Simulation for fun and profit

One really useful thing that you can do with programming skills is simulate things that you don't fully understand. This is something that hard science people do all the time: it turns out that there are some kinds of problems that aren't solvable in a deductive, analytic kind of way, but if you can write a simulation of some of the broad influences on the problem, then you can sometimes get a good idea what's going on.

For example, John Conway's famous Game of Life can be understood as a representation of an evolutionary process and has been the object of tons of study via computer simulation.

In statistics there are many many examples of simulation that actually help people learn more about data. For example, there's a technique called the "bootstrap" that involves taking your data and generating synthetic samples from it via simulation, which statisticans can use to squeeze out more information from the original data. There's also a wide array of techniques called Monte Carlo simulations in statistics that can be used to do freakishly powerful stuff. Simulation is great.

However, for our purposes in this class, the most useful thing about simulations is how they allow us to explore how probability affects real-world results. It turns out that a lot of truths in probability are very counterintuitive to our unaided imagines, but are really easy to simulate using a computer, and, due to a very happy mathematical fact called the law of large numbers (which we'll learn more about in the stats section of the course), as you repeat a probabilistic simulation a lot of times (which a computer can do very fast), you're likely, in many cases, to be able to observe behavior that reflects the aggregate behavior that we'd expect to see in the real world.

Here's an example. There's a very famous probability problem called "Monty Hall." I have some more details about it in a later lesson, but here's the crux of it. There was an old game show called "Let's Make a Deal" (Monty Hall was the host) and one the games they played involved presenting the contestant with three closed doors. Behind one of the door was a car, and behind the other two was a booby prize, like a goat or something. (I dunno, goats are cute, I'd be cool with that.)

The contestant would choose a door, and then Monty Hall would open one of the remaining doors not chosen by the contestant and reveal a goat. Then Monty Hall would ask the contestant: do they want to change with their original door, or do they want to switch? Then, after the contestant makes their final choice, Monty reveals what's behind the door they landed on, and the contestant gets the prize.

Pause for a moment, and think about the game. Can you figure out a good strategy for it? Should the contestant always stick with their original door?

Should they always switch? Should they flip a coin? Does it matter at all? Think about it, and try to justify your answer. I'll be back below…

…

…

…

…

…

…

…

…

…

…

…

…

…

…

…

…

…

…

…

…

…

…

Almost everyone who hasn't seen this problem concludes that it doesn't matter—stick, switch, flip a coin, either way, you have a 1 in 3 chance of getting the car. Actually, this is wrong—the correct strategy is to always switch to the last closed door after Monty opens a door with a goat. (But don't feel bad if you didn't realize that, like I said, almost nobody who doesn't know the problem realizes that, including math professors.)

Suppose you don't believe me. This is really counterintuitive! How can it possibly make a difference, we start the game with one goat and three doors?!!!?!?? Well, this is a probability problem, so we can simulate it—and taking a look at the results might convince you that maybe I'm telling you the truth after all.

Here's some code to simulate the Monty Hall problem. It uses object-oriented programming concepts; please see the lesson about object-orientation to understand what's going on here. It turns out that object-oriented programming, as a style of writing code, is really well suited to simulations, because you can do things like make a player and its strategies one class, and a round of the game another class, and then just have them interact in order to produce your result.

```
import random

class Round(object):
    def __init__(self):
        self.doors = ["a", "b", "c"]
        self.car = random.choice(self.doors)

    def pick(self, door):
        self.player_choice = door

    def open_door(self):
        available_to_open = [x for x in self.doors if x not in [self.player_choice, self.ca
        return random.choice(available_to_open)

    def evaluate(self):
        return self.player_choice == self.car  # true if player chose the car, false if not


class Player(object):
    def __init__(self, strategy):
        self.strategy = strategy

    def first_pick(self):
        doors = ["a", "b", "c"]
        first_choice = random.choice(doors)
        self.first_choice = first_choice
        return first_choice

    def final_pick(self, open_door):
        switch_choice = [x for x in ["a", "b", "c"] if x not in [self.first_choice, open_do
        if strategy == "stick":
            return self.first_choice
        elif strategy == "switch":
            return switch_choice
        else:  # default strategy is assumed to be "random"
            return random.choice([self.first_choice, switch_choice])


class MontySimulation(object):
    def __init__(self, player):
        self.player = player
        self.wins = 0

    def report(self):
        percentage_won = 100 * self.wins / (self.rounds)
        print("Player with {} in {} rounds won {}% of rounds".format(self.player.strategy, s

    def play(self, rounds):
        self.rounds = rounds
        for r in range(rounds):
            current_round = Round()
            first_pick = self.player.first_pick()
            current_round.pick(first_pick)
            opened = current_round.open_door()
            final_pick = self.player.final_pick(opened)
            current_round.pick(final_pick)  # it's perfectly ok to call the method a second
            won = current_round.evaluate()
            if won:
                self.wins += 1
        self.report()
```

I've added some comments to the code to make it a bit more readable, but I won't explain it line by line (maybe in class). I want you to try to make sense of it, and then figure out how to run it. (I'll post an explanation of how to run it after a few lines of whitespace.)

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

Here's a good way to run this code, at 10000 times per strategy, which should run very quickly and is highly likely to produce reports reflecting the underlying probabilities of winning. As you can see if you run this code, switching is a much better strategy than sticking!

```
strategies = ["stick", "switch", "random"]
for strategy in strategies:
    player = Player(strategy)
    game = MontySimulation(player)
    game.play(10000)
```

We will talk about Monty Hall later on in the course, and I'll give you the mathematical reasoning behind the correct strategy.

Incidentally, while object-oriented programming is a good match for simulation, it isn't necessary. To prove it, here's another Monty Hall simulation I wrote a couple years in a programming language that doesn't even support object-oriented programming. The part that does the simulation actually comes out shorter than the example above (although the linked code also generates a webpage to see the simulation live, which you can play with here if you want). If you want to see what the same simulation looks like in another language, go check it out… I'll bet you can understand the broad outlines, even though you've never been exposed to the other language.

In the remainder of this class, I will liberally use simulation to demonstrate important concepts in probability and statistics. I encourage you to poke around in those simulations, change their assumptions (for example, can you figure out a way to change the rules of Monty Hall to change the player's optimal strategy? If you can, simulate it to check!), and poke around the edges in order to get a feel for the knowledge they represent. I think you'll find that being able to dig your hands into a probability/statistics process will make it more understandable and intuitive than just looking at some mathematical pr