Probability Simulations

A so-called Monte Carlo simulation is a method for analysing the outcome of certain random events (such as casino games) using the capacity of computers to simulate random numbers and perform many trials.

A simulation gives a good picture of what is most likely to happen in reality, provided:

- ***The simulation accurately models the true situation.***
- ***The simulation runs a sufficiently large number of trials.***

A classic 'gambling strategy' which is at the root of many so-called 'sure-fire systems' is the Martingale Strategy. In its simplest form, it should ensure you win, eventually, for any game of chance with a non-zero chance of winning, provided you can continue to play:

1) Bet your initial stake (eg £1).
   If you win, you'll win £1 and be £1 better off. If not…
2) Double your stake (eg £2).
   If you win, the £2 winnings cover your losses, and you're £1 better off. If not…
3) Repeat step 2 until you eventually win.
   Doubling your stake every time ensures you cover all previous losses, plus £1.

In order to accurately simulate this without risking real money, we can make use of a computer simulation. One very important element to consider is the total bankroll of the player – clearly, there will come a point when doubling your stake is no longer an option, because you are broke, or reached the point where you can't justify risking any more.

The catch:
The maths of the Martingale Strategy checks out. There's a 50% chance that you win your stake after 1 game (assuming a simple fair game). There's a 25% chance that you lose the first round, but you win the second, and a 12.5% chance that it takes you 3 rounds, etc.

Since each of these is mutually exclusive, we can find the probability of winning *eventually* by calculating the value of the infinite sum:

$$\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\cdots$$

Tip: to find this sum, notice that *doubling* it gives you $1+\frac{1}{2}+\frac{1}{4}+\cdots$, which is one more than the original sum. Solve $2S = 1 + S$ to find the value.

So what's the catch? There's a 100% chance of *eventually* winning, but in reality nobody has an unlimited bankroll. What's the chance of winning within 10 rounds or less (risking a maximum stake of £1024 if our original stake was £1)? We can calculate it using this:

$$\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\cdots+\frac{1}{1024}$$

It's close to 100%, but not quite 100%. The question is, would you bet £1000 for a 99% chance of winning £1? It's a near-certain bet, but is the return worth the potential loss?
Turn over to see how we can simulate variations on this with Python…

Probability Simulation *with Python*

Because we want to run a large number of trials, it would be helpful to create a dedicated function which automatically runs a simulation of a given game, with a particular stake:

```
1   import random
2
3   def gamble(stake, p, winnings=None):
4     if winnings == None:
5       winnings = stake / p
6     if random.random() < p:
7       return winnings - stake
8     else:
9       return - stake
```

We will be relying on the pseudo-random numbers provided by Python's **random** module, so import that at the very beginning.
Define the function, which will allow you to bet a certain amount (your **stake**), on a game with a certain probability of winning (**p**), with an *optional argument* **winnings** which defaults to **None**, determining how much you win if you win. The optional 'keyword argument' allows us to define the winnings for ourselves if we choose, but if not, the function has default behaviour built-in (in this case, it uses **stake / p** to make the game perfectly fair).

The function works by using **random** to decide if the game is won or lost. If lost, the function returns a negative value (**– stake**), because you have lost the money bet. If won, the function returns **winnings – stake** (you have spent **stake** but won **winnings**).

Test your function out by trying some simple trials to begin with. Notice how I can start with a certain amount of cash, and repeatedly update its value using my **gamble** function:

```
11  cash = 100
12  cash += gamble(50, 0.5)
13  print(cash)
14
15  cash += gamble(cash / 2, 0.1)
16  print(cash)
17
18  cash += gamble(20, 0.3, 50)
19  print(cash)
```

We provide our simulated player with £100, and he proceeds to bet £50 on a game with probability of winning 50%. His cash is increased by the return amount, which means it either goes up £50, or down £50.
After this, he gambles half his remaining cash on a game with a 10% chance of success. He either loses it, or gets back $10 \times$ his stake.
Finally, he gambles £20 on a 30% chance to win £50.
Try running this a few times to see what happens.

The real power of our code lies in the ability to run a full simulation. The Martingale Strategy would look something like this:

```
11  def martingale(bankroll, initial_stake):
12    stake = initial_stake
13    while bankroll >= 0:
14      result = gamble(stake, 0.49, winnings=stake*2)
15      bankroll += result
16      if result > 0:
17        return bankroll
18      stake *= 2
19      if stake > bankroll:
20        return bankroll
21
22  for i in range(40):
23    print(martingale(1000, 10) - 1000)
```

This function lets us decide how much we can afford to risk, and how much our initial stake should be. Remember, for Martingale, the final amount you win is fixed, but the only limit to what you can lose is your initial bankroll.
This version pays double if you win, but, like a roulette wheel, has a 49% chance of winning each time. If you win, you stop, and if you can't afford to double your stake, you stop. Repeat a few times using a **for** loop and see what you notice.

Simulating Randomness continued…

A *random walk* is a journey, typically in 2 dimensions, where the direction is determined randomly. We can use Python Turtle to visualise this:

```
1   import turtle, random
2
3   angles = [-10, 10]
4
5   def random_walk(t):
6     for i in range(40):
7       direction = random.choice(angles)
8       t.fd(10)
9       t.lt(direction)
10
11  bob = turtle.Turtle()
12  random_walk(bob)
```

We'll use the **random.choice** function from the **random** module, which selects at random from the given list. Try messing with the options – I've given options of either $10°$ or $-10°$.

The **random_walk** function moves the turtle forward 10 pixels, then turns either $10°$ left or $-10°$ left (that is, $10°$ right), repeating 40 times.

It may seem overkill to define a function specifically for this task, but if we want to send a whole flock (herd? Nest. Probably.) of turtles on a random walk, it will come in handy:

```
14  num_turtles = 10
15  turtles = []
16  for i in range(num_turtles):
17    turtles.append(turtle.Turtle())
18
19  for turtle in turtles:
20    random_walk(turtle)
```

This starts by creating a list of 10 turtle objects. With a nest of turtles this large (or bale? Possibly bale of turtles), we give them numbers rather than names.
In Python, we can just loop through the whole list, calling each one 'turtle' as we go if we like.

If, instead of turning a certain amount, we set the **heading** randomly to one of two different possibilities, we can create a moving tree diagram of turtles. See where they end up…

```
1   import turtle, random
2
3   angles = [20, 40]
4
5   def random_walk(t):
6     for i in range(8):
7       heading = random.choice(angles)
8       t.fd(30)
9       t.seth(heading)
10
11  num_turtles = 10
12  turtles = []
13  for i in range(num_turtles):
14    turtles.append(turtle.Turtle())
15
16  for turtle in turtles:
17    random_walk(turtle)
```

This version gives choices of 20 or 40 degrees heading, which means at each point the turtle will choose a heading at random, and move forward a certain amount in that direction.

You can experiment with changing the colour of each path using **t.pencolor(r, g, b)** where **r**, **g** and **b** are set using **random.randint(0, 255)**. You can also set the width of the lines to be different for each turtle so that the overlapping paths are more obvious.

Binomial Simulation

```
1   import random
2
3   def binomial(n, p):
4     successes = 0
5     for i in range(n):
6       if random.random() < p:
7         successes += 1
8     return successes
9
10  print(binomial(20, 0.5))
```

The binomial probability distribution deals with a fixed number of independent trials, with a fixed probability of success (eg tossing a coin 20 times).

This **binomial** function takes as its input **n** and **p**, the number of trials and the probability of success. The loop within runs each trial in turn, keeping track of how many times the trial is successful.
*Note that **random.random()** returns a uniformly distributed random number from 0 to 1, so it will be below, say, 0.7 around 70% of the time.*

We can get a bit more sophisticated by running many of these experiments. In the example above, we toss a fair coin 20 times and record the number of Heads. If we do this 100 times, how often would you expect to get exactly 10 Heads? More than 12? Less than 5?

```
1   import random
2
3   def binomial(n, p):
4     successes = 0
5     for i in range(n):
6       if random.random() < p:
7         successes += 1
8     return successes
9
10  num_trials = 20
11  p_success = 0.5
12
13  results = {}
14  for i in range(100):
15    num = binomial(num_trials, p_success)
16    if num in results:
17      results[num] += 1
18    else:
19      results[num] = 1
20
21  for i in range(num_trials + 1):
22    if i in results:
23      graph = "x" * results[i]
24      print(f"{i: 3}: {graph}")
25    else:
26      print(f"{i: 3}: ")
```

This code records results in a **dictionary**, which lets us record results in *key, value* pairs.

*Use **print(results)** somewhere in your code to see what it looks like.*

It should add an entry every time a new result comes out. For instance, the first time the **binomial** function gives us a 10, the dictionary will have **10: 1** written into it. The next time, since **10** is in the dictionary (it's one of the keys), the value (1) will be increased by 1 to give **10:2**.

The final bit of code is a nice way to visualise your results. We could just print out the numbers (just print **results[i]** instead of **graph**), which would be helpful if we did this many times, but a string of **x** characters effectively plots a graph for us.

Experiment with the binomial function, trying different numbers of trials and different probabilities to visualise the shape of the resulting distribution.
Eg:

- A biased coin shows heads 55% of the time. How many throws would someone need to make before they could be pretty confident that the coin was really unfair?
- How likely is it that, if I roll a fair 6-sided die 30 times, that I get exactly 5 sixes?
- If a train has a 10% chance of being delayed on any given day, what is the chance of going 10 days without having any delays?