Simulation – Experiments and Trials

15-110 – Monday 11/23

Learning Goals

- Recognize and use methods from the **random library** to implement randomness

- Use **Monte Carlo methods** to estimate the answer to a question

- Organize **animated simulations** to observe how systems evolve over time

Randomness

Random Library

Sometimes we want the computer to generate random numbers, for use in simulation, testing, or other purposes. We'll do this using the built-in library `random`.

The function `random.randint(start, end)` picks a random number between `start` and `end`, inclusive.

```
import random
num = random.randint(1, 10)
```

Other Random Functions

There are two more functions that act on **lists** which will be useful for producing randomness in code:

```
lst = [ "A", "B", "C", "D", "E"]
random.choice(lst) # chooses an element randomly

random.shuffle(lst) # destructively shuffles the list
```

Most functions in the random library are based on `random.random()`, which generates a random floating point number in the range [0.0, 1.0).

Activity: Pick a Random Functions

You do: which function would you use to simulate...

... flipping a coin?

... rolling a die?

... drawing a card?

Common Random Actions

Q: How would you **flip a coin** using Python's random library?

A: Use `random.choice(["Heads", "Tails"])`

Q: How would you **roll a die**?

A: Use `random.randint(1, 6)`

Q: How would you **draw a card**?

A: Make a deck, use `random.shuffle(deck)`, and check `deck[0]`.

Computing Randomness

How is it possible for us to generate random numbers this way?

Randomness is difficult to define, either philosophically or mathematically. Here's a practical definition: given a truly random sequence, there is **no gambling strategy possible** that allows a winner in the long run.

But computers are **deterministic** – given an input, a function should always return the same output. Circuits should not behave differently at different points in time. So how does the random library work?

True Randomness

To implement truly random behavior, we can't use an algorithm. Instead, we must gather data from **physical phenomena** that can't be predicted.

Common examples are atmospheric noise, radioactive decay, or thermal noise from a transistor.

This kind of data is impossible to predict, but it's also slow and expensive to measure.

Pseudo-Randomness

Most programs instead use **pseudo-random numbers** for casual purposes. A **pseudo-random number generator** is an algorithm that produces numbers which look 'random enough'.

These algorithms work by taking as input a number x_i and running it through an algorithm to calculate x_{i+1}. For the next random number, the function uses x_{i+1} as input to generate x_{i+2}, and so on.

By calling the function repeatedly, the algorithm generates a **sequence** of numbers that appear to be random to the casual observer.

Pseudo-Randomness is 'Good Enough'

The number sequence generated by a pseudo-random number generator isn't *truly* random; if someone figures out the algorithm, they can predict the results. But it is random enough to use for casual purposes.

Python's random API uses an algorithm called the Mersenne Twister to generate pseudo-random numbers.

Monte Carlo Methods

Randomness in Simulation

Most simulations use randomness in some way; otherwise, every run of the simulation will produce the same result.

This randomness means that the same simulation might have multiple different outcomes on the same input model. A single run of a simulation is not a good estimate of the true average outcome.

To find the truth in the randomness, we need to use probability!

Law of Large Numbers

The Law of Large Numbers states that if you perform an experiment multiple times, the average of the results will approach the **expected value** as the number of trials grows.

This law works for simulation as well! We can calculate the expected value of an event by simulating it a large number of times.

We call programs that repeat simulations this way **Monte Carlo methods**, after the famous gambling district in the French Riviera.

Monte Carlo Method Structure

If we put our simulation code in the function `runTrial()`, and want to find the odds that a simulation 'succeeds', a Monte Carlo method might take the following format:

```
def getExpectedValue(numTrials):
    count = 0
    for trial in range(numTrials):
        result = runTrial() # run a new simulation
        if result == True:  # check the result
            count = count + 1
    return count / numTrials # return the probability
```

Monte Carlo Example

Every year, SCS holds the Random Distance Race. The length of this race is determined by rolling two dice. **What is the expected number of laps a runner will need to complete?**

```
import random
def runTrial():
    return random.randint(1, 6) + random.randint(1, 6)

def getExpectedValue(numTrials):
    lapCount = 0
    for trial in range(numTrials):
        lapCount += runTrial()
    return lapCount / numTrials
```

Activity: Monte Carlo Methods

You do: what are the odds that a runner in the Random Distance Race will need to run 10 or more laps?

Write the code to run the trial. You can modify the code from the previous slide.

Advanced Simulations

Designing a Simulation

We now have all the individual parts of a simulation. All that remains is to **combine** these components to design a useful simulation.

Let's do an advanced example by simulating a **zombie outbreak**. Our goal will be to determine how long it takes for the whole world to become zombies based on different zombie infection rates.

A zombie infection rate is how likely you are to become a zombie if you encounter a zombie. In other words, how effective are the zombies?

Warning: prepare for a lot of code!

Zombie Outbreak Model

Let's simulate our world as a 2D grid. Zombies will move around, but humans will stay still (they're hiding).

Model: start with 20 humans and 5 zombies. Also start with an infection rate.

View: humans and zombies will both be squares. Humans are green, zombies are purple.

Rules: every step, move each zombie one square in a random direction on the grid. If a zombie is touching (bordering) a human, use the infection rate to determine if the human is turned into a zombie.

Programming the Model

```
def makeModel(data):
    data["rate"] = 0.5 # 50% chance a human becomes infected on contact
    data["size"] = 20  # grid is 20 x 20
    # A 'creature' has a row, a column, and a species- human or zombie
    data["creatures"] = [ ]
    # Start with 20 humans and 5 zombies randomly placed
    for human in range(20):
        data["creatures"].append({ "row" : random.randint(0, 19),
                                   "col" : random.randint(0, 19),
                                    "species" : "human" })
    for zombie in range(5):
        data["creatures"].append({ "row" : random.randint(0, 19),
                                   "col" : random.randint(0, 19),
                                   "species" : "zombie" })
```

Programming the View

```
def makeView(data, canvas):
    # Draw an underlying grid
    cellSize = 20
    for row in range(data["size"]):
        for col in range(data["size"]):
            canvas.create_rectangle(col*cellSize, row*cellSize,
                                    (col+1)*cellSize, (row+1)*cellSize)
    # Then draw creatures on top
    for creature in data["creatures"]:
        row = creature["row"]
        col = creature["col"]
        if creature["species"] == "human":
            color = "green"
        else:
            color = "purple"
        canvas.create_rectangle(col*cellSize, row*cellSize,
                                (col+1)*cellSize, (row+1)*cellSize, fill=color)
```

Programming the Rules – Zombies Move

```
def runRules(data, call):
    zombies = []  # For checking if bordering with humans
    for creature in data["creatures"]:
        if creature["species"] == "zombie":
            zombies.append(creature)
            # Move in a random direction
            move = random.choice([[-1, 0], [1, 0], [0, -1], [0, 1]])
            creature["row"] += move[0]
            creature["col"] += move[1]
            # Make sure they don't move offscreen!
            if not onscreen(creature, data["size"]):
                creature["row"] -= move[0]
                creature["col"] -= move[1]

# Need to be within both the width and the height
def onscreen(creature, size):
    return 0 <= creature["row"] < size and 0 <= creature["col"] < size
```

Programming the Rules – Infecting Humans

```
def runRules(data, call):
                ...
    for creature in data["creatures"]:
        if creature["species"] == "human":
            # Check if any zombie is touching this human
            for zombie in zombies:
                if bordering(creature["row"], creature["col"],
                             zombie["row"], zombie["col"]):
                    odds = random.random() # roll the dice, figuratively
                    if odds < data["rate"]:
                        creature["species"] = "zombie" # zombify!

def bordering(row1, col1, row2, col2):
    # Bordering if in the same row and at most one col apart or vice versa
    if row1 == row2 and abs(col1 - col2) <= 1:    return True
    elif col1 == col2 and abs(row1 - row2) <= 1:    return True
    else:    return False
```

Programming the Rules – Detecting The End

```
def runRules(data, call):
    if allZombies(data["creatures"]):
        print(call) # number of 'days' that have passed
        exit() # this exits the program
    ...

def allZombies(creatures):
    for creature in creatures:
        if creature["species"] == "human":
            return False # any humans? not done yet
    return True
```

Using Simulations

Once we've programmed a robust simulation, we can **change the starting state** to see how it changes the simulation. This is especially useful when we want to **predict** certain things about the world.

We can check predictions more quickly by making `timeRate` smaller (calling the simulation more often).

For example: how long will it take for the whole world to become zombies...

- In our current code?
- If we start with more or fewer humans?
- If we start with a higher infection rate?

Sidebar: Calculating Outcomes

If we want to explore the simulation, we can run it with the visualization on.

If we just want to find the **average results**, we can move the `makeModel` and `runRules` code to all take place in a single function (where the time loop becomes a while loop). Have that function return the number of days it takes to zombify all the humans.

When we run this function with `getExpectedValues`, we can find the expected amount of time left for the human race.

Sidebar: Calculating Outcomes Code

```
def runTrial():
    data = { }
    makeModel(data)
    daysPassed = 0
    while not allZombies(data["creatures"]):
        ... # put runRules body here
        daysPassed += 1
    return daysPassed

def getExpectedValue(numTrials):
    lapCount = 0
    for trial in range(numTrials):
        lapCount += runTrial()
    return lapCount / numTrials

print(getExpectedValue(100))
```

Learning Goals

- Recognize and use methods from the **random library** to implement randomness
- Use **Monte Carlo methods** to estimate the answer to a question
- Organize **animated simulations** to observe how systems evolve over time
- **Feedback:** https://bit.ly/110-feedback