#5-3: Experimentation

CS SCHOLARS – PROGRAMMING

Announcements

- Will review `drawPixelArt` / `gradebookSummary` at beginning of class tomorrow

- What should we cover for the bonus Tuesday lecture? Will run a poll on Slack
  - Options include: CS ethics, recent CS trends, CS history, efficiency, dictionaries, limits of computation, and more!

Learning Goals

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

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

Randomness

Random Functions

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

Recall the random library, which we learned about early in the program. This module included several useful functions we can use:

```
random.random() # pick a random float between 0-1

random.randint(x, y) # pick a random number in a range


random.choice(lst) # chooses an element randomly
random.shuffle(lst) # destructively shuffles the list
```

Computing Randomness

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

Randomness is difficult to define, either philosophically or mathematically. Here is 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'. Each number the algorithm generates acts as a starting place to generate the next one.

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

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.

Monte Carlo Methods

Randomness in Simulation

Using randomness in a simulation 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
```

Another Monte Carlo Example

Here's a more complicated example. If we draw a hand of five cards, what are the odds that that hand forms a **straight** in Poker (five card values in a row, like 7-8-9-10-Jack)?

First, we need to represent a deck of cards. We'll use a list, and each card will be a two-element list (suit and value). We can 'draw' five cards from the deck by shuffling the deck and slicing the first five values.

```
def generateDeck():
    deck = []
    for suit in ["Club", "Diamond", "Heart", "Spade"]:
        for value in range(2, 15): # 2 to Ace (14)
            deck.append([suit, value])
    return deck
```

Another Monte Carlo Example

To test whether a hand is a straight, extract the card values, sort them, and check whether each value is exactly one smaller than the next in the list.

```
def isStraight(hand):
    values = []
    for card in hand:
        values.append(card[1])
    values.sort()
    for i in range(len(values)-1):
        if values[i] != values[i+1] - 1:
            return False
    return True
```

Another Monte Carlo Example

To calculate the odds, just keep shuffling the deck and drawing cards!

```
def calculateOdds(trials):
    count = 0
    deck = generateDeck()
    for trial in range(trials):
        random.shuffle(deck)
        hand = deck[:5] # first five cards
        if isStraight(hand):
            count += 1
    return count / trials
```

Activity: Monte Carlo Methods

You do: what are the odds that a hand in Poker forms a **flush** (five cards of the same suit)? Do those odds change if we play Poker with hands of six cards instead of five?

Write the code to find the odds of this happening. You can start from the code we wrote in the previous slides.

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**.

Goal: we want to determine **how many days** 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 in random locations. Also start with an infection rate.

Rules: every second, 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.

View: humans and zombies will both be squares. Humans are green (healthy), zombies are purple (infected).

Programming the Model

```
def init(data):
    data.rate = 0.5 # 50% chance a human becomes infected on contact
    data.size = 20 # grid is 20 x 20
    data.numCalls = 0 # track number of days passed
    # 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):
        # each creature has a row, col, and species in a list
        data.creatures.append([ random.randint(0, data.size-1),
                                random.randint(0, data.size-1), "human" ])
    for zombie in range(5):
        data.creatures.append(([ random.randint(0, data.size-1),
                                 random.randint(0, data.size-1), "zombie" ])
```

Programming the View

```
def redrawAll(canvas, data):
    # Draw an underlying grid
    cellSize = 400 / data.size # 400 is the window size
    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[0]
        col = creature[1]
        species = creature[2]
        if 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 timerFired(data):
    zombies = [] # To check if zombies are close to humans
    for creature in data.creatures:
        if creature[2] == "zombie": # species
            zombies.append(creature)
            # Move in a random direction
            move = random.choice([[-1, 0], [1, 0], [0, -1], [0, 1]])
            creature[0] += move[0] # row
            creature[1] += move[1] # col
            # Make sure they don't move offscreen!
            if not onscreen(creature, data.size):
                creature[0] -= move[0]
                creature[1] -= move[1]

# Need to be within both the width and the height
def onscreen(creature, size):
    return 0 <= creature[0] < size and 0 <= creature[1] < size
```

Programming the Rules – Infecting Humans

```
def timerFired(data):
                ...
    for creature in data.creatures:
        if creature.species == "human":
            # Check if any zombie is touching this human
            for zombie in zombies:
                if bordering(creature[0], creature[1],
                             zombie[0], zombie[1]):
                    odds = random.random() # roll the dice, figuratively
                    if odds < data.rate:
                        creature[2] = "zombie" # zombify!

# If in the same row and at most one apart, you're bordering
def bordering(row1, col1, row2, col2):
    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 timerFired(data):
    data.numCalls += 1 # each call is a 'day'
    if allZombies(data.creatures):
        print(data.numCalls) # number of 'days' that have passed
        exit() # this exits the program
    ...

def allZombies(creatures):
    for creature in creatures:
        if creature[2] == "human":
            return False # any humans? not done yet
    return True
```

Experimenting with Simulations

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 `rate` 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?

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 call the `init` and `timerFired` functions from a new 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 find the expected amount of time left for the human race. Monte Carlo solves the problem!

Calculating Outcomes Code

```
def runTrial():
    data = Model()
    init(data)
    while not allZombies(data.creatures):
        timerFired(data)
    return data.numCalls

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

print(getExpectedValue(100))
```

Learning Goals

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

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