Simulating The Monty Hall Problem

The Monty Hall problem is a well-known puzzle in probability derived from an American game show, *Let's Make a Deal*. (The original 1960s-era show was hosted by Monty Hall, giving this puzzle its name.) Intuition leads many people to get the puzzle wrong, and when the Monty Hall problem is presented in a newspaper or discussion list, it often leads to a lengthy argument in letters-to-the-editor and on message boards.

The game is played like this:

1. The game show set has three doors. A prize such as a car or vacation is behind one door, and the other two doors hide a valueless prize called a Zonk; in most discussions of the problem, the Zonk is a goat.
2. The contestant chooses one door. We'll assume the contestant has no inside knowledge of which door holds the prize, so the contestant will just make a random choice.
3. The smiling host Monty Hall opens one of the other doors, always choosing one that shows a goat, and always offers the contestant a chance to switch their choice to the remaining unopened door.
4. The contestant either chooses to switch doors, or opts to stick with the first choice.
5. Monty calls for the remaining two doors to open, and the contestant wins whatever is behind their chosen door.

Let's say a hypothetical contestant chooses door #2. Monty might then open door #1 and offer the chance to switch to door #3. The contestant switches to door #3, and then we see if the prize is behind #3.

The puzzle is: what is the best strategy for the contestant? Does switching increase the chance of winning the car, decrease it, or make no difference?

The best strategy is to make the switch. It's possible to analyze the situation and figure this out, but instead we'll tackle it by simulating thousands of games and measuring how often each strategy ends up winning.

Approach

Simulating one run of the game is straightforward. We will write a `simulate()` function that uses Python's `random` module to pick which door hides the prize, the contestant's initial choice, and which doors Monty chooses to open. An input parameter controls whether the

contestant chooses to switch, and `simulate()` will then return a Boolean telling whether the contestant's final choice was the winning door.

Part of the reason the problem fools so many people is that in the three-door case the probabilities involved are 1/3 and 1/2, and it's easy to get confused about which probability is relevant. Considering the same game with many more doors makes reasoning about the problem much clearer, so we'll make the number of doors a configurable parameter of the simulation script.

Solution

The simulation script is executed from the command line. If you supply no arguments, the script will use three doors and run 10,000 trials of both the switching and not-switching strategies. You can supply `--doors=100` to use 100 doors and `--trials=1000` to run a smaller number of trials.

```
#!/usr/bin/env python3

"""Simulate the Monty Hall problem.

"""

import argparse, random

def simulate(num_doors, switch, verbose):
    """(int, bool): bool

    Carry out the game for one contestant.  If 'switch' is True,
    the contestant will switch their chosen door when offered the chance.
    Returns a Boolean value telling whether the simulated contestant won.
    """

    # Doors are numbered from 0 up to num_doors-1 (inclusive).

    # Randomly choose the door hiding the prize.
    winning_door = random.randint(0, num_doors-1)
    if verbose:
        print('Prize is behind door {}'.format(winning_door+1))

    # The contestant picks a random door, too.
    choice = random.randint(0, num_doors-1)
    if verbose:
        print('Contestant chooses door {}'.format(choice+1))

    # The host opens all but two doors.
    closed_doors = list(range(num_doors))
    while len(closed_doors) > 2:
        # Randomly choose a door to open.
        door_to_remove = random.choice(closed_doors)

        # The host will never open the winning door, or the door
        # chosen by the contestant.
        if door_to_remove == winning_door or door_to_remove == choice:
            continue

        # Remove the door from the list of closed doors.
        closed_doors.remove(door_to_remove)
        if verbose:
            print('Host opens door {}'.format(door_to_remove+1))

    # There are always two doors remaining.
    assert len(closed_doors) == 2

    # Does the contestant want to switch their choice?
    if switch:
```

```
    if switch:
        if verbose:
            print('Contestant switches from door {} '.format(choice+1), end='')

        # There are two closed doors left.  The contestant will never
        # choose the same door, so we'll remove that door as a choice.
        available_doors = list(closed_doors) # Make a copy of the list.
        available_doors.remove(choice)

        # Change choice to the only door available.
        choice = available_doors.pop()
        if verbose:
            print('to {}'.format(choice+1))

    # Did the contestant win?
    won = (choice == winning_door)
    if verbose:
        if won:
            print('Contestant WON', end='\n\n')
        else:
            print('Contestant LOST', end='\n\n')
    return won


def main():
    # Get command-line arguments
    parser = argparse.ArgumentParser(
        description='simulate the Monty Hall problem')
    parser.add_argument('--doors', default=3, type=int, metavar='int',
                        help='number of doors offered to the contestant')
    parser.add_argument('--trials', default=10000, type=int, metavar='int',
                        help='number of trials to perform')
    parser.add_argument('--verbose', default=False, action='store_true',
                        help='display the results of each trial')
    args = parser.parse_args()

    print('Simulating {} trials...'.format(args.trials))

    # Carry out the trials
    winning_non_switchers = 0
    winning_switchers = 0
    for i in range(args.trials):
        # First, do a trial where the contestant never switches.
        won = simulate(args.doors, switch=False, verbose=args.verbose)
        if won:
            winning_non_switchers += 1

        # Next, try one where the contestant switches.
        won = simulate(args.doors, switch=True, verbose=args.verbose)
        if won:
            winning_switchers += 1

    print('    Switching won {0:5} times out of {1} ({2}% of the time)'.format(
            winning_switchers, args.trials,
            (winning_switchers / args.trials * 100 ) ))
    print('Not switching won {0:5} times out of {1} ({2}% of the time)'.format(
            winning_non_switchers, args.trials,
            (winning_non_switchers / args.trials * 100 ) ))


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

A sample run:

```
-> code/monty-hall.py
Simulating 10000 trials...
    Switching won  6639 times out of 10000 (66.39% of the time)
Not switching won  3357 times out of 10000 (33.57% of the time)
->
```

Our simulation confirms the result: it's better to switch, which wins the car more often. If you switch, you have a 2/3 probability of winning the car; if you don't switch, you'll only win the car 1/3 of the time. The numbers from our simulation bear this out, though our random trials usually won't result in percentages that are *exactly* 66.6% or 33.3%.

If you supply the `--verbose` switch, the simulator will print out each step of the game so you can work through some examples. Be sure to use `--trials` to run a smaller number of trials:

```
-> code/monty-hall.py  --verbose --trials=2
Simulating 2 trials...
Prize is behind door 2
Contestant chooses door 3
Host opens door 1
Contestant LOST

Prize is behind door 3
Contestant chooses door 1
Host opens door 2
Contestant switches from door 1 to 3
Contestant WON

Prize is behind door 2
Contestant chooses door 3
Host opens door 1
Contestant LOST

Prize is behind door 3
Contestant chooses door 3
Host opens door 1
Contestant switches from door 3 to 2
Contestant LOST

    Switching won     1 times out of 2 (50.0% of the time)
Not switching won     0 times out of 2 (0.0% of the time)
->
```

Code Discussion

The command-line arguments are parsed using the `argparse` module, and the resulting values are passed into the `simulate()` function.

When there are **num_doors** doors, `simulate()` numbers the doors from 0 up to **num_doors**-1. `random.randint(a, b)()` picks a random integer from the range **a** to **b**, possibly choosing one of the endpoints, so here we use `random.randint(0, num_doors-1)()`.

To figure out which doors the host will open, the code makes a list of the currently closed doors, initially containing all the integers from 0 to **num_doors**-1. Then the code loops, picking a random door from the list to open. By our description of the problem, Monty will never open the contestant's door or the one hiding the prize, so the loop excludes those two doors and picks a different door. The loop continues until only two doors remain, so Monty will always open **num_doors**-2 doors.

To implement the contestant's switching strategy, we take the list of closed doors, which is now 2 elements long, and remove the contestant's current choice. The remaining element is therefore the door they're switching to.

Lessons Learned

This approach to answering a question, where we randomly generate many possible inputs, calculate the outcomes, and summarize the results, is called Monte Carlo simulation and has a long history, having been first developed in the 1940s by mathematicians working on the Manhattan Project to build an atomic bomb.

In the case of the Monty Hall problem, the simulation is straightforward to program and we can figure out an analytical result, so it's easy to inspect the output and verify that the program is correct. Often, though, simulations are for attacking problems too complicated to be solved beforehand and then checking for correctness is much harder. The programmers will need to carefully validate their code by unit-testing the simulation's internal functions and adding checks for internal correctness. `monty-hall.py` does this in a small way with an `assert` statement that will raise an exception if the number of closed doors is not equal to 2, which would indicate some sort of failure in the **`simulate()`** function's logic or input data. We could potentially add more such checks:

```
assert 0 <= winning_door < num_doors, 'Winning door is not a legal door'
assert 0 <= choice < num_doors, 'Contestant choice is not a legal door'
```

In our simple simulation, these assertions are never going to fail, but perhaps we might make changes to **`simulate()`** that make the function incorrect, or perhaps someday a bug in Python's **`random`** module will come to light.

References

http://en.wikipedia.org/wiki/Monty_Hall_problem
Discusses the history of the problem and various approaches to solving it.

http://library.lanl.gov/cgi-bin/getfile?00326867.pdf
"Stan Ulam, John von Neumann, and the Monte Carlo Method" (1987), by Roger Eckhardt, is a discussion of the very first Monte Carlo simulation and some of the mathematical problems encountered while implementing it. This first simulation modelled neutrons diffusing through fissionable material in an effort to determine whether a chain reaction would occur.

(The PDF also features a discussion of how random number generators work, written by Tony Warnock.)

http://www.youtube.com/watch?v=0W978thuweY
A condensed episode of an episode of the original game show, showing Monty Hall's quick wit in action. Notice that the original game is more complicated than the Monty Hall

puzzle as described above because Monty has many more actions available to him: he can offer the choice of an unknown prize or an unknown amount of cash, or suggest trading in what you've already won for a different unknown prize.