat: 6695
                        ---> Car: 3365
Start: 10000
                        ---> Goat: 1654
         ---> Car: 3305
                        ---> Car: 1651

Process finished with exit code 0
```

```
/usr/local/bin/python3.13 /Users/azizmutabanna/Documents/CS 313E/Term Project/testcase_keepstrat.py

The result of keeping the original guess is winning 33.53% of the time.

Tree of Counts when Keeping Original Guess:
                        ---> Goat: 6647
         ---> Goat: 6647
                        ---> Car: 0
Start: 10000
                        ---> Goat: 0
         ---> Car: 3353
                        ---> Car: 3353

Process finished with exit code 0
```

```
/usr/local/bin/python3.13 /Users/azizmutabanna/Documents/CS 313E/Term Project/testcase_changestrat.py

The result of always changing the second guess is winning 66.09% of the time.

Tree of Counts when Changing Next Guess:
                        ---> Goat: 0
         ---> Goat: 6609
                        ---> Car: 6609
Start: 10000
                        ---> Goat: 3391
         ---> Car: 3391
                        ---> Car: 0

Process finished with exit code 0
```

GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

```
/usr/local/bin/python3.13 /Users/azizmutabanna/Documents/CS 313E/Term Project/testcase_beststrat.py

The best strategy is to change your guess due to its 66.10% win rate.

Process finished with exit code 0
```

My main function's/test case output does the following three things:

1) It ensures that all classes and functions work appropriately. If the *Door* or *Doors* classes were not written correctly, then the *montyhall_gameplay()* function would not have worked properly, which means that the *montyhall_simulator()* function also would not have worked properly (which *testcase_beststrat.py* determined worked fine). If the *DecisionNode* and *DecisionTree* classes weren't working correctly, then errors would have been raised, the counts would have been incorrect, or some other issue would have presented itself when the tree objects were printed.
2) It identifies which strategy is the best for winning the car by comparing the percentage of times a car is won when each strategy is implemented a specified number of times.
3) It prints out the trees for each of the three strategy options to provide a visualization of what is actually happening behind the scenes at every stage of the decision-making process while playing the game defined by the Monty Hall Problem.

GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

Output Interpretations

Based on the output of my main function/test cases, I can make the following interpretations from the results of my simulation of the Monty Hall Problem:

- The strategy that gives the best chances at winning the car is *changing* the second guess from the first guess to the other remaining unopened door. I know this to be true because the test case output showed that randomizing the second guess gave a 50% chance at winning the car, keeping the original guess only gave a 33% chance, and changing from the original guess gave a 66% chance of winning the game.
- Using the visualizations of the results of each of the available strategies at each stage of the game (the printed decision trees), the probability of selecting a goat or a car at any stage when using any strategy can be easily estimated by sight alone.
    - At the first level of the tree, representing the first, randomized guess, we can quantitatively see how there is a ⅓ chance of selecting the car and a ⅔ chance of selecting the goat.
        - When randomizing the second guess, the probability of selecting either a goat or a car quantitatively/visually becomes ½ each.
        - Since there is a ⅔ chance of first picking a door with a goat behind it, there is quantitatively/visually a 100% chance of next picking the door with the car behind it *if you switch from your original guess*.
        - Otherwise, if you first pick a door with a goat behind it and stick with your guess, then there is a ⅔ chance of getting the goat and quantitatively/visually a 100% chance of keeping it.
- Thus, using the visualizations/printed decision trees, we can also see how the skewed results of the first guess affect the results of the second guess. When you're at the stage of making your second guess, you are not existing in an isolated, independent situation; you are existing within the specific situation that your first guess put you in.

GitHub Repository: https://github.com/amutabanna/mutabanna-term-proj.git

Program Evaluation

A lot of the expected weaknesses for my program were identified by Pylint: some of my classes didn't have enough public methods, and some of my functions had too many branches and needed to be condensed into something more readable, concise, and clear. Specifically, the *Door* class, the *Doors* class, and the *DecisionNode* class all only had one out of two required public methods, with the *Door* class only having *print_door()*, the *Doors* class only having *set_up_doors()*, and the *DecisionNode* class only having *print_node()*. Additionally, the *montyhall_gameplay()* function had 14 branches (like loops and conditionals), which went 2 branches over the Pylint limit of 12 branches per function.

First, within the *Door* class, I added the attribute accessor *open_door()*, and within the *DecisionNode* class, I added the attribute mutator *decide()*. This brought up the public methods of both classes to two, and it also added to the number of externally usable methods in my program (of which I utilized both). In order to solve the last two problems, I essentially killed two birds with one stone: within the *Doors* class, I added a method that removes a *Door* object from the *door_list* attribute if it matches a previously opened—or "revealed"—door. By doing this, I eliminated the necessity for a piece of code within the *montyhall_gameplay()* function, which allowed me to delete the code block, replace it with the *remove_door()* method, and bring down the number of branches in the function to within the accepted limit.

I think the manner in which I could most significantly improve upon my program in the future is if I better incorporate the tree objects I created into my calculations for deciding which strategy option gives the best shot at winning the car. At the moment, the tree objects only hold counts, but they could perhaps instead hold percentages or something else more meaningful to the average viewer if I further improve upon this program in the future.