

Catalyst conversation on AI

IIT ROPAR - Minor in AI

23rd Jan, 2025

Case Study: Rolling a Dice

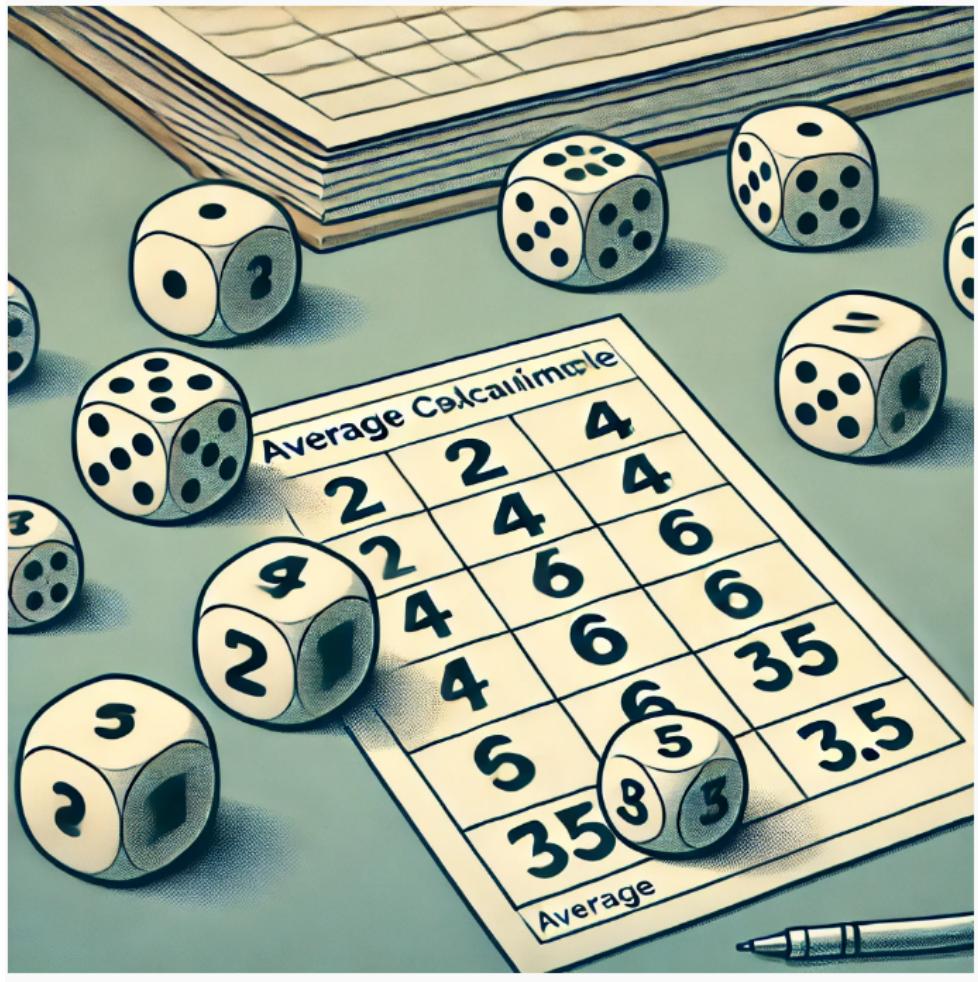


Figure 1: Dice rolls

Random Number Experiment

Let us imagine a class activity where students roll a dice 100 times each. After completing their rolls, they record the numbers and calculate the average. One student might roll these numbers: 2, 4, 6, 3, 5, and so on. Adding them up gives a total, for example, $2 + 4 + 6 + 3 + 5 + \dots = 350$. Dividing this total by 100 rolls, the average is $350 \div 100 = 3.5$. Interestingly, when other students complete the same activity, their averages also come very close to 3.5.

The teacher repeats this experiment multiple times and finds that the average remains consistent around 3.5. This observation raises the question: why does this happen?

Why does the average come out to 3.5?

Let us take an example. If you roll a dice a few times, you might get numbers like 1, 4, 6, 3, and 5. Adding these numbers gives you $1 + 4 + 6 + 3 + 5 = 19$. Now divide this sum by the number of rolls, which is 5. The average becomes $19 \div 5 = 3.8$. The next time you roll the dice, you might get different numbers like 2, 3, 5, 6, and 4. This time, the total is $2 + 3 + 5 + 6 + 4 = 20$, and the average is $20 \div 5 = 4.0$. If you keep repeating this process, you will see the averages getting closer and closer to 3.5.

What is special about 3.5?

The number 3.5 is interesting because it is exactly halfway between the smallest number (1) and the largest number (6). For example, if you add all the numbers from 1 to 6, you get $1 + 2 + 3 + 4 + 5 + 6 = 21$. Divide this by 6 (the total number of numbers), and you get $21 \div 6 = 3.5$. This is why 3.5 keeps appearing as the average when you roll a dice many times.

Real-Life Connection

Think of a situation where you distribute candies to children. If each child gets between 1 and 6 candies, and you do this for a large group of children, on average, each child will get around 3 or 4 candies. This shows how averages work in everyday scenarios.

Heat Map from Pin Code Data

A survey was conducted with 500 students, and their pin codes were collected. These pin codes were used to find their locations. These locations were then marked on a map. The result was a type of map called a heat map.

What is a heat map?

A heat map is a visual way to show how concentrated something is in different areas. For example, if many students come from one specific area, that part of the map will appear bright or intense in color. If fewer

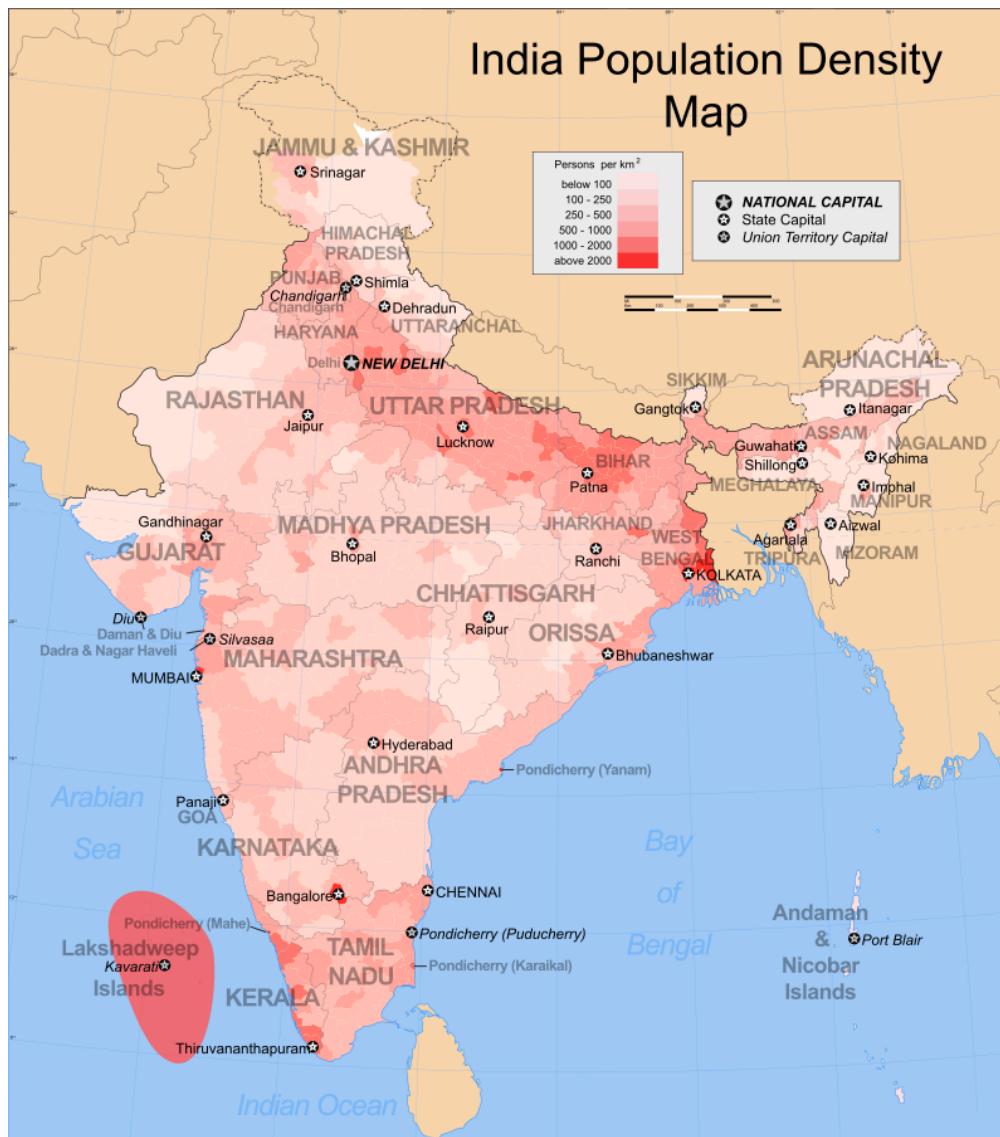


Figure 2: Example of a Heat Map showing population density in different areas.

students come from another area, it will appear lighter or less intense.

Example of a Heat Map

Imagine you are marking the houses of 500 students on a city map. If 100 students live in one neighborhood, that area will look very bright on the heat map. If only 5 students live in another neighborhood, that part of the map will look faint. This way, the heat map makes it easy to understand where most students are located.

Properties of the Heat Map

- **High concentration areas:** Bright or intense colors show where most students are located.

- **Low concentration areas:** Light or faint colors show where fewer students are located.
- **Clear distribution:** It provides a clear visual of how students are spread across different locations.

Real-Life Uses of Heat Maps

Heat maps are not just for students. They are used in many fields. For example:

- **Traffic Analysis:** To show where traffic jams happen most often.
- **Weather Reports:** To display temperature or rainfall in different areas.
- **Business Planning:** To identify where most customers are located.

This helps people make better decisions based on the data.

The Magic of Prompting: Getting Started with AI

IIT Ropar – Minor in AI

25th Jan, 2025

1. The Birthday Surprise: More Common Than You Think!

Imagine a classroom full of students, just like you. Now, if I asked you, "Do you think two people in this room share the same birthday?" you might instinctively say "Nah, that's pretty unlikely!" But here's a mind-bender: if there are just 23 people in a room, there's a surprisingly high chance that two of them will have the same birthday!

It's called the "Birthday Paradox," but it's not really a paradox. It's a quirky way of seeing how probabilities work. The chances of one person sharing your birthday are low. But the chances of any two people in a group sharing any birthday are much higher than you'd think.

Now, this is not just a fun fact; it's our starting point for understanding how to talk to computers. It might seem weird that something like the birthday paradox could be related to computers. However, this simple idea will help introduce you to the main point of this lesson: the power of asking effectively! We'll show you how to ask a computer to do amazing things using something called "prompting".



Figure 1: A common birthday!

2. Your First Tool: *Google Sheets*

Before we start asking computers to do complex things, let's get familiar with a simple tool: Google Sheets. Think of it like a super-powered notebook that lives on the internet. It's a great way to store data, play around with it, and see it in action.

Think of Google Sheets like this:

Online Notebook: Access it from any device.

Data Organizer: Store and manage numbers, text, and more.

Visual Helper: See your data with charts and graphs.

Smart Formulas: Use built-in functions to make calculations and more.

We can do something basic: sort some numbers. It's like arranging books on a shelf. Just type some numbers in a column. Google Sheets can also make random numbers for you using =RANDBETWEEN(1,100) which means a random number between 1 and 100 will appear in the cell. Copy this formula to other cells, and each will give you a new random number!

To sort the numbers, we can use the SORT() formula: =SORT(A1:A10). This will take the numbers from cell A1 to A10 and show you the sorted values in the next column. If you add more numbers to column A, the sorted numbers will also update automatically!

	A	B
1	61	12
2	68	13
3	48	13
4	67	48
5	13	58
6	97	61
7	70	67
8	58	68
9	12	70
10	13	97

Figure 2: Sorted values, using Google Sheets

3. The Art of Asking: Prompting with Google Sheets

Now, let's try something a little more challenging. Let's say, we want to highlight numbers that appear more than once in our sheet. This is where "prompting" comes in. "Can you highlight the duplicate numbers in my sheet" might sound simple enough for a human, but a computer needs more specific instructions.

This is where we use the power of effective prompting. Instead of searching through the cells manually, we can simply ask the google sheets to do it for us by creating small programs for it.

There are two ways to do it:

1. Using Google Apps Script (A Bit of Programming):

Google Sheets lets us add programs, or "scripts", using something called Google Apps Script. We can use it to automate tasks and ask the computer to do what we want. It might sound like rocket science, but with AI assistance, anyone can do it!

We can ask an AI like ChatGPT or Gemini for help. For example, we can prompt: "I have some numbers in column B of a Google Sheet. If any two consecutive cells have the same number, please highlight it. Give me a Google script for it." The AI will then give you a piece of code that does what you asked for.

Then, copy the script, go to 'Extensions' then 'Apps Script' in Google Sheets. Paste your code into the editor, save it, and click 'Run'. Now, your Google Sheet will automatically highlight duplicate entries.

2. Using Conditional Formatting (No Programming Needed!):

Alternatively, you can use a built-in feature called Conditional Formatting. No need to write code here. Again, we can ask an AI something like "Is there any way I can do this without using Google script?" and it will find a solution!

The AI might give a response like "Yes, you can do it using conditional formatting. Here's how." It will probably suggest something like this formula: $=\text{AND}(B1=B2, B1<>"")$. We can copy this formula, go to 'Format', 'Conditional Formatting' and paste it in the custom formula section. Now your sheet will highlight the duplicates automatically!

	A	B
1	61	12
2	68	13
3	48	13
4	67	48
5	13	58
6	97	61
7	70	67
8	58	68
9	12	70
10	13	97

Figure 3: Repeated value highlighted!

4. Google Colab: Your Python Playground

Now, it's time to take it to the next level and venture into the world of Python. It is one of the most popular programming languages for AI. Think of Python like a language that a computer can easily understand. You can give it instructions to perform tasks, and it can do it very quickly and efficiently.

To get started with Python, we can use Google Colab. Think of it as an online notebook where you can write and run Python code, all in your browser. It's free and easy to use.

To get started, just go to colab.research.google.com.

The code that we wrote in google sheets can be easily replicated using Python as well. This time, let's revisit our birthday paradox using Python.

We can ask our AI assistants again! For example: "Create a list with 365 numbers. Pick numbers randomly from this list, add them to a new list until a duplicate number appears, and then show the number of elements in the new list".

The AI might give you a piece of code similar to this:

```
import random

def birthday_paradox():
    all_birthdays = list(range(1, 366)) # List of all possible birthdays
    room = [] # A list to represent our group

    while True:
        pick_number = random.choice(all_birthdays) # Pick a random birthday
        if pick_number in room:
            print("We have a match! This happened after", len(room) + 1, "people")
            return # Stop the simulation if a match is found
        else:
            room.append(pick_number) # Add the new person to the group
birthday_paradox()
```

You can copy-paste this code into Google Colab and run it. This code will keep adding random "birthdays" to a list until it finds a duplicate, showing you how many people it took to find a match. Most of the time, it will be close to 23.

```

import random

def birthday_paradox():
    all_birthdays = list(range(1, 366)) # List of all possible birthdays
    room = [] # A list to represent our group
    while True:
        pick_number = random.choice(all_birthdays) # Pick a random birthday
        if pick_number in room:
            print("We have a match! This happened after", len(room) + 1, "people")
            return # Stop the simulation if a match is found
        else:
            room.append(pick_number) # Add the new person to the group
    birthday_paradox()

→ We have a match! This happened after 10 people

```

Figure 4: Execution of the code in Google Colab. Oh, just 10 people this time!

The Key: Asking Questions Effectively

Through the above examples, you have learned about the skill of Prompting. It's the art of communicating with AI effectively, where the key is to learn how to ask the right questions.

Be Clear: Make your needs specific.

Be Flexible: Adjust your requests based on AI responses.

Experiment: Try out different questions and methods.

This lesson was not just about learning Google Sheets or Python. It was more about learning the key skill of 'Prompting' which will be very useful for understanding and working with the ever-growing world of AI.

You don't need to be a computer expert or a coder. You just need to know how to ask. And the best way to learn is to start asking, playing around and experimenting with your prompts!

So, go ahead and start exploring the amazing world of AI, one question at a time.

Minor in AI

AI-Assisted Programming

1 Case study: Solve a coding problem with AI

Imagine you're a student tasked with creating a complex calendar program. Traditionally, this would require extensive coding knowledge and time. However, with AI-assisted programming, you can accomplish this task efficiently, even with limited coding experience. This real-world scenario illustrates the power and potential of AI in programming education and practice.

2 The Problem and AI Solution

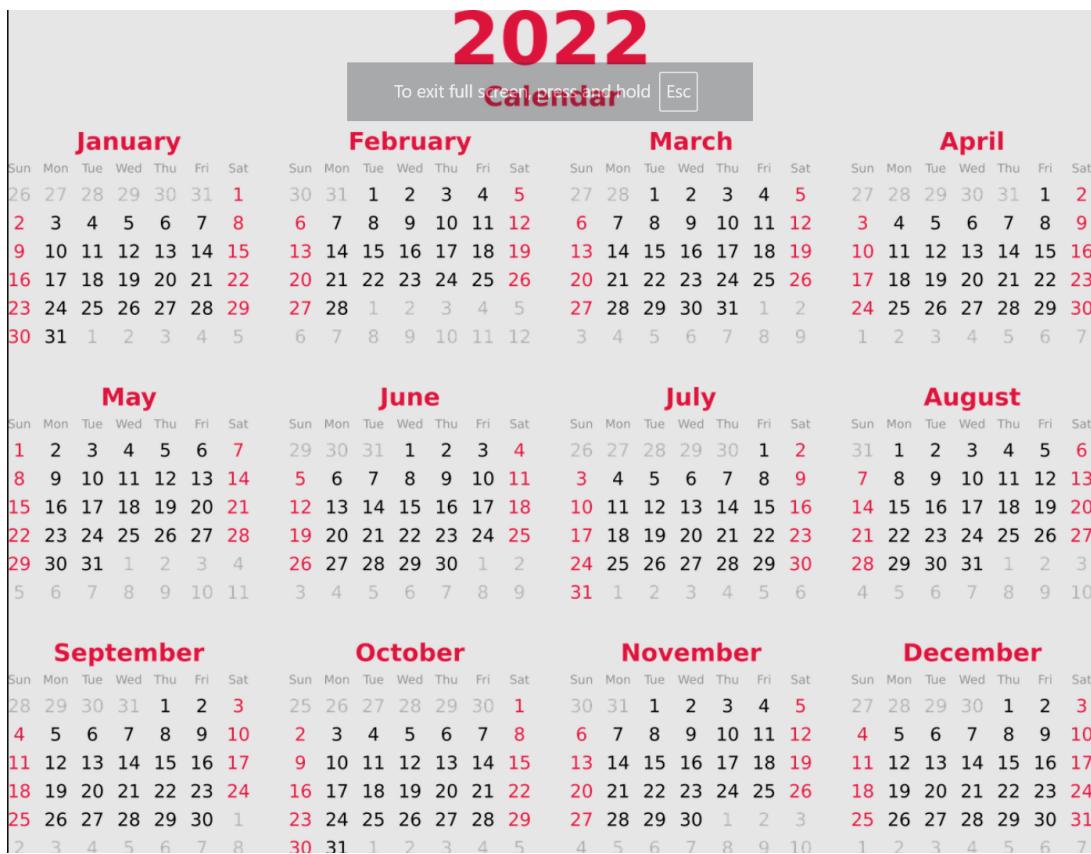
The challenge in programming education is balancing the need for fundamental coding skills with the rapid advancement of AI tools. AI-assisted programming, as demonstrated in this lecture, offers a solution by allowing students to focus on problem-solving and logic while using natural language prompts to generate code.

2.1 Key Concepts

- **AI-Powered Prompts:** Using natural language to generate Python code in Google Colab.
- **Basic Python Programming:** Creating simple programs for tasks like printing numbers and generating calendars.
- **The Birthday Paradox:** Exploring probability through code simulation.

2.2 Demonstration: Calendar Generation

Here's an example of how AI can assist in creating a calendar program:



```

1 # AI-generated code based on the prompt:
2 # "Given a year, display the calendar of that year in a user-friendly way"
3
4 import calendar
5
6 year = int(input("Enter a year: "))
7
8 for month in range(1, 13):
9     print(calendar.month(year, month))

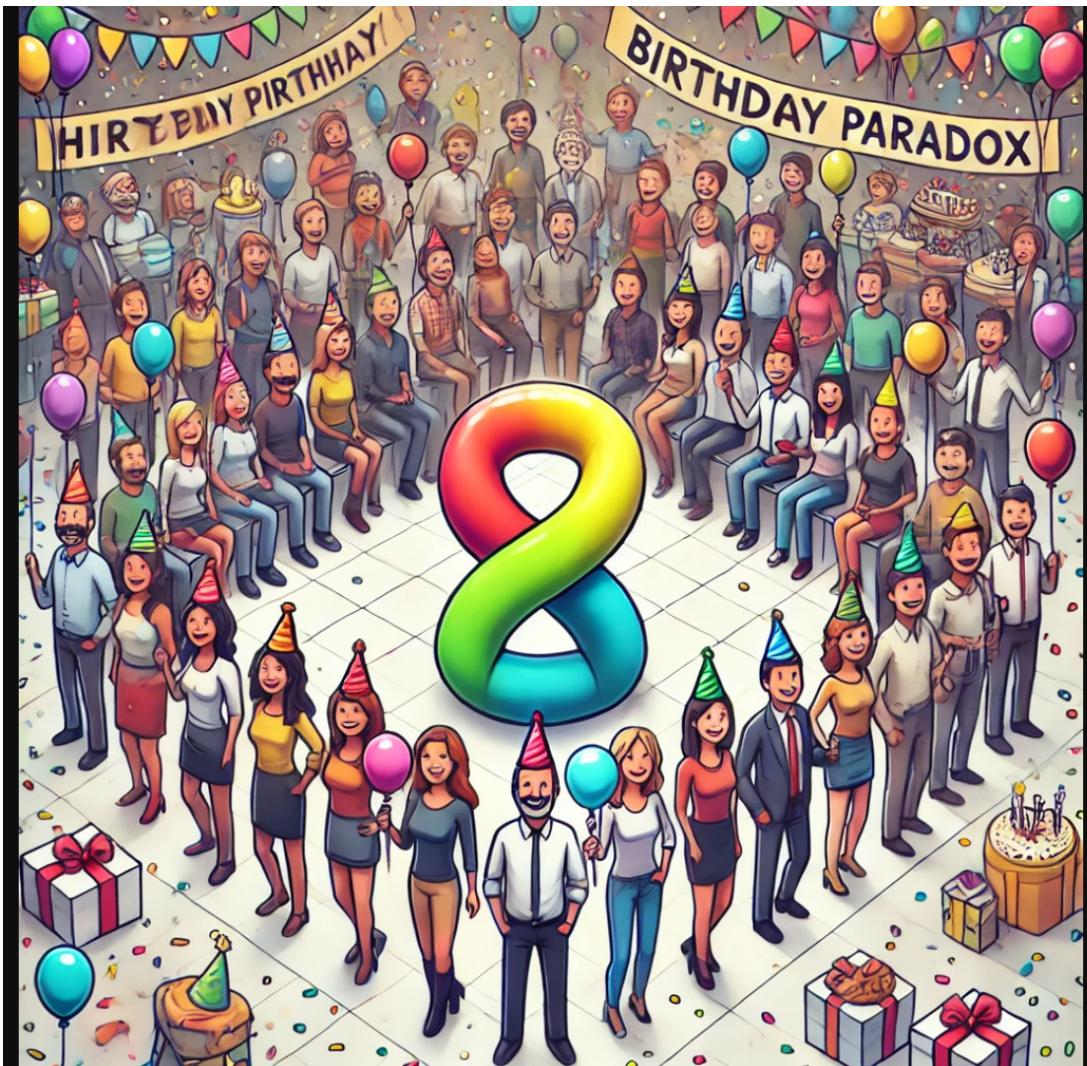
```

This code, generated through an AI prompt, demonstrates how complex tasks can be simplified using AI-assisted programming.

3 The Birthday Paradox: A Case Study

The lecture used the Birthday Paradox to illustrate the power of AI-assisted coding in exploring mathematical concepts.

3.1 Problem Statement



In a group of n people, the probability of at least two people sharing the same birthday can be surprisingly high, even for relatively small groups. This phenomenon is known as the **Birthday Paradox**.

3.2 AI-Assisted Solution

We simulated the probability of shared birthdays using Python. Below is the AI-generated code with comments to explain each step:

```

1 import random
2
3 # Function to simulate the birthday paradox
4 def birthday_paradox(num_people, num_simulations):
5     """
6         Simulates the birthday paradox and calculates the probability
7         of at least two people sharing a birthday.
8
9     Parameters:
10        num_people (int): Number of people in the group.
11        num_simulations (int): Number of simulations to run.

```

```

12
13     Returns:
14     float: Probability of at least two people sharing a birthday.
15     """
16
17     matches = 0 # Counter for simulations with shared birthdays
18
19     # Run the simulation num_simulations times
20     for _ in range(num_simulations):
21         # Generate random birthdays for the group
22         birthdays = [random.randint(1, 365) for _ in range(num_people)]
23
24         # Check if there are duplicate birthdays
25         if len(birthdays) != len(set(birthdays)):
26             matches += 1 # Increment matches if duplicates found
27
28     # Return the probability of shared birthdays
29     return matches / num_simulations
30
31 # Example usage of the function
32 result = birthday_paradox(25, 10000)
33 print(f"Probability of a shared birthday in a group of 25: {result:.2f}")

```

3.3 Explanation of the Code

1. **Imports:** The `random` module is used to generate random integers, simulating birthdays.

2. **Function Definition:**

- `birthday_paradox(num_people, num_simulations)`: This function calculates the probability of shared birthdays.

• **Parameters:**

- `num_people`: The size of the group (e.g., 25 people).
- `num_simulations`: Number of times the simulation is repeated (e.g., 10,000 times for accuracy).

3. **Simulations:**

- A loop runs the simulation `num_simulations` times.
- For each simulation, a list of `num_people` random birthdays (integers between 1 and 365) is generated.
- The function checks for duplicates by comparing the length of the `birthdays` list with the length of the `set(birthdays)` (a set automatically removes duplicates).

4. **Count Matches:**

- If duplicates are found, the `matches` counter is incremented.

5. Calculate Probability:

- The probability is computed as the ratio of `matches` to `num_simulations`.

6. Result:

- For a group of 25 people, the simulation typically shows that the probability of at least two people sharing a birthday is around 0.57 (57%).

3.4 Key Insight

This code demonstrates the counterintuitive nature of the Birthday Paradox: even with a group as small as 25 people, there is a high likelihood of shared birthdays due to combinatorial probabilities.

4 Conclusion

AI-assisted programming offers a new paradigm in coding education and practice. It allows students to focus on problem-solving and logic while leveraging AI to handle syntax and implementation details. However, it's crucial to balance this with understanding fundamental programming concepts.

Key takeaways:

- AI can significantly speed up the coding process, especially for beginners.
- Understanding the problem and formulating the right prompts is crucial.
- AI-assisted coding can help explore complex concepts like the Birthday Paradox.
- Traditional programming skills remain important for deeper understanding and problem-solving.

5 Additional Resources

For those interested in exploring further:

- [Wikipedia: Birthday Paradox](#)
- [Library of Babel Website](#)

Prompting with a Pinch of Logic and Code: Your AI Starter Kit

IIT Ropar - Minor in AI

28th Jan, 2025

Introduction: The Magic of Giving Instructions

Imagine you're in a kitchen, and you're trying to bake a cake. You have all the ingredients, but you don't know exactly how to put them together. So, you call your friend, a great baker, and say, "Hey, I have flour, sugar, eggs, and butter. I want to make a cake. Can you help me?"

That, in its simplest form, is what we are going to learn: how to give the right instructions to get what you want. In our case, instead of your friend, we'll be talking to AI. This book will teach you how to give instructions (we call these instructions "prompts") to AI tools, and we'll see that just like any recipe, the way you phrase your request matters a lot. It's a bit like learning a new language, but a language that machines understand – and it's all about logic and a bit of code! So, let's dive in!

1 Talking to AI – The Power of Prompts

We're beginning our journey into the world of Artificial Intelligence, and like learning any new language, we need to understand how to communicate with it. The first step in doing this is through something we call "prompting." A prompt is simply an instruction that we give to an AI. It's how we tell it what we want it to do.

Here's a simple example: imagine we want AI to create a list for us, we can give it a prompt like "Write me a list with 10 random numbers in it, then print it."

We might think this is a straightforward request, but as we will see, the way we phrase our prompts can change the outcome drastically.

1.1 Starting Simple: Creating Lists

Let's take our number list example a bit further. We can ask AI to create a list following a certain pattern:

"Write me a list where the first two numbers are one and two. After that, the third number should be the sum of the first and second. The fourth number should be the sum of the second and third, and so on. Keep doing this for 15 numbers, and then print the entire list."

The AI can follow these instructions and generate a sequence like 1, 2, 3, 5, 8, and so on. This might remind you of something called the “Fibonacci sequence”.

A screenshot of an AI code editor interface. At the top left is a "Generate" button with a gear icon. To its right is a text input box containing the instruction: "Write me a list where the first two numbers are one and two. After that, the third number should be the sum of the first and second. The fourth number should be the sum of the second and third, and so on. Keep doing this for 15 numbers, and then print the entire list." To the right of the input box is a magnifying glass search icon. Below the input box are navigation buttons: '< 1 of 1 >' and 'Undo changes' followed by a link 'Use code with caution'. The main area shows a Python script with a play button icon:

```
▶ def generate_list():
    num_list = [1, 2]
    for i in range(2, 15):
        next_num = num_list[i-1] + num_list[i-2]
        num_list.append(next_num)
    print(num_list)

generate_list()
```

At the bottom left is a copy icon, and to its right is the generated list: [1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987].

1.2 Refining Our Prompts: Less is More

Sometimes, we can get the same result using a much shorter prompt. In our case, instead of writing lengthy instructions, we could have simply said “Generate a Fibonacci sequence list starting with 1 and 2, and give me the first 15 numbers.” See how much shorter this instruction is compared to our previous version?

The lesson here is: try to be clear, and try to be concise. It is all about learning how to get to the point when giving instructions to the AI.

1.3 Beyond Simple Numbers: Transforming Lists

Now, let’s imagine we have a list of numbers, and we want to create a new list based on the original one. For example, if our original list is 1, 2, 3, 4, 5... we want to generate a new list where every number is one more than the corresponding number in the original. So, the new list would be 2, 3, 4, 5, 6....

This might seem basic, but this concept of transforming lists is very useful. A countdown timer is a real-world example of a transforming list. Think of each second or day going by and how it decreases a countdown timer; this represents a type of sequence with direct applications in real life.

2 Logic and Conditions: Making AI Smarter

Now, let’s add another layer of intelligence to our interactions with AI. Let’s explore the concept of conditions.

2.1 Checking Conditions: The “If” Logic

Imagine we want to create a student’s report card. We can prompt the AI by giving it details like the student’s name, and the marks in three subjects. Once the AI knows this, it needs to process the information. We can tell the AI:

“Read the name and marks in three subjects for a student. Calculate the total marks, the average marks, and then display the report card. Also, assign a grade to the student based on their marks. If the student’s grade is F, then also output: grade needs improvement.”

```
# Input student details
name = input("Enter student's name: ")
subject1 = float(input("Enter marks in subject 1: "))
subject2 = float(input("Enter marks in subject 2: "))
subject3 = float(input("Enter marks in subject 3: "))

# Calculate total and average marks
total_marks = subject1 + subject2 + subject3
average_marks = total_marks / 3

# Assign a grade based on the average marks
if average_marks >= 90:
    grade = "A"
elif average_marks >= 80:
    grade = "B"
elif average_marks >= 70:
    grade = "C"
elif average_marks >= 60:
    grade = "D"
elif average_marks >= 50:
    grade = "E"
else:
    grade = "F"

# Display the report card
print("\n--- Report Card ---")
print("Student Name:", name)
print("Subject 1 Marks:", subject1)
print("Subject 2 Marks:", subject2)
print("Subject 3 Marks:", subject3)
print("Total Marks:", total_marks)
print("Average Marks:", average_marks)
print("Grade:", grade)

# Output additional message for grade F
if grade == "F":
    print("Grade needs improvement.")
```

```
→ Enter student's name: Aakash
Enter marks in subject 1: 40
Enter marks in subject 2: 51
Enter marks in subject 3: 67

--- Report Card ---
Student Name: Aakash
Subject 1 Marks: 40.0
Subject 2 Marks: 51.0
Subject 3 Marks: 67.0
Total Marks: 158.0
Average Marks: 52.666666666666664
Grade: E
```

Using “if” conditions like this makes the AI understand the logic of grades, and it makes decisions based on our inputs.

2.2 Randomness and Conditions: Let the Games Begin

Another example of incorporating logic is when we play games with AI. Take a game like rock-paper-scissors. If the computer picks paper and we pick rock, then the computer wins. Otherwise, if we pick paper and the computer picks rock, then we win.

The screenshot shows a code editor interface with the following details:

- Header:** "Generate" button, status message: "rock-paper-scissors. If the computer picks paper and we pick rock, then the computer wins. Otherwise, if we pick paper and the computer picks rock, then we win.", navigation buttons: "< 1 of 1 > Undo changes Use code with caution".
- Code Area:** Generated code may be subject to a licence | nrfletcher/csc-202 | laksh29/Rock-Paper-Scissors | athenanoggle/rock-paper-scissors-game | tdgo/MIS132
import random

def play_rock_paper_scissors():
 """Plays a game of Rock-Paper-Scissors against the computer."""

 user_choice = input("Enter your choice (rock, paper, or scissors): ").lower()
 possible_actions = ["rock", "paper", "scissors"]

 if user_choice not in possible_actions:
 print("Invalid choice. Please try again.")
 return

 computer_choice = random.choice(possible_actions)
 print(f"\nYou chose {user_choice}, computer chose {computer_choice}.\n")

 if user_choice == computer_choice:
 print("Both players selected {user_choice}. It's a tie!")
 elif user_choice == "rock":
 if computer_choice == "scissors":
 print("Rock smashes scissors! You win!")
 else:
 print("Paper covers rock! You lose.")
 elif user_choice == "paper":
 if computer_choice == "rock":
 print("Paper covers rock! You win!")
 else:
 print("Scissors cuts paper! You lose.")
 elif user_choice == "scissors":
 if computer_choice == "paper":
 print("Scissors cuts paper! You win!")
 else:
 print("Rock smashes scissors! You lose.")

play_rock_paper_scissors()
- Output Area:** "Enter your choice (rock, paper, or scissors): rock", "You chose rock, computer chose scissors.", "Rock smashes scissors! You win!"

3 Real-World Applications: Beyond Theory

The examples that we saw in the previous chapters are not just abstract ideas. These ideas are applied every day in real life situations and are used to solve problems.

- **Timers and Counters:** Remember how we created a list where each number was one more than the previous? This is at the heart of how timers and counters work.
- **Decision Making:** Using ‘if’ conditions, we can get AI to make decisions such as to grade a report card or to declare who won in rock-paper-scissors.

4 Going a Bit Deeper: Playing with Text and Files

Let’s look at some more complicated examples of prompting and coding.

4.1 Text Transformation

Imagine we want to transform a string (a word, or a sentence). Suppose we have a sentence “Hello World”. We can ask the AI to change every letter in the sentence, by shifting it by three places in the alphabet.

4.2 Working with Files

Now, let’s try and work with large text files. Imagine we have a very big book on the computer. We can ask the AI to open this book, and count how many times the letter “A” appears in the file.

Generate

in the file sherlock.txt, how many times the letter "A" appears

< 1 of 1 > [Use code with caution](#)

```
# prompt: in the file sherlock.txt, how many times the letter "A" appears

def count_letter_occurrences(filepath, letter):
    """Counts the occurrences of a specific letter in a file.

    Args:
        filepath: The path to the file.
        letter: The letter to count.

    Returns:
        The number of times the letter appears in the file,
        or -1 if the file does not exist.
    """
    try:
        with open(filepath, 'r') as file:
            content = file.read().upper() # Read and convert to uppercase for case-insensitivity
            return content.count(letter.upper())
    except FileNotFoundError:
        return -1

# Example usage
filepath = "sherlock.txt"
letter_to_count = "A"

occurrences = count_letter_occurrences(filepath, letter_to_count)

if occurrences != -1:
    print(f"The letter '{letter_to_count}' appears {occurrences} times in the file.")
else:
    print(f"File '{filepath}' not found.")
```

The letter 'A' appears 36142 times in the file.

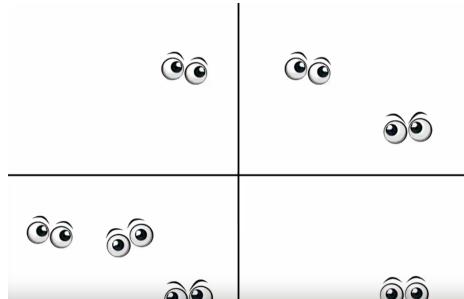
Conclusion: Your Journey Begins Here

We have just scratched the surface of what's possible with AI. By learning how to write good prompts and understanding some basic programming and logic, you can unlock the power of Artificial Intelligence.

Building Smarter Python Programs: Decisions, Loops, and Error Handling

1 Introduction: The Story of the Eyes

Imagine you’re looking at a picture. It’s a grid, and inside each of the four cells, there’s an eye staring back at you. What do you see? Are they all the same eyes, or are they different? Are they human eyes or animal eyes? Are they looking in the same direction?



Your mind probably started weaving a little story. Maybe they’re suspicious eyes, or maybe they’re surprised. Maybe they’re all watching you. Your mind is naturally deciding, sorting, and connecting based on what you see. It is automatically looping through the various scenarios you can imagine.

That’s precisely what we’re going to learn how to do in Python: make decisions, repeat actions, and handle unexpected situations, just like your mind does when you look at a picture.

2 Python Thinks Like You: The Power of Decisions

Our minds make decisions constantly. Should we wear a jacket? Should we cross the street? Programming languages, like Python, need to make decisions too. That’s why they include logic constructs that allow programs to “choose” different paths based on certain conditions, just like you choose a story for the eyes.

Note: The symbol in code represents a **space character**. It is used to explicitly indicate spaces, you might find them in your python code in this document.

2.1 The `if` Statement: Making Choices

Let's start by taking an age from the user. We need to define it like: `age = 40`. Instead of us directly defining the value of the variable, let's make the user define it using `input()`.

Now, let's say we want to validate the age someone has provided. We don't want negative ages, and we don't want zero. If we need to check the value of `age`, that's where the `if` statement comes in.

```
1 age = int(input("Enter the age:")) # Get age from user, convert  
    to integer  
2  
3 if age < 0:  
4     print("Your age cannot be negative")
```

2.2 The `elif` Statement: Adding More Conditions

But what if the age is zero? This is where `elif` (else if) comes in.

```
1 age = int(input("Enter the age:")) # Get age from user, convert  
    to integer  
2  
3 if age < 0:  
4     print("Your age cannot be negative")  
5 elif age == 0:  
6     print("Age cannot be zero")
```

2.3 The `underlineelse` Statement: The Default Path

Finally, if the age is not negative and not zero, it must be a valid age. We can use `else` to handle this default case.

```
1 age = int(input("Enter the age:")) # Get age from user, convert  
    to integer  
2  
3 if age < 0:  
4     print("Your age cannot be negative")  
5 elif age == 0:  
6     print("Age cannot be zero")  
7 else:  
8     print("Age is valid")
```

2.4 Combining Decisions: Let us calculate your birth year!

```

1 age = int(input("Enter the age:")) # Get age from user, convert
   to integer
2
3 if age < 0:
4     print("Your age cannot be negative")
5 elif age == 0:
6     print("Age cannot be zero")
7 else:
8     print("Age is valid")
9     current_year = int(input("Enter the current year:"))
10    birth_year = current_year - age
11    print(f"You were born in {birth_year}")

```

The line `print(f"You were born in {birth_year}")` is an example of an

2.5 Example to find : Pass or Fail

Listing 1: Pass or Fail Decision

```

1 # Input: Student's score
2 score = int(input("Enter the student's score:"))
3
4 # Checking the grade
5 if score >= 50:
6     print("Pass")
7 elif score >= 40:
8     print("Supplementary Exam")
9 else:
10    print("Fail")

```

Explanation

- If the score is **50 or above**, the student passes.
- If the score is **between 40 and 49**, they need to take a **supplementary exam**.
- If the score is **below 40**, they fail.

2.6 Python Code for determining grading system

Listing 2: Grade Assignment

```

1 # Input: Student's score
2 score = int(input("Enter the student's score:"))
3
4 # Checking the grade
5 if score >= 90:
6     print("Grade: A")
7 elif score >= 80:
8     print("Grade: B")
9 elif score >= 70:
10    print("Grade: C")

```

```
11 elif score >= 60:  
12     print("Grade : D")  
13 else:  
14     print("Grade : F")
```

2.7 Explanation

- If the score is **90 or above**, the grade is **A**.
 - If the score is between **80 and 89**, the grade is **B**.
 - If the score is between **70 and 79**, the grade is **C**.
 - If the score is between **60 and 69**, the grade is **D**.
 - If the score is **below 60**, the grade is **F**.

3 Looping: Doing Things Again and Again

Remember how your mind looped through the four eyes, considering the possibilities? In programming, we use loops to repeat actions. Let's take an example. Say we have a list of numbers, we want to check if they are negative or positive. A loop would be useful.

3.1 The for Loop: Iterating through Items

Let's say we have a list of numbers:

```
1 numbers = [2, -4, 89, -74, 68]
```

We can use a `for` loop to go through each number and decide whether it is positive or negative.

```
numbers = [2, -4, 89, -74, 68]
2
3 for num in numbers:
4     if num < 0:
5         print(f"{num} is a negative number")
6     else:
7         print(f"{num} is a positive number")
```

This loop will go through each number in the list, assigning the number to the variable `num`, and then checking whether it's negative or not.

Python 3.11
known limitations

```
1 numbers = [2, -4, 89, -74, 68]
2
3 for num in numbers:
4     if num < 0:
5         print(f"{num} is a negative number")
6     else:
7         print(f"{num} is a positive number")
```

[Edit this code](#)

Print output (drag lower right corner to resize)

```
2 is a positive number
-4 is a negative number
89 is a positive number
-74 is a negative number
68 is a positive number
```

Frames Objects

Global frame		list				
numbers	1	0	1	2	3	4
num	68	2	-4	89	-74	68

Loops allow us to execute a block of code multiple times. Here are two examples demonstrating the use of loops in Python.

3.2 Example 1: Sum of First N Natural Numbers using a for Loop

The following Python program calculates the sum of the first N natural numbers using a **for** loop.

3.2.1 Python Code

Listing 3: Sum of First N Natural Numbers

```
1 # Input: Number of terms
2 N = int(input("Enter a number:"))
3
4 # Initializing sum
5 total = 0
6
7 # Using a for loop to calculate sum
8 for i in range(1, N + 1):
9     total += i
10
11 print("Sum of first", N, "natural numbers is:", total)
```

3.2.2 Explanation

- The user inputs a number N , which determines how many natural numbers to sum.
- A variable **total** is initialized to 0.
- The **for** loop iterates from 1 to N , adding each number to **total**.
- Finally, the total sum is printed.

3.3 Example 2: Finding Factorial using a while Loop

The following Python program calculates the factorial of a given number using a **while** loop.

3.3.1 Python Code

Listing 4: Factorial Calculation

```
1 # Input: Number for factorial
2 num = int(input("Enter a number:"))
3
4 # Initializing factorial
5 factorial = 1
```

```
6
7 # Using a while loop to calculate factorial
8 while num > 0:
9     factorial *= num
10    num -= 1
11
12 print("Factorial is:", factorial)
```

3.3.2 Explanation

- The user inputs a number whose factorial is to be calculated.
- A variable `factorial` is initialized to 1.
- The `while` loop runs until the number reduces to 0.
- In each iteration, the current number is multiplied with `factorial`, and the number is decremented.
- The final factorial value is printed.

4 Conclusion: Your Programming Journey Begins

We've covered a lot: making decisions with `if`, `elif`, and `else`; looping through lists with `for`; and visualizing code with Python Tutor. This is just the start of your programming journey. As you practice and explore, you'll learn more ways to create amazing programs that not only perform tasks but also think logically, just like the human mind.

Minor in AI

Python Programming Basics

1 Case Study

Imagine you're at a casino, flipping a coin repeatedly. You're trying to determine if the coin is fair or biased. How would you keep track of the number of heads and tails? This real-world scenario introduces us to the concept of loops and conditional statements in programming.

2 Introduction



Figure 1: Evolution from tapes to disks.

The evolution of storage devices, from magnetic tapes to modern disks, drawing an analogy to the development of programming languages. Just as storage technology progressed from bulky cassettes to compact disks and eventually to high-capacity hard drives, programming languages have evolved to become more efficient and user-friendly. This evolution is driven by the need to store and process increasing amounts of data, much like how programming languages have developed to handle more complex tasks. This comparison highlights the importance of understanding the fundamental principles behind programming languages, as it allows us to appreciate the reasons behind certain design choices and potentially innovate new solutions.

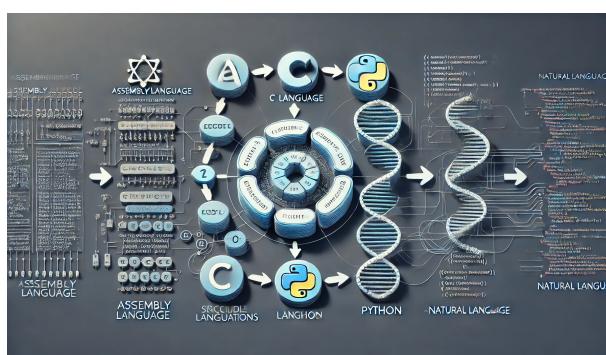


Figure 2: Conceptual illustration of Evolution from Assembly language to C to Python to natural human language.

3 Loops and Ifs

3.1 Problem Definition

The problem we're addressing is how to efficiently count the number of heads and tails in a series of coin flips. This introduces us to several key programming concepts:

1. Data storage (lists)

2. Loops (for and while)
3. Conditional statements (if-else)
4. Variable manipulation

3.2 Solution: Python Code

Let's look at a Python solution to this problem:

```
1 # Define the list of coin flips
2 flips = ['H', 'T', 'T', 'H', 'H', 'T']
3
4 # Initialize counters for heads and tails
5 head_count = 0
6 tail_count = 0
7
8 # Iterate through each flip in the list
9 for f in flips:
10     if f == 'H': # If the flip is a head, increment head_count
11         head_count += 1
12     elif f == 'T': # If the flip is a tail, increment tail_count
13         tail_count += 1
14
15 # Print the final count of heads and tails
16 print(f"Head count: {head_count}")
17 print(f"Tail count: {tail_count}")
```

This code demonstrates:

- List usage to store coin flip results
- A 'for loop' to iterate through the list
- Conditional statements to check for heads or tails
- Incrementing counters

3.3 Alternative Solution: While Loop

We can also solve this problem using a while loop:

```
1 # Define the list of coin flips
2 flips = ['H', 'T', 'T', 'H', 'H', 'T']
3
4 # Initialize counters for heads and tails
5 head_count = 0
6 tail_count = 0
7
8 # Initialize loop index and total number of flips
9 i = 0
10 n = len(flips) # Get the length of the list
11
12 # Loop through each element in the list using a while loop
13 while i < n:
14     if flips[i] == 'H': # If the flip is a head, increment head_count
15         head_count += 1
16     elif flips[i] == 'T': # If the flip is a tail, increment tail_count
17         tail_count += 1
18     i += 1 # Move to the next index
19
20 # Print the final count of heads and tails
21 print(f"Head count: {head_count}")
```

```
22 print(f"Tail count: {tail_count}")
```

This version introduces:

- The concept of indexing in lists
- Use of a ‘while loop’ with a counter
- Pre-computing the length of the list for efficiency

4 Bitwise Operators

In addition to basic arithmetic and logical operations, Python supports bitwise operators. These operators manipulate individual bits of integers, providing efficient low-level operations that are widely used in various applications.

4.1 Types of Bitwise Operators

The bitwise operators are:

- **AND** : Performs a bitwise AND operation. Returns 1 if both corresponding bits are 1, otherwise 0.

Example: $1010_2 \& 1100_2 = 1000_2$

- **OR** : Performs a bitwise OR operation. Returns 1 if at least one of the corresponding bits is 1, otherwise 0.

Example: $1010_2 | 1100_2 = 1110_2$

- **NOT** : Performs a bitwise negation (1’s complement). Inverts all bits, changing 1s to 0s and 0s to 1s.

Example (8-bit representation): $\sim 1010_2 = 0101_2$

- **Left Shift** : Shifts the bits to the left by a specified number of positions, filling with 0s on the right. This operation is equivalent to multiplying by 2^n , where n is the number of shifts.

Example: $1010_2 << 2 = 101000_2$

- **Right Shift** : Shifts the bits to the right by a specified number of positions. If it is a logical right shift, it fills the leftmost bits with 0s. If it is an arithmetic right shift, it fills with the sign bit (for negative numbers). Equivalent to integer division by 2^n .

Example: $1010_2 >> 2 = 10_2$

- **1’s Complement**: Inverts all bits of a number (same as bitwise NOT). Used in some old systems for representing negative numbers.

Example (8-bit representation): 1’s complement of $00001010_2 = 11110101_2$

- **2’s Complement**: Obtained by taking the 1’s complement of a number and adding 1. It is the standard way to represent negative numbers in binary systems.

Example (8-bit representation): 2’s complement of 00001010_2

1’s complement: 11110101_2

Adding 1: 11110110_2

4.2 Example: Bitwise AND

Let's look at how the bitwise AND operator works:

```
1 a = 12 # 1100 in binary
2 b = 25 # 11001 in binary
3 result = a & b # 1100 & 11001 = 01100 (8 in decimal)
4 print(f"Result of {a} & {b} = {result}")
```

Step-by-step Explanation:

- Convert decimal numbers to binary:

$$12_{10} = 1100_2, \quad 25_{10} = 11001_2$$

- Perform bitwise AND on corresponding bits:

$$01100_2 \& 11001_2 = 01000_2$$

- Convert the binary result back to decimal:

$$01000_2 = 8_{10}$$

- Print the result: Result of 12 & 25 = 8

4.3 Applications of Bitwise Operators in AI

Bitwise operators are commonly used in AI and machine learning in the following ways:

- **Feature Engineering:** Used to optimize storage and processing of binary features in datasets.
- **Efficient Computations:** Speeds up neural network operations, such as activation functions and matrix transformations.
- **Genetic Algorithms:** Helps implement mutation and crossover operations efficiently.
- **Image Processing:** Used in image masking, thresholding, and edge detection.
- **Cryptography:** Assists in encryption and hashing techniques used in secure AI applications.

5 Key Takeaways

This section covers fundamental Python concepts, including Boolean operations with logical operators (AND, OR, NOT) and bitwise operators, highlighting the difference between the two. It explores Python's core data types such as integers, floats, characters, strings, and Booleans, along with conditional statements like the if-else structure. Looping constructs, including for and while loops, are discussed with emphasis on list operations—creating lists, accessing elements, and variable management such as initialization and incrementing. The importance of code visualization using Python Tutor is

demonstrated by analyzing memory frames and variable states during execution. Practical examples include a coin toss simulation program that counts heads and tails using loops and an exploration of one's complement and two's complement concepts. Additionally, programming best practices are introduced, such as loop optimization techniques like pre-computing list lengths, along with an overview of the significance of understanding fundamental principles in programming. The section also briefly introduces the concept of loop rolling and unrolling for efficient execution.

Learning Python Through A Fun Battle Game

IIT Ropar - Minor in AI

1st Feb, 2025

Have you ever talked about something with a friend on WhatsApp – maybe how much you love a particular brand of shoes – and then, *bam*, those same shoes start popping up in ads all over the internet? It's like your phone is reading your mind! You might wonder, is WhatsApp sharing your private chats? Is Google listening in? The truth is, probably not directly. There's something a bit more clever going on called *Federated Learning*. It's a way for computers to learn from data without actually seeing your data.

While we won't be building something as complex as that just yet, we are about to start building our own game. It is so cool that you can learn a lot of concepts of programming and Artificial Intelligence just by building a simple game like that. And today, you will. Yes, you will build your own battle game. And the best part? We're going to do it using just three basic things you probably already know in Python: `if` statements, `for` or `while` loops and `print` statements.

That's right. If you know how to print something on the screen, use `if` conditions and use loops, then you already have everything you need to create your own battle game, analyze winning strategies and even make a simple AI opponent to play against. Sounds amazing, right? Let's jump in!



Figure 1: Let's get started!

Building Our Text-Based Battle Game

Our game is a simple battle between two players. Here's the breakdown:

1. **Two Players:** We need two participants for a battle, which we'll simply call Player 1 and Player 2.
2. **Health:** Each player starts with 100 health points (HP).
3. **Attacks:** Players will take turns attacking each other, causing damage and reducing the opponent's health.
4. **Turn-Based:** The game proceeds turn by turn. Player 1 attacks, then Player 2, and so on.
5. **Text-Based:** This is a text-based game. You will type commands, and the game's results will be shown to you in the terminal or interpreter (Google Colab in our case).
6. **The Goal:** Whoever reaches zero health first loses the game.

Let's start coding. We will be using Google Colab, a free online platform where you can write and run Python code.



Figure 2: The learning begins!

Setting Up the Game

First, let's create the initial variables that hold the health information for our players and set the game up. Let's open up a notebook in Colab and put this code inside a cell:

```
1 Player1.HP = 100  
2 Player2.HP = 100
```

Here, we've just created two variables: `Player1.HP` and `Player2.HP`, each starting with a value of 100. These represent the health points of each player.

The Game Loop

Next, we need our game to keep running until one player runs out of health. For this, we will use a `while` loop which will continue until one of the player's health becomes zero. Let's add the following code:

```
1 turn = 1
2 while Player1_HP > 0 and Player2_HP > 0:
3     print("\n-----Turn Start-----")
4     print('Player 1 HP:', Player1_HP)
5     print('Player 2 HP:', Player2_HP)
```

Let's break this down:

- `turn = 1`: We create a variable called `turn` and set it to 1 initially. This will keep track of whose turn it is.
- `while Player1_HP > 0 and Player2_HP > 0::`: This is our `while` loop. It will keep running as long as both players' health is greater than zero.
- Inside the loop, we print "Turn Start", followed by each player's current health. The `\n` in "`n-----Turn Start-----`" moves the output to a new line so it is more readable.

Determining Who's Turn it Is

Now we need to implement our `if` statement to determine whose turn it is. This part is crucial because it ensures that players take turns. Look at the following code and we will see how it works:

```
1 if turn == 1:
2     print("Player 1's Turn")
3     input("Press ENTER to attack:")
4     damage = random.randint(10, 20)
5     print("Generating attack.....")
6     time.sleep(2)
7     Player2_HP = Player2_HP - damage
8     print("Player 1 has attacked Player 2 for", damage, "damage")
9     !
10    turn = 2
11 else:
12     print("Player 2's Turn")
13     input("Press ENTER to attack:")
14     damage = random.randint(10, 20)
15     print("Generating attack.....")
16     time.sleep(2)
17     Player1_HP = Player1_HP - damage
18     print("Player 2 has attacked Player 1 for", damage, "damage")
19     !
20    turn = 1
```

Let's understand it, step by step:

- `if turn == 1::`: This is an `if` statement which checks if the `turn` variable is equal to 1. If it is, it means it's Player 1's turn.

- Inside the `if` block:
 - We print `Player 1's Turn.`
 - We get an input from the player, telling them to press ENTER to attack.
 - We generate a random damage amount between 10 and 20 using `random.randint(10, 20)`. To do this, we first need to add the following line of code at the top of the notebook:

```

1 import random
2 import time
3

```

This imports the necessary `random` library that helps us generate random integers and `time` library to make the delay.

- We print "Generating Attack", and add a delay of 2 seconds using `time.sleep(2)` – this is simply to make the game feel a bit more realistic.
- We then reduce Player 2's health by the generated damage: `Player2_HP = Player2_HP - damage.`
- We print the damage dealt by Player 1 to Player 2 using another print statement and an emoji at the end.
- Finally, we change `turn` to 2 (`turn = 2`). This ensures that next time, it will be player 2's turn.

- `else::` If the `turn` is *not* 1, then the code inside the `else` block runs (meaning it is player 2's turn) and Player 2 attacks. The steps are almost the same as for player 1. The health of Player 1 is decreased, and the turn is changed to 1.

Determining the Winner

After the `while` loop ends, it means that one player has reached zero health. We need to determine the winner:

```

1 if Player1_HP > 0:
2     print('Player 1 is the winner')
3 else:
4     print('Player 2 is the winner')

```

If `Player1_HP` is greater than 0, it means that Player 1 is the winner. Otherwise, Player 2 is the winner.

And that's our basic game! You can now copy and paste all of the above blocks of code into a Google Colab notebook, and run the game. You'll see each player attacking turn-by-turn and you will see who wins.

```

-----Turn Start-----
Player 1s HP: 20
Player 2s HP: 24
Player 1's Turn
Press ENTER to attack:
Generating attack.....*
```

```

Player 1 has attacked Player 2 for 14 damage! ✘
```



```

-----Turn Start-----
Player 1s HP: 20
Player 2s HP: 10
Player 2's Turn
Press ENTER to attack:
Generating attack.....*
```

```

Player 2 has attacked Player 1 for 10 damage! ✘
```



```

-----Turn Start-----
Player 1s HP: 10
Player 2s HP: 10
Player 1's Turn
Press ENTER to attack:
Generating attack.....*
```

```

Player 1 has attacked Player 2 for 17 damage! ✘
```

```

Player 1 is the winner 🎉
```

Figure 3: A screen shot of a sample output from the game, showing turns, health, and damage.

Making it Fair: Analyzing the Game

Our game is fun, but is it fair? Does one player have an advantage over the other? Let's put our game to the test by simulating multiple game matches. Let's not be human and play this game ourselves, rather let's have the machine play this game over and over again and tell us if this is a fair game.

Let's wrap the game code in a function so that it's easy to repeat multiple times:

```

1 def game():
2     Player1_HP = 100
3     Player2_HP = 100
4     turn = 1
5
6     while Player1_HP > 0 and Player2_HP > 0:
7         if turn == 1:
8             damage = random.randint(10, 20)
9             Player2_HP = Player2_HP - damage
10            turn = 2
11        else:
12            damage = random.randint(10, 20)
```

```

13         Player1_HP = Player1_HP - damage
14         turn = 1
15
16     if Player1_HP > 0:
17         return 'Player 1',
18     else:
19         return 'Player 2'

```

Now let's write some code to simulate 10000 games:

```

1 num_games = 10000
2 Player1_wins = 0
3 Player2_wins = 0
4
5 for i in range(num_games):
6     winner = game()
7     if winner == 'Player 1':
8         Player1_wins = Player1_wins + 1
9     else:
10        Player2_wins = Player2_wins + 1

```

Here, we're running the `game()` function 10,000 times. We are keeping count of how many times player 1 has won and how many times player 2 has won. At the end, we'll have a count of wins for each player.

To visualize this, we will create a bar graph using `matplotlib`, a Python visualization library. Let's add this code at the end:

```

1 import matplotlib.pyplot as plt
2 plt.figure(figsize=(5, 5))
3 x_axis = ['Player 1', 'Player 2']
4 y_axis = [Player1_wins, Player2_wins]
5
6 plt.bar(x_axis, y_axis)
7 plt.ylabel('Number of wins')
8 plt.title('Game Simulation Results')
9 plt.show()

```

This code will generate a bar graph showing the number of wins for each player. If you run this code, you will see that the game is biased towards player 1, as the game begins with his turn.

Making an AI Opponent: Strategic Choices

Now, let's make our game more interesting by adding an AI opponent. Instead of us choosing to attack, the computer will make a decision to attack or defend for us, making it the AI opponent.

To do this, we need to add another action: the option to *defend*. Defending will reduce the damage taken by 50 percent. We will add two new variables at the beginning of our `game()` function:

```

1 Player1_defending = False
2 Player2_defending = False

```

These variables will keep track of whether a player is currently defending. We'll then update our code to ask for input whether to attack or defend:

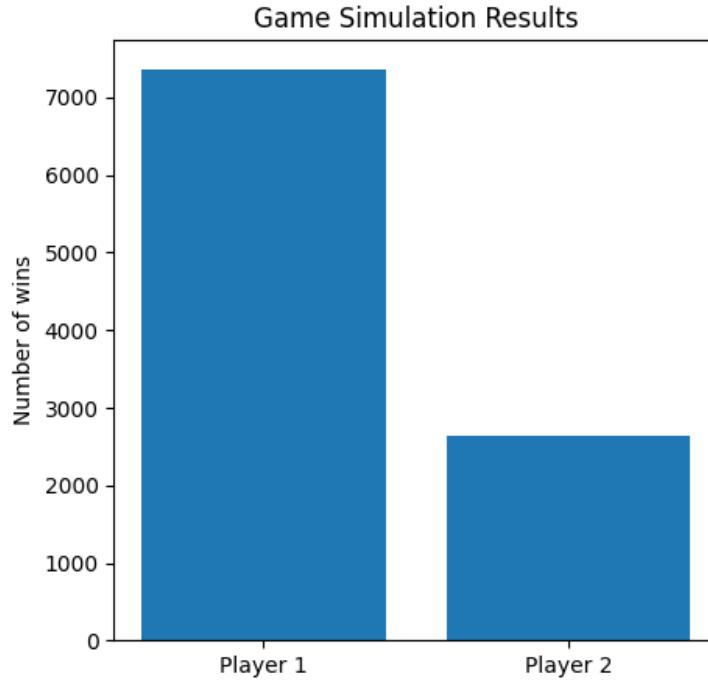


Figure 4: A sample bar chart showing the number of wins for each player, with player 1's bar much bigger than that of player 2.

```

1  if turn == 1:
2      print("Player 1's Turn")
3      action = input("Choose attack or defend: ")
4      if action == 'attack':
5          damage = random.randint(10, 20)
6          print("Generating attack....")
7          time.sleep(2)
8          if Player2_defending:
9              damage = damage // 2
10             Player2_defending = False
11             Player2_HP = Player2_HP - damage
12             print("Player 1 has attacked Player 2 for", damage, " "
13             damage!)
14         else:
15             Player1_defending = True
16             print("Player 1 is defending")
    turn = 2

```

If the player chooses to attack, the code does what it did previously – deal a random damage between 10 and 20 to the other player. The only difference is now that if the other player was defending, we halve the damage being dealt. If the player chooses to defend, then that player's `Playerx_defending` variable is set to `True`, to indicate that that player is defending. And when it's the next

player's turn, that defending variable gets set to `False`, no matter what the other player chooses to do, whether to attack or defend. The damage is only halved if one of the players was defending *during* the other player's attack.

We need to also update this same code for Player 2. And finally, to create our AI opponent, we will write a function for the computer's choice:

```

1 def computer_choice(Player1_HP, Player2_HP, Player1_defending,
2     Player2_defending):
3     if Player1_defending:
4         return 'attack'
5     elif Player2_defending:
6         return 'attack'
7     elif Player1_HP < 30:
8         return 'attack'
9     elif Player2_HP < 20:
10        return 'defend'
11    elif random.random() < 0.7:
12        return 'attack' # 70% chance to attack
13    else:
14        return 'defend' # 30% chance to defend

```

Here, the computer makes a decision to either attack or defend. Let's understand the code:

- First, if either of the players is defending, it chooses to attack.
- If Player 1 has less than 30 HP, it chooses to attack.
- If Player 2 has less than 20 HP, the computer chooses to defend.
- Otherwise, it randomly chooses to attack with a 70% chance, and defend with a 30% chance.

Now let's modify the player 2 input part to include this computer choice:

```

1 else:
2     # Player 2's turn
3     print("Player 2's Turn")
4     action = computer_choice(Player1_HP, Player2_HP,
5         Player1_defending, Player2_defending)
6     if action == 'attack':
7         damage = random.randint(10, 20)
8         print("Generating attack.....")
9         time.sleep(2)
10        if Player1_defending:
11            damage = damage // 2 # Integer division
12            Player1_defending = False
13            Player1_HP = Player1_HP - damage
14            print("Player 2 has attacked Player 1 for", damage, "
15            damage!")
16        else:
17            Player2_defending = True
18            print("Player 2 is defending")
19            turn = 1

```

In the modified code above, we're calling our `computer_choice` function which takes the relevant parameters and makes a choice for player two. We no longer require user input here, since the computer chooses the action.

Run this code. You will be able to play the game against your new AI opponent.

```
-----Turn Start-----
Player 1s HP: 28
Player 2s HP: 41
Player 1's Turn
Choose attack or defend: attack
Generating attack.....*
Player 1 has attacked Player 2 for 8 damage! ✘

-----Turn Start-----
Player 1s HP: 28
Player 2s HP: 33
Player 2's Turn
Generating attack.....*
Player 2 has attacked Player 1 for 15 damage! ✘

-----Turn Start-----
Player 1s HP: 13
Player 2s HP: 33
Player 1's Turn
Choose attack or defend: attack
Generating attack.....*
Player 1 has attacked Player 2 for 18 damage! ✘

-----Turn Start-----
Player 1s HP: 13
Player 2s HP: 15
Player 2's Turn
Generating attack.....*
Player 2 has attacked Player 1 for 13 damage! ✘
Player 2 is the winner!
```

Figure 5: Sample output screenshot for the improvised version of the game!

And there you have it! You've built a text-based battle game, analyzed its fairness, and created a simple AI opponent, all using the basic building blocks of Python: `if` statements, `while` loops, and `print` statements.

You've come a long way! This is just the beginning of your journey. You're well on your way to learning the fundamentals of Python and Artificial Intelligence through engaging and fun methods. Remember, keep experimenting and keep asking questions, and you can do a lot of cool things just using this basic knowledge. Keep practicing!

Minor in AI

Python: Lists, Tuples, and Dictionaries

1 Storing customer data

Imagine you're a data scientist working on a project to analyze customer data for an e-commerce company. You have information about thousands of customers, including their names, ages, purchase history, and preferences. How would you efficiently store and manipulate this data to extract meaningful insights?

This is where Python's data structures come into play. In this lesson, we'll explore three fundamental data structures in Python: lists, tuples, and dictionaries. These powerful tools allow us to organize, store, and manipulate data effectively, making them essential for tasks ranging from simple data analysis to complex machine learning algorithms.



Figure 1: Storing customer data

2 Lists: Versatile and Mutable

Lists are one of the most commonly used data structures in Python. They are ordered, mutable (changeable), and can contain elements of different data types.

2.1 Operating with Lists

```
1 # Creating a list of customer ages
2 ages = [25, 30, 22, 35, 28]
3
4 # Adding a new customer age
5 ages.append(40)
6 print("Adding an element: ",ages)
7
8 # Removing specific element using remove()
```

```
9 ages.remove(30)
10 print("After removing 30:", ages)
11
12 # Removing element at a specific index using pop()
13 removed_age = ages.pop(2)
14 print("Removed age at index 2:", removed_age)
15 print("List after pop():", ages)
16
17 # Sorting the list
18 ages.sort()
19 print("Sorted ages:", ages)
20
21 # Sorting in descending order
22 ages.sort(reverse=True)
23 print("Sorted ages (descending):", ages)
24
25 # Reversing the list
26 ages.reverse()
27 print("Reversed ages:", ages)
28
29 # Counting occurrences of an element
30 ages = [25, 30, 22, 35, 28, 40, 25]
31 count_25 = ages.count(25)
32 print("Number of times 25 appears:", count_25)
33
34 # Finding the index of an element
35 index_35 = ages.index(35)
36 print("Index of 35:", index_35)
37
38 # Inserting an element at a specific index
39 ages.insert(2, 33)
40 print("After inserting 33 at index 2:", ages)
41
42 # Extending a list with another list
43 more_ages = [45, 50]
44 ages.extend(more_ages)
45 print("Extended list:", ages)
```

These operations demonstrate the versatility of lists, including removing elements, sorting, reversing, counting occurrences, finding indices, inserting elements, and extending lists.

2.2 List Comprehension

List comprehension is a powerful feature that allows us to create new lists based on existing ones concisely:

```
1 # Creating a new list with ages increased by 1
2 increased_ages = [age + 1 for age in ages]
3 print(increased_ages) # Output: [26, 31, 24, 36, 29, 41]
4
5 # Filtering for ages over 30
6 over_30 = [age for age in ages if age > 30]
7 print(over_30) # Output: [35, 40]
```

3 Tuples: Immutable and Efficient

Tuples are similar to lists but are immutable, meaning their contents cannot be changed after creation. This makes them useful for storing data that shouldn't be modified. While tuples are immutable, there are still several operations we can perform on them:

```
1 # Creating a tuple of customer information
2 customer = ("John Doe", 30, "john@example.com")
3
4 # Accessing tuple elements
5 name, age, email = customer
6 print(f"Name: {name}, Age: {age}, Email: {email}")
7
8 # Creating a tuple
9 customer_data = ("John Doe", 30, "john@example.com", "New York", "Tech",
10   30, 30)
11
12 # Counting occurrences of an element
13 count_30 = customer_data.count(30)
14 print("Number of times 30 appears:", count_30)
15
16 # Finding the index of an element
17 index_email = customer_data.index("john@example.com")
18 print("Index of email:", index_email)
19
20 # Tuple unpacking with * for multiple values
21 name, age, *rest = customer_data
22 print("Name:", name)
23 print("Age:", age)
24 print("Remaining data:", rest)
25
26 # Converting tuple to list for more flexibility
27 customer_list = list(customer_data)
28 customer_list.append("Additional Info")
29 print("Modified list:", customer_list)
30
31 # Converting back to tuple if needed
32 customer_data_updated = tuple(customer_list)
33 print("Updated tuple:", customer_data_updated)
```

These operations showcase counting elements, finding indices, tuple unpacking, and converting between tuples and lists for added flexibility.

4 Dictionaries: Key-Value Pairs

Dictionaries are unordered collections of key-value pairs, making them ideal for storing and retrieving data efficiently. Dictionaries in Python offer various operations for managing key-value pairs efficiently.

```
1 # Creating a dictionary of customer information
2 customer_info = {
3     "name": "Jane Smith",
4     "age": 28,
5     "email": "jane@example.com",
6     "purchases": ["laptop", "phone", "headphones"]
7 }
8
```

```

9 # Accessing dictionary values
10 print(customer_info["name"]) # Output: Jane Smith
11
12 # Using .get() to safely access a key
13 print(customer_info.get("loyalty_points", "NA - added as default"))
14 # Output: NA (since it's not added yet)
15
16 # Adding a new key-value pair
17 customer_info["loyalty_points"] = 500
18
19 # Using .get() again after adding the key
20 print(customer_info.get("loyalty_points", "Not available"))
21 # Output: 500
22
23 # Iterating through dictionary items
24 for key, value in customer_info.items():
25     print(f"{key}: {value}")
26
27 # Merging additional information into the dictionary
28 additional_info = {"loyalty_points": 500, "membership": "Gold"}
29 customer_info.update(additional_info)
30 print("After updating:", customer_info)
31
32 # Creating a new dictionary to merge
33 contact_info = {"phone": "555-1234", "address": "123 Main St"}
34 merged_info = {**customer_info, **contact_info}
35 print("Merged dictionary:", merged_info)
36
37 # Removing a key-value pair using del
38 del merged_info["membership"]
39 print("After deleting 'membership':", merged_info)
40
41 # Removing a key-value pair using pop()
42 removed_age = merged_info.pop("age")
43 print("Removed age:", removed_age)
44 print("After popping 'age':", merged_info)
45
46 # Getting all keys and values
47 print("Keys:", list(merged_info.keys()))
48 print("Values:", list(merged_info.values()))
49
50 # Creating a dictionary from two lists
51 keys = ["department", "salary"]
52 values = ["IT", 75000]
53 employee_details = dict(zip(keys, values))
54 print("Employee details:", employee_details)
55
56 # Copying a dictionary
57 employee_details_copy = employee_details.copy()
58 print("Copied dictionary:", employee_details_copy)

```

These operations demonstrate merging dictionaries, removing key-value pairs, accessing keys and values, creating dictionaries from lists, and copying dictionaries.

By utilizing these operations, you can more effectively manipulate lists, tuples, and dictionaries in your Python programs, allowing for more complex data handling and management.

5 Choosing the Right Data Structure

Data Structure	Ordered	Mutable	Constructor	Example
List	Yes	Yes	<code>[]</code> or <code>list()</code>	<code>[5.7, 4, 'yes', 5.7]</code>
Tuple	Yes	No	<code>()</code> or <code>tuple()</code>	<code>(5.7, 4, 'yes', 5.7)</code>
Set	No	Yes	<code>{...}</code> * or <code>set()</code>	<code>{5.7, 4, 'yes'}</code>
Dictionary	No	Yes**	<code>{...}</code> or <code>dict()</code>	<code>{'Jun': 75, 'Jul': 89}</code>

Figure 2: Comparison between different data types to store data

When working with data, it's crucial to choose the appropriate data structure:

- Use lists when you need an ordered, mutable collection of items.
- Use tuples for immutable collections, especially when working with fixed data.
- Use dictionaries when you need fast lookups based on unique keys.

6 Conclusion

Understanding these data structures is fundamental to effective programming in Python, especially in the context of AI and data science. Lists provide flexibility, tuples offer immutability, and dictionaries enable efficient data retrieval. By mastering these structures, you'll be better equipped to handle complex data manipulation tasks and build more efficient algorithms.

As you continue your journey in AI and machine learning, you'll find these data structures at the core of many advanced concepts and techniques. Practice using them in various scenarios to solidify your understanding and improve your problem-solving skills.

AI Alchemy with Python: Sets, Functions, and Modularity

Welcome, Future AI Wizards!

This module is your friendly guide to understanding fundamental Python concepts that will form the bedrock of your journey into the exciting world of Artificial Intelligence. We'll focus on sets, built-in functions, and the power of modularity in your code. Let's dive in!



Figure 1: Buckle up, buckaroos!

From Sunlight to Supercomputers: An Analogy for Problem Solving

Imagine a world without electricity. A world where the only source of light is the sun. Life would be very different, right? Our activities would be limited to

daylight hours.

Now, imagine someone inventing the candle. This is progress! We can now have light even after sunset, extending our day.

But a candle isn't perfect. It's smoky, needs constant attention, and doesn't provide a lot of light. So, we invent the light bulb! Brighter, cleaner, and more efficient.

This progression from sunlight to candles to light bulbs illustrates a key principle in problem-solving and in AI: **evolution**. We start with a basic solution, identify its limitations, and then innovate to create something better. This continuous cycle of improvement is at the heart of AI development. From simple command-line interfaces, to mainframes, to cloud computing and serverless architectures, we build upon previous solutions to develop better solutions in the field of AI.

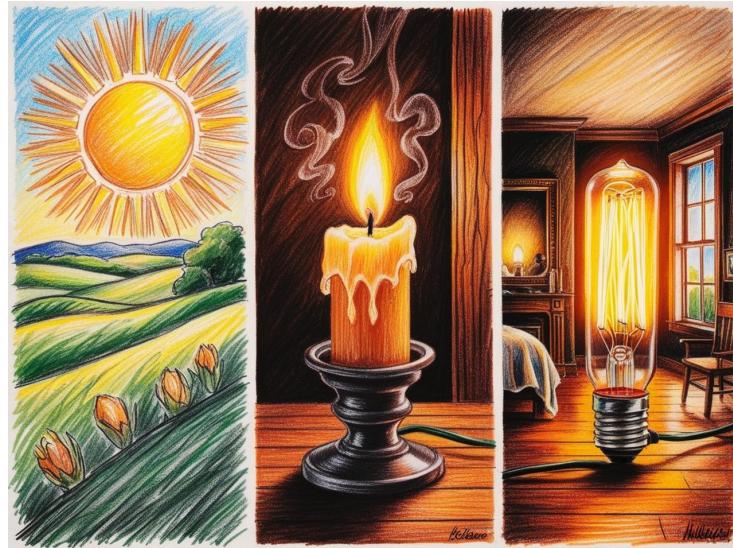


Figure 2: Evolution of problem-solving

Working with Sets

Let's talk about sets in Python. Think of a set as a collection of unique items. Imagine you have two bags of marbles.

- Bag A: Contains marbles with numbers 1, 2, and 3.
- Bag B: Contains marbles with numbers 3, 4, and 5.

A set only keeps unique values, so if there are 2 '1's in Bag A, the set will take only one '1'.

Now, what if you want to combine these bags? That's where set operations come in.

```
1 # Defining our sets
2 set_a = {1, 2, 3}
3 set_b = {3, 4, 5}
```

1. Union

The union of two sets combines all the elements from both sets, removing any duplicates. It's like emptying both bags into a new, larger bag.

```
1 # Using the union method
2 set_union = set_a.union(set_b)
3 print(set_union) # Output: {1, 2, 3, 4, 5}
4
5 # Another way to achieve the same result is using the "|" operator
6 set_union_2 = set_a | set_b
7 print(set_union_2) # Output: {1, 2, 3, 4, 5}
```

2. Intersection

The intersection of two sets finds the elements that are common to both. It's like finding the marbles that are present in *both* Bag A and Bag B.

```
1 # Using the intersection method
2 set_intersection = set_a.intersection(set_b)
3 print(set_intersection) # Output: {3}
4
5 # Another way to achieve the same result is using the "&" operator
6 set_intersection_2 = set_a & set_b
7 print(set_intersection_2) # Output: {3}
```

3. Difference

The difference between two sets finds the elements that are present in the first set but *not* in the second set. It's like finding the marbles that are in Bag A but not in Bag B.

```
1 # Using the difference method
2 set_difference = set_a.difference(set_b)
3 print(set_difference) # Output: {1, 2}
```

Handy Built-in Functions

Python comes with many built-in functions that can save you time and effort. Let's look at a few useful ones.

1. abs(): Absolute Value

This function returns the absolute value of a number. Think of it as the distance from zero, regardless of whether the number is positive or negative.

```
1 number = -10
2 absolute_value = abs(number)
3 print(absolute_value) # Output: 10
```

2. pow(): Power

This function calculates the power of a number. For example, 2 raised to the power of 4 (2^4).

```
1 base = 2
2 exponent = 4
3 result = pow(base, exponent)
4 print(result) # Output: 16
```

3. round(): Rounding Numbers

This function rounds a number to a specified number of decimal places.

```
1 pi_value = 3.14159
2 rounded_pi = round(pi_value, 2) # Round to 2 decimal places
3 print(rounded_pi) # Output: 3.14
```

4. divmod(): Division with Remainder

This function performs division and returns both the quotient and the remainder.

```
1 numerator = 10
2 denominator = 3
3 quotient, remainder = divmod(numerator, denominator)
4 print(quotient, remainder) # Output: 3 1
5 print(type((quotient, remainder))) # Output: <class 'tuple'>
```

Notice that `divmod()` returns the result as a *tuple*. A tuple is an immutable data structure that's used to store related data together. A key learning is that function designers choose the right datatypes to store the information in; in this example, Python has intelligently given the result in a tuple so that it is immutable, and we are prevented from changing the results of an operation.

5. max(), min(), and sum(): List Operations

These functions are useful for working with lists of numbers.

- `max()`: Returns the largest number in the list.
- `min()`: Returns the smallest number in the list.

- `sum()`: Returns the sum of all the numbers in the list.

```

1 numbers = [1, 5, 2, 8, 3]
2 maximum = max(numbers)
3 minimum = min(numbers)
4 total = sum(numbers)
5
6 print(maximum) # Output: 8
7 print(minimum) # Output: 1
8 print(total)   # Output: 19

```

6. `sqrt()` and `factorial()` from the `math` Module

Before you can use these functions, you need to import the `math` module.

```

1 import math
2
3 number = 16
4 square_root = math.sqrt(number)
5 print(square_root)
6
7 number = 4
8 factorial = math.factorial(number)
9 print(factorial)

```

The lesson here is, some basic math functions like `sum`, `min`, `max` are built into Python. However, for more complex mathematical functions like square root and factorial, you need to import the `math` module.

Working with Strings

Strings are sequences of characters. Python provides many built-in functions for manipulating strings.

```

1 text = " Hello Tuesday "

```

1. `len()`: String Length

This function returns the number of characters in a string, including spaces.

```

1 string_length = len(text)
2 print(string_length) # Output: 17

```

2. `upper()` and `lower()`: Case Conversion

These functions convert a string to uppercase or lowercase, respectively.

```

1 uppercase_text = text.upper()
2 lowercase_text = text.lower()
3
4 print(uppercase_text) # Output:    HELLO TUESDAY
5 print(lowercase_text) # Output:    hello tuesday

```

3. replace(): Replacing Substrings

This function replaces a specified substring with another substring.

```
1 new_text = text.replace("Tuesday", "Holiday")
2 print(new_text) # Output: Hello Holiday
```

4. strip(): Removing Whitespace

This function removes leading and trailing whitespace from a string.

```
1 stripped_text = text.strip()
2 print(stripped_text) # Output: Hello Tuesday
```

5. split(): Splitting Strings

This function splits a string into a list of substrings based on a specified delimiter.

```
1 words = stripped_text.split(" ")
2 print(words) # Output: ['Hello', 'Tuesday']
3
4 # Another Example
5 text = "one,two,three"
6 numbers = text.split(",")
7 print(numbers) # Output: ['one', 'two', 'three']
```

6. startswith(): Checking String Start

This function checks if a string starts with a specified prefix.

```
1 starts_with_hello = stripped_text.startswith("Hello")
2 print(starts_with_hello) # Output: True
```

7. find(): Finding Substrings

This function finds the first occurrence of a substring within a string and returns its index. If the substring is not found, it returns -1.

```
1 index = stripped_text.find("Tue")
2 print(index) # Output: 6
3
4 index = stripped_text.find("Wed")
5 print(index) # Output: -1
```

Writing Modular Code with Functions

As your programs grow, it's essential to organize your code into reusable blocks. That's where functions come in.

A function is a named block of code that performs a specific task. It can take inputs (arguments) and return an output (return value).

```
1 def string_length(text):
2     count = 0
3     for char in text:
4         count += 1
5     return count
6
7 text = "Hello Python"
8 length = string_length(text)
9 print(length) # Output: 12
```

1. Default Parameters

You can provide default values for function parameters. If the caller doesn't provide a value for the parameter, the default value is used.

```
1 def greet(name="Guest"):
2     print("Hello, " + name)
3
4 greet("Alice") # Output: Hello, Alice
5 greet()         # Output: Hello, Guest
```

Inline Execution with Lambda Functions

Lambda functions are small, anonymous functions that can be defined inline. They are often used for short, simple operations.

```
1 cube = lambda x: x * x * x
2 print(cube(3)) # Output: 27
3
4 number = lambda x: x * 10
5 print(number(3)) # Output: 30
```

Congratulations! You've taken your first steps towards mastering Python for AI. You've learned about sets, built-in functions, strings, functions, and lambda expressions. Keep practicing and exploring, and you'll be well on your way to becoming an AI expert!

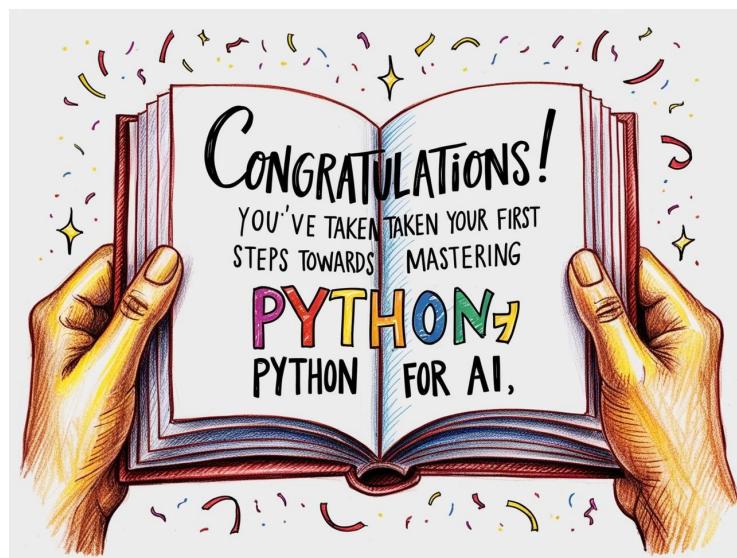


Figure 3: Keep exploring AI with Python

Minor in AI

Notes

04 March

Title: Mastering Data Visualization with Matplotlib: From Basics to Stunning Plots!

Syllabus:

- Customize Plots for Better Visualization
- Modify colors, markers, and line styles for better clarity.
- Add legends, annotations, and grid lines to improve interpretability.
- Handle Multiple Subplots: Use plt.subplot() or plt.subplots() to create multiple visualizations in a single figure.
- Adjust layout spacing using plt.tight_layout().
- Save and Display Plots Efficiently, Export figures in different formats (.png, .jpg, .pdf), Optimize figure size and resolution for presentations and reports.

Activities carried out:

- Design of the city: Place Charles de Gaulle, 12 avenues
- NetLogo model discussed: Path
- We don't need plots at all places – Stats dashboard from Championship Trophy
- Worm Graphs

What Intern did:

Case 1: An Intern was asked to analyze data on ice cream sales and shark attacks over the summer months. He noticed that as ice cream sales go up, shark attacks also increase! To prove his point, he made a line plot connecting the two trends.

"Look! The lines move together! That must mean eating ice cream causes sharks to attack!"

Case 2: Intern was asked to analyze how concert ticket prices impact the number of people attending a concert for different events. He decided to use a histogram to show the relationship.

"I made a histogram showing how many concerts had different ticket price ranges! That should explain everything, right?"

Code Snippets that we did hands-on

(Refer to video and collab for more)

```
import matplotlib.pyplot as plt

# Create a figure
plt.figure(figsize=(10, 5))

# Sample datasets
datasets = [1, 2, 3, 4, 5]

# Accuracy of Model A on different datasets
accuracy_model_a = [85, 88, 90, 87, 92]

# Accuracy of Model B on different datasets
accuracy_model_b = [80, 83, 85, 86, 89]

# First subplot - Model A Accuracy
plt.subplot(1, 2, 1)
plt.plot(datasets, accuracy_model_a, marker='o',
color='blue', label="Model A")
plt.title("Model A Accuracy")
plt.xlabel("Dataset")
plt.ylabel("Accuracy (%)")
plt.ylim(auto=True)
plt.grid()
plt.legend()

# Second subplot - Model B Accuracy
plt.subplot(1, 2, 2)
plt.plot(datasets, accuracy_model_b, marker='s',
color='red', label="Model B")
plt.title("Model B Accuracy")
plt.xlabel("Dataset")
plt.ylabel("Accuracy (%)")
plt.ylim(auto=True)
plt.grid()
plt.legend()

# Adjust layout and show the plots
plt.tight_layout()
plt.show()
```

```
import matplotlib.pyplot as plt

# Training cycles (iterations where AI learns more memes)
training_cycles = list(range(1, 11))

# Accuracy of AI in recognizing dank memes over time
meme_recognition_accuracy = [30, 40, 50, 60, 70, 78, 83, 87, 90, 93]

# Create the plot
plt.figure(figsize=(8, 5))
plt.plot(training_cycles, meme_recognition_accuracy,
marker='o', linestyle='-', color='purple', label="AI Meme Detector")

# Add titles and labels
plt.title("AI Learning to Recognize Dank Memes")
plt.xlabel("Training Cycle (More Memes Shown)")
plt.ylabel("Accuracy (%)")
plt.ylim(20, 100) # Setting limits for better visualization
plt.grid()
plt.legend()

# Save the figure as an image
plt.savefig("meme_ai_accuracy.png", dpi=300)

# Show the plot
plt.show()
```

```
import matplotlib.pyplot as plt
import numpy as np

# Generate sample data
x = np.linspace(0, 10, 100)
y = np.sin(x)

# Create a figure with optimized size and resolution
plt.figure(figsize=(10, 6), dpi=300)
# Plot data
plt.plot(x, y, label='Sine Wave', linewidth=2)

# Labels and title
plt.xlabel('X-axis', fontsize=14)
plt.ylabel('Y-axis', fontsize=14)
plt.title('Optimized Plot for Presentation and Reports', fontsize=16)

# Grid and legend
plt.grid(True, linestyle='--', alpha=0.6)
plt.legend(fontsize=12)

# Adjust layout
plt.tight_layout()

# Save as high-resolution image
plt.savefig('optimized_plot.png', dpi=300,
bbox_inches='tight')

# Show plot
plt.show()
```

Building Intelligent Systems: Object-Oriented "Pro"gramming Using AI-Powered Coding

IIT Ropar - Minor in AI

6th Feb, 2025

The Supermarket Challenge - Why Object-Oriented Programming?

Imagine Rajesh, the owner of a busy supermarket. He's facing a problem: too many salespeople tripping over each other, long queues at the checkout, and constant worries about keeping track of the stock. He dreams of a supermarket where inventory and billing are automated. He wants to build a system that accurately tracks which items are in stock and how many items are there, and also automatically calculate how much a customer owes.



Figure 1: A crowded supermarket scene vs. a streamlined, automated checkout system!

How can Rajesh achieve this dream?

Initially, Rajesh might think of writing a set of step-by-step instructions for the computer to follow. This is called **procedural programming**. It's

like giving the computer a recipe: do this, then do that, then do this other thing. While this can work for simple tasks, it can quickly become complicated and difficult to manage as the supermarket grows and adds more products or introduces new features like membership cards and discounts.

To understand how procedural programming works, let's assume the user has bought a notebook, pen and an eraser with their corresponding prices as 150, 20 and 5 Rs, and quantities as 100, 200 and 300.

```
1 # Product details in inventory
2 product_names = ['notebook', 'pen', 'eraser']
3 product_price = [150, 20, 5]
4 product_qty = [100, 200, 300]
5
6 # Function to display all the products
7 def display_products():
8     print("Available Products:")
9     for i in range(len(product_names)):
10         print(f"{i+1}. {product_names[i]} - Rs.{product_price[i]} - "
11             f"Quantity: {product_qty[i]}")
12 display_products()

Available Products:
1. notebook - Rs.150 - Quantity: 100
2. pen - Rs.20 - Quantity: 200
3. eraser - Rs.5 - Quantity: 300
```

Figure 2: Output!

That is where **object-oriented programming (OOP)** comes to the rescue. OOP is a way of organizing our code to represent things in the real world as **objects**. Think of each product in the supermarket as an object, each with its own characteristics (name, price, quantity) and actions it can perform (display its details, update its stock level).

This module will guide you through transitioning from procedural programming to OOP, using Rajesh's supermarket as a real-world example and leveraging the power of AI-powered coding assistants to help you along the way. By the end, you'll be equipped to build intelligent systems that are easier to understand, maintain, and extend.

Introduction to Object-Oriented Programming

Object-oriented programming is a programming paradigm centered around "objects." An object is a self-contained entity that combines data and code to manipulate that data. Let's break down the key concepts:

- **Class:** Think of a class as a blueprint or a template for creating objects. In Rajesh's supermarket, the "Product" class defines what a product is (its attributes) and what it can do (its behaviors or methods).

- **Object:** An object is an instance of a class. So, a "notebook" or "pen" would be objects created from the "Product" class.
- **Attributes:** Attributes are the characteristics or properties of an object. For a "Product" object, the attributes might be its name, price, and the quantity in stock.
- **Methods:** Methods are actions or behaviors that an object can perform. For a "Product" object, methods might include displaying its details or updating its stock level.

Building the "Product" Class

Let's create a "Product" class in Python to represent the products in Rajesh's supermarket.

```

1  class Product:
2      def __init__(self, name, price, quantity):
3          self.name = name
4          self.price = price
5          self.quantity = quantity
6
7      def display(self):
8          print(f"{self.name} - Rs.{self.price} (Stock: {self.
9              quantity})")
10
11     def update_stock(self, qty):
12         if self.quantity >= qty:
13             self.quantity -= qty
14             return self.price * qty
15         else:
16             print(f"Not enough stock for {self.name}. Available: {self.
17                 quantity}")
18             return 0

```

- `class Product:` This line defines a new class named "Product."
- `def __init__(self, name, price, quantity):`: This is the constructor method. It's called automatically when you create a new object of the "Product" class.
 - `self`: `self` always represents the instance (the object itself) that is calling the method.
 - `name, price, quantity`: These are the parameters used to initialize the object's attributes.
 - `self.name = name, self.price = price, self.quantity = quantity`: These lines assign the values passed as arguments to the object's attributes.
- `def display(self):`: This is a method to display the product's details. It prints the name, price, and stock quantity.

- `def update_stock(self, qty):`: This is a method to update the stock quantity after a customer makes a purchase.
 - It checks if there is enough stock before reducing the quantity.
 - If enough stock is available, it reduces the quantity and returns the total price of the purchased items.
 - If not enough stock is available, it prints a message and returns 0.

Creating Product Objects

Now that we have the "Product" class, we can create objects (instances) of that class.

```

1 # Create a list of Product instances
2 products = [
3     Product("Laptop", 50000, 10),
4     Product("Phone", 20000, 5),
5     Product("Headphones", 3000, 15)
6 ]

```

This code creates three "Product" objects: a "Laptop," a "Phone," and "Headphones." Each object has its own set of attributes (name, price, quantity) with different values. This is the beauty of OOP.

Calculating the Bill

Now let's create a function to calculate the bill for a customer.

```

1 def calculate_bill(products, cart):
2     total_bill = 0
3     print("\nProcessing Order...\n")
4
5     for product in products:
6         if product.name in cart:
7             qty = cart[product.name]
8             cost = product.update_stock(qty)
9             if cost > 0:
10                 print(f"{qty} x {product.name} added to bill: Rs.{cost}")
11                 total_bill += cost
12
13     print("\nOrder Processed Successfully!")
14     return total_bill

```

This code iterates through a customer's cart (a dictionary where keys are product names and values are the quantities to be purchased), updates the stock levels using the product object's `update_stock` method and calculates the total bill.

Streamlining User Input

To interact with the system, we need to gather the products a customer wants to buy:

```

1 # Get input for the cart from user
2 cart = {}
3 while True:
4     product_name = input("Enter the product name (or 'done' to
5         finish): ")
6     if product_name.lower() == 'done':
7         break
8     quantity = int(input("Enter the quantity: "))
9     cart[product_name] = quantity

```

This code allows the user to enter product names and quantities, adding them to a dictionary called "cart".

Putting It All Together

Let's put all the code together and run the supermarket billing system.

```

1 # Display available products before purchase
2 print("Available Products:")
3 for product in products:
4     product.display()
5
6 # Calculate the bill based on the cart
7 total = calculate_bill(products, cart)
8 print(f"\nTotal Bill Amount: Rs.{total}")
9
10 # Display product stock after the purchase
11 print("\nStock After Purchase:")
12 for product in products:
13     product.display()

```

```

Available Products:
Laptop - Rs.50000 (Stock: 10)
Phone - Rs.20000 (Stock: 5)
Headphones - Rs.3000 (Stock: 15)
Enter the product name (or 'done' to finish): Laptop
Enter the quantity: 2
Enter the product name (or 'done' to finish): done

Processing Order...

2 x Laptop added to bill: Rs.100000

Order Processed Successfully!

Total Bill Amount: Rs.100000

Stock After Purchase:
Laptop - Rs.50000 (Stock: 8)
Phone - Rs.20000 (Stock: 5)
Headphones - Rs.3000 (Stock: 15)

```

Figure 3: Final output!

This code displays the available products, prompts the user to enter their desired products and quantities, calculates the total bill amount, and then displays the remaining stock levels of each product after the purchase.

AI Powered Coding

As we've seen, transitioning to object-oriented programming offers significant advantages for building complex systems like Rajesh's supermarket. OOP allows us to organize code in a more modular and reusable way, making it easier to maintain and extend.

The process of transitioning from procedural to object-oriented programming, and AI tools like Gemini have assisted a great deal to streamline and help speed up the coding process.



Figure 4: AI tools assisting in code generation and optimization.

OOPS - Encapsulation & Inheritance

IIT Ropar - Minor in AI

7th Feb, 2025

1 Enchanted Animal Sanctuary

Welcome to the Enchanted Animal Sanctuary, a magical place where various creatures coexist harmoniously. The sanctuary's caretakers need a system to manage their unique inhabitants. Let's explore how Object-Oriented Programming (OOP) can help create an efficient and secure management system.



2 The Challenge

The sanctuary needs to:

1. Keep track of different animals
2. Protect sensitive information about the creatures
3. Allow for easy addition of new animal types

3 The Solution: Magical Creature Management System

Let's dive into our Python-based solution that showcases encapsulation and inheritance.

```
1 class Animal:
2     def __init__(self, name):
3         self._name = name      # Protected member
4         self.__species = "Magical Creature" # Private member
5
6     def get_species(self):    # Public method to access private
7         member
8         return self.__species
9
10    class Dog(Animal):
11        def __init__(self, name):
12            super().__init__(name)
13            self.__species = "Enchanted Canine" # Private member
14            specific to Dog
15
16        def get_species(self):    # Public method to access private
17            member
18            return self.__species
19
20        def bark(self):
21            return f"{self._name} says Woof with a magical echo!"
22
23 # Let's add a new inhabitant to our sanctuary
24 buddy = Dog("Buddy")
25
26 print(f"Inhabitant name: {buddy._name}")
27 print(buddy.bark())
28 print(f"Species: {buddy.get_species()}")
```

OUTPUT:

Inhabitant name: Buddy
Buddy says Woof with a magical echo!
Species: Enchanted Canine

4 Explaining the Magic

4.1 Encapsulation: Protecting Our Magical Secrets

In our `Animal` class:

- `self._name` is protected (single underscore). It's like a light invisibility cloak – accessible but signaling "handle with care".
- `self.__species` is private (double underscore). It's under a powerful concealment charm, hidden from direct outside access.

Why is this important? Imagine if anyone could change an animal's species at will – chaos in the sanctuary!

4.2 Inheritance: Passing Down Magical Traits

Our `Dog` class inherits from `Animal`:

- It gets all the basic "magical creature" properties.
- It adds its own twist with a special bark and a unique species.

This is like magical genetics – dogs inherit general animal traits but have their own special abilities.

4.3 Method Overriding: Customizing Our Magic

The `Dog` class overrides the `get_species` method: This is like a specialized enchantment overriding the general one from `Animal`. This override changes the behavior of the `get_species` method for `Dog` objects:

- In the `Animal` class, it returns `Magical Creature`.
- In the `Dog` class, it returns `Enchanted Canine`.

4.4 Accessing Our Magical Properties

```
1 buddy = Dog("Buddy")
2 print(buddy.bark())
3 # Output: Buddy says Woof with a magical echo!
```

Here, we're calling a public method. It's like asking Buddy to perform a trick – anyone can do this!

```
1 print(f"Species: {buddy.get_species()}")
2 # Output: Species: Enchanted Canine
```

We're using a public method to access a private attribute. It's like using a special spell to reveal hidden information.

```
1 print(f"Name: {buddy._name}") # Output: Name: Buddy
```

Directly accessing a protected member. It works, but it's frowned upon – like using a forbidden spell.

```
1 # print(buddy._species) # This would raise an AttributeError
```

Trying to access a private member directly. It's like attempting to break through a powerful protection spell – it just won't work!

5 Flying Twist

5.1 Adding a new inhabitant

A pigeon named "pichku" flies into our sanctuary and insists upon staying here. We then add pigeons too into our system.

```
1 class Pigeon(Animal):
2     def __init__(self, name):
3         super().__init__(name)
4         self._species = "Neighbourhood Pigoen"
5         self.flies = True
6
7     def get_species(self):
8         return self._species
9
10    def coo(self):
11        return f"{self._name} says Coo with a magical echo!"
12
13 # Let's add a new inhabitant to our sanctuary
14 pichku = Pigeon("Pichku")
15
16 print(f"Inhabitant name: {pichku._name}")
17 print(pichku.coo())
18 print(f"Species: {pichku.get_species()}")
19 print(f"Can fly: {pichku.flies}")
```

OUTPUT:

```
Inhabitant name: Pichku
Pichku says Coo with a magical echo!
Species: Neighbourhood Pigoen
Can fly: True
```

5.2 Finding Flying Inhabitants

Now, we create a function to find total number of inhabitants that can fly in our sanctuary.

```
1 # function to count number of Animals that can fly
2 def count_flying_animals(animals):
3     count = 0
4     for animal in animals:
5         if hasattr(animal, 'flies') and animal.flies:
6             count += 1
7     return count
8
```

```

9 # Current inhabitants of our sanctuary
10 buddy = Dog("Buddy")
11 pichku = Pigeon("Pichku")
12 brownie = Dog("Brownie")
13 tweety = Pigeon("Tweety")
14
15 # list of all inhabitants
16 animalsList = [buddy, pichku, brownie, tweety]
17
18 # count of flying inhabitants
19 print(count_flying_animals(animalsList))

```

OUTPUT:

2

There are total 2 flying inhabitants in our sanctuary currently.

6 The Results

Our Magical Creature Management System successfully:

1. Keeps track of different animals (like Buddy, Pichku)
2. Protects sensitive information (the private `_species`)
3. Allows for easy addition of new animal types (we can create more classes inheriting from `Animal`) through code reusability.

7 Your Turn: Expand the Sanctuary

Now that you've seen how our Magical Creature Management System works, it's time for you to add your own enchantment to the sanctuary!

7.1 Challenge: Introduce Magical Felines

The sanctuary has decided to welcome a new type of inhabitant: Enchanted Cats. Your mission, should you choose to accept it, is to create a `Cat` class similar to our `Dog` class.

Here's what you need to do:

1. Create a new class named `Cat` that inherits from `Animal`.
2. Initialize it with a name, just like the `Dog` class.
3. Set a private species attribute specific to cats.
4. Add a method called `meow()` that returns a string with the cat's name and a meow sound.

Here's a template to get you started:

```
1 class Cat(Animal):
2     pass
3
4 # Test your class here
5 # whiskers = Cat("Whiskers")
6 # print(whiskers.meow())
7 # print(f"Species: {whiskers.get_species()}")
```

7.2 Hints

- Remember to use `super().__init__(name)` to properly initialize the parent class.
- Use double underscores (`__`) for the private species attribute.
- Get creative with your magical meow sound!

Once you've implemented the `Cat` class, try creating a few cat instances and test out their methods. How does accessing their attributes compare to what we saw with the `Dog` class?

Try doing this challenge yourself rather than using AI tools to by comparing with the `Dog` class so, that you can reinforce your understanding of inheritance and encapsulation in Python.

8 Conclusion

Through the power of OOP, we've created a flexible and secure system for our Enchanted Animal Sanctuary. Encapsulation keeps our magical creatures' secrets safe, while inheritance allows us to easily expand our sanctuary with new types of enchanted animals.

Note: Please Try solving these questions to build better understanding, I have attached a solution in last page, please attempt on your own before you go to the solution

Question 1: (loop and conditional statements, Lists)

Display a menu with the following options:

1. Add a new task
2. View all tasks
3. Mark a task as done
4. Exit the program

Allow the user to add tasks to a list.

Display the list of tasks with their corresponding numbers.

Let the user mark a task as done, removing it from the list.

Keep running in a loop until the user chooses to exit.

Question 2: (Dictionaries, simple function)

Write a Python program to manage **Student Attendance**. Your program should:

1. Display a menu with the following options:
 - Add student attendance
 - Check student attendance
 - Exit the program
2. Allow the user to add attendance for a student by entering their name and marking them as either '**present**' or '**absent**'.
3. Allow the user to check a student's attendance status by entering their name.
4. Keep running in a loop until the user chooses to exit.

Ensure that your program handles invalid inputs gracefully.

Regards,

Arunav Ghosh(TA)

Solutions:

Question 1:

```
todo_list = []
```

```
while True:
```

```
    print("\n--- To-Do List Manager ---")
```

```
    print("1. Add task")
```

```
    print("2. View tasks")
```

```
    print("3. Mark task as done")
```

```
    print("4. Exit")
```

```
choice = input("Enter your choice (1-4): ")
```

```
if choice == '1':
```

```
    task = input("Enter a new task: ")
```

```
    todo_list.append(task)
```

```
    print("Task added successfully!")
```

```
elif choice == '2':
```

```
    if not todo_list:
```

```
        print("Your to-do list is empty.")
```

```
else:
    print("Your To-Do List:")
    for index, task in enumerate(todo_list, start=1):
        print(f"{index}. {task}")

elif choice == '3':
    if not todo_list:
        print("Your to-do list is empty.")
    else:
        print("Your To-Do List:")
        for index, task in enumerate(todo_list, start=1):
            print(f"{index}. {task}")

        task_num = input("Enter the number of the task to mark as done: ")
        if task_num.isdigit() and 1 <= int(task_num) <= len(todo_list):
            done_task = todo_list.pop(int(task_num) - 1)
            print(f"Marked '{done_task}' as done!")
        else:
            print("Invalid task number.")

elif choice == '4':
    print("Thank you for using the To-Do List Manager. Goodbye!")
    break

else:
    print("Invalid choice. Please try again.")
```

Question 2:

```
def manage_student_attendance():

    attendance = {}

    while True:

        print("\n--- Student Attendance Manager ---")

        print("1. Add student attendance")
        print("2. Check student attendance")
        print("3. Exit")

        choice = input("Enter your choice (1-3): ")

        if choice == '1':

            name = input("Enter student name: ")

            status = input("Enter status ('present' or 'absent'): ").lower()

            if status in ['present', 'absent']:

                attendance[name] = status

                print(f"Attendance for {name} marked as {status}.")"

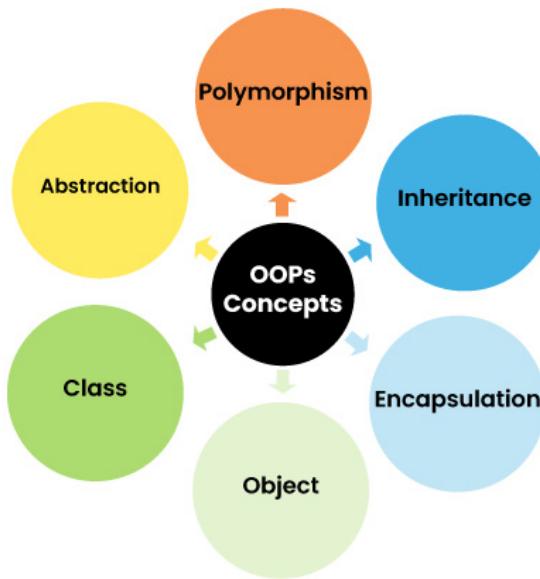
            else:

                print("Invalid status. Please enter 'present' or 'absent'.")
```

```
elif choice == '2':  
    name = input("Enter student name to check: ")  
    if name in attendance:  
        print(f"{name}'s attendance status: {attendance[name]}")  
    else:  
        print(f"{name} not found in attendance list.")  
  
elif choice == '3':  
    print("Exiting Student Attendance Manager.")  
    break  
  
else:  
    print("Invalid choice. Please try again.")  
  
manage_student_attendance()  
  
#note this last line runs the function
```

Minor in AI

Object-Oriented Paradigm in Python



1 Building Software Like a House

Imagine you are building a house. Before you start construction, you need a blueprint that outlines the structure, the number of rooms, and the layout. This blueprint is like a class in programming—it defines the structure and behavior of objects. Once the blueprint is ready, you can build multiple houses (objects) based on it. Each house will have the same structure but can have different attributes, like color or furniture. This is the essence of Object-Oriented Programming (OOP), a programming paradigm that uses objects and classes to structure code.

OOP principles like classes, objects, inheritance, and polymorphism can be applied to create a battle game in Python. By using OOP, we can simplify the code, make it more modular, and easily extend it by adding new characters with unique abilities.

2 Details

2.1 Problem Statement

When building a game, especially one with multiple characters and functionalities, the code can quickly become complex and hard to manage. For example, if each character has different abilities, writing separate code for each character would lead to redundancy and make the code difficult to maintain. This is where OOP comes into play.

2.2 How OOP Solves the Problem

OOP allows us to create a blueprint (class) for a player, which includes common attributes like health, attack, and defense. We can then create different characters (objects) based on this blueprint, each with its own special abilities. This approach reduces redundancy and makes the code more modular and easier to extend.

2.2.1 Key Concepts

- **Class:** A blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that the objects will have. For example, the `Player` class defines attributes like `name`, `hp`, and `defending`, and methods like `attack` and `defend`.
- **Object:** An instance of a class. For example, `player1` and `player2` are objects of the `Player` class.
- **Inheritance:** A mechanism that allows a new class to inherit properties and methods from an existing class. For example, the `Warrior`, `Healer`, and `Rogue` classes inherit from the `Player` class and add their own special abilities.
- **Polymorphism:** The ability of different objects to respond to the same method in different ways. For example, the `use_special` method behaves differently for each character class.

3 Code Implementation

The code provided in the session demonstrates how to use OOP to create a battle game. Here's a breakdown of the key components:

```

1 \begin{lstlisting}[style=pythonstyle, caption={Player class with OOP
2     concepts in Python}]
3 import random
4 import time
5
5 class Player:
6     # Constructor method to initialize player attributes
7     def __init__(self, name):
8         self.name = name # Player name
9         self.hp = 100 # Player's health points
10        self.defending = False # Flag to check if player is defending
11        self.specialUsed = False # Flag to check if special move has
12            been used
13
13    # Method for attacking an opponent
14    def attack(self, opponent):
15        damage = random.randint(10, 20) # Generate random damage
16            between 10 and 20
17        print("Generating attack....")
18        time.sleep(2) # Simulate attack delay
19
19        if opponent.defending: # If opponent is defending, reduce
20            damage
21            damage = damage // 2
22            opponent.defending = False # Reset opponent's defending
23                status
24
24        opponent.hp -= damage # Deduct damage from opponent's HP
25        print(self.name, "attacks", opponent.name, "for", damage, "
26            damage!")
26
# Method to enable defense mode

```

```
27     def defend(self):
28         self.defending = True # Set defending flag to True
29         print(self.name, "is defending!")
30
31     # Placeholder method for a special move
32     def use_special(self, opponent):
33         pass # Special move logic can be implemented later
```

4 Explanation of the Code

4.1 Class Definition

The class `Player` is defined using the `class` keyword, representing a player in a game.

4.2 Constructor Method (`__init__`)

- The `__init__` method is a constructor that initializes the player's attributes.
- `self.name`: Stores the player's name.
- `self.hp`: Stores the player's health points, initialized to 100.
- `self.defending`: A boolean flag to track whether the player is in defense mode.
- `self.specialUsed`: A boolean flag to track if the player has used their special move.

4.3 `attack()` Method

- This method allows the player to attack an opponent.
- The damage is randomly generated between 10 and 20 using `random.randint(10, 20)`.
- A message is printed: "Generating attack...." to indicate an attack is happening.
- A delay of 2 seconds is introduced using `time.sleep(2)` to simulate attack preparation time.
- If the opponent is defending, the damage is halved.
- The opponent's `hp` is reduced by the computed damage.
- A message is displayed showing the attack details.

4.4 `defend()` Method

- This method sets `self.defending` to `True`, indicating that the player is in defense mode.
- A message is printed to indicate that the player is defending.

4.5 `use_special()` Method

- This method is a placeholder for implementing a special attack.
- The `pass` keyword ensures that the method is syntactically correct without executing any logic.
- This method can be extended to include a powerful attack or special ability.

5 Conclusion

The `Player` class demonstrates the fundamental principles of Object-Oriented Programming (OOP), including:

- Encapsulation: The attributes and methods are bundled into a class.
- Abstraction: The `use_special()` method is defined but not yet implemented, showing how abstraction works.
- Interaction: The `attack()` method allows objects (players) to interact with each other.

This class can be further extended by implementing the `use_special()` method and introducing additional game mechanics.

The `Player` class serves as the base class for all characters. It includes methods for attacking, defending, and using special abilities. The `Warrior`, `Healer`, and `Rogue` classes inherit from `Player` and override the `use_special` method to implement their unique abilities.

5.1 Gameplay

The game allows two players to choose their characters and take turns attacking, defending, or using their special abilities. The game continues until one player's health drops to zero. The code is designed to be modular, making it easy to add new characters or modify existing ones.

6 Conclusion

Object-Oriented Programming is a powerful paradigm that helps in organizing and structuring code, especially in complex applications like games. By using classes and objects, we can create reusable and modular code that is easier to maintain and extend. OOP principles like inheritance and polymorphism can be used to create a battle game with multiple characters, each with unique abilities.

6.1 Key Takeaways

- OOP helps in organizing code by creating reusable blueprints (classes) for objects.
- Inheritance allows us to create new classes based on existing ones, reducing redundancy.

- Polymorphism enables different objects to respond to the same method in different ways.
- OOP makes code more modular, easier to maintain, and extendable.

6.2 Resources

- [Colab Notebook for the Session](#)

Minor in AI

File Handling in Python

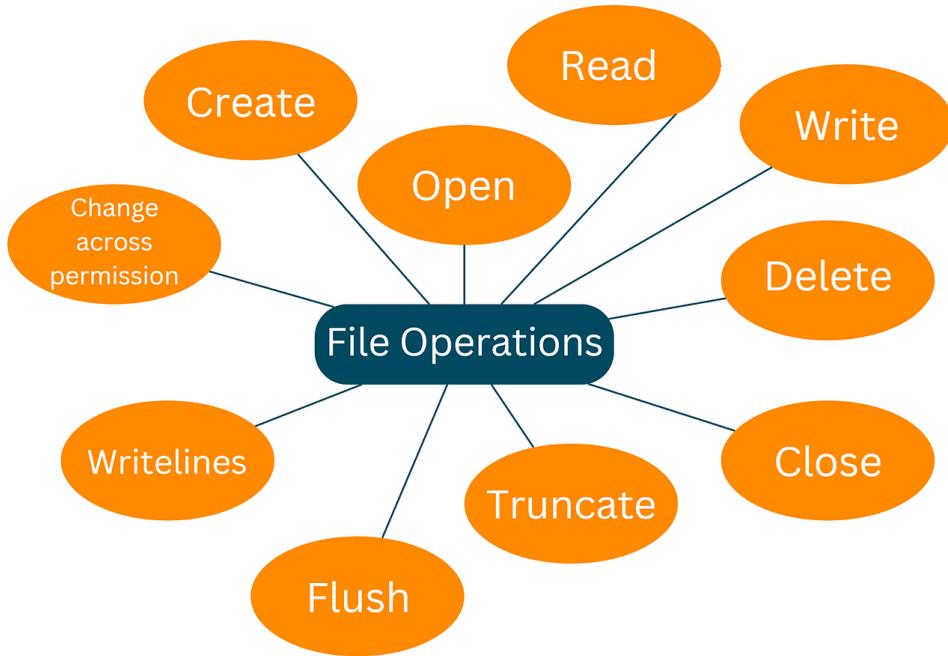


Figure 1: File Handling

1 Smart Book-Keeping with Python File Handling

Imagine you're a teacher managing records for hundreds of students. You need to store their names, grades, and attendance. How would you handle this data efficiently? This is where file handling in Python comes to the rescue.

File handling allows us to store and retrieve large amounts of data persistently. Unlike variables that lose their values when a program ends, files retain information even after the program closes. This makes files crucial for managing data in real-world applications.

2 Understanding File Handling

File handling in Python involves working with external files for storing and retrieving data. It's a fundamental concept in programming that allows us to work with persistent data.

2.1 Key Concepts

- **Opening a file:** Before we can work with a file, we need to open it using the `open()` function.
- **File modes:** These determine how we interact with the file (read, write, append).
- **Reading from a file:** Extracting data from the file.
- **Writing to a file:** Adding/Appending new data to the file.
- **Closing a file:** It's crucial to close files after operations to free up system resources.

2.2 File Modes

- ‘r’: Read mode
- ‘w’: Write mode (overwrites existing content)
- ‘a’: Append mode (adds to existing content)
- ‘r+’: Read and write mode without overwriting
- ‘w+’: Read and write mode with overwriting
- ‘a+’: Append and read mode

3 Code Implementation

Let's implement a simple student record system to demonstrate file handling:

```

1 # Open file in write mode and initialize with a header
2 with open('student_records.txt', 'w') as file:
3     file.write("Student Records\n") # Writing the title of the records
4     file.write("-----\n") # Adding a separator line
5
6 # Function to add a student record to the file
7 def add_student(name, grade):
8     with open('student_records.txt', 'a') as file: # Open file in
9         append mode
10        file.write(f"{name}: {grade}\n") # Write the student's name and
11        grade
12
13 # Function to read and display all student records from the file
14 def read_records():
15     with open('student_records.txt', 'r') as file: # Open file in read
16         mode
17     print(file.read()) # Read and print the contents of the file
18
19 # Adding student records
20 add_student("Alice", "A") # Adding Alice with grade A
21 add_student("Bob", "B") # Adding Bob with grade B
22 add_student("Charlie", "A-") # Adding Charlie with grade A-
23
24 # Reading and displaying the records
25 read_records() # Display all student records

```

This code creates a file, adds student records, and then reads and displays them.

4 Diving Deeper

4.1 Exception Handling

4.1.1 Basic Example: Handling File Not Found

```
1 # Attempting to open a file that may not exist
2 try:
3     with open('nonexistent_file.txt', 'r') as file:
4         content = file.read() # Trying to read the file's content
5 except FileNotFoundError:
6     # Handles the case where the file does not exist
7     print("The file does not exist.")
```

4.1.2 Handling Multiple Exceptions

```
1 try:
2     num = int(input("Enter a number: ")) # Taking user input and
3         # converting to integer
4     result = 10 / num # Performing division
5 except ValueError:
6     # Handles the case where input is not a valid integer
7     print("Invalid input! Please enter a number.")
8 except ZeroDivisionError:
9     # Handles division by zero error
10    print("Cannot divide by zero!")
```

4.1.3 Using finally for Cleanup

```
1 try:
2     file = open("example.txt", "r") # Attempting to open a file
3     content = file.read() # Reading the file content
4 except FileNotFoundError:
5     # Handles the case where the file does not exist
6     print("File not found.")
7 finally:
8     # This block always executes, regardless of exceptions
9     print("Execution completed.")
```

4.2 Working with CSV Files

4.2.1 What is CSV?

CSV (Comma-Separated Values) files are commonly used for storing tabular data. Each line in a CSV file represents a row, and the columns are separated by commas. CSV files are widely used in data science and machine learning for storing datasets, as they provide an easy way to represent data in a simple text format.

4.2.2 Where is CSV?

In Machine Learning, CSV files are commonly used to store structured datasets, including both input data and target data. After reading the data from a CSV file, it is often preprocessed and used for training machine learning models.

4.2.3 Accessing CSV:

Using libraries like `csv` in Python, we can easily read and write these files for data preprocessing and model training.

```
1 import csv # Importing the CSV module
2
3 # Writing to a CSV file
4 with open('students.csv', 'w', newline='') as file:
5     writer = csv.writer(file) # Create a writer object to write to the
       CSV file
6     writer.writerow(["Name", "Grade"]) # Writing the header row
7     writer.writerow(["Alice", "A"]) # Writing data for Alice
8     writer.writerow(["Bob", "B"]) # Writing data for Bob
9
10 # Reading from a CSV file
11 with open('students.csv', 'r') as file:
12     reader = csv.reader(file) # Create a reader object to read the CSV
       file
13     for row in reader: # Iterating over each row in the CSV file
14         print(row) # Print each row of data
```

Explanation:

- `csv.writer` is used to write data to a CSV file.
- `csv.reader` is used to read data from a CSV file.
- The `writerow()` method writes a single row of data to the CSV file.
- The `for row in reader` loop reads each row from the file, which is then printed.

5 Conclusion

File handling in Python provides a powerful way to work with persistent data. It's essential for many real-world applications, from simple record-keeping to complex data analysis followed by ML and AI. By understanding file handling, one can create more robust and useful programs.

6 Additional Resources

- [Google Colab File](#)

Welcome to Algorithmic Thinking with Python and AI

Welcome, aspiring Python and AI explorers! This module is designed to gently guide you through the fascinating world of algorithms, the essential building blocks of any intelligent system. We'll be taking a "slow thinking" approach, carefully dissecting each concept and writing code from scratch to solidify your understanding.

What are Algorithms and Why Should You Care?

Imagine you're teaching a toddler how to make a peanut butter and jelly sandwich. You wouldn't just say, "Make a sandwich!" You'd break it down into simple, step-by-step instructions:

1. Take out two slices of bread.
2. Open the peanut butter jar.
3. Spread peanut butter on one slice of bread.
4. Open the jelly jar.
5. Spread jelly on the other slice of bread.
6. Put the two slices of bread together.
7. Enjoy your sandwich!

These step-by-step instructions are an algorithm!

Algorithms are simply a set of well-defined instructions for solving a problem. We need to study them because they are the foundation for writing efficient, high-quality code. While tools like ChatGPT can generate code, understanding the underlying logic is crucial for developing truly powerful AI solutions. Think of algorithms as the weights you lift in the gym. You don't just *want* the result (stronger muscles), you get there by doing the work (lifting the weights). Each algorithm you master strengthens your problem-solving abilities.

In this module, we'll focus on understanding the *intuition* behind each algorithm. We'll learn how to "think slow," break down complex problems, and then translate those steps into Python code.

Python Basics Refresher

Before diving into complex algorithms, let's refresh some fundamental Python concepts.



Figure 1: Taking it step-by-step!

Printing to the Screen

The `print()` function is your best friend for displaying information.

```
1 print("Hello") # Outputs: Hello
2 print("World") # Outputs: World
```

Each `print()` statement displays its output on a new line.

Loops: Doing Things Repeatedly

What if you wanted to print "Hello" five times? You could write `print("Hello")` five times. But that's tedious! Loops allow you to execute a block of code multiple times. The `for` loop is particularly useful.

```
1 for i in range(0, 5):
2     print("Hello")
```

This code prints "Hello" five times, each on a new line.

Let's break down this `for` loop:

- `for i in range(0, 5)`: This line tells Python to iterate through a sequence of numbers generated by the `range()` function.
- `range(0, 5)`: This function generates numbers starting from 0 and going up to (but *not including*) 5. So, it produces the sequence: 0, 1, 2, 3, 4.
- `i`: The variable `i` takes on each value from the `range()` sequence in each iteration of the loop. It acts as a counter.
- `print("Hello")`: This line is executed in each iteration of the loop.

Example: Printing numbers from 1 to 10

```
1 for i in range(1, 11):
2     print(i)
```

Here, `range(1, 11)` generates numbers from 1 to 10 (inclusive).

Lists: Collections of Data

A list is an ordered collection of items. You can store numbers, strings, or even other lists in a list!

```
1 my_list = [10, 20, 30, 40, 50, 60, 70]
```

- `my_list`: This is the name of the list.
- `[10, 20, 30, 40, 50, 60, 70]`: These are the elements of the list, enclosed in square brackets.

Accessing List Elements:

You can access individual elements in a list using their *index*. The index starts at 0 for the first element, 1 for the second, and so on.

```
1 print(my_list[0]) # Outputs: 10 (the first element)
2 print(my_list[1]) # Outputs: 20 (the second element)
```

List Length:

The `len()` function returns the number of elements in a list.

```
1 n = len(my_list)
2 print(n) # Outputs: 7
```

Iterating Through a List:

You can use a `for` loop to access each element of a list.

```
1 n = len(my_list)
2 for i in range(0, n):
3     print(my_list[i])
```

This code will print each element of `my_list` on a new line.

Linear Search: Finding a Needle in a Haystack

Imagine you have a list of student names, and you want to check if a specific name is on that list. Linear search is the simplest way to do this. You go through the list, one name at a time, until you find the name you're looking for or reach the end of the list.

The "Slow Thinking" Approach:

Let's say our list is `[1, 5, 4, 2, 8]` and we're searching for 4. Think about how *you* would do this, but *really slowly*:

1. Look at the first number: 1. Is it 4? No.
2. Look at the second number: 5. Is it 4? No.
3. Look at the third number: 4. Is it 4? Yes! We found it!

Python Code:

```

1 my_list = [1, 5, 4, 2, 8]
2 n = len(my_list)
3 found = False # Initially, we haven't found the number
4 key = 4 # The number we're searching for
5
6 for i in range(0, n):
7     if my_list[i] == key:
8         found = True # We found it!
9         break # No need to search further
10
11 if found:
12     print("Found")
13 else:
14     print("Not Found")

```

- `found = False`: This variable acts as a flag. It starts as `False` and becomes `True` if we find the number.
- `key = 4`: This variable stores the number we're searching for.
- `if my_list[i] == key`: This checks if the current element (`my_list[i]`) is equal to the number we're searching for (`key`).
- `break`: Once the element is found, the `break` statement is used to exit the loop.

Selection Sort: Putting Things in Order

Imagine you have a deck of cards that are not sorted (i.e not ordered). Now, you want to sort them in ascending order (from the smallest to the largest card). The selection sort algorithm is a step-by-step recipe for accomplishing this task.

The "Slow Thinking" Approach:

Let's say you have the numbers: `[5, 1, 4, 2, 8]`

1. Find the smallest number: Scan the entire list and find the smallest number, which is 1.
2. Swap: Put the smallest number at the beginning of the list. Swap 5 and 1: `[1, 5, 4, 2, 8]`
3. Repeat: Now, ignore the first element (1 is already in the correct position). Find the smallest number in the *remaining* list (`[5, 4, 2, 8]`), which is 2.
4. Swap: Swap 5 and 2: `[1, 2, 4, 5, 8]`
5. Continue: Repeat this process for the rest of the list.

Python Code:

```

1 my_list = [5, 1, 4, 2, 8]
2 n = len(my_list)
3
4 for i in range(0, n):
5     min_idx = i # Assume the current element is the minimum
6     for j in range(i + 1, n):
7         if my_list[j] < my_list[min_idx]:
8             min_idx = j # Found a smaller element
9
10    # Swap the current element with the minimum element
11    my_list[i], my_list[min_idx] = my_list[min_idx], my_list[i]
12
13 print(my_list) # Outputs: [1, 2, 4, 5, 8]

```

- `for i in range(0, n):`: This outer loop iterates through each position in the list.
- `min_idx = i`: This assumes that the element at the current position `i` is the minimum.
- `for j in range(i + 1, n):`: This inner loop iterates through the *remaining* list (from `i + 1` to the end).
- `if my_list[j] < my_list[min_idx]`: This checks if the current element `my_list[j]` is smaller than the current minimum element `my_list[min_idx]`. If it is, we update `min_idx` to the new minimum's index.
- `my_list[i], my_list[min_idx] = my_list[min_idx], my_list[i]`: This line swaps the element at the current position `i` with the smallest element we found (at index `min_idx`).

Conclusion

Congratulations! You've taken your first steps into the world of algorithms. Remember the power of "slow thinking." By breaking down complex problems into simple steps and translating those steps into code, you can master any algorithm. Keep practicing, and you'll be well on your way to building intelligent systems.

Sort and search

IIT Ropar - Minor in AI

13th Feb, 2025

1 Efficient Book Lookup in a Digital Library

Imagine you are working in a university digital library. The library contains millions of books categorized by their ISBN numbers. Your task is implementing a system that efficiently retrieves book details based on ISBN queries. A naive linear search approach would be too slow, making the system inefficient for users.

1.1 Challenges

- The dataset is vast, containing millions of books.
- Users expect real-time search results.
- A traditional linear search would result in high computational time.
- The system needs to be scalable for future expansions.

1.2 Solution: Implementing Binary Search

Since the ISBN numbers are pre-sorted, Binary Search can be applied effectively. Instead of scanning each book sequentially, Binary Search divides the dataset into halves, reducing the search complexity. This optimization significantly improves query response time.

1.3 Implementation Steps

1. Ensure the book database is sorted by ISBN.
2. Use Binary Search to locate the desired ISBN efficiently.
3. Retrieve and display the book information to the user.

1.4 Results and Benefits

- Query response time improved significantly from linear to logarithmic scale.
- Enhanced user experience with faster search results.

- Scalability ensured as the system grows with new book additions.

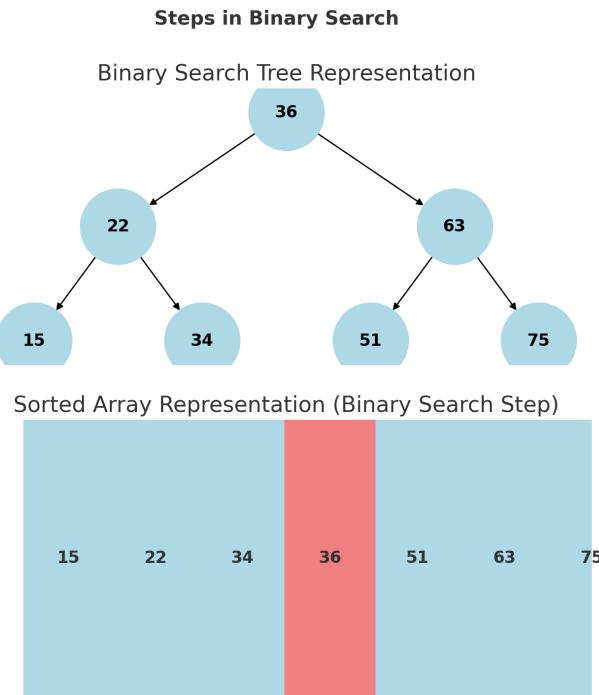


Figure 1: Binary Seacrh

2 Binary Search Code Implementation

Below is the Python implementation of Binary Search with detailed comments:

Listing 1: Binary Search

```

1 # Function to perform Binary Search
2 def binary_search(arr, target):
3     """
4         Performs binary search on a sorted array.
5         :param arr: List of sorted elements
6         :param target: The element to search for
7         :return: Index of target if found, else -1
8     """
9     left, right = 0, len(arr) - 1 # Initialize search bounds
10
11    while left <= right:
12        mid = left + (right - left) // 2 # Find the middle index
13
14        if arr[mid] == target:
  
```

```

15     return mid # Target found, return index
16 elif arr[mid] < target:
17     left = mid + 1 # Ignore left half
18 else:
19     right = mid - 1 # Ignore right half
20
21 return -1 # Target not found
22
23 # Example usage:
24 data = [15, 22, 34, 36, 51, 63, 75]
25 target_value = 36
26 result = binary_search(data, target_value)
27 print("Element found at index:" if result != -1 else "Element not
    found", result)

```

2.1 Explanation of Code

- The function accepts a sorted list and a target value.
- It initializes two pointers, `left` and `right`, to mark the search boundaries.
- A `while` loop runs as long as `left` is less than or equal to `right`.
- The middle index `mid` is calculated to divide the search space.
- If the middle element matches the target, its index is returned.
- If the middle element is smaller, we move the left boundary to `mid + 1`.
- If the middle element is larger, we move the right boundary to `mid - 1`.
- If the loop terminates without finding the target, `-1` is returned.

3 Sorting Student Attendance Records

A school maintains attendance records of students in small-sized classes. The records are kept in an unordered list, and the administration needs a simple way to sort them based on roll numbers before generating reports.

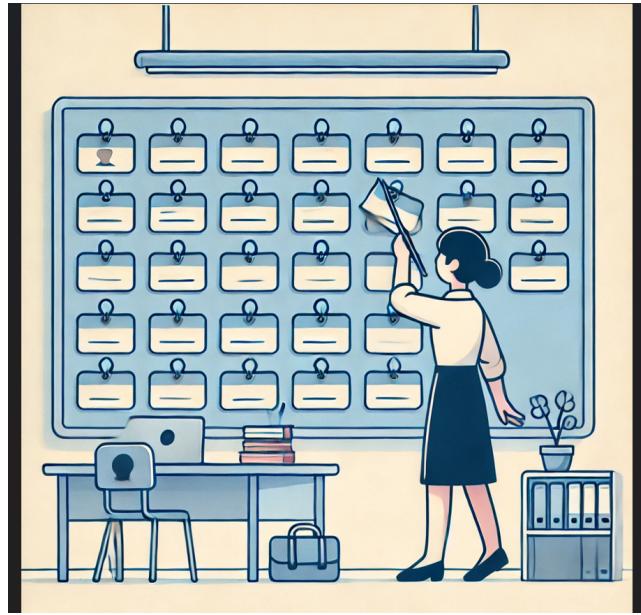


Figure 2: Sort Attendance records

3.1 Challenges

- The number of students per class is relatively small (typically 30-50).
- The sorting process needs to be implemented quickly without complex algorithms.
- The school's system has limited computational resources.
- Teachers without technical expertise should be able to understand and apply the sorting process.

3.2 Solution: Implementing Bubble Sort

Since the dataset is small, Bubble Sort is a viable choice due to its straightforward implementation. The algorithm repeatedly compares adjacent elements and swaps them if they are in the wrong order. Despite its $O(n^2)$ complexity, it performs adequately for small lists.

3.3 Implementation Steps

1. Retrieve the list of student attendance records.
2. Apply Bubble Sort to arrange the records by roll number.
3. Store the sorted records for generating reports.

3.4 Results and Benefits

- Simple implementation requiring minimal programming knowledge.
- Efficient sorting for small datasets without additional computational overhead.
- Easy to debug and modify if needed.
- Suitable for educational institutions with basic IT infrastructure.

4 Algorithm Process and Example

The Bubble Sort algorithm follows these steps:

1. Start from the first element.
2. Compare the current element with the next element.
3. If the current element is greater than the next element, swap them.
4. Move to the next element and repeat steps 2 and 3 until the last element.
5. Repeat the process for $n - 1$ passes, where n is the length of the list.

4.1 Example

Consider the unsorted list: [5, 3, 8, 4, 2]

4.2 Step-by-Step Execution

We perform multiple passes over the list, where in each pass, the largest unsorted element bubbles up to its correct position.

4.2.1 Pass 1

5	3	8	4	2
---	---	---	---	---

1. Compare 5 and 3, swap: → [3, 5, 8, 4, 2]
2. Compare 5 and 8, no swap: → [3, 5, 8, 4, 2]
3. Compare 8 and 4, swap: → [3, 5, 4, 8, 2]
4. Compare 8 and 2, swap: → [3, 5, 4, 2, 8]

Total comparisons: 4

4.2.2 Pass 2

1. Compare 3 and 5, no swap: → [3, 5, 4, 2, 8]
2. Compare 5 and 4, swap: → [3, 4, 5, 2, 8]
3. Compare 5 and 2, swap: → [3, 4, 2, 5, 8]

Total comparisons: 3

4.2.3 Pass 3

1. Compare 3 and 4, no swap: → [3, 4, 2, 5, 8]
2. Compare 4 and 2, swap: → [3, 2, 4, 5, 8]

Total comparisons: 2

4.2.4 Pass 4

1. Compare 3 and 2, swap: → [2, 3, 4, 5, 8]

Total comparisons: 1

After 4 passes, the list is sorted: [2, 3, 4, 5, 8]

5 Code Implementation and Explanation

Below is a Python implementation of the Bubble Sort algorithm with comments explaining each step:

Listing 2: Bubble Sort Algorithm

```
1 def bubble_sort(arr):
2     """
3         Function to perform Bubble Sort on a given list.
4         :param arr: List of elements to be sorted
5     """
6     n = len(arr)
7
8     # Traverse through all elements in the list
9     for i in range(n - 1):
10         swapped = False # Track if any swaps occur
11
12         # Last i elements are already in place
13         for j in range(n - i - 1):
14             if arr[j] > arr[j + 1]:
15                 arr[j], arr[j + 1] = arr[j + 1], arr[j] # Swap if
16                     # elements are in wrong order
17                 swapped = True
18
19         # If no swaps occurred, the array is already sorted
20         if not swapped:
21             break
```

```
21
22 # Example usage:
23 data = [5, 3, 8, 4, 2]
24 bubble_sort(data)
25 print("Sorted array:", data)
```

Explanation:

- The function takes an array as input and sorts it in ascending order.
- It iterates through the array multiple times.
- In each pass, adjacent elements are compared, and if needed, swapped.
- The process is optimized by introducing a `swapped` flag, which stops the algorithm early if the list becomes sorted before completing all passes.

6 Summary

Sorting algorithms are essential for efficient data retrieval and management in different applications. Binary Search is highly effective for large-scale applications like digital library systems, providing efficient, scalable, and real-time book retrieval to enhance performance and user satisfaction. On the other hand, Bubble Sort remains a practical choice for small-scale applications such as school attendance management, where ease of implementation is prioritized over efficiency. Both algorithms demonstrate the importance of selecting appropriate sorting techniques based on the specific needs of an application.

Algorithm Adventures: Sorting, Searching, and Solving the Teacher's Nightmare!

Minor in AI, IIT Ropar
14th February, 2025

Quick Sort Algorithm: A Step-by-Step Guide for Beginners

Welcome back to the world of sorting algorithms! In this module, we'll explore a particularly clever and powerful sorting algorithm called **Quick Sort**.

We will take things slowly, with plenty of examples, and show you how to implement Quick Sort in Python. Finally, we'll discuss how to measure how efficient an algorithm is using a concept called Time Complexity. Let's begin!

The Teacher's Dilemma - An Intuitive Introduction to Quick Sort

Imagine two school teachers tasked with arranging students in a line according to their heights. The students, naturally, prefer standing with their friends and end up in a random order every day. The teacher's challenge is to quickly sort them into ascending order of height.

Let's say the students are initially standing in this order (representing their heights):

7, 6, 1, 5, 4, 9, 2, 3

Now, these teachers came up with a unique method. One teacher, let's call her Teacher A, stands at the beginning of the line. The other, Teacher B, stands at the end.

Teacher A's job is to send the taller students to the *back* of the line. Teacher B, on the other hand, sends the shorter students to the *front*. The idea is that by constantly shuffling students, we can progressively sort the entire line.

The Problem:

Initially, Teacher A looks at the first student (height 7) and tells him to move to the back, as he's tall. But the student protests, "Tall compared to whom?". Teacher B faces a similar problem with the last student (height 3). They need a common reference point.

The Solution:

The teachers decide to pick a student from the middle as a reference point. In our example, there are 8 students, so the "middle" student is the one at

position 4, with height 5. They call this the **Pivot**.

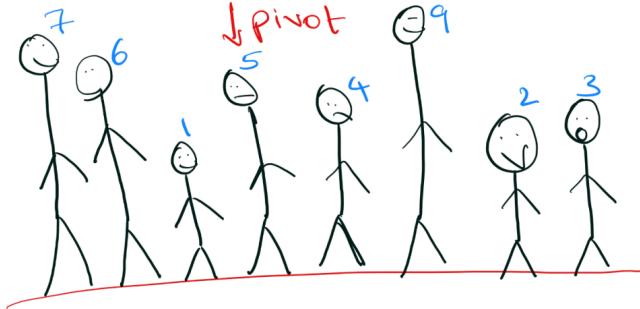


Figure 1: A diagram showing the students lined up, with an arrow pointing to the middle student (height 5) labelled "Pivot"

Now, Teacher A compares everyone to the Pivot (5). If a student is taller than 5, they're sent to the back. Teacher B compares everyone to the Pivot as well. If a student is shorter than 5, they go to the front.

The Shuffling Begins:

1. Teacher A tells the first student (7) to go to the back, and Teacher B tells the last student (3) to go to the front. They swap places, giving us:
3, 6, 1, 5, 4, 9, 2, 7
2. Teacher A moves one position to the right, looking at the student with height 6. Teacher B moves one position to the left, looking at the student with height 2. Both are taller and shorter respectively compared to Pivot (5). They swap places, giving us:
3, 2, 1, 5, 4, 9, 6, 7
3. Teacher A moves to the student with height 1, noting that it is not taller compared to Pivot (5). So, the teacher moves to the next student with height 5. Teacher A tells the student with height 5 to go to the back and Teacher B tells the student with height 4 to go to the front. They swap places, giving us:
3, 2, 1, 4, 5, 9, 6, 7
4. As teachers move positions one after the other, Teacher A looks to the position of Teacher B & Teacher B looks at the position of Teacher A. Since both teachers find out that Teacher A is at the left half where all elements are smaller than pivot and Teacher B is at the right half where all elements are bigger than pivot, they realize that their job is done. Teacher A stops at position 4, the left half and Teacher B stops at position 5, the right half.

The Result:

Notice something interesting! All the students shorter than 5 are now on the left side of the line, and all the students taller than 5 are on the right!

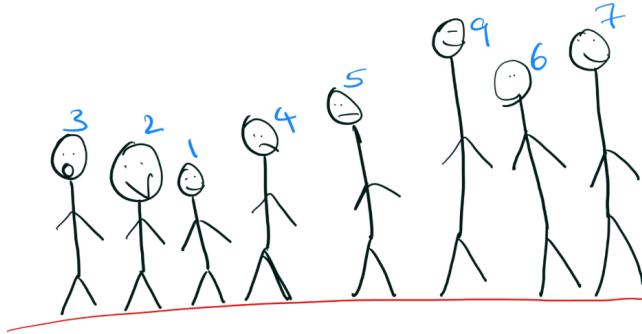


Figure 2: Result of the partition!

This is what Quick Sort does at its core. It picks a pivot, divides the list into two parts (smaller and larger than the pivot), and then repeats the process on each of those parts. By breaking down the problem into smaller and smaller chunks, we can efficiently sort the entire list.

Breaking it Down - The Quick Sort Algorithm

Now that we've got the intuitive understanding down, let's formalize the Quick Sort algorithm.

1. **Choose a Pivot:** Select an element from the list to serve as the pivot. In our teacher example, we picked the middle element. In practice, pivots can be chosen in various ways (first element, random element, etc.).
2. **Partition:** Rearrange the list so that all elements smaller than the pivot are placed before it, and all elements greater than the pivot are placed after it. The pivot ends up in its final sorted position. The same thing the teachers did in previous section. This step is known as **Partitioning**.
3. **Recursion:** Recursively apply steps 1 and 2 to the sub-lists of smaller and larger elements. This means we repeat the whole process on the left side elements of Pivot and right side elements of Pivot.

Python Implementation - Bringing Quick Sort to Life

Let's write some Python code to implement the Quick Sort algorithm. We'll start by defining a `partition` function, mirroring the teacher's actions of dividing the list around the pivot.

Listing 1: Partition function in Python

```
1 def partition(my_list, start, end):
2     t1 = start
3     t2 = end
4     pivot = my_list[(start+end)//2]
5
6     while True:
7         while my_list[t1]<pivot:
8             t1 += 1
9         while my_list[t2]>pivot:
10            t2 -= 1
11        if t1>=t2:
12            return t2
13
14        #swap
15        my_list[t1], my_list[t2] = my_list[t2], my_list[t1]
16        t1 += 1
17        t2 -= 1
```

This `partition` function takes the list (`my_list`), a `start` index, and an `end` index as input. It selects the middle element as the `pivot`. Then, using a `while` loop, it iterates through the list, shifting elements according to whether they are less than or greater than the `pivot`.

Next, let's implement the `quicksort` function, which uses the `partition` function recursively to sort the list.

Listing 2: Quicksort function in Python

```
1 def quicksort(my_list, start, end):
2     if start<end:
3         t2 = partition(my_list, start, end)
4         quicksort(my_list,start,t2)
5         quicksort(my_list,t2+1,end)
```

This function checks if there's more than one element in the current sub-list (`if start < end`). If so, it calls the `partition` function to divide the sub-list around the `pivot`. Then, it recursively calls itself on the two resulting sub-lists.

Let's try it out with our initial student height list:

Listing 3: Example usage of quicksort

```
1 my_list = [7,6,1,5,4,9,2,3]
2 quicksort(my_list,0, len(my_list)-1)
3 print(my_list) # Output: [1, 2, 3, 4, 5, 6, 7, 9]
```

Congratulations! You've successfully implemented Quick Sort in Python!

Measuring Efficiency - Time Complexity

Now we know how Quick Sort works, but how do we compare it to other sorting algorithms? This is where the concept of **Time Complexity** comes in.

What is Time Complexity?

Time complexity is a way of describing how the *number of operations* an algorithm performs grows as the *input size* grows. We are talking about how

many instructions (or operations) the computer has to execute to complete the task. Instead of measuring time in seconds (which can vary based on the computer), we count the number of instructions.

Consider these example instructions:

Listing 4: Time complexity Example 1

```
print("Hello") #1 instruction

print("Hello") #2 instructions
print("Hello")

for i in range(0,10):
    print("Hello") #10 instructions

for i in range(0,10):
    print("Hello") #20 instructions
    print("Hello")
```

So, number of instructions can be measured as above.

The Big O Notation

We use something called **Big O notation** to express time complexity. The "O" stands for "order of," and it gives us a general idea of how the algorithm scales.

For example, **O(n)** (read as "order of n") means the number of operations grows linearly with the input size (n). If you double the input size, you roughly double the number of operations. An example is the linear search algorithm.

Listing 5: Linear search algorithm

```
key = 5
n = len(my_list) #number of instructions is n
for i in range(0,len(my_list)):
    if(my_list[i]==key):
        print("Found")
```

The code has linear time complexity of $O(n)$.

Understanding $O(n^2)$

If the algorithm executes as follows:

Listing 6: $O(n \text{ squared})$

```
for i in range(0,n):
    for j in range(0,n):
        print(my_list[i])

    print(my_list[i])
```

Then time complexity will be $n * n + n = n^2 + n$. However, as n becomes very large, n^2 dwarfs n. For example:

If $n = 10$, the number of instructions will be: $10 * 10 + 10 = 110$ instructions

If $n = 100$, the number of instructions will be: $100 * 100 + 100 = 10100$ instructions

If $n = 1000$, the number of instructions will be: $1000 * 1000 + 1000 = 1001000$ instructions

You can see that as n becomes very large, the $+ n$ part is relatively insignificant compared to n^2 . So, we ignore it and say the time complexity is $O(n^2)$.

Logarithmic Time - $O(\log n)$

Consider the following algorithm: binary search. The code has logarithmic time complexity. It turns out that the time complexity of the binary search algorithm is $O(\log n)$.

The reason binary search achieves $O(\log n)$ complexity lies in its divide-and-conquer approach. With each comparison, the algorithm effectively halves the search space. Let's say you have a list of ' n ' items.

1st comparison: You check the middle element. If it's not the target, you eliminate half of the list. You're left with approximately $n/2$ elements to search.

2nd comparison: You check the middle element of the remaining half. Again, you eliminate half of what's left. You're now looking at roughly $n/4$ elements.

3rd comparison: You're down to about $n/8$ elements.

This halving continues until you either find the target element or the search space is exhausted (meaning the element is not in the list).

So, how many times can you divide ' n ' by 2 until you get down to 1? This is essentially what the logarithm (base 2) is asking. In mathematical terms, we're looking for ' k ' such that:

$$\frac{n}{2^k} = 1$$

This can be rewritten as:

$$n = 2^k$$

Taking the logarithm base 2 of both sides:

$$\log_2(n) = k$$

Therefore, the number of comparisons k is proportional to $\log_2(n)$. Since the base of the logarithm is a constant, we express the time complexity in Big O notation as $O(\log n)$. This means that as the size of the input (n) increases, the number of operations required by binary search grows logarithmically, making it very efficient for searching large sorted lists.

Listing 7: Binary search algorithm

```
#Binary Search - REVISION OF PREVIOUS CLASS
my_list = [3,4,6,7,9,12,16,17]
target = 13
ans = -1
start = 0
end = len(my_list)-1

while(start<=end):
    mid = (start+end)//2
```

```

if(my_list[mid]==target):
    ans = mid
    break
elif(my_list[mid]>target):
    end = mid-1
elif (my_list[mid]<target):
    start = mid+1

print(ans)

```

Time Complexity of Quick Sort

The time complexity of Quick Sort is a bit more nuanced. In the **best-case** and **average-case**, Quick Sort has a time complexity of $O(n \log n)$. However, in the **worst-case** (when the pivot is consistently the smallest or largest element), it degrades to $O(n^2)$.

The **best-case** and **average-case** occur when the partitioning step divides the list into roughly equal halves. Then, since teachers are working together to find the correct position in Quick Sort, it saves time in sorting.

That's why, it is named as `quick sort`.

Despite the possibility of a worst-case scenario, Quick Sort is often favored in practice due to its excellent average-case performance and is often used in implementations.

Congratulations on completing this journey through Quick Sort! We started with an intuitive example of teachers sorting students, translated that intuition into a Python implementation, and finally, learned how to measure the efficiency of algorithms using Time Complexity. With this knowledge, you're well-equipped to tackle more complex sorting problems and appreciate the power of algorithmic thinking!

Vectors and Matrices: A Gentle Introduction with Python

Introduction

Welcome to the world of Vectors and Matrices! This book is designed to gently introduce you to these powerful mathematical concepts and how they're used in Python. You might be surprised to know that you've already encountered them in your daily life.

Location, QR Codes, and Images – Vectors and Matrices in the Real World

Imagine sharing your location with a friend. You likely use Google Maps or a similar app. When you share, you're essentially sending your GPS coordinates. These coordinates look something like this: 13.03° N, 77.56° E.

What is this? It's a set of two numbers that represent your position on the Earth's surface. These two numbers are your latitude and longitude. If we ignore the "N" and "E" and just focus on the numerical values, we have (13.03, 77.56). This is a *point* defined by two values. It can be considered as a location data.

This point describes our location north of the Equator and east of the Prime Meridian. This simple example shows how two numbers can represent a location on a two-dimensional surface.



*GPS Location Marker on a Map

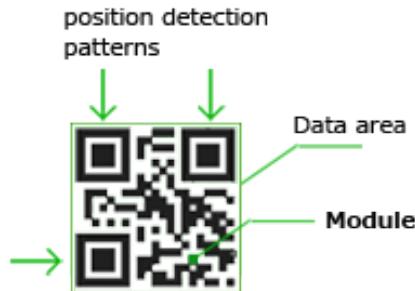
Now think about student data stored in a spreadsheet. Each row might represent a student, and each column a piece of information about them, like their class and percentage. For example:

Student	Class	Percentage
John	10	85
Jane	11	92

Each row of this data effectively represents a student using a series of values. These values define a data of that student.

Consider QR codes, which are everywhere today, linking us to websites or enabling payments. If we zoom in, we see that a QR code is made of small squares, each of which is either black or white.

Let's say we have a small QR code made up of 30 rows and 30 columns. This gives us 900 individual squares. Each square can be either black (represented by 0) or white (represented by 1). This QR code essentially stores information in a grid.



*Close-up of a QR Code

Finally, think about digital images. You've surely seen a lot of them. Images are composed of *pixels*, which are the smallest units of an image. Each pixel has a color value. So, an image can be represented as a grid of pixels, with each pixel having a specific color.

These examples demonstrate that data in the real world can be organized in different ways. Sometimes it is enough to have only one list of values to represent something. Sometimes you need data in the form of grids. These "lists" and "grids" are the basis of *vectors* and *matrices*, which we'll explore further in this book.

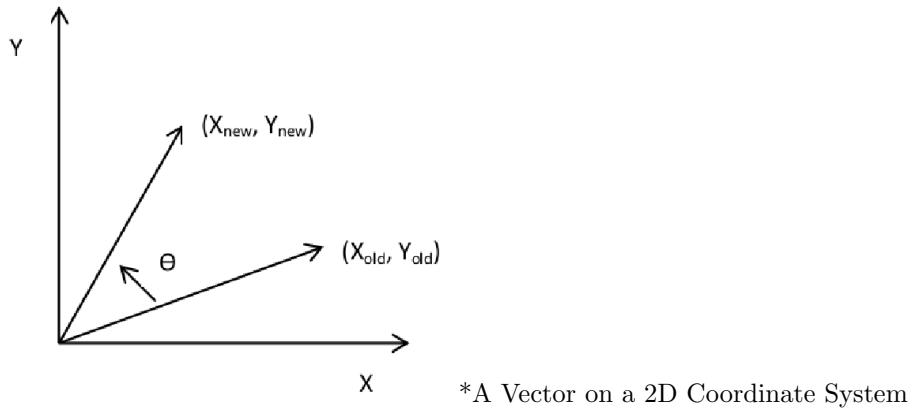
1 Vectors – One-Dimensional Data

A **vector** is essentially a list of numbers. Think of it as a way to represent a point in space. Let's start with a two-dimensional space, like a piece of graph paper.

Imagine a coordinate system with an X-axis and a Y-axis. The point where they meet is called the *origin*, represented as $(0, 0)$. Any point on this paper can be described by its X and Y coordinates.

For example, the point $(3, 4)$ means that you move 3 units along the X-axis and 4 units along the Y-axis from the origin.

Now, connect the origin $(0, 0)$ to the point $(3, 4)$ with a straight line. This line, with a direction from the origin to the point, is a **vector**.



Mathematically, we represent a vector using its coordinates, often written in a column format:

$$\begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

This vector represents a direction and a distance from the origin to the point $(3, 4)$.

The point $(0, 0)$ itself can also be represented as a vector called the **zero vector**. A zero vector has no length and no specific direction.

Vector Operations

So, what can you *do* with vectors? You can perform arithmetic operations on them, similar to how you work with regular numbers.

1.1 Scalar Multiplication

Scalar multiplication means multiplying a vector by a single number (a scalar). This changes the length of the vector, scaling it up or down.

For instance, if we have a vector $\mathbf{v} = (2, 3)$ and multiply it by the scalar 2, we get a new vector:

$$2 \cdot \mathbf{v} = (2 \cdot 2, 2 \cdot 3) = (4, 6)$$

This new vector $(4, 6)$ is twice as long as the original vector $(2, 3)$ and points in the same direction. The point simply went further away from the origin.

Multiplying by a scalar less than 1 makes the vector shorter. Multiplying by a negative scalar reverses the direction of the vector.

1.2 Vector Addition

Vector addition means adding two vectors together. This combines their directions and distances.

For instance, if we have vectors $\mathbf{v} = (2, 3)$ and $\mathbf{w} = (6, 4)$, adding them together is done by adding the corresponding elements:

$$\mathbf{v} + \mathbf{w} = (2 + 6, 3 + 4) = (8, 7)$$

The resulting vector $(8, 7)$ represents a new point that you would reach by following the direction and distance of both original vectors in succession.

Geometrically, you can visualize this by placing the tail of vector \mathbf{w} at the head of vector \mathbf{v} . The resulting vector $\mathbf{v} + \mathbf{w}$ is the vector from the origin to the head of \mathbf{w} .

2 Matrices – Two-Dimensional Data

A **matrix** is a grid of numbers arranged in rows and columns. Think of it as a table or a spreadsheet.

For example, a matrix \mathbf{M} could look like this:

$$\mathbf{M} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

This is a 2×2 matrix because it has 2 rows and 2 columns. The numbers inside the matrix are called its *elements*.

Transforming Vectors with Matrices

One of the most powerful things you can do with matrices is to *transform* vectors. This means applying a matrix to a vector to change its direction or length.

To multiply a matrix by a vector, the number of columns in the matrix must equal the number of rows in the vector.

For example, let's multiply the matrix \mathbf{M} from above by the vector $\mathbf{v} = (2, 3)$:

$$\mathbf{M} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad \mathbf{v} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

The result \mathbf{r} is a new vector and can be found as follows:

$$\mathbf{r} = \mathbf{M} \cdot \mathbf{v} = \begin{bmatrix} 1 \cdot 2 + 2 \cdot 3 \\ 3 \cdot 2 + 4 \cdot 3 \end{bmatrix} = \begin{bmatrix} 8 \\ 18 \end{bmatrix}$$

The resulting vector is $(8, 18)$. The original vector $(2, 3)$ has been transformed into a new vector $(8, 18)$ by applying the matrix \mathbf{M} .

Multiplying with a matrix is also called **linear transformation**. The matrix \mathbf{M} is known as **transformation matrix**.

3 Linear Transformations

Different matrices cause different transformations. Applying the *identity matrix*, for example, will result in the same vector.

Identity matrix:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

In general, in the transformation matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

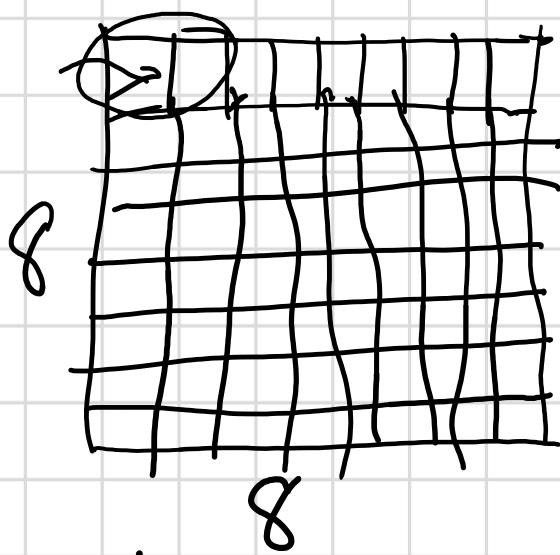
- The value of a controls scaling along the X-axis.
- The value of d controls scaling along the Y-axis.
- The value of b controls rotation with respect to the Y-axis.
- The value of c controls rotation with respect to the X-axis.

Different values for these parameters will result in different transformations of the original vector.

Why This Matters?

Vectors and matrices are fundamental building blocks in many fields, including:

- **Computer graphics:** Representing images, animations, and 3D models.
- **Machine learning:** Storing data, performing calculations, and building models.
- **Physics:** Modeling motion, forces, and other physical phenomena.



1st 2nd 3rd

$$2 \times 2 \times 2 = 8$$

$$1^{\text{st}} \dots 6^{\text{th}} = \underbrace{2 \times 2 \times \dots \times 2}_{6^{\text{th}}} = 2^6 \text{ ways}$$

2^{100} ways,

$$9^{100} = 81,00,000$$

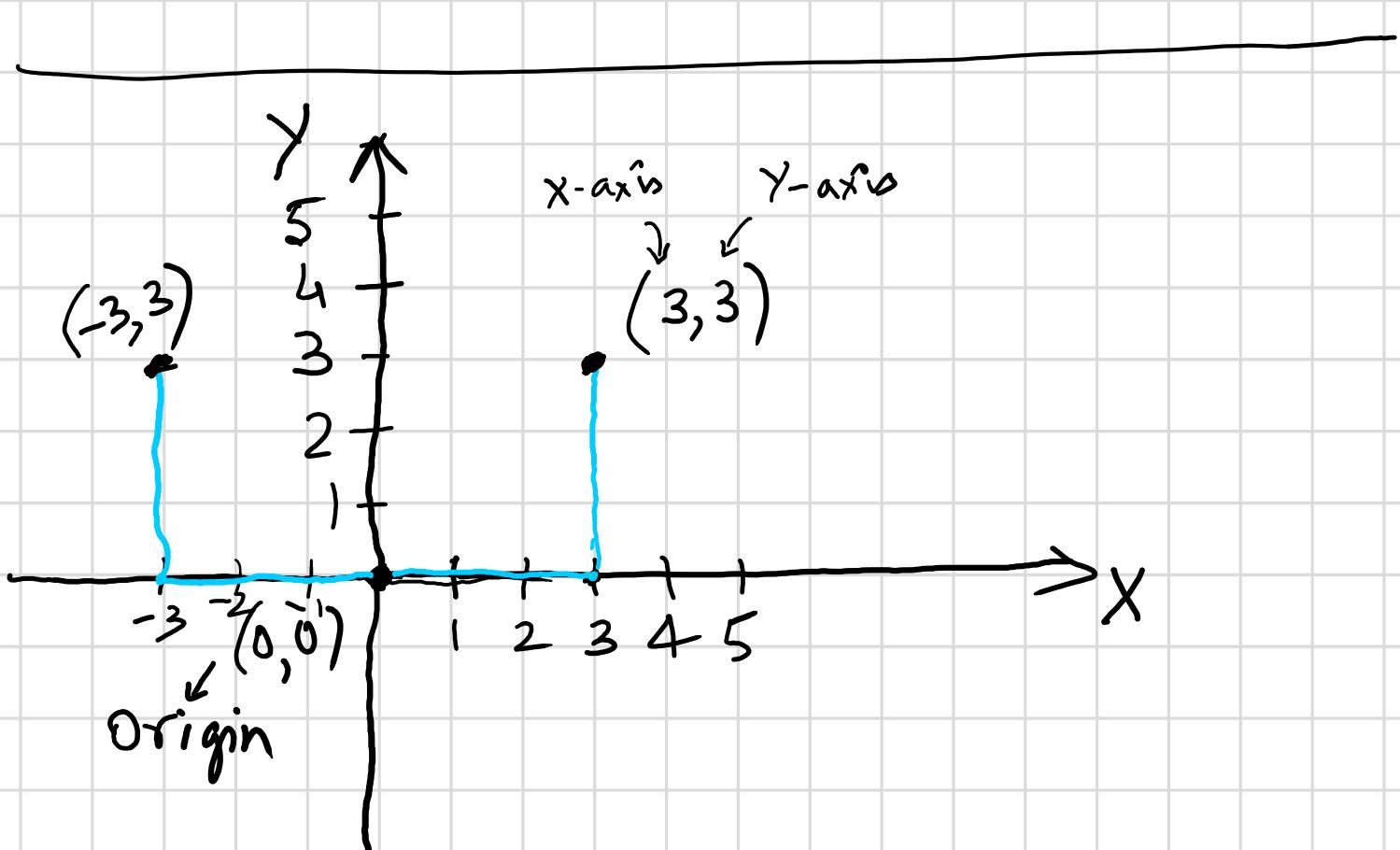
00
01
10
11

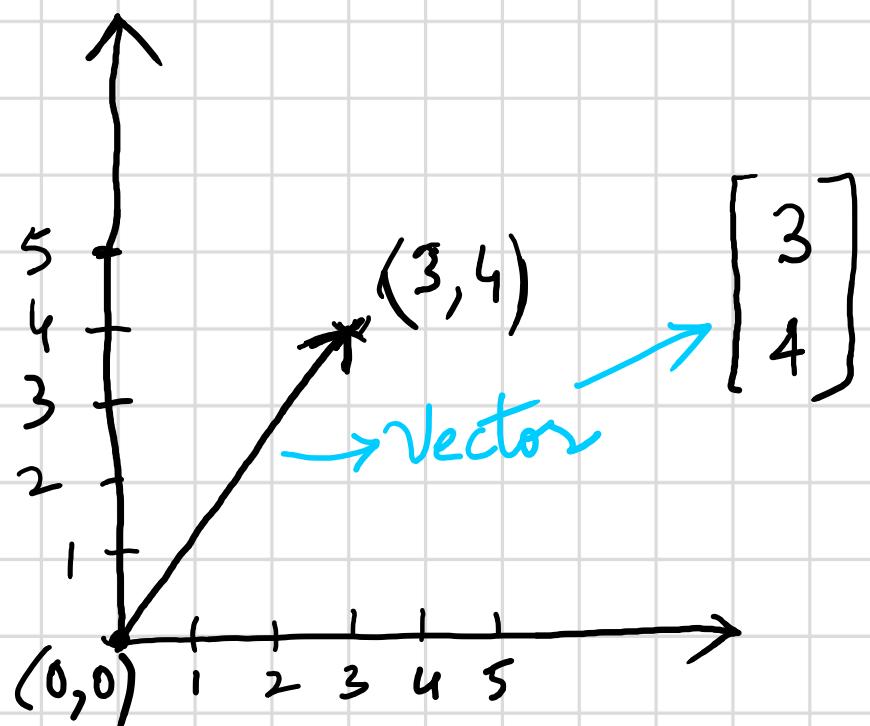
1st cell $\begin{matrix} 0 \\ 1 \end{matrix}$

2nd cell $\begin{matrix} 0 \\ 1 \end{matrix}$

1 st	2 nd	1
-----------------	-----------------	---

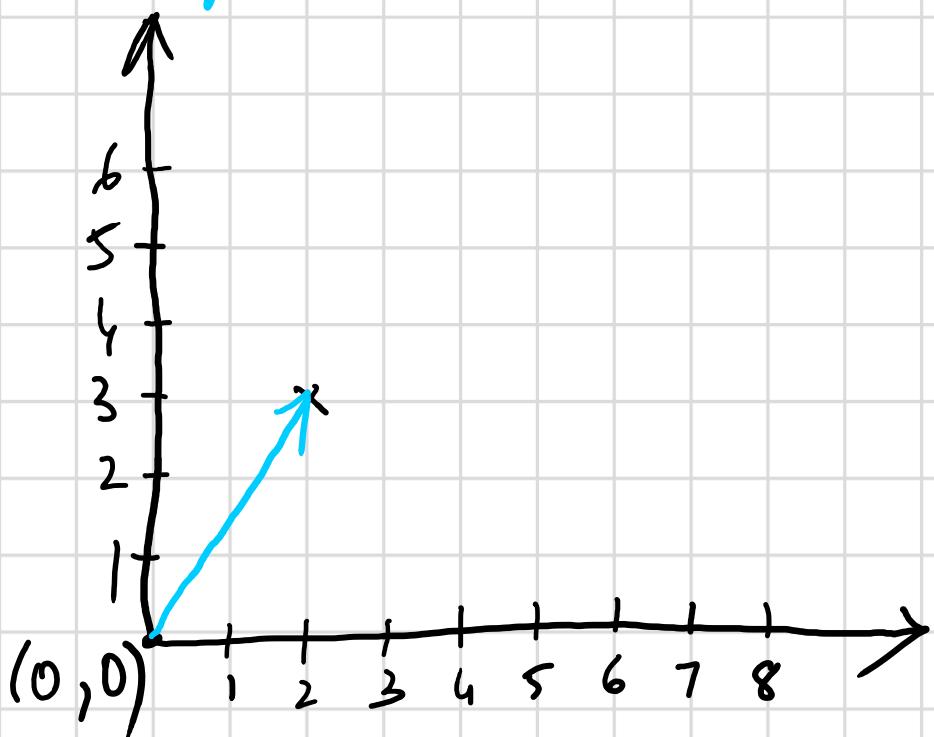
$$2 \times 2 = 4$$





$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow \text{Zero Vector}$$

What can you do with vectors?



$$u = \begin{bmatrix} 2 \\ 3 \end{bmatrix}_{2 \times 1} \Rightarrow 2 \times u =$$

$$u = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

$$v = \begin{pmatrix} 6 \\ 4 \end{pmatrix}$$

$$w = \begin{pmatrix} 8 \\ 7 \end{pmatrix}$$

$$1u + 1v = w$$

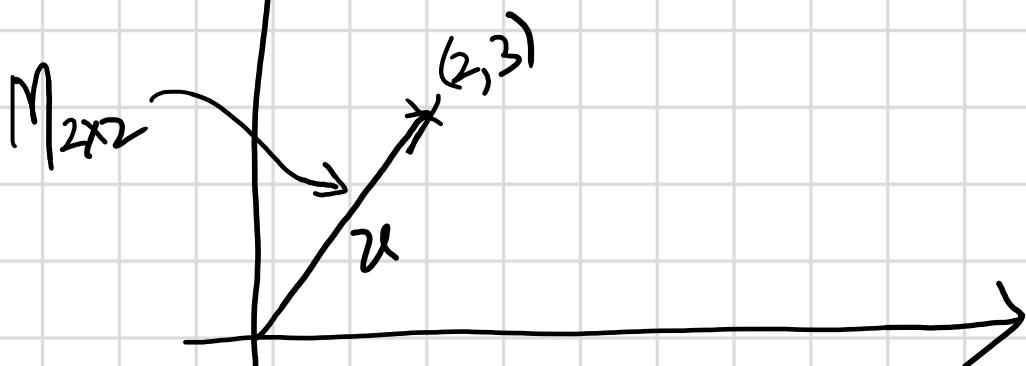
$$u = \begin{pmatrix} 2 \\ 3 \end{pmatrix}_{2 \times 1}$$

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}_{2 \times 2}$$

$$M \times u \Rightarrow \underline{\underline{2 \times 2}} \quad \underline{\underline{2 \times 1}} = \text{result}_{2 \times 1}$$

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}_{2 \times 2} \begin{pmatrix} 2 \\ 3 \end{pmatrix}_{2 \times 1} = \begin{pmatrix} 1 \times 2 + 2 \times 3 \\ 3 \times 2 + 4 \times 3 \end{pmatrix}_{2 \times 1} = \begin{pmatrix} 8 \\ 18 \end{pmatrix}_{2 \times 1}$$

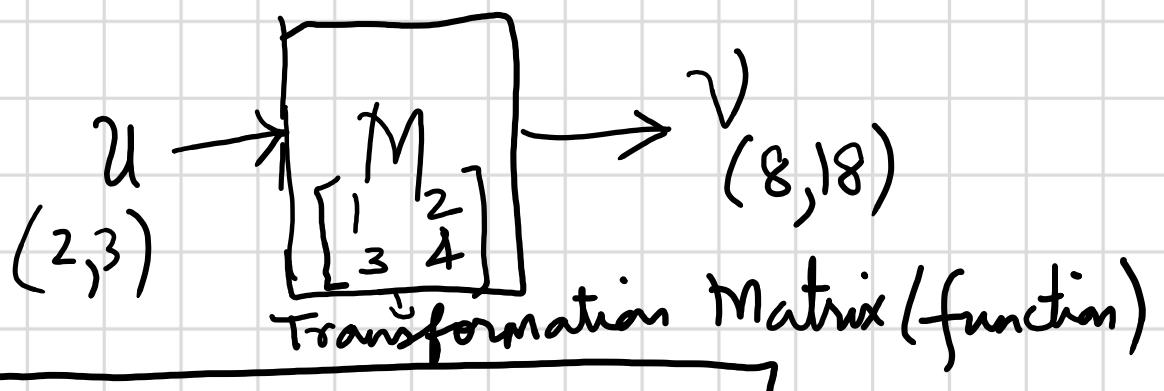
$\checkmark (8, 18)$



$$v = M u$$

Using a Matrix M , we transformed a point u into a point v .





Linear Transformation

$$N = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_{2 \times 2} \quad u = \begin{pmatrix} 2 \\ 3 \end{pmatrix}_{2 \times 1}$$

$N \times u = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 1 \times 2 + 0 \\ 0 + 1 \times 3 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$

Identity Matrix

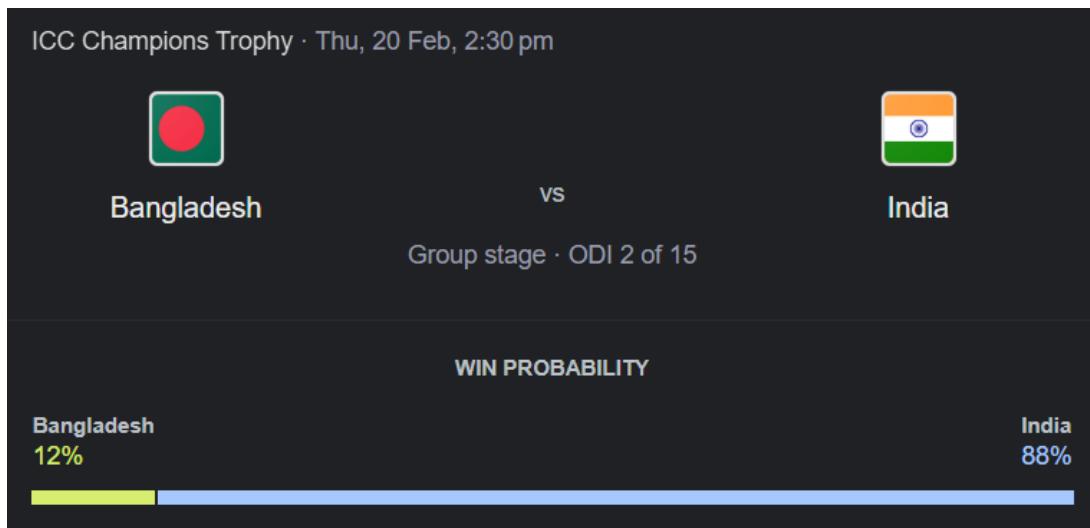
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}$$

$M * u = v$



Probability in AI

Minor in AI



Live Cricket Match Predictions

Imagine watching a cricket match where Google shows live win probability percentages. Why do these never reach 100%? This real-time analysis demonstrates probability theory in action, considering:

- Player performance history
- Weather conditions
- Team dynamics
- Historical match data

This real-time prediction is not mere magic—it is a powerful application of probability theory in action. In **AI**, probability provides the mathematical framework to model uncertainty and make informed decisions in dynamic environments, from sports analytics to self-driving cars.

The central challenge we address is: **How can we model and compute the likelihood of events when multiple, interrelated factors are at play?** This document explores fundamental concepts in probability, essential rules, real-world case studies, and practical Python implementations to bridge theory with real-life applications in AI.

1 Core Concepts in Probability

1.1 Basic Terminology

- **Sample Space:** The set of all possible outcomes in an experiment. For example, in a dice roll, $\{1, 2, 3, 4, 5, 6\}$.
- **Outcome:** A single possible result from the sample space.
- **Event:** A specific subset of the sample space; for instance, rolling an even number.

1.2 Cromwell's Rule

Cromwell's Rule cautions against assigning probabilities of exactly 0 or 1 (except in logically impossible or certain cases). This principle maintains a realistic level of uncertainty:

- **Example:** In live sports predictions, even if one team appears dominant, its winning probability is capped below 100% (e.g., 99.95%).

1.3 Independent vs. Dependent Events

- **Independent Events:** The outcome of one event does not affect another (e.g., successive coin flips).
- **Dependent Events:** The outcome of one event influences the probability of another (e.g., drawing cards from a deck without replacement).

1.4 Probability Rules

Addition Rule: For events that may overlap,

$$P(A \text{ or } B) = P(A) + P(B) - P(A \text{ and } B)$$

calculates the probability of either event A or event B occurring, subtracting $P(A \text{ and } B)$ to avoid double-counting cases where both events happen. Here, $P(A)$ and $P(B)$ are the individual probabilities of events A and B , while $P(A \text{ and } B)$ represents the probability that both occur together.

Multiplication Rule: For two events,

$$P(A \text{ and } B) = P(A) \times P(B) \quad (\text{if independent})$$

This follows from the definition of independence, which states that the occurrence of event A does not influence the probability of event B , meaning $P(B|A) = P(B)$.

For **dependent events**, the rule adjusts to:

$$P(A \text{ and } B) = P(A) \times P(B|A)$$

calculates the probability of both events A and B occurring, where $P(B|A)$ represents the conditional probability of event B occurring given that event A has already occurred.

2 Applications

2.1 Live Cricket Match Analysis

- **Scenario:** Using live data (score, player performance, weather) to update win probabilities.
- **Insight:** Cromwell's Rule ensures probabilities never reach absolute 0 or 1, acknowledging inherent uncertainty.

2.2 Handy Games

- **Independent Example:** Flipping a coin twice.
- **Dependent Example:** Drawing two cards sequentially from a deck without replacement.

2.3 Self-Driving Car Scenario

- **Scenario:** Evaluating a self-driving car's environment where the sample space consists of all possible road conditions and obstacles.
- **Application:** The car's AI must process these probabilities in real time to execute safe maneuvers.

2.4 Text File Character Frequency Analysis

- **Scenario:** Analyzing character frequencies in a text file.
- **Application:** By analyzing a text file to compute the frequency of characters, one can assign shorter codes to more frequent characters (Huffman coding), thereby reducing the overall data size.

3 Python Implementation Example

The following Python snippet demonstrates how to simulate probability.

Card Draw Probability Simulator

```
1 import random
2
3 deck = ['red']*26 + ['black']*26
4 random.shuffle(deck)
5
6 def draw_card(n=1):
7     return [deck.pop() for _ in range(n)]
8
9 # Probability of first two being red
10 first_two_red = len([1 for _ in 1000 if draw_card(2) == ['red', 'red']])
11                           ])/1000
```

4 Connection between AI and Probability

Probability theory is the backbone of several AI and machine learning methodologies:

- **Bayesian Networks:** For reasoning under uncertainty.
- **Markov Chains:** Model where the next state depends on the current state.
- **Monte Carlo:** To simulate complex systems and approximate probabilities.
- **Natural Language Processing:** For predicting word sequences and translations.

-
- **Data Compression:** Using character frequency analysis to design efficient encoding schemes.

5 Conclusion

5.1 Key Takeaways

- **Modeling Uncertainty:** Probability theory provides essential tools to quantify and manage uncertainty in real-world applications.
- **Fundamental Rules:** Understanding sample space, events, and the addition/multiplication rules is vital for effective probability calculations.
- **Real-World Relevance:** From live sports predictions to self-driving cars and data compression, probability underpins many advanced AI systems.
- **Hands-On Learning:** Implementing these concepts in Python reinforces theoretical understanding through practical simulation.

The AI Engineer's Toolkit: Data Distributions, Simulation Tools, and Hypothesis Testing for Effective Model Development

Minor in AI, IIT Ropar
18th Feb, 2025

A Tale of Turtles and Traffic

Imagine you're observing a group of turtles randomly wandering around in a circular area. Each turtle starts from the center and takes random steps in any direction. Some might venture far, others might stay close to the origin. Some might seem to move in straight lines. You would be left wondering:

- Will they all eventually return to where they started?
- What kind of patterns do their movements create?
- Can we use this to understand other things around us?

These turtles, though simple, represent complex real-world scenarios. They could be:

- **Products moving in the market:** Understanding how your product spreads and reaches different customers.
- **Weather prediction:** Modeling the random movement of air masses.
- **Traffic movement:** Analyzing how cars distribute themselves on the roads.
- **Crowd behavior:** Simulating how people move in a large gathering.

These are chaotic scenarios, but we can still gain insights by understanding **data distributions**. This chapter will give you the tools to explore and analyze the data patterns in such scenarios.

Simulation Tools - NetLogo

To truly grasp data and its patterns, simulation can be a powerful tool. **NetLogo**, created by a popular university, is one such tool. Let's see how it can help us.

The Birthday Paradox

Remember the classic "Birthday Paradox"? It asks: how many people do you need in a room for there to be a 50% chance that two of them share a birthday? The answer, surprisingly, is only 23.

NetLogo allows you to simulate this. It picks 23 random birthdays and highlights any duplicates. By running the simulation multiple times, you can observe the frequency of shared birthdays, visually confirming the paradox.

Random Walk Simulation

Going back to our turtle friends, NetLogo offers a "Random Walk 360" simulation. Here, hundreds of "turtles" start at the origin and take random steps. Key parameters include:

- **Number of Turtles:** How many turtles are moving around?
- **Ring Radius:** The boundary within which the turtles move.
- **Random Step:** The size of each random step the turtle takes.

As the turtles move, NetLogo tracks:

- **Average Distance:** The average distance of all turtles from the origin.
- **Standard Deviation:** The spread of turtles around the average distance.

Graphs show how these metrics change over time. This allows you to visualize the overall trend of the turtle movements and draw conclusions about the scenarios it represents.

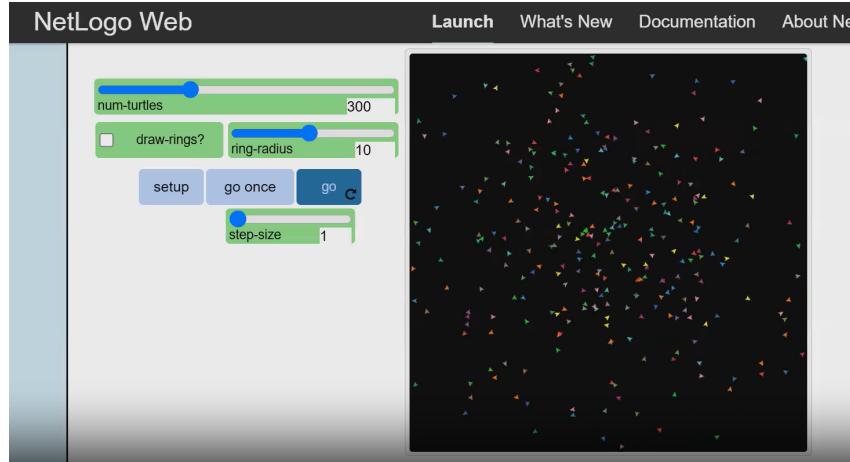


Figure 1: NetLogo Turtle Walk Simulation

Understanding Data Distributions

Data distributions are the backbone of data analysis. They describe how data points are spread across a range of values. Recognizing these distributions helps us understand the underlying processes generating the data and choose the right analysis techniques.

Uniform Distribution

Imagine rolling a fair six-sided die. Each number (1 to 6) has an equal chance of appearing ($1/6$ probability). This is a **uniform distribution**.

In a uniform distribution, all outcomes in the sample space have equal probability. Other examples include:

- Tossing a fair coin (Heads or Tails).
- Drawing a card from a shuffled deck.
- Rock-paper-scissors.

In code, we can simulate rolling a die multiple times and visualize the results using a **histogram**:

```
1 import matplotlib.pyplot as plt
2 import random
3
4 def roll_dice(num_rolls):
5     results = [random.randint(1, 6) for _ in range(num_rolls)]
6     return results
7
8 # Simulate 100 dice rolls
9 rolls = roll_dice(100)
10
11 # Create a histogram
12 plt.hist(rolls, bins=range(1, 8), align='left', rwidth=0.8)
13 plt.title('Uniform Distribution: Dice Rolls')
14 plt.xlabel('Dice Value')
15 plt.ylabel('Frequency')
16 plt.show()
```

Listing 1: Uniform Distribution Simulation

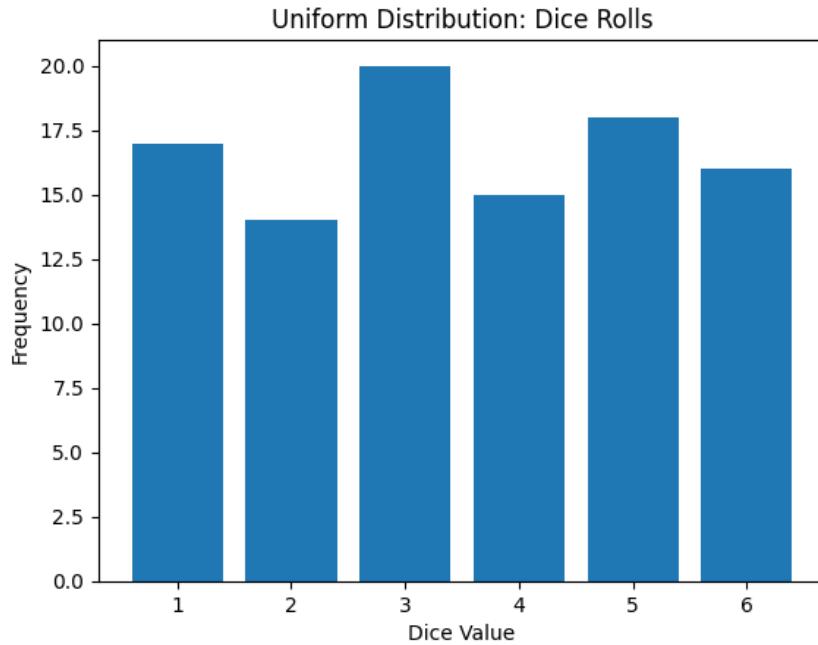


Figure 2: Uniform Distribution Histogram

Normal Distribution

Now, consider the heights of students in a class. Most students will have heights close to the average. Fewer students will be exceptionally tall or short. This is a **normal distribution**, also known as a Gaussian distribution.

The normal distribution is bell-shaped, with the mean (average) at the center. The data is symmetrically distributed around the mean. Many real-world phenomena follow a normal distribution, such as:

- Weights of newborns.
- Blood pressure levels.
- IQ scores.

Simulating a normal distribution:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Generate 1000 random numbers from a normal distribution
5 # with mean 50 and standard deviation 10
6 data = np.random.normal(loc=50, scale=10, size=1000)
7
8 # Create a histogram
9 plt.hist(data, bins=30)
10 plt.title('Normal Distribution')
11 plt.xlabel('Value')
12 plt.ylabel('Frequency')
13 plt.show()

```

Listing 2: Normal Distribution Simulation

Poisson Distribution

Think about the number of customers arriving at a store per hour. The number might fluctuate depending on the time of day or day of the week. The traffic will be very high at the peak hours and very low at the off-peak hours. This is a **Poisson distribution**.

The Poisson distribution models the number of events occurring in a fixed interval of time or space. It's useful for:

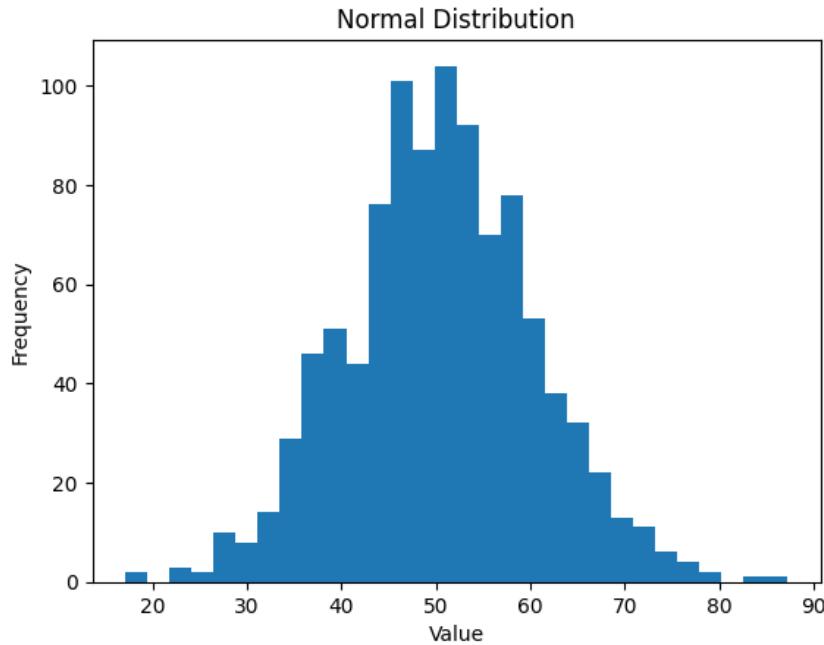


Figure 3: Normal Distribution

- Website clicks per minute.
- Number of emails received per day.
- Number of accidents at an intersection per day.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Generate 1000 random numbers from a Poisson distribution
5 # with lambda = 5 (average number of events)
6 data = np.random.poisson(lam=5, size=1000)
7
8 # Create a histogram
9 plt.hist(data, bins=15)
10 plt.title('Poisson Distribution')
11 plt.xlabel('Number of Events')
12 plt.ylabel('Frequency')
13 plt.show()

```

Listing 3: Poisson Distribution Simulation

Binomial Distribution

Imagine flipping a coin 10 times. Each flip is a **Bernoulli trial**: it has two possible outcomes (Heads or Tails) with a certain probability. The number of heads you get in 10 flips follows a **binomial distribution**.

The binomial distribution models the number of successes in a fixed number of independent Bernoulli trials. It's useful for:

- Defective items in a batch of products.
- Number of users who click on an ad.
- Number of patients who recover from a disease.

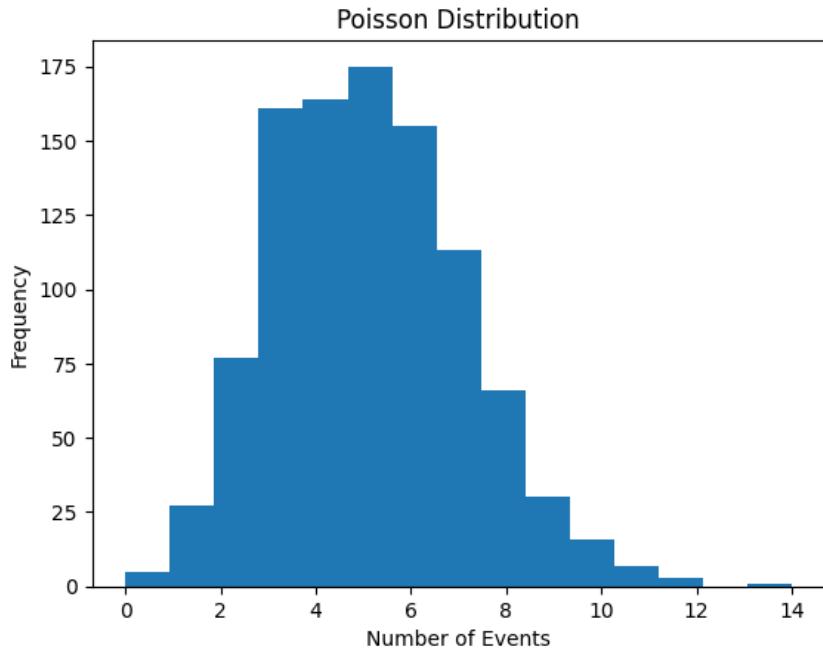


Figure 4: Poisson Distribution Graph

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Generate 1000 random numbers from a Binomial distribution
5 # with n = 10 (number of trials) and p = 0.5 (probability of success)
6 data = np.random.binomial(n=10, p=0.5, size=1000)
7
8 # Create a histogram
9 plt.hist(data, bins=11)
10 plt.title('Binomial Distribution')
11 plt.xlabel('Number of Successes')
12 plt.ylabel('Frequency')
13 plt.show()

```

Listing 4: Binomial Distribution Simulation

Hypothesis Testing

Let's say you've developed a new AI model, and you want to compare its performance to an existing model. How do you determine if the new model is truly better? This is where **hypothesis testing** comes in.

The Core Concepts

- **Null Hypothesis (H₀):** A statement you're trying to disprove. It usually states that there is no effect or no difference.
- **Alternative Hypothesis (H₁):** A statement that contradicts the null hypothesis. It suggests there *is* an effect or a difference.
- **P-value:** The probability of observing the data (or more extreme data) if the null hypothesis is true.
- **Significance Level (alpha):** A threshold for rejecting the null hypothesis. Commonly set at 0.05.
- **Confidence Interval:** Gives a range of values that the mean of the sample could take within a certain confidence.

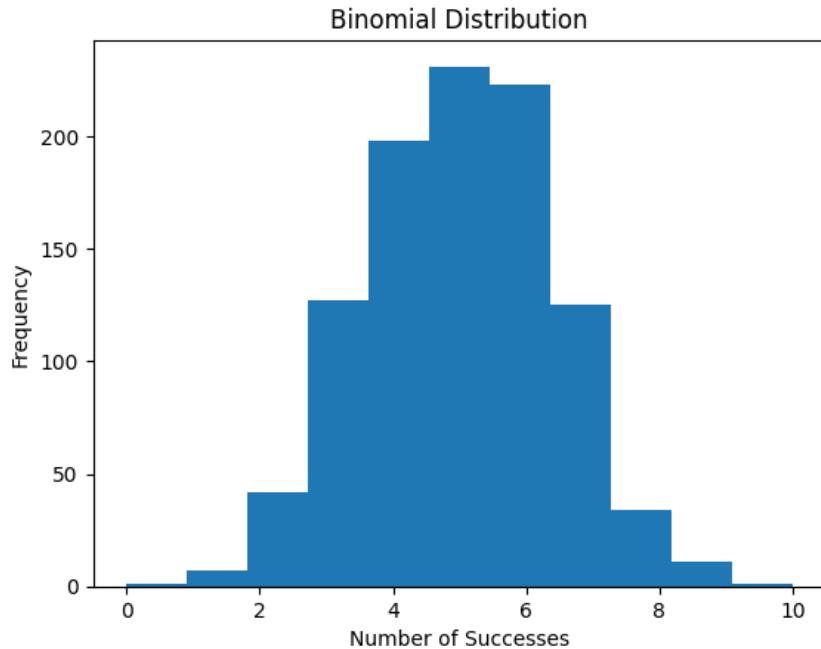


Figure 5: Binomial Distribution Graph

- **Type 1 Error (False Positive):** Rejecting the null hypothesis when it is actually true.
- **Type 2 Error (False Negative):** Failing to reject the null hypothesis when it is actually false.

One-Tailed vs. Two-Tailed Tests

- **One-Tailed Test:** Used when you have a specific direction in mind (e.g., "The new model is *better* than the old one"). It can test if a parameter is only greater than or less than a certain value.
- **Two-Tailed Test:** Used when you're simply interested in whether there's a difference, without specifying a direction (e.g., "The new model is *different* from the old one").

The Decision Rule

- If the p-value is *less* than the significance level (alpha), reject the null hypothesis. This suggests there's strong evidence against the null hypothesis.
- If the p-value is *greater* than the significance level (alpha), fail to reject the null hypothesis. This means there's not enough evidence to reject the null hypothesis.

Example: Comparing Two AI Models

Let's say you have two AI models (A and B) for image recognition. You test them on 50 different datasets and record the accuracy of each model on each dataset.

Your **Null Hypothesis (H₀)** is: "There is no difference in the accuracy of Model A and Model B."

Your **Alternative Hypothesis (H₁)** is: "There is a difference in the accuracy of Model A and Model B." (This is a two-tailed test because you're not assuming one model is better than the other).

Here's how you can perform a t-test in Python:

```

1 import numpy as np
2 from scipy import stats
3
4 # Accuracy data for Model A and Model B (replace with your actual data)
5 model_a_accuracy = np.array([0.86, 0.88, 0.90, 0.85, 0.92, ..., 0.89]) # 50 values
6 model_b_accuracy = np.array([0.84, 0.87, 0.89, 0.83, 0.91, ..., 0.88]) # 50 values
7
8 # Perform a paired t-test

```

```

9 t_statistic, p_value = stats.ttest_rel(model_a_accuracy, model_b_accuracy)
10
11 # Set the significance level
12 alpha = 0.05
13
14 # Make the decision
15 if p_value < alpha:
16     print("Reject the null hypothesis: There is a significant difference between the
17     models.")
18 else:
19     print("Fail to reject the null hypothesis: There is no significant difference
20     between the models.")
21
22 print("P-value:", p_value)

```

Listing 5: T-test for Comparing AI Models

This code calculates the t-statistic and p-value using a paired t-test (because you're comparing the same datasets with two different models). If the p-value is less than 0.05, you reject the null hypothesis and conclude that there's a statistically significant difference between the two models.

The Bigger Picture

Understanding data distributions, leveraging simulation tools, and conducting hypothesis tests are essential skills for any AI Engineer. These techniques allow you to:

- Gain insights from complex data.
- Validate your AI models.
- Make informed decisions.
- Build robust and reliable AI systems.

This module has provided a foundational understanding of these concepts. As you progress in your AI journey, remember to keep exploring, experimenting, and questioning. The world of AI is constantly evolving, and a curious and analytical mind is your greatest asset. Good luck!

From Chances to Choices: Mastering Probability Distributions & Hypothesis Testing

Syllabus:

- Understand Probability Distributions: Explain different types of probability distributions (e.g., uniform, normal, binomial, Poisson) and their real-world applications.
- Identify Key Properties: Describe the characteristics (mean, variance, skewness) of common probability distributions.
- Formulate Hypotheses: Differentiate between null and alternative hypotheses in statistical testing.
- Perform Hypothesis Testing: Apply basic hypothesis testing techniques (e.g., t-test, chi-square test) to make data-driven decisions.
- Interpret Results: Analyze p-values and confidence intervals to draw meaningful conclusions from statistical tests.
- Connect Concepts to AI & ML: Recognize the role of probability distributions and hypothesis testing in machine learning and data science.

Distributions:

1. Uniform Distribution

It is a distribution where all outcomes have equal probability. Every value in a given range is equally likely to occur. It is flat, constant probability across the range. Uniform sampling ensures each data point has an equal chance of being included in your sample. This distribution does not have peaks or clustering of values.

How to identify?

- Check if the values are evenly spread across the range.

Examples:

- Rolling a fair die: Each face of a fair die has an equal chance of landing face up (1/6 probability).
- Random number generators: These algorithms are designed to produce sequences of numbers that are uniformly distributed within a given range.
- Drawing a card from a well-shuffled deck: Each card in the deck has an equal chance of being drawn.
- Lotteries: In a fair lottery, each ticket has an equal chance of being the winning ticket.

Why do we need random numbers?

- It is Foundation of many algorithms. Many ML algorithms rely on randomness, whether for initializing weights in neural networks, shuffling data, or sampling. Uniform distributions are the go-to for generating those random numbers. It provides a fair starting point. You're not biasing your algorithm towards certain values from the outset.
- Weights in neural networks are often initialized with random numbers from a uniform distribution within a certain range. This helps the model learn effectively.
- When building AI models for things like traffic flow or customer behaviour, uniform distributions can help model events that have an equal chance of occurring.

2. Normal Distribution (Gaussian)

It is a bell-shaped distribution where most values cluster around a central mean, and fewer values appear as you move away from the center. It is symmetric around the mean, peaks at the mean, tails off on both sides. Most values are near the mean, extreme values are rare. The normal distribution, often called the "bell curve" or Gaussian distribution, is one of the most common probability distributions in statistics. It describes many natural phenomena where data tends to cluster around a central mean, with fewer and fewer data points occurring further away from that mean.

How to Identify:

Data follows a symmetric bell curve.

Examples:

- Test scores (most students score near the average, few score very high/low).
- Check out student height and weight data set analysis here: <https://www.kaggle.com/datasets/burnoutminer/heights-and-weights-dataset>
- Similar to blood pressure, heart rates in a population often show a normal distribution.
- Size of organisms: In many species, size follows a normal distribution.
- Many machine learning algorithms assume that the data follows a normal distribution, or that the errors in the model are normally distributed. This assumption simplifies the mathematics and makes it easier to analyze the model. For example, linear regression models often assume that the errors are normally distributed.
- Understanding normal distributions helps in data analysis, feature engineering, model building, and interpreting results.
- To simulate in spreadsheet: =NORM.INV(RAND(), mean, standard_dev)

3. Binomial Distribution

Number of successes in n trials of a yes/no experiment. **Example:** Flipping a coin 10 times and counting heads.

When is it used?

The binomial distribution is used in many situations where you have repeated independent trials with two possible outcomes:

Key Differences from Normal Distribution:

- **Discrete:** The binomial distribution is discrete, meaning the number of successes can only be whole numbers (0, 1, 2, ...). The normal distribution is continuous.
- **Shape:** The binomial distribution can be skewed, especially if p is close to 0 or 1. As n gets larger and p is closer to 0.5, it starts to look more like a normal distribution.

A factory produces light bulbs. The probability of a bulb being defective is 0.02. You take a sample of 50 bulbs. What's the probability that exactly 2 are defective?

In a spread sheet cell, type =BINOM.DIST(2, 50, 0.02, FALSE) – refer video for more reference

4. Poisson distribution

The Poisson distribution is a discrete probability distribution that describes the probability of a given number of events occurring in a fixed interval of time or space if these events occur with a known average rate and independently of the time since the last event. It's useful for modelling rare events. Here are some examples:

Examples:

- Number of phone calls received by a call center per hour: If a call center receives an average of 10 calls per hour, the Poisson distribution can be used to calculate the probability of receiving, say, 15 calls in an hour, or 5 calls in an hour.
- Number of cars passing a specific point on a highway per minute: Traffic flow can often be modeled using the Poisson distribution. If the average is 20 cars per minute, you can calculate the probability of 25 cars passing in a minute, or only 10.
- Number of customers entering a store per hour: Similar to call centers, customer arrivals at a store can often be modeled using the Poisson distribution.
- Number of typos on a page: If a typist makes an average of 2 typos per page, the Poisson distribution can be used to calculate the probability of a page having 0 typos, 1 typo, 3 typos, etc.
- Word Count Distributions: The distribution of word counts in a document or corpus can sometimes be approximated by a Poisson distribution, especially for less frequent words. This can be useful in tasks like text classification or topic modeling.
- Spam Detection: The number of certain keywords or phrases in an email (which might be relatively rare) could be modeled using a Poisson distribution. A high count of such keywords might suggest spam.

Hypothesis and Tests:

Refer to the video on how we built the premise for it.

Test Type	Null Hypothesis (H_0)	Alternative Hypothesis (H_1)	When to Reject H_0	When to Accept (Fail to Reject) H_0
One-Tailed (Right-Tailed)	The new method is not better than the old one.	The new method is better than the old one.	If $p < 0.05$, reject H_0 → Significant improvement.	If $p \geq 0.05$, fail to reject H_0 → No significant improvement.
One-Tailed (Left-Tailed)	The new method is not worse than the old one.	The new method is worse than the old one.	If $p < 0.05$, reject H_0 → Significant decline.	If $p \geq 0.05$, fail to reject H_0 → No significant decline.
Two-Tailed	No difference between methods.	The new method is different (could be better or worse).	If $p < 0.05$, reject H_0 → Significant difference.	If $p \geq 0.05$, fail to reject H_0 → No significant difference.

What Does the 5% Significance Level ($\alpha = 0.05$) Mean?

The significance level (α) is the threshold we set to decide when to reject H_0 .

Why 5%?

- $\alpha = 0.05$ means we allow a 5% chance of mistakenly rejecting H_0 (Type I Error).
- In other words, we accept a 5% risk of saying "something is happening" when actually it's just random noise.

If $p < 0.05$ (below the threshold):

- The observed result is so rare under H_0 that we reject H_0 and say there's likely a real effect.

If $p \geq 0.05$ (above the threshold):

- The observed result is not rare enough → Fail to reject H_0 (not enough evidence to prove an effect).

Definitions:

Hypothesis Testing

Hypothesis testing is a way to check if a claim about data is true. It compares observed results with what we expect. If the difference is significant, we reject the initial assumption. It helps in decision-making by analyzing whether an effect is real or just due to chance.

Confidence Interval

A confidence interval gives a range of values where we expect the true value to be. For example, a 95% confidence interval means we are 95% sure the actual value falls within that range. It helps estimate uncertainty in data and is commonly used in statistics.

P-Value

The p-value tells us how likely it is to see our results if the initial assumption (null hypothesis) is true. A small p-value (like below 0.05) suggests strong evidence against the null hypothesis, meaning the observed effect is likely real and not just due to random chance.

T-Test

A t-test is used to compare the means of two groups to check if they are significantly different. It helps in determining if a change or difference between groups happened due to chance or an actual effect. It's commonly used in experiments and research studies.

Null Hypothesis

The null hypothesis assumes no real effect or difference exists. It is the starting point of hypothesis testing, stating that any observed changes are due to chance. If data strongly contradicts it, we reject the null hypothesis and accept an alternative explanation.

Alternate Hypothesis

The alternate hypothesis is the opposite of the null hypothesis. It suggests there is a real effect, difference, or relationship between variables. If enough evidence is found against the null hypothesis, we accept the alternate hypothesis, indicating the observed change is meaningful.

Note:

Refer to video for more explanation.

The class also referred to

- NetLogo simulation tool
- Jamovi for statistical analysis

NumPy for Machine Learning Beginners: A Hands-On Guide Using Google Colab

February 21, 2025

1 Introduction: Why NumPy?

Imagine you're trying to teach a computer to recognize cats in pictures. You need to feed it tons of images, and each image is broken down into millions of numbers representing the color of each pixel. Now, imagine trying to store and manipulate all those numbers using regular Python lists. It would be incredibly slow and inefficient!

That's where NumPy comes in. NumPy, short for "Numerical Python," is a powerful tool that makes working with large amounts of numerical data in Python much faster and easier. It's the foundation upon which many machine learning libraries are built. In this book, we'll start from scratch and learn how to use NumPy in Google Colab to prepare our data for machine learning models.

2 Lists in Python - A Quick Review

Before diving into NumPy, let's quickly revisit lists in Python. You've probably used them before, but let's make sure we're all on the same page.

A list is simply a collection of items. You create a list using square brackets [] and separate the items with commas:

```
my_list = [1, 2, 3, 4, 5]
print(my_list) # Output: [1, 2, 3, 4, 5]
```

You can access individual items in a list using their index (position), starting from 0:

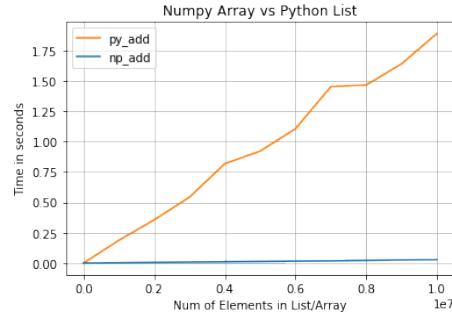
```
print(my_list[0]) # Output: 1
print(my_list[2]) # Output: 3
```

One interesting feature of lists is that they can hold different data types:

```
mixed_list = [1, "hello", 3.14, True]
print(mixed_list) # Output: [1, 'hello', 3.14, True]
```

This flexibility is useful in some cases, but it comes with a performance cost when dealing with large numerical datasets.

3 The Need for Speed: Why NumPy Arrays?



In machine learning, we often work with huge datasets containing mostly numbers. Lists can store these numbers, but they are generally slow and computationally expensive for large datasets. Imagine trying to perform a simple arithmetic operation on millions of numbers in a list!

This is where NumPy arrays shine. NumPy arrays are like lists, but they are designed specifically for efficient numerical operations. They are much faster and more powerful than lists when it comes to handling numerical data.

4 Introducing NumPy Arrays

Think of NumPy arrays as super-charged lists specifically designed for numbers. They are the core data structure for numerical computations in Python.

4.1 Importing NumPy

To use NumPy, you first need to import it into your Python environment. A standard practice is to import NumPy with the alias `np`:

```
import numpy as np
```

This allows you to refer to NumPy functions and objects using the shorter `np` prefix.

4.2 Creating NumPy Arrays

You can create a NumPy array from a Python list using the `np.array()` function:

```
my_list = [1, 2, 3, 4, 5]
my_array = np.array(my_list)
print(my_array) # Output: [1 2 3 4 5]
```

NumPy arrays can only store data of the *same* type. This might seem like a disadvantage compared to lists, but it's a key reason why they are so efficient for numerical computations.

4.3 Accessing Elements and Shape

Accessing elements in a NumPy array is similar to lists:

```
print(my_array[0]) # Output: 1  
print(my_array[3]) # Output: 4
```

To determine the “size” of a NumPy array, you use the `.shape` attribute:

```
print(my_array.shape) # Output: (5,)
```

This tells you the array has one dimension and 5 elements.

5 NumPy’s Power: Creating Arrays with Ease

One of NumPy’s strengths is its ability to create arrays quickly and easily.

5.1 np.arange(): Generating Number Sequences

The `np.arange()` function is similar to Python’s `range()` function, but it creates a NumPy array instead of a list.

```
numbers = np.arange(0, 10) # Numbers from 0 to 9  
print(numbers) # Output: [0 1 2 3 4 5 6 7 8 9]  
  
even_numbers = np.arange(2, 21, 2) # Even numbers from 2 to 20  
print(even_numbers) # Output: [ 2 4 6 8 10 12 14 16 18 20]  
  
reverse_numbers = np.arange(20, 9, -3) # Reverse array with step  
size of -3  
print(reverse_numbers) # Output: [20 17 14 11]
```

5.2 np.zeros() and np.ones(): Arrays Filled with Zeros or Ones

Often, you need to create arrays filled with a specific value, like zeros or ones. NumPy provides `np.zeros()` and `np.ones()` for this:

```
zeros_array = np.zeros(10) # Array of 10 zeros  
print(zeros_array) # Output: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]  
  
ones_array = np.ones(5) # Array of 5 ones  
print(ones_array) # Output: [1. 1. 1. 1. 1.]
```

By default, these functions create arrays of floating-point numbers. You can specify the data type using the `dtype` argument:

```
integer_zeros = np.zeros(5, dtype=int)
print(integer_zeros) # Output: [0 0 0 0 0]
```

5.3 Multidimensional Arrays

You can create two-dimensional (or higher) arrays using `np.zeros()` and `np.ones()` by passing a tuple as the shape:

```
matrix_of_zeros = np.zeros((3, 5)) # 3 rows, 5 columns
print(matrix_of_zeros)
```

Output:

```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

This creates a 3×5 matrix (3 rows and 5 columns) filled with zeros.

Another way to interpret this is as a “stack” of 3 arrays, each containing 5 elements.

Similarly:

```
three_d_array = np.zeros((2,3,5)) # 2 "Layers", 3 rows, 5 columns
print(three_d_array)
```

This creates a 3D array of shape $(2, 3, 5)$. You can think of it as two layers where each layer is the matrix created earlier.

The `.shape` attribute gives you the dimensions of the array:

```
print(matrix_of_zeros.shape) # Output: (3, 5)
print(three_d_array.shape) # Output: (2,3,5)
```

6 NumPy’s Powerful Functions

NumPy provides a wide range of functions for performing mathematical operations on arrays. These functions are highly optimized for performance.

6.1 Basic Mathematical Operations

```
my_array = np.array([1, 16, 23, 45, 67, 21, 5])

print(np.max(my_array)) # Output: 67 (Maximum element)
print(np.min(my_array)) # Output: 1 (Minimum element)
```

```
print(np.sqrt(25)) # Output: 5.0 (Square root)
print(np.abs(-10)) # Output: 10 (Absolute value)
print(np.exp(0)) # Output: 1.0 (Exponential e^0)
print(np.mean(my_array)) # Output: 25.428
```

6.2 Applying Functions to Entire Arrays

One of NumPy's most powerful features is the ability to apply functions to entire arrays at once:

```
numbers = np.array([1, 4, 9, 16, 25])
square_roots = np.sqrt(numbers)
print(square_roots) # Output: [1. 2. 3. 4. 5.]
```

6.3 Universal Functions (ufuncs)

NumPy's "universal functions" (ufuncs) are functions that operate element-wise on arrays. You can find a comprehensive list of ufuncs in the NumPy documentation. Some examples include: `power`, `lcm`, `gcd`, trigonometric functions (`sin`, `cos`, `tan`), comparison functions, and logical functions. You can find these in the universal function documentation. Just search "NumPy Universal Functions" on Google.

6.4 Sorting

Sorting arrays is also incredibly easy with NumPy:

```
unsorted_array = np.array([5, 2, 8, 1, 9])
sorted_array = np.sort(unsorted_array)
print(sorted_array) # Output: [1 2 5 8 9]
```

Sorting can also be applied to strings. Lowercase are ordered after uppercase, following ASCII tables.

```
string_array = np.array(["Aman", "vidae", "khushu", "Shashan", "Harman"])
print(np.sort(string_array))
```

On 2D arrays, each individual array is sorted, but not together.

```
two_d_array = np.array([[4,2,3],[6,10,4]])
print(np.sort(two_d_array))
```

For arrays containing boolean values, `False` comes before `True` because Python represents `False` as 0 and `True` as 1.

7 Array Arithmetic and Comparison

NumPy makes it easy to perform arithmetic and comparison operations on arrays.

7.1 Scalar Operations

You can perform arithmetic operations between an array and a single number (a “scalar”):

```
my_array = np.array([1, 2, 3, 4, 5])
result = my_array + 2
print(result) # Output: [3 4 5 6 7]

result2 = my_array * 6
print(result2) # Output: [ 6 12 18 24 30]

print(my_array % 6) # Output: [1 2 3 4 5]
```

7.2 Element-wise Operations

You can also perform arithmetic operations between two arrays of the same shape. The operations are performed element-wise:

```
array1 = np.array([1, 2, 3, 4])
array2 = np.array([1, 0, 1, 0])
result = array1 + array2
print(result) # Output: [2 2 4 4]
```

Similarly:

```
print(array1 > array2) # Output: [False True True True]
```

7.3 Matrix Multiplication

NumPy offers two ways to do multiplication. As seen earlier, it can be performed element-wise with the `*` symbol.

However, we can also conduct proper matrix multiplication with `@`:

```
matrix1 = np.array([[1,2,3],[4,5,6],[7,8,9]])
matrix2 = np.array([[1,2,3],[4,5,6],[1,2,3]])
print(matrix1 @ matrix2)
```

8 Conclusion: NumPy - Your Foundation for Machine Learning

NumPy is a powerful library that provides efficient tools for working with numerical data in Python. It's an essential foundation for machine learning, allowing you to store and manipulate large datasets quickly and easily. By understanding the concepts covered in this guide, you'll be well-equipped to tackle more advanced machine learning tasks.

NumPy: Advanced Array Operations

IIT Ropar - Minor in AI

February 21, 2025

1 Weather Data Analysis Using NumPy

A meteorological department collects daily temperature readings across 10 cities for a month. The department wants to analyze trends, including average temperatures, hottest and coldest days, and temperature fluctuations. Using NumPy, perform operations such as indexing, aggregation, reshaping, and boolean filtering to extract meaningful insights.

NumPy is a powerful library for numerical computing in Python. In this guide, we'll explore advanced array operations that are crucial for data manipulation in machine learning tasks.

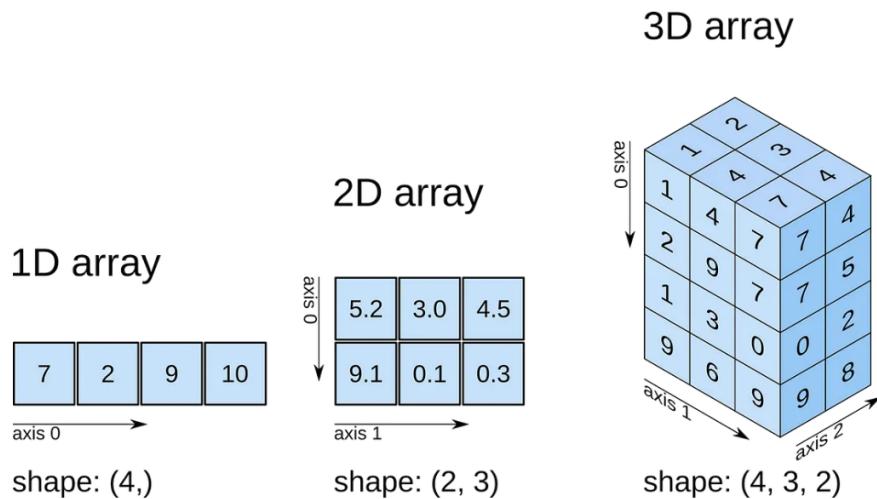


Figure 1: Illustration of 1D, 2D, and 3D NumPy arrays

2 Creating and Manipulating NumPy Arrays

Let's start by creating a simple NumPy array:

```
import numpy as np
np1 = np.array([3,4,5,6,7,8])
print(np1)
# Output:
# [3 4 5 6 7 8]
```

This creates a 1-dimensional array with 6 elements.

3 Array Indexing and Slicing

NumPy provides powerful indexing capabilities:

3.1 Positive Indexing

You can use positive indices to access elements from the start of the array:

```
print(np1[3])
# Output: 6
```

This returns the 4th element(i.e. element at 3rd index) from the start of the array.

3.2 Negative Indexing

You can use negative indices to access elements from the end of the array:

```
print(np1[-3]) # Output: 6
```

This returns the third element(i.e. element at -3 index) from the end of the array.

3.3 Array Slicing

Slicing allows you to extract a portion of the array:

```
print(np1[0:-1])
# Output: [3 4 5 6 7]
print(np1[2:])
# Output: [5 6 7 8]
print(np1[:5])
# Output: [3 4 5 6 7]
print(np1[:])
# Output: [3 4 5 6 7 8]
```

These operations demonstrate various ways to slice the array, including from the start, to the end, and with specific ranges.

3.4 Using Step Size while Array Slicing

You can also specify a step size when slicing:

```
print(np1[0:-2:2])
# Output: [3 5]
print(np1[::-2])
# Output: [3 5 7]
```

This allows you to select every nth element of the array.

4 Multidimensional Arrays

NumPy excels at handling multidimensional arrays:

```
np2 = np.array([[1,2,3,4,5,6,7], [8,9,10,11,12,13,14]])
print(np2)
# Output:
# [[ 1  2  3  4  5  6  7]
# [ 8  9  10 11 12 13 14]]
```

This creates a 2-dimensional array with 2 rows and 7 columns.

4.1 Slicing 2D Arrays

You can slice 2D arrays using comma-separated indices:

```
print(np2[:,2:5])
# Output:
# [[ 3  4  5]
# [10 11 12]]
```

This selects all rows and columns 2 through 4.

5 Modifying Array Elements

NumPy allows for easy modification of array elements:

```
np2[1,4] = 20
print(np2)
# Output:
# [[ 1  2  3  4  5  6  7]
# [ 8  9  10 11 20 13 14]]

np2[:, 1] = [20,30]
print(np2)
# Output:
# [[ 1 20  3  4  5  6  7]]
```

```
# [ 8 30 10 11 20 13 14]]  
  
np2[0, :] = [1,1,1,1,1,1,1]  
print(np2)  
# Output:  
# [[ 1 1 1 1 1 1 1 ]]  
# [ 8 30 10 11 20 13 14]]
```

These operations demonstrate how to modify single elements, columns, and rows of a 2D array.

6 Creating Special Arrays

NumPy provides functions to create arrays with specific properties:

```
outer = np.ones((5,5))  
print(outer)  
# Output:  
# [[1. 1. 1. 1. 1.]  
# [1. 1. 1. 1. 1.]  
# [1. 1. 1. 1. 1.]  
# [1. 1. 1. 1. 1.]  
# [1. 1. 1. 1. 1.]  
  
inner = np.zeros((3,3))  
print(inner)  
# Output:  
# [[0. 0. 0.]  
# [0. 0. 0.]  
# [0. 0. 0.]  
  
inner[1,1] = 9  
print(inner)  
# Output:  
# [[0. 0. 0.]  
# [0. 9. 0.]  
# [0. 0. 0.]  
  
outer[1:4, 1:4] = inner  
print(outer)  
# Output:  
# [[1. 1. 1. 1. 1.]  
# [1. 0. 0. 0. 1.]  
# [1. 0. 9. 0. 1.]  
# [1. 0. 0. 0. 1.]
```

```
# [1. 1. 1. 1. 1.]
```

This creates a 5x5 array of ones and inserts a 3x3 array of zeros (with a 9 in the center) into numpy array named outer.

7 Reshaping Arrays

Reshaping allows you to change the dimensions of an array:

```
np2 = np.array([1,2,3,4,5,6,7,8])
np_2d = np2.reshape(2,2,2)
print(np_2d)
# Outer:
# [[[1 2]
# [3 4]]
#
# [[5 6]
# [7 8]]]

np_2d = np2.reshape(2,-1,2)
print(np_2d)
# Output:
# [[[1 2]
# [3 4]]
#
# [[5 6]
# [7 8]]]
```

`np2.reshape(2,2,2)` reshapes the 1D array into a 3D array with 2 "layers", each containing 2 rows and 2 columns.

`np2.reshape(2,-1,2)` does the same thing as above, it reshapes the 1D array, in such a way that outermost layer has 2 elements and innermost layer has 2 elements, but in this case the middle layer's no. of elements were figured out by Numpy itself(since -1 written) rather than given by us.

8 Copying Arrays

When working with arrays, it's important to understand the difference between views and copies:

```
a = np.array([1,2,3,4,5])
b = a.copy()
b[1] = 100

print(b)
# Output: [ 1 100 3 4 5]
```

```
print(a)
# Output: [1 2 3 4 5]

c = a
c[1] = 200
print(c)
# Output: [ 1 200 3 4 5]
print(a)
# Output: [ 1 200 3 4 5]
```

This demonstrates that modifying a copy of an array (i.e. b in this case) doesn't affect the original array.

Whereas, modifying view of an array (i.e. c in this case) does affect the original array.

9 Conclusion

These advanced NumPy operations form the foundation for efficient data manipulation in machine learning tasks. By mastering these techniques, you'll be well-equipped to handle complex numerical computations and data pre-processing for your machine learning models.

AI in Agriculture and Ethical AI Development

Minor in AI

22-02-2025

1 Transformative AI Applications in Agriculture

Real-World Impact Case Studies

Case 1: Phenobot System

- **Problem:** Manual crop monitoring (2 hours/acre) vs AI solution (5 minutes/acre)
- **Solution:** \$100 IoT device with Raspberry Pi + U-Net segmentation
- **Outcome:** 40% faster identification of drought-resistant crops

Case 2: See & Spray Technology

- **Problem:** 70% herbicide waste in traditional spraying
- **Solution:** Real-time ML weed detection
- **Outcome:** 90% chemical reduction (Blue River Tech)

In today's rapidly evolving technological landscape, artificial intelligence (AI) is transforming traditional industries, particularly agriculture. Consider a farmer who deploys a low-cost IoT device called a *phenobot* to monitor crop health. This device captures images of crops, which are then analyzed using AI techniques to detect issues such as pest infestations, water stress, or nutrient deficiencies in real time.

2 AI in Agriculture: Solving Real-World Problems

2.1 The Problem

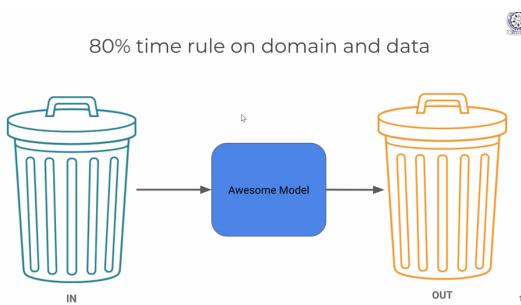
Traditional farming methods face several challenges:

- Inefficient resource management
- Late detection of crop diseases and pests
- Difficulty in predicting optimal harvest times
- Environmental concerns due to overuse of chemicals
- Labor-intensive and time-consuming manual inspections
- Prone to human error and inconsistent data collection

2.2 How AI Addresses the Problem

AI offers compelling solutions through:

1. **Precision Farming:** AI-powered equipment uses sensors and GPS to optimize planting, watering, and fertilizing.
2. **Crop Monitoring:** Computer vision and machine learning analyze crop health, enabling early detection of issues.



3. **Predictive Analytics:** AI algorithms analyze various data sources to predict weather patterns and crop yields.
4. **Automated Pest Control:** AI-driven systems can identify and target pests with precision, reducing chemical use.
5. **Image Analysis with Deep Learning:** Implementing segmentation models such as U-Net to analyze images and extract meaningful traits (e.g., leaf size and health).

3 AI Development Principles

3.1 Thumb Rules for AI Model Building

Dr. Praveen's 80/20 Principles

1. **80% Time on Data:** Allocate 80% of time for domain understanding and data preparation.
2. **20% Time on Models:** Reserve 20% of time for model building and refinement.
3. **80% Accuracy MVP:** Aim for an initial accuracy of 80% for MVP development.
4. **Hybrid Approach:** Use minimal labeled data effectively and consider hybrid approaches.
5. **Sustainability First:** Optimize for energy efficiency.

3.2 Ethical Considerations

- **Asimov Updated:** AI should benefit all ecosystems, not just humans.
- **Green AI:** A 10MB model running on solar beats 10GB cloud model.
- **Explainability:** Farmers must trust and understand predictions.
- **Environmental Impact:** Evaluate and minimize the environmental footprint of AI systems.
- **Fairness:** Ensure the technology benefits a broad spectrum of stakeholders.

5 thumb-rules to building and deploying models

1. Spend 80% of your time on understanding the domain; cleaning, preparing and insights into the data (EDA)
2. Spend less than 20% of your time in building and refining the model!
3. MVP: Spend 80% of that 20% time in building a model that gives at least 80% accuracy (or Error of 20%) in the wild rather than 99% accuracy in validation set!
4. Spend only 20% of the 20% model building time in making it accurate!
5.
 - a. Less labeled data -> rely on prior knowledge or science-based models or rule-based models.
 - b. More labeled data -> consider yourself very lucky (go for deep learning)!
 - c. Are you in-between? -> Hybrid approach (data + science models)

Figure 1: Enter Caption

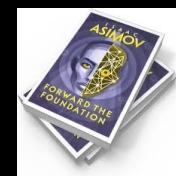
4 Key Takeaways

- Data quality beats model complexity
- Start with explainable models
- Every AI system has environmental costs
- Cross-disciplinary knowledge is crucial
- AI enhances decision-making in farming through data-driven insights
- Precision agriculture enabled by AI optimizes resource use and reduces waste
- AI contributes to sustainable farming practices by reducing chemical usage and environmental impact

5 Conclusion

AI in agriculture represents a significant leap forward in addressing food security and sustainability challenges. By enabling precision farming, improving crop monitoring, and optimizing resource use, AI is helping farmers increase productivity while reducing environmental impact. As technology continues to evolve, we can expect even more innovative applications of AI in agriculture, further transforming the way we grow and manage our food resources.

"The 100 slide AI"



ISAAC ASIMOV'S 3LAWS:-

1. A robot may not injure a human being or, through inaction, allow a human being to come to harm.
2. A robot must obey orders given to it by human beings except where such orders would conflict with the first law. [Robot shouldn't harm]
3. A robot must protect its own existence as long as protection does not conflict with the first or second law. [Robot should help human being without thinking about its life]

PROPOSAL FOR MODIFICATION TO THE 3 LAWS FOR AI

1. AI should benefit all the earth and the inhabitants (not alone humans. → Benefit other parts of the ecosystem as well.)
2. It should enhance the quality of life at all planes of existence (not only time and work efficiency)
→ It should accompany mental wellness.
3. If the user is human, augment the intelligence and perceptions of the user (liberation and not making them more dependent on them).
→ It must be your critical reviewer, it must help you think differently.
→ Never let go off your mental capability

Does google map make us better navigators or it makes us dependent on it?

→ Making us dependent.

Can you give 'Mukti' to the user rather than making the user dependent?

use CHAT GPT to challenge yourself to maintain your cognitive thinking & your abilities.

"CONCERN FOR DECREASE IN REAL INTELLIGENCE"

THUMB RULES TO BUILDING AND DEPLOYING MODELS:-

1. Spend 80% of your time on understanding the domain; cleaning, preparing and insights into the data (EDA).



* Data Analysis
80% → Build your data as the data you get is never complete

2. 20% → Refining your model.

3. MVP (Minimum Viable Prototype):- 80% of 20% time → building a model that gives 80% accuracy atleast in the wild rather than 99% accuracy in validation set. → Random data

4. Spend only 20% of the 20% model building time in making it accurate.

5. Don't go looking for labeled data.

→ MINIMAL DATA ↴

Less labeled data → rely on prior knowledge OR rule-based models

JACKPOT More labeled data → Go for deep learning

BALANCE IN Between → hybrid (data + science model)

Accuracy + Reliability

80% Time rule on domain data



Rules for building Minimum Viable product - A)

Occam's Razor :- Simplest explanation is usually the best.

'principle of parsimony' :- If given 2 models, always choose the simpler one.

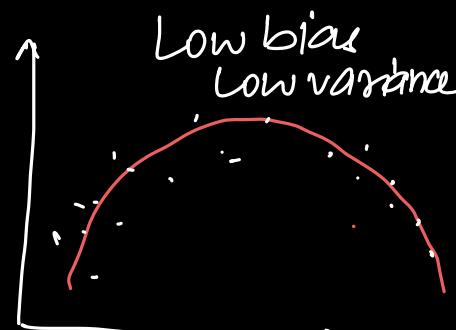
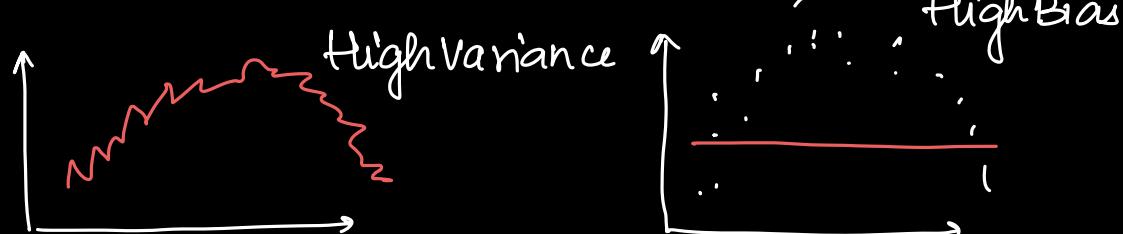
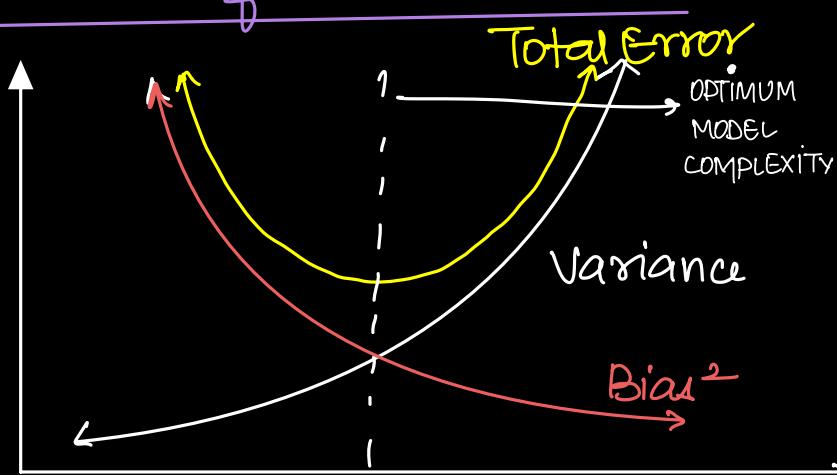
Can you explain reliability of the Model? Say yes

→ To explain, use the simplest possible model (say if-else) with the fewest parameters

As someone working for the world, you not only need to build smaller UML, but also models with less footprints less data, minimal amount, environmental footprint

→ ETHICAL
RESPONSIBILITY

Variations of Occam's Razor :-



REAL WORLD APPLICATIONS :-

AGRICULTURE :-

Monitoring of farms:- Sensing:-

Input, Data
(images, audio etc)

Storing of information.

Temp | Humidity Measurement

IOT Devices:-

Cold storages, Transporting, logistics

Network of physical devices(things) embedded with hardware, software and connectivity which enables these to connect, collect and exchange data.

"IOT AND AI"

Soil Informatics, irrigation, phenotyping, Micro-climate

Automated Variety Selection :-

Yield ↑

Pests and Disease

Climate Adaption

Quality Improvement

Increase Gene diversity

Specific Traits

Sustainable Production

Reduced Input Resources

which variety satisfies all / most of these conditions?

How do you select them

PHENOBOT → Phenotype (External trade / visible)
Cloud Computing

"Thresholding"

UNET :- convolution & Extraction

The Memoryless Magic Trick: Predicting the Future with Markov Chains

Minor in AI, IIT Ropar
24th Feb, 2025

Predicting the Future is Not Magic, it's Maths!

A Mall, an Offer, and a Mystery

Imagine you're assisting Ramesh, the owner of a bustling mall. He's planning to launch a new festive offer, but he wants to be strategic. He needs to choose the *right* place in the mall to maximize customer engagement and get the most bang for his buck.

The mall has four main areas:

- **Entrance (E):** Where customers first enter the mall.
- **Clothing Section (C):** Filled with trendy apparel.
- **Electronics Section (X):** Packed with the latest gadgets.
- **Food Court (F):** A place to relax and grab a bite.

Ramesh wonders, “*Where should I roll out this offer to get the most people excited about it?*”

Some people suggest the Entrance, where everyone walks through. Others say the middle of the mall or the center, but that's very vague.

Which area do *you* think is best? Let's keep this question in mind as we explore Markov Chains. We'll revisit it later and see how we can use this powerful tool to help Ramesh make the best decision.



Figure 1: Shopping case Study!

Probability: The Foundation of Our Future

Before diving into Markov Chains, let's refresh our understanding of probability. Think about these simple scenarios:

- **Tossing a Fair Coin:** When you flip a fair coin, there's an equal chance of getting heads or tails. This means the probability of getting heads is $1/2$ (or 50%), and the probability of getting tails is also $1/2$ (or 50%).
- **Drawing Balls from a Bag:** Imagine a bag containing 5 red balls and 6 blue balls. What's the probability of picking a red ball?
 - There are a total of 11 balls (5 red + 6 blue).
 - The probability of picking a red ball is $5/11$.

Now, let's add a twist! Suppose you pick a red ball, but *don't* put it back in the bag. What's the probability of picking another red ball on your second try?

- Now there are only 10 balls left in the bag.
- And only 4 of them are red.
- So, the probability of picking another red ball is now $4/10$.

This brings us to an important concept:

- **Dependent Events:** Events where the outcome of one affects the probability of the other. In our ball example, the probability of picking a red ball the second time *depended* on whether we picked a red or blue ball the first time, and if we replaced it.
- **Independent Events:** In contrary, if we put the ball back into the bag after each draw, the outcome of each draw would not have affected the other and thus called independent events.

States and Transitions: A Tale of 1000 Students

Let's consider another scenario to introduce the core concepts of Markov Chains. Imagine we're observing 1000 students. At any given moment, a student can be in one of two states:

- **Studying**
- **Scrolling through Social Media**

Now, let's introduce some probabilities:

- **A student currently studying has a 60% (0.6) chance of continuing to study.** This also means they have a 40% (0.4) chance of switching to scrolling through social media.
- **A student currently scrolling through social media has an 80% (0.8) chance of continuing to scroll.** And only 20% (0.2) of these students start studying.

We can visualize this as a diagram with two circles (representing the states) and arrows showing the transitions between them, labeled with the corresponding probabilities.

Let's say that at time T0, there are 500 students studying and 500 students scrolling through social media. At time T1 (the next moment), how many students will be studying?

- Out of the initial 500 studying, 60% will continue to study: $500 * 0.6 = 300$
- Out of the initial 500 scrolling, 20% will switch to studying: $500 * 0.2 = 100$

So, at time T1, we expect $300 + 100 = 400$ students to be studying. Similarly, we can calculate the number of students scrolling through social media.

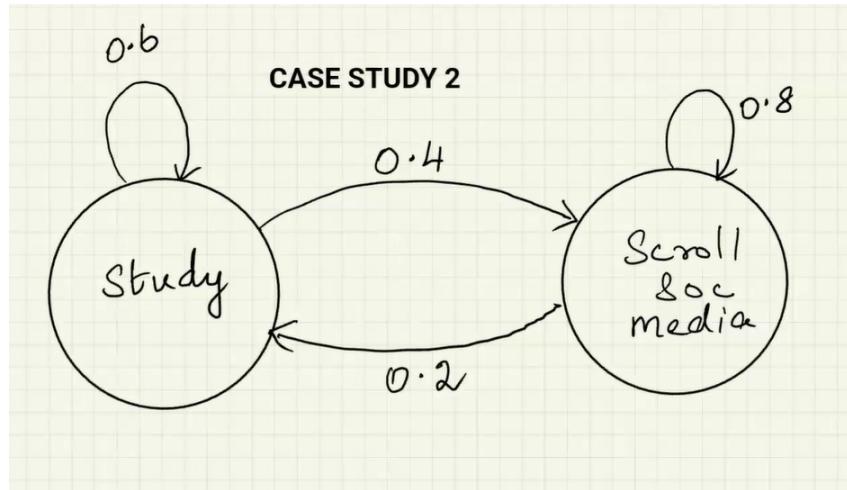


Figure 2: A state diagram showing the "Studying" and "Social Media" states, with arrows indicating transition probabilities (0.6, 0.4, 0.8, 0.2).

Predicting the Future with Excel

We can extend this concept to predict what happens in the future. Let's use a spreadsheet (like Google Sheets or Microsoft Excel) to simulate this process.

1. **Set up columns:** Create columns for Time (T0, T1, T2...), Number of Students Studying, and Number of Students Scrolling.
2. **Enter initial values:** In row T0, enter 500 for both "Studying" and "Scrolling."
3. **Write formulas:** In row T1, enter the formulas to calculate the number of students in each state based on the probabilities:
 - Studying (T1) = (Studying (T0) * 0.6) + (Scrolling (T0) * 0.2)
 - Scrolling (T1) = (Studying (T0) * 0.4) + (Scrolling (T0) * 0.8)
4. **Use the fill handle:** Select the cells with formulas and drag the fill handle down to automatically calculate the values for T2, T3, and so on.

What you'll notice is that over time, the number of students in each state will converge to a stable value. It doesn't matter what the initial numbers are, given large enough time. The student ratio will converge to roughly 33/67 due to probabilities.

The Transition Matrix: A Formal Representation

What we've been working with can be formally represented using a **transition matrix**. This is a square matrix that shows the probabilities of transitioning between different states.

For our student example, the transition matrix would be:

	Studying	Scrolling
Studying	0.6	0.4
Scrolling	0.2	0.8

Each row represents the current state, and each column represents the next state. For example, the value in the first row and second column (0.4) represents the probability of a student transitioning from the "Studying" state to the "Scrolling" state.

	A	B	C
1	Study	Scroll Soc. Media	
2	T0	500	500
3	T1	400	600
4	T2	360	640
5	T3	344	656
6	T4	337.6	662.4
7	T5	335.04	664.96
8	T6	334.016	665.984
9	T7	333.6064	666.3936
10	T8	333.44256	666.55744
11	T9	333.377024	666.622976
12	T10	333.3508096	666.6491904

Figure 3: A screenshot of a spreadsheet showing the simulation of student transitions, with the numbers converging over time.

Back to the Mall: Using Markov Chains for Decision Making

Now, let's revisit Ramesh and his festive offer at the mall! To help him make the best decision, we need data about customer movement between the four areas: Entrance (E), Clothing (C), Electronics (X), and Food Court (F). Based on Ramesh's observation of customer flow, we've constructed the following transition matrix, visualized below, showing the probabilities of customers moving between areas:

	A	B	C	D	E	F	G	H	I	J	K
1	E	C	X	F			E	C	X	F	
2	From E	0.1	0.5	0.2	0.2		1000	0	0	0	
3	From C	0.3	0.4	0.2	0.1		100	500	200	200	1000
4	From X	0.2	0.3	0.4	0.1		220	330	260	190	1000
5	From F	0.1	0.1	0.3	0.5		192	339	271	198	1000
6							194.9	332.7	274	198.4	1000
7							193.94	332.57	274.64	198.85	1000
8							193.978	332.275	274.813	198.934	1000
9							193.9363	332.2363	274.856	198.9714	1000
10							193.9328e	332.21661274.86834	198.98219	1000	

Figure 4: Mall Customer Transition Data and Simulation

The transition matrix is represented as:

	E	C	X	F
E	0.1	0.5	0.2	0.2
C	0.3	0.4	0.2	0.1
X	0.2	0.3	0.4	0.1
F	0.1	0.1	0.3	0.5

As evidenced by the simulation data, starting with 1000 customers entering the mall at the Entrance (E), the number of customers at each location over time settles into a relatively stable state. For example, this means if a customer is in the Clothing section now, there is 30% (0.3) chance that they move to Entrance, 40% (0.4) chance that they remain in Clothing section, 20% (0.2) chance that they move to Electronics, and 10% (0.1) chance that they move to Food court. The simulation (see Image) shows that

the Clothing (C) section consistently attracts a significant proportion of customers after a few iterations (time steps).

Therefore, based on this analysis, the Clothing section is the most strategic location to launch the Festive Offer for Ramesh to maximize customer engagement.

Markov Chains: Key Concepts

Here's a summary of the key concepts we've covered:

- **Markov Chain:** A mathematical model describing a system that transitions between states based on probabilities.
- **State:** A specific condition or situation of the system (e.g., "Studying," "Scrolling," "In the Entrance," "In the Clothing Section").
- **Transition Probability:** The probability of moving from one state to another.
- **Transition Matrix:** A matrix containing all the transition probabilities between the states.
- **Markov Property (Memoryless Property):** The future state depends only on the current state and not on the past history.
- **Convergence:** A state is reached where the proportion of individuals in each is balanced after a long time.

Real-World Applications

Markov Chains are used in many different areas, including:

- **Weather Forecasting:** Predicting the weather based on current conditions and transition probabilities between different weather states (sunny, rainy, cloudy).
- **Speech Recognition:** Recognizing spoken words by analyzing the sequence of sounds and their transition probabilities.
- **Finance:** Modeling stock prices and other financial data.
- **Recommendation Systems:** Predicting what products or movies a user might like based on their past behavior.
- **Reinforcement Learning:** This technique helps train AI agents by rewarding and penalizing specific actions.
- **Hidden Markov Models:** A model where the states are unknown, used for prediction of sequencing.

You've now taken your first steps into the world of Markov Chains! By understanding the basic concepts and applying them to practical examples, you can start using this powerful tool to model probabilistic systems and make better decisions. As you continue your AI journey, explore the various applications of Markov Chains and delve deeper into the mathematical foundations. Keep experimenting, keep learning, and you'll be amazed at what you can achieve!

Unlock Your Data Superpowers: A Beginner's Guide to Pandas

Minor in AI, IIT Ropar

25th Feb, 2025

Welcome, future data wizards! This module is your launchpad into the wonderful world of Pandas, a powerful Python library that will become your best friend when working with data. We'll start with an intuitive explanation of the need for Pandas, then dive into its core components, and finally, explore essential operations to manipulate data like a pro. Let's begin!

The OTT Platform Problem - Why Pandas?

Imagine you're building your own Online TV (OTT) Platform, a competitor to Netflix or Disney+. What cool new feature would you add? Think about it. Maybe AI-generated storylines based on user preferences? Or perhaps multi-angle viewing with live commentary?

Now, consider this: to make these features *actually* work, you need a LOT of data. You need information about:

- **Each Movie/Show:** Title, Genre, Release Year, Director, Actors, Ratings, Reviews, Specifications, etc.
- **Each User:** Viewing History, Ratings Given, Search Queries, Subscription Type, Devices Used, location, demographic.
- **Platform Performance:** Streaming Quality, Server Load, Ad Revenue, Subscription Numbers.

All this data needs to be organized, cleaned, and analyzed to provide recommendations, fix bugs, and make smart business decisions. Imagine trying to do this with just basic Python lists and dictionaries... It would be a nightmare!

That's where Pandas comes in. Pandas is designed to handle large amounts of *structured data* efficiently. It's like a super-powered spreadsheet for Python, making data manipulation a breeze.

Understanding Different Types of Data

Before we dive into Pandas, let's take a step back and classify the different types of data we encounter in the real world. Understanding these data types will give you a better appreciation for what Pandas is trying to solve.

1. **Cross-Sectional Data:** This is a snapshot of data at a *single point in time*. Think of it like taking a survey of students in a class to understand how many of them understand the day's lecture. You're collecting data from many individuals (students) at one specific time.



Figure 1: A snapshot of a classroom with students raising their hands to indicate understanding.

2. **Time Series Data:** This is data collected *over a period of time*. Imagine tracking the daily stock prices of a company for the last 20 years. You're collecting data from one entity (the company) over a long period of time.

3. **Panel Data:** This is a *combination of both cross-sectional and time series data*. Think of tracking the performance of 10 different OTT platforms over a year, for a specified time. You have multiple entities (OTT platforms) and data collected over a period of time (a year).

Pandas was born out of the need to efficiently work with panel data. Operations that were previously slow became significantly faster thanks to the data structures and algorithms implemented in Pandas.

Introducing Pandas - From Panel Data to DataFrames

You might have wondered where the name "Pandas" comes from. Well, it's derived from "Panel Data"! Initially designed to handle panel data effectively, Pandas has evolved to handle a wide variety of structured data. This library emerged when a company wanted to analyze huge sets of panel data and wanted the processing to be fast. Now, Pandas is used for virtually every type of data processing!

The core of Pandas is the **DataFrame**.



Figure 2: Pandas - The data manipulation tool!

A DataFrame is like a table (rows and columns), where each column can have a different data type (numbers, text, dates, etc.). It's a powerful way to organize and represent data, and it's the primary data structure you'll be working with in Pandas.

But what makes a DataFrame so special? It's the fact that it can hold heterogeneous Data. The different types of data are combined to form one collective organization. Each attribute, whether strings or integers, can exist with the same data frame.

Think back to our OTT platform example. You can create a DataFrame to hold movie data, where each row represents a movie, and the columns represent the movie's title, genre, release year, rating, and so on. The power of the data frame is also that you can frame data together!

Getting Started with Pandas - Basic DataFrames

Let's create a simple DataFrame to see how it works. Open your Python editor (or a Colab notebook if you're using Google Colab) and type in the following code:

```
1 import pandas as pd
2
3 # Create a dictionary of data
4 data = {
5     'Name': ['Alice', 'Bob', 'Charlie'],
6     'Age': [25, 30, 28],
7     'City': ['New York', 'London', 'Paris']
8 }
9
10 # Create a DataFrame from the dictionary
11 df = pd.DataFrame(data)
12
13 # Print the DataFrame
14 print(df)
```

Listing 1: Creating a Basic DataFrame

This code does the following:

- (a) `import pandas as pd`: This line imports the Pandas library and gives it the alias `pd`. This is a standard convention, and it means you can now access Pandas functions using `pd`. (e.g., `pd.DataFrame`).
- (b) `data = { ... }`: This creates a Python dictionary called `data`. The keys of the dictionary are the column names ('Name', 'Age', 'City'), and the values are lists containing the data for each column.
- (c) `df = pd.DataFrame(data)`: This is the magic line! It creates a Pandas DataFrame named `df` from the `data` dictionary. The dictionary's keys become the column names, and the lists become the column data.
- (d) `print(df)`: This line prints the DataFrame to your console.

Run the code, and you should see something like this:

```
Name    Age      City
0   Alice    25  New York
1     Bob    30    London
2 Charlie   28     Paris
```

Congratulations! You've created your first Pandas DataFrame. Notice how Pandas automatically assigns row labels (0, 1, 2) to each row.

Essential DataFrame Operations

Now that you have a DataFrame, let's explore some essential operations you can perform to manipulate and analyze your data. We'll expand upon what was covered before.

(a) Data Types:

You can check the data types of each column in your DataFrame using the `.dtypes` attribute. For example:

```
1 print(df.dtypes)
```

Listing 2: Checking Data Types

This will output something like:

```
Name      object
Age       int64
City      object
dtype: object
```

Notice that Pandas infers the data type of the 'Name' and 'City' columns as `object` (which is generally used for strings) and the 'Age' column as `int64` (a 64-bit integer). Pandas can also infer floating point numbers and Booleans.

(b) Accessing Data:

- **Selecting Columns:** You can select a single column using square brackets:

```
1 print(df['Name'])
```

Listing 3: Selecting Single Column

This will print the 'Name' column. You can select multiple columns by passing a list of column names within the square brackets:

```
1 print(df[['Name', 'Age']])
```

Listing 4: Selecting Multiple Columns

This will print the 'Name' and 'Age' columns. Notice the double square brackets! The outer brackets are for accessing the DataFrame, and the inner brackets create a list of column names.

- **Selecting Rows Using ‘.loc[]’ and ‘.iloc[]’:** Pandas provides two primary ways to select rows: ‘.loc[]’ (label-based) and ‘.iloc[]’ (integer-based).
 - `.loc[]`: Accesses rows by label (index name) and columns by name.
 - `.iloc[]`: Accesses rows and columns by integer position.

```

1 import pandas as pd
2
3 data = {
4     'Name': ['Aryan', 'Aman', 'Akhil', 'Akash', 'Ayush'],
5     'Age': [25, 30, 22, 28, 24],
6     'City': ['Pune', 'Blore', 'Hyd', 'Pune', 'Delhi'],
7     'Salary': [60000, 60000, 45000, 70000, 55000]
8 }
9
10 df = pd.DataFrame(data)
11
12 # Get Akhil's Name and Age using .loc[]
13 akhil_info = df.loc[2, ['Name', 'Age']]
14 print(akhil_info)
15
16 # Get Aryan's Name and Age using .iloc[]
17 aryan_info = df.iloc[0, [0, 1]]
18 print(aryan_info)
```

Listing 5: Selecting Rows and Columns Using loc and iloc

- **Selecting Rows with `.head()` and `.tail()`:** You can select the first few rows using the `.head()` method and last few rows using the `.tail()` method.

```

1 print(df.head(2)) # Display the first 2 rows
2 print(df.tail(1)) # Display the last row
```

Listing 6: Selecting Rows Using head() and tail()

This gives a great high level view about the data.

(c) Filtering Data:

One of the most powerful features of Pandas is the ability to filter data based on conditions. For example, let's say you want to find all the people in your DataFrame who are older than 27. You can do this using boolean indexing:

```

1 import pandas as pd
2
3 data = {
4     'Name': ['Aryan', 'Aman', 'Akhil', 'Akash', 'Ayush'],
5     'Age': [25, 30, 22, 28, 24],
6     'City': ['Pune', 'Blore', 'Hyd', 'Pune', 'Delhi'],
7     'Salary': [60000, 60000, 45000, 70000, 55000]
8 }
9
10 df = pd.DataFrame(data)
11
12 filtered_df = df[df['Age'] < 25]
13 print(filtered_df)
```

Listing 7: Filtering Data Using Boolean Indexing

This code does the following:

- `df['Age'] < 25`: This creates a *boolean series* where each element is `True` if the corresponding age is less than 25, and `False` otherwise.
- `df[...]`: This uses the boolean series to select only the rows where the condition is `True`.

You can combine multiple conditions using logical operators (`&` for "and", `|` for "or"). For example, to find all people who are in Pune with salary more than 50000:

```

1 import pandas as pd
2
3 data = {
4     'Name': ['Aryan', 'Aman', 'Akhil', 'Akash', 'Ayush'],
5     'Age': [25, 30, 22, 28, 24],
6     'City': ['Pune', 'Blore', 'Hyd', 'Pune', 'Delhi'],
7     'Salary': [60000, 60000, 45000, 70000, 55000]
8 }
9
10 df = pd.DataFrame(data)
11
12 salary = df[(df['Salary'] > 50000) & (df['City'] == 'Pune')]
13 print(salary)

```

Listing 8: Filtering with Multiple Conditions

Working with Missing Data

Real-world data is messy. It often contains missing values, which can cause problems if you try to perform calculations or analyses. Pandas provides tools to handle these missing values.

- (a) **Representing Missing Data:** Pandas uses `NaN` (Not a Number) to represent missing numerical values. The string counterpart is `None`.
- (b) **Detecting Missing Values:** You can use the `.isnull()` method to detect missing values in your DataFrame. This method returns a boolean DataFrame where `True` indicates a missing value and `False` indicates a non-missing value.

```

1 import pandas as pd
2
3 data = {
4     'Team': ['Ind', 'Aus', 'Eng'],
5     'Score': [250, None, None],
6     'Result': ['W', 'W', 'L']
7 }
8
9 df = pd.DataFrame(data)
10 print(df)
11
12 print("\n", df.isnull())

```

Listing 9: Detecting Missing Values using `isnull()`

Output:

	Team	Score	Result
0	Ind	250.0	W
1	Aus	NaN	W
2	Eng	NaN	L

	Team	Score	Result
0	False	False	False
1	False	True	False
2	False	True	False

- (c) **Handling Missing Values:**

- `dropna()`: This method removes rows or columns containing missing values.

```

1 import pandas as pd
2
3 data = {
4     'Team': ['Ind', 'Aus', 'Eng'],
5     'Score': [250, None, None],
6     'Result': ['W', 'W', 'L']
7 }
8
9 df = pd.DataFrame(data)

```

```

10
11 df_dropped = df.dropna()
12 print("\n", df_dropped, "\n")

```

Listing 10: Removing Missing Values using dropna()

This will drop any row with at least one missing value. Output:

	Team	Score	Result
0	Ind	250.0	W

- **fillna()**: This method fills missing values with a specified value.

```

1 import pandas as pd
2
3 data = {
4     'Team': ['Ind', 'Aus', 'Eng'],
5     'Score': [250, None, None],
6     'Result': ['W', 'W', 'L']
7 }
8
9 df = pd.DataFrame(data)
10
11 df_filled = df.fillna(0)
12 print(df_filled)

```

Listing 11: Filling Missing Values using fillna()

Output:

	Team	Score	Result
0	Ind	250.0	W
1	Aus	0.0	W
2	Eng	0.0	L

This gives a new data frame, and all NaNs are now 0. You can also fill missing values with the mean, median, or mode of a column.

This is only a brief introduction, but a strong beginning to understanding Pandas, DataFrames, and essential operations for data manipulation. Think of Pandas as a language that can interpret the requirements of a Data Scientist, and run through large numbers of data to filter, process, analyze, and give data insight to help your company grow.

Minor in AI

Pandas I

25 Feb 2025

Class Notes

Topics: Data Frames, Clean and prepare data frames

Pandas was created by Wes McKinney in 2008 while he was working at a financial firm called AQR Capital Management. He needed a powerful and flexible tool for analyzing large financial datasets, but existing tools like Excel and R were either too slow or lacked essential functionalities. The name "pandas" comes from the term "Panel Data", which is a statistical term referring to multidimensional structured datasets commonly used in econometrics and data analysis.

Before pandas, Python lacked a dedicated data analysis library. NumPy was useful for numerical computing, but it did not handle structured data (tables with labels) efficiently. pandas bridged the gap by introducing DataFrames, which made Python a strong alternative to R for data science.

Concepts discussed in class:

- Time series data, cross sectional data and panel data
- Data Frames
- In place
- Why we need query processing
- Applications from e-commerce sites
- Missing values and cleaning data frames

Program on what Pandas can do:

```
import pandas as pd
# Sample data for 5 e-commerce products
data = {
    'Product Domain': ['Electronics', 'Clothing', 'Electronics', 'Home & Kitchen', 'Books'],
    'Model': ['Smartphone X', 'T-Shirt', 'Laptop Pro', 'Coffee Maker', 'Python Programming'],
    'Color': ['Black', 'Blue', 'Silver', 'Red', 'N/A'],
```

```

'Price': [899.99, 19.99, 1299.99, 49.99, 29.99],
'Rating': [4.5, 4.0, 4.8, 3.5, 5.0]
}

# Create the DataFrame
df = pd.DataFrame(data)

# Display the entire DataFrame
print("Original DataFrame:\n", df)

# Accessing columns
print("\nProduct Models:\n", df['Model'])

# Accessing rows using index
print("\nSecond product:\n", df.iloc[1])

# Filtering data
print("\nProducts with rating >= 4.5:\n", df[df['Rating'] >= 4.5])

# Sorting data
print("\nProducts sorted by price:\n", df.sort_values(by='Price'))

```

Operation to understand the capability

```

import pandas as pd
df = pd.read_csv("sales_data-1.csv")
print(df)
df.fillna({'Price': df['Price'].mean()}, inplace=True)
print(df)
df['Product'] = df['Product'].str.replace('-', ' ').str.title()
print(df)
df.plot()

```

Understanding Basics in Pandas - Data Types

```

data = {'name': ['Alice', 'Bob', 'Charlie'], 'age': [25, 30, 35]}
df1 = pd.DataFrame(data)

data2 = {'id': [1, 2, 3], 'age': [25, 30, 35]}
df2 = pd.DataFrame(data2)

print(df1.dtypes, "\n")
print(df2.dtypes)
print("\n")

df1['name'] = df1['name'].astype('string')
print(df1.dtypes)

```

Working with Data Frames

```

import pandas as pd

# Sample DataFrame (replace with your actual data)
data = {'Name': ['Aryan', 'Aman', 'Akhil', 'Akash', 'Ayush'],
        'Age': [25, 30, 22, 28, 24],
        'City': ['Pune', 'Blore', 'Hyd', 'Pune', 'Delhi'],
        'Salary': [60000, 60000, 45000, 70000, 55000]}

df = pd.DataFrame(data)

# Displaying data
print("Head:\n", df.head(2)) # First 2 rows
print("\nTail:\n", df.tail(2)) # Last 2 rows

# Selecting columns
names = df['Name']
print("\nNames:\n", names)

# Selecting multiple columns
name_and_age = df[['Name', 'Age']]

```

```

print("\nName and Age:\n", name_and_age)

# Selecting rows and columns using loc[] (label-based indexing)
akhil_info = df.loc[2, ['Name', 'Age']] # Row with label 0, columns "Name" and "Age"
print("Akhil Info:\n", akhil_info)

# Selecting rows and columns using iloc[] (integer-based indexing)
aryan_info = df.iloc[0, [0, 1]] # Row with index 1, columns with indices 0 and 1
print("Aryan Info:\n", aryan_info)

print("\n")
young_people1 = df[df['Age'] < 25]
print("Below 25:\n", young_people1)

print("\n")
# Filtering and conditional selection
young_people2 = df[df['Age'] < 25]
print("Below 25:\n", young_people2)

# Multiple conditions
high_earners_in_london = df[(df['Salary'] > 50000) & (df['City'] == 'Pune')]
print("High Earners in Pune:\n", high_earners_in_london)

```

Working with Missing Values

```

import pandas as pd

# Create a sample DataFrame with some missing values
data = {"Team": ['India', 'Australia', 'England', 'South Africa'],
        'Score': [250, None, 200, 220],
        'Overs': [50, 50, 45, None],
        'Result': ['Won', 'Lost', 'Won', 'Lost']}
df = pd.DataFrame(data)

```

```
# Check for missing values
print("Missing values:")
print(df.isnull())

# Drop rows with any missing values
df_dropped = df.dropna()
print("\nDataFrame after dropping rows with missing values:")
print(df_dropped)

# Fill missing values with a specific value (e.g., 0)
df_filled = df.fillna(0)
print("\nDataFrame after filling missing values with 0:")
print(df_filled)
```

```
import pandas as pd
import numpy as np

# Create a sample DataFrame with missing values
data = {'Student': ['Alice', 'Bob', 'Charlie'],
        'Test 1': [8, np.nan, 7],
        'Test 2': [9, 6, 8],
        'Test 3': [7, np.nan, 9],
        'Test 4': [6, 8, 7],
        'Test 5': [10, 7, 9]}
df = pd.DataFrame(data)

# Fill missing values with 5
df_filled_with_5 = df.fillna(5)

# Print the original, mean filled, and 5 filled DataFrames
print("Original DataFrame:\n", df)
print("\nDataFrame filled with 5:\n", df_filled_with_5)
```

Panda 2.0

Welcome to Panda 2.0!

If you're just starting your journey with Python and data analysis, you've come to the right place. This book will guide you through the fundamentals of Pandas, a powerful library for working with data. We'll be using Google Colab, a fantastic environment for writing and running Python code right in your browser.



Figure 1: Pandas can't code but we can do a lot with pandas framework.

1 Combining Datasets - A Real-World Scenario

Imagine you're running an online store. You have two separate spreadsheets: one contains information about your products (like their ID, brand, and price), and the other contains information about your sales (like the product ID, the date of sale, and the quantity sold).

You realize that to gain better insights – like knowing which brands are selling the most or whether higher-priced items have lower sales – you need to combine these two spreadsheets. This is where Pandas comes to the rescue! Pandas helps you bring these separate tables together, clean up any inconsistencies, and analyze the combined data to answer your business questions.

The core of this process is something called “merging” dataframes. Let's dive deeper and explore how Pandas lets us achieve this.

2 Understanding DataFrames

Before we start, let's quickly recap what a Pandas DataFrame is. Think of it as a table in a spreadsheet. It has rows and columns, where each column can hold a different type of data (numbers, text, etc.). You can create a DataFrame from various sources, like a list of dictionaries or even directly from a CSV file (more on that later!).

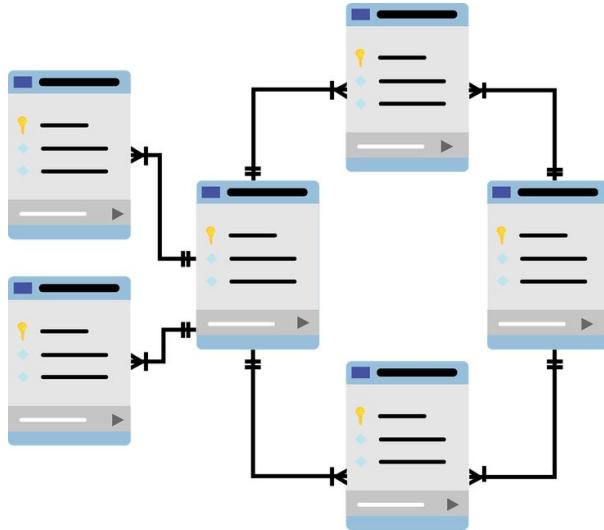


Figure 2: Data Frames can be merged using pandas.

3 Merging DataFrames - Different Approaches

When merging DataFrames, we need a common column (or columns) that connects the two tables. In our online store example, the “product ID” is the common link between the product data and the sales data. But what happens if one table has a product ID that doesn’t exist in the other? That’s where different types of merges come into play.

Imagine you are combining the following two tables/dataframes:

Product Data (DataFrame 1):

Product ID	Brand	Price
1	Brand A	20
2	Brand B	30
3	Brand C	25
4	Brand D	40
5	Brand E	50

Sales Data (DataFrame 2):

- **Inner Merge:** This keeps only the rows where the “Product ID” exists in *both* DataFrames. In our example, it would only include products with IDs 1, 2, 3, and

Product ID	Sales	Quantity
1	100	5
2	150	8
3	120	6
4	200	10
6	80	4

4. Product ID 5 will be discarded from the first data frame and Product ID 6 will be discarded from the second dataframe.

Example Code:

```
inner_merge = pd.merge(df1, df2, how='inner', on='CustomerID')
print(inner_merge)
```

- **Outer Merge:** This keeps all rows from *both* DataFrames. If a “Product ID” exists in one DataFrame but not the other, the missing values will be filled with “NaN” (Not a Number), representing missing data. So, it would have Product IDs from 1 to 6.

Example Code:

```
outer_merge = pd.merge(df1, df2, how='outer', on='CustomerID')
print(outer_merge)
```

- **Left Merge:** This keeps all rows from the *left* DataFrame (the first one you specify) and the matching rows from the *right* DataFrame (the second one). If a “Product ID” in the left DataFrame doesn’t exist in the right, the corresponding columns from the right DataFrame will be filled with “NaN”. So, you will have product IDs from 1 to 5.

Example Code:

```
left_merge = pd.merge(df1, df2, how='left', on='CustomerID')
print(left_merge)
```

- **Right Merge:** This keeps all rows from the *right* DataFrame and the matching rows from the *left* DataFrame. If a “Product ID” in the right DataFrame doesn’t exist in the left, the corresponding columns from the left DataFrame will be filled with “NaN”. You will have product IDs from 1, 2, 3, 4, and 6.

Example Code:

```
right_merge = pd.merge(df1, df2, how='right', on='CustomerID')
print(right_merge)
```

Essentially:

- **Inner:** Intersection

- **Outer:** Union

The choice of which merge to use depends on the analysis you're trying to do and the questions you're trying to answer. If you only want information for products that have both product details *and* sales data, an inner merge is the way to go. If you want *all* product details, even if there are no sales recorded, a left merge is appropriate.

Example Code:

```
import pandas as pd
df = pd.read_csv("name_of_your_data_file.csv")

# First DataFrame: Customer info
df1 = pd.DataFrame({
    'CustomerID': [1, 2, 3, 4],
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 35, 40]
})

# Second DataFrame: Orders
df2 = pd.DataFrame({
    'CustomerID': [1, 2, 5],
    'OrderID': [101, 102, 103],
    'Amount': [500, 300, 700]
})

print("df1:\n", df1)
print("df2:\n", df2)
```

Output:

```
df1:
   CustomerID      Name  Age
0            1     Alice  25
1            2       Bob  30
2            3  Charlie  35
3            4    David  40

df2:
   CustomerID  OrderID  Amount
0            1      101     500
1            2      102     300
2            5      103     700
```

4 Descriptive Statistics - Understanding Your Data

Once you have your combined DataFrame, you'll want to understand the data better. Pandas provides a handy function called `.describe()` that gives you a summary of the numerical columns.

For each numerical column, `.describe()` calculates:

- **Count:** The number of non-missing values.
- **Mean:** The average value.
- **Standard Deviation:** A measure of how spread out the values are.
- **Minimum:** The smallest value.
- **25th Percentile (Q1):** The value below which 25% of the data falls.
- **50th Percentile (Q2):** The median value.
- **75th Percentile (Q3):** The value below which 75% of the data falls.
- **Maximum:** The largest value.

This information helps you quickly get a sense of the distribution and range of your data.

5 Visualizing Your Data

Pandas also integrates well with plotting libraries like Matplotlib and Seaborn. You can use the `.plot()` function to create various types of charts directly from your DataFrame, to see how the data changes over the dataset.'

```
#if df is our dataframe we can plot a graph like this
df.plot()
```

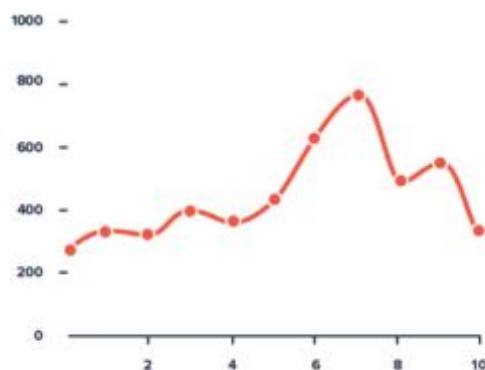


Figure 3: Graph by using `df.plot()`.

6 Handling Missing Data

Real-world data is often messy and contains missing values. Pandas provides tools for handling these “NaN” values.

- **Identifying Missing Data:** You’ll see “NaN” in your DataFrame where data is missing.
- **Filling Missing Data:** You can fill “NaN” values with different strategies:
 - **Filling with a Constant Value:** Replace all “NaN”s with a specific value, like 0 or the average.
 - **Filling with the Mean/Median/Mode:** Replace “NaN”s with the mean, median, or mode of the column.
 - **Forward Fill:** Replace “NaN”s with the value from the previous row.
 - **Backward Fill:** Replace “NaN”s with the value from the next row.
 - **Interpolation:** Estimate missing values based on the surrounding data. You could take the average of the rows before and after the NaN values, or use statistical techniques to estimate the data.
 - **Drop NA:** Remove all of the rows with NaN values.
- **Dropping Missing Data:** If you have a lot of missing data in a particular row, you can simply remove those rows from your DataFrame.

The choice of which method to use depends on the nature of your data and the analysis you’re trying to perform. Be careful when filling missing data, as you don’t want to introduce bias or distort your results.

Code From Class

```
import pandas as pd
import numpy as np

# Sample DataFrame with NaN values
data = {
    'item_id': [101, 102, 103, 104, 105],
    'item': ['Apple', 'Banana', 'Orange', np.nan, 'Grapes'],
    'price': [50, np.nan, 30, 40, np.nan],
    'sales': [200, 150, np.nan, 180, 220]
}

df = pd.DataFrame(data)

print("Original DataFrame with NaN values:")
print(df)

# 1. Fill NaN with a specific value
df1 = df.fillna(0)

# 2. Fill NaN in a specific column with a fixed value
df2 = df.copy()
```

```

df2['price'] = df2['price'].fillna(df2['price'].mean()) #  

    Filling NaN with mean value  
  

# 3. Fill NaN using forward fill (propagate last valid value  

#      forward)  

df3 = df.fillna()  
  

# 4. Fill NaN using backward fill (propagate next valid value  

#      backward)  

df4 = df.bfill()  
  

# 5. Fill NaN with the median value of the column  

df5 = df.copy()  

df5['price'] = df5['price'].fillna(df5['price'].median())  
  

# 6. Fill NaN with the mode (most frequent value) of a column  

df6 = df.copy()  

df6['item'] = df6['item'].fillna(df6['item'].mode()[0])  
  

# 7. Fill NaN using interpolation  

df7 = df.copy()  

df7['sales'] = df7['sales'].interpolate()  
  

# 8. Fill NaN using a dictionary of values for each column  

df8 = df.fillna({'item': 'Unknown', 'price': 45, 'sales': 95})  
  

# 9. Drop rows containing NaN values  

df9 = df.dropna()  
  

# Displaying results
print("\nFill\u00d7NaN\u00d7with\u00d70:\n", df1)
print("\nFill\u00d7NaN\u00d7in\u00d7'price'\u00d7with\u00d7mean:\n", df2)
print("\nForward\u00d7Fill\u00d7(ffill):\n", df3)
print("\nBackward\u00d7Fill\u00d7(bfill):\n", df4)
print("\nFill\u00d7NaN\u00d7in\u00d7'price'\u00d7with\u00d7median:\n", df5)
print("\nFill\u00d7NaN\u00d7in\u00d7'item'\u00d7with\u00d7mode:\n", df6)
print("\nInterpolate\u00d7missing\u00d7values\u00d7in\u00d7'sales':\n", df7)
print("\nFill\u00d7NaN\u00d7with\u00d7a\u00d7dictionary:\n", df8)
print("\nDrop\u00d7rows\u00d7with\u00d7NaN:\n", df9)

```

Output:

Original DataFrame with NaN values:
item_id item price sales
0 101 Apple 50.0 200.0
1 102 Banana NaN 150.0
2 103 Orange 30.0 NaN
3 104 NaN 40.0 180.0
4 105 Grapes NaN 220.0

Fill NaN with 0:
item_id item price sales

```
0      101    Apple   50.0  200.0
1      102    Banana   0.0   150.0
2      103    Orange  30.0   0.0
3      104        0   40.0  180.0
4      105    Grapes   0.0  220.0
```

Fill NaN in 'price' with mean:

```
item_id    item  price  sales
0      101    Apple   50.0  200.0
1      102    Banana  40.0  150.0
2      103    Orange  30.0   NaN
3      104        NaN  40.0  180.0
4      105    Grapes  40.0  220.0
```

Forward Fill (ffill):

```
item_id    item  price  sales
0      101    Apple   50.0  200.0
1      102    Banana  50.0  150.0
2      103    Orange  30.0  150.0
3      104    Orange  40.0  180.0
4      105    Grapes  40.0  220.0
```

Backward Fill (bfill):

```
item_id    item  price  sales
0      101    Apple   50.0  200.0
1      102    Banana  30.0  150.0
2      103    Orange  30.0  180.0
3      104    Grapes  40.0  180.0
4      105    Grapes   NaN  220.0
```

Fill NaN in 'price' with median:

```
item_id    item  price  sales
0      101    Apple   50.0  200.0
1      102    Banana  40.0  150.0
2      103    Orange  30.0   NaN
3      104        NaN  40.0  180.0
4      105    Grapes  40.0  220.0
```

Fill NaN in 'item' with mode:

```
item_id    item  price  sales
0      101    Apple   50.0  200.0
1      102    Banana   NaN  150.0
2      103    Orange  30.0   NaN
3      104    Apple   40.0  180.0
4      105    Grapes   NaN  220.0
```

Interpolate missing values in 'sales':

```
item_id    item  price  sales
0      101    Apple   50.0  200.0
1      102    Banana   NaN  150.0
2      103    Orange  30.0  165.0
```

```

3      104      NaN    40.0   180.0
4      105  Grapes      NaN   220.0

Fill NaN with a dictionary:
  item_id      item  price  sales
0      101    Apple  50.0  200.0
1      102  Banana  45.0  150.0
2      103  Orange  30.0   95.0
3      104  Unknown  40.0   180.0
4      105  Grapes  45.0   220.0

Drop rows with NaN:
  item_id      item  price  sales
0      101    Apple  50.0  200.0

```

7 Loading Data from CSV Files

Instead of defining a dataframe using the `pd.DataFrame()` command, you can also load data from your computer directly. CSV, or comma-separated values, is a common file format for storing tabular data, such as spreadsheets or databases.

You can upload the CSV files using Google Colab. After the data is in Colab, use the `read_csv` command to read the files.

```

import pandas as pd
df = pd.read_csv("name_of_your_data_file.csv")
print(df)

```

8 Bringing It All Together - A Case Study

Let's revisit our online store example. You now have the product data and sales data combined into a single DataFrame. You've cleaned up any missing values and calculated descriptive statistics.

Now you can start answering interesting questions:

- What is the average price of products sold in each quantity range?
- Which brands have the highest sales?
- Is there a correlation between price and quantity sold?
- Which industry had the fastest growth adoption rate over any consecutive three-year period?

By using Pandas to manipulate and analyze your data, you can gain valuable insights that can help you improve your business.

Conclusion

Pandas is an indispensable tool for anyone working with data in Python. By understanding the concepts we've covered in this book, you'll be well-equipped to tackle a wide range of data analysis tasks. So, get out there, experiment with your own datasets, and discover the power of Pandas! Good luck!

Minor in AI

Batch 04 – Notes

28 Feb 2025

Title: Mastering Data Visualization with Matplotlib: From Basics to Stunning Plots!

Syllabus:

- Understand the Basics of Matplotlib
- Import Matplotlib and set up a basic plotting environment.
- Create Basic Plots
- Generate line plots using plt.plot().

Matplotlib is a powerful Python library for data visualization, widely used for creating static, animated, and interactive plots. It provides a variety of chart types, including line graphs, bar charts, histograms, and scatter plots, making it ideal for exploratory data analysis. Matplotlib integrates well with libraries like NumPy and pandas, allowing seamless plotting of structured data. It offers extensive customization options, enabling users to modify colors, labels, and styles. The library supports multiple backends, making it versatile for different environments. With its ease of use and detailed control over graphical representations, Matplotlib is a fundamental tool for data scientists and analysts.

Note: Please refer to class video for more insights

Sample Program to generate a line plot:

Go to a weather site and collect the data of temperatures for a week or hour wise for a day and then plot the graph.

```
import matplotlib.pyplot as plt

# Sample weather data (replace with your actual data)
days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
temperatures = [25, 28, 25, 26, 27, 29, 30]

# Create the line plot
plt.plot(days, temperatures)

# Add labels and title
plt.xlabel("Day of the week")
plt.ylabel("Temperature (°C)")
plt.title("Weekly Temperature Trend")

# Display the plot
plt.show()
```

Read data from spread sheet and load to pandas and print

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_excel('sh-data.xlsx', sheet_name='Sheet1')
plt.bar(df['Student Name'], df['Height'])
plt.xlabel('Student Name')
plt.ylabel('Height')
plt.title('Student Heights')
plt.show()
```

Comparison Graph:

```
import matplotlib.pyplot as plt

# Data: Days of the week
days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]

# Data: Code written (in lines) & Bugs found
code_written = [50, 120, 200, 180, 300, 50, 20] # Lines of Code
bugs_found = [5, 10, 20, 25, 40, 5, 1] # Number of bugs

# Create the plot
plt.figure(figsize=(8, 5))

# Plot Code Written
plt.plot(days, code_written, linestyle='-', color='b', label="Code Written (LOC)")

# Plot Bugs Found
plt.plot(days, bugs_found, linestyle='--', color='r', label="Bugs Found")

# Add labels, title, and legend
plt.xlabel("Days of the Week")
plt.ylabel("Lines of Code / Bugs Found")
plt.title("Code Written vs. Bugs Found Over a Week")
plt.legend() # Show legend
plt.grid(True, linestyle="--", alpha=0.6)

# Show the plot
plt.show()
```

Matplotlib : Data Visualization

Minor in AI, IIT Ropar

28th Feb, 2025

1 Background

Working as intern for a tech startup, we are helping them visualize the weekly data on code production and bug detection. We want to gain insights into development process and identify potential areas for improvement.

2 Objective

To create an informative and visually appealing representation of the startup's weekly coding and debugging data using Matplotlib, a powerful Python library for data visualization.

3 Implementation

3.1 Step 1: Data Collection and Preparation

The startup provided us with the following data for a week:

- Days of the week: Monday to Sunday
- Lines of code written each day: [50, 120, 200, 180, 300, 50, 20]
- Bugs found each day: [5, 10, 20, 15, 30, 5, 1]

3.2 Step 2: Setting Up the Environment

We begin by importing the necessary libraries:

```
import matplotlib.pyplot as plt
```

3.3 Step 3: Creating the Visualization

We use Matplotlib to create a line plot that compares the lines of code written to the bugs found:

```
import matplotlib.pyplot as plt

# Data: Days of the week
days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]

# Data: Code written (in lines) & Bugs found
code_written = [50, 120, 200, 180, 300, 50, 20] # Lines of Code
bugs_found = [5, 10, 20, 15, 30, 5, 1] # Number of bugs

# Create the plot
plt.figure(figsize=(8, 5))
```

```

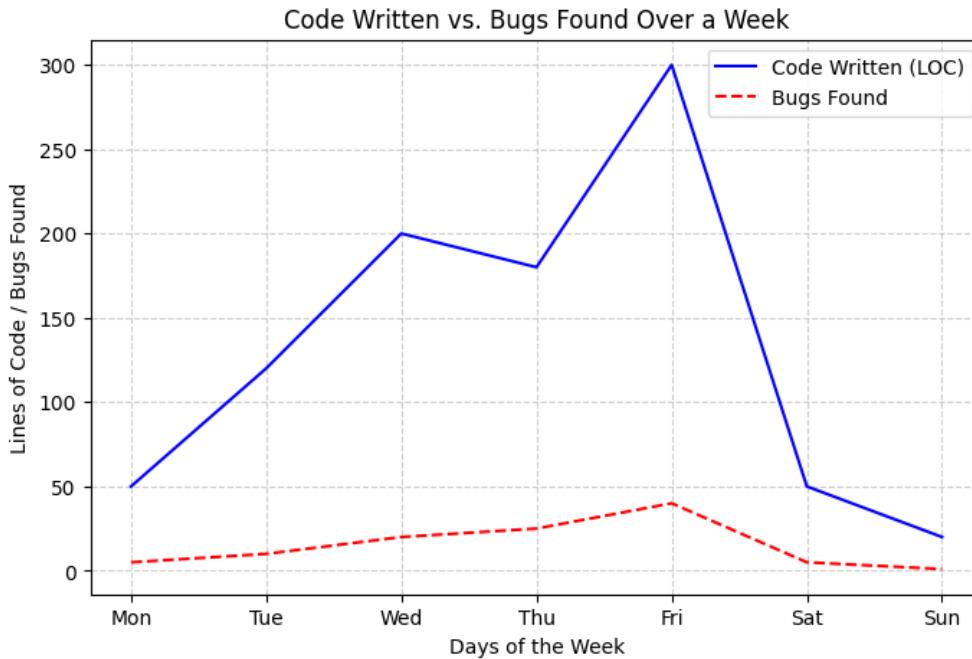
# Plot Code Written
plt.plot(days, code_written, linestyle='--', color='b', label="Code Written (LOC)")

# Plot Bugs Found
plt.plot(days, bugs_found, linestyle='--', color='r', label="Bugs Found")

# Add labels, title, and legend
plt.xlabel("Days of the Week")
plt.ylabel("Lines of Code / Bugs Found")
plt.title("Code Written vs. Bugs Found Over a Week")
plt.legend() # Show legend
plt.grid(True, linestyle="--", alpha=0.6)

# Show the plot
plt.show()

```



Upon examining the plot, we observe:

- A peak in code production on Friday (300 lines)
- A corresponding peak in bugs found on Friday (30 bugs)
- Lower code production and bug detection on weekends

3.4 Step 5: Bugs not found

Let's suppose, `bugs_found = [5, 10, 20, None, 30, 5, 1]`
i.e., for Thursday we didn't have Bugs data then,

To handle the missing data for Thursday:

- We could calculate the mean of the available bug counts, excluding the outlier (Friday's count).
- Alternatively, we could use interpolation based on surrounding days' data.

3.5 Step 6: Saving the Visualization

To preserve the visualization for future reference:

```
plt.savefig('weekly_code_bugs_report.png', dpi=300, bbox_inches='tight')
```

4 Matplotlib Basics

Matplotlib is a powerful Python library for creating static, animated, and interactive visualizations. It provides a MATLAB-like interface for creating plots and figures.

Key features:

- Wide variety of plot types
- High degree of customization
- Integration with NumPy and Pandas
- Support for multiple output formats (PNG, PDF, SVG, etc.)

5 Basic Plotting

The most common way to use Matplotlib is through its pyplot interface:

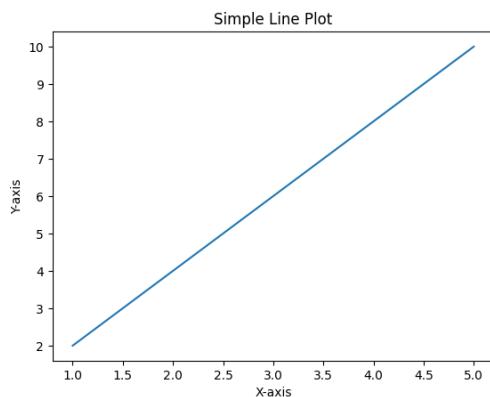
```
import matplotlib.pyplot as plt

# Create data
x = [1,2,3,4,5]
y = [2,4,6,8,10]

# Create line plot
plt.plot(x, y)

# Add labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Simple Line Plot')

# Display the plot
plt.show()
```



6 Plot Types

Matplotlib supports many types of plots:

- Line plots: `plt.plot(x, y)`
- Scatter plots: `plt.scatter(x, y)`
- Bar plots: `plt.bar(x, y)`
- Histograms: `plt.hist(x)`
- Box plots: `plt.boxplot(x)`
- Pie charts: `plt.pie(x)`

7 Customizing Plots

Plots can be customized in various ways:

- Colors: `plt.plot(x, y, color='red')`
- Line styles: `plt.plot(x, y, linestyle='--')`
- Markers: `plt.plot(x, y, marker='o')`
- Axis limits: `plt.xlim(0, 10), plt.ylim(0, 20)`
- Legends: `plt.legend(['Data 1', 'Data 2'])`
- Grid lines: `plt.grid(True)`

8 Weekly Temperature Trend

Take weekly data on temperature of your area from any website and put it in a python list. // We display it using matplotlib and observe the trend.

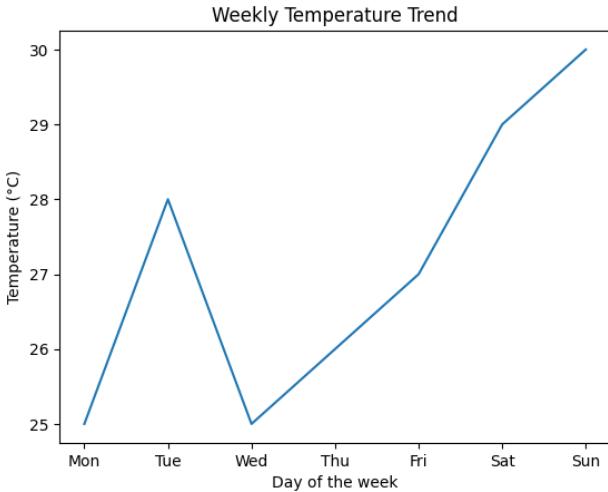
```
import matplotlib.pyplot as plt

# Sample weather data (replace with your actual data)
days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
temperatures = [25, 28, 25, 26, 27, 29, 30]

# Create the line plot
plt.plot(days, temperatures)

# Add labels and title
plt.xlabel("Day of the week")
plt.ylabel("Temperature (°C)")
plt.title("Weekly Temperature Trend")

# Display the plot
plt.show()
```



9 Interactive Plotting

9.1 ECG Signal Graph

Let's suppose you are a researcher studying the topic of heart health and analyzing vast data on ECG signal. Clearly, the ECG signal data in table format will be very bland. Therefore, you can take that data and transform it into a interactive and much easier to understand visualization(just like below one).

Try changing the sliders of Heart Rate and Amplitude.

```
import numpy as np
import matplotlib.pyplot as plt
import ipywidgets as widgets
from IPython.display import display

# Generate x values (time)
x = np.linspace(0, 2, 500)

# Function to create a simulated ECG signal
def ecg_signal(x, heart_rate, amplitude):
    p_wave = np.sin(10 * np.pi * x) * np.exp(-3 * (x - 0.5)**2) # P wave
    qrs_complex = np.sin(30 * np.pi * x) * np.exp(-100 * (x - 1)**2) # QRS
    t_wave = np.sin(10 * np.pi * x) * np.exp(-10 * (x - 1.5)**2) # T wave
    return amplitude * (p_wave + qrs_complex + t_wave) * np.sin(heart_rate * np.pi * x)

# Function to update plot
def update_plot(heart_rate, amplitude):
    y = ecg_signal(x, heart_rate, amplitude)
    plt.figure(figsize=(10, 4))
    plt.plot(x, y, color='red', linewidth=2)

    # Labels and grid
    plt.title("Interactive ECG Signal")
    plt.xlabel("Time (s)")
    plt.ylabel("Voltage (mV)")
    plt.axhline(0, color='black', linewidth=0.5, linestyle="--")
    plt.grid(True, linestyle='--', alpha=0.5)
```

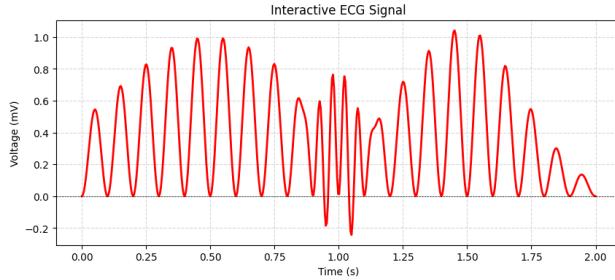
```

plt.show()

# Create interactive sliders
heart_rate_slider = widgets.FloatSlider(min=5, max=20, step=0.5, value=10, description="Heart Rate")
amplitude_slider = widgets.FloatSlider(min=0.5, max=2.0, step=0.1, value=1.0, description="Amplitude")

# Display interactive plot
display(widgets.interactive(update_plot, heart_rate=heart_rate_slider, amplitude=amplitude_slider))

```



Don't worry you don't need to understand the above code fully, just try to understand how things are working from a top level(like how function is called). Revise sin, cos, exponential functions and how to use them using numpy library.

9.2 Simple Linear Function

Earlier we plotted a simple line plot, we can also make it interactive by creating a simple linear function and adding sliders for slope and intercept(if these terms seems foreign to you, revise different equations of 2 D line, one of them is $y = mx + c$, where m = slope, c = intercept).

Try changing sliders for Slope and Intercept.

```

import numpy as np
import matplotlib.pyplot as plt
import ipywidgets as widgets
from IPython.display import display

# Function to plot a straight line
def plot_line(slope, intercept):
    x = np.linspace(-10, 10, 100)
    y = slope * x + intercept

    plt.figure(figsize=(6, 4))
    plt.plot(x, y, color='blue', linewidth=2, label=f"y = {slope}x + {intercept}")

    # Labels and grid
    plt.title("Simple Linear Function")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.axhline(0, color='black', linewidth=0.5, linestyle="--")
    plt.axvline(0, color='black', linewidth=0.5, linestyle="--")
    plt.legend()
    plt.grid(True, linestyle='--', alpha=0.5)

    plt.show()

# Sliders for slope and intercept
slope_slider = widgets.FloatSlider(min=-5, max=5, step=0.5, value=1, description="Slope")

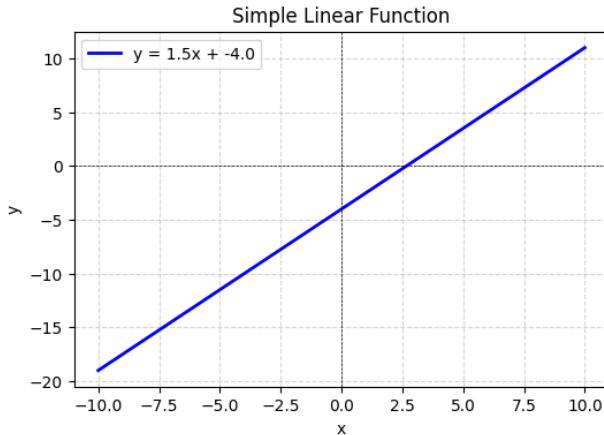
```

```

intercept_slider = widgets.FloatSlider(min=-10, max=10, step=1, value=0, description="Intercept")

# Display interactive plot
display(widgets.interactive(plot_line, slope=slope_slider, intercept=intercept_slider))

```



In the above code, try tweaking a few things and understand the `plot_line(slope, intercept)` function.

9.3 Simple Linear Classifier

Below is a simple linear classifier. Its basically a line plotted in b/w the data points in a way that it forms 2 groups of data (let's say class 0 and class 1). You can adjust the sliders for slope and intercept to see how the data points in class 0 and class 1 gets changed.

This isn't a machine learning model(what's this?? will be covered in later sessions) though, as we are here just segregating the data points. A real classifier would automatically learn the best slope and intercept from the training data.(This will be covered in later sessions too, so don't worry if you don't understand this)

```

import numpy as np
import matplotlib.pyplot as plt
import ipywidgets as widgets
from IPython.display import display

# Generate random data points
np.random.seed(0)
X = np.random.rand(100, 2) # 100 points in 2D
y = (X[:, 0] + X[:, 1] > 0).astype(int) # Simple linear boundary: x + y > 0

# Function to plot decision boundary
def plot_classifier(slope, intercept):
    plt.figure(figsize=(6, 5))

    # Plot data points
    plt.scatter(X[y == 0][:, 0], X[y == 0][:, 1], color='red', label="Class 0")
    plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='blue', label="Class 1")

    # Plot decision boundary
    x_vals = np.linspace(-3, 3, 100)
    y_vals = -slope * x_vals - intercept # Line equation: y = -mx - b
    plt.plot(x_vals, y_vals, 'green', linewidth=2, label="Decision Boundary")

```

```

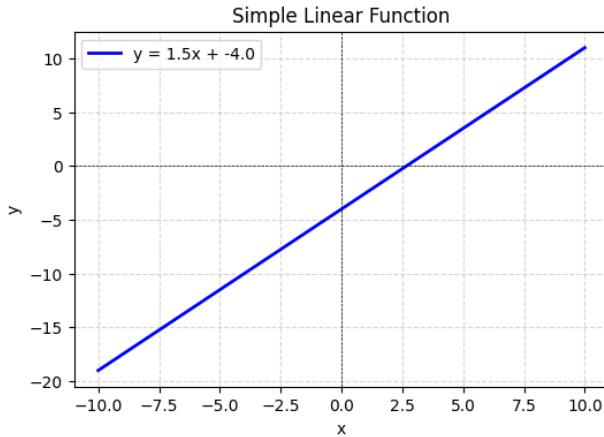
# Labels and grid
plt.title("Simple Linear Classifier")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.axhline(0, color='black', linewidth=0.5, linestyle="--")
plt.axvline(0, color='black', linewidth=0.5, linestyle="--")
plt.legend()
plt.grid(True, linestyle='--', alpha=0.5)

plt.show()

# Sliders for slope and intercept of the decision boundary
slope_slider = widgets.FloatSlider(min=-3, max=3, step=0.1, value=1, description="Slope")
intercept_slider = widgets.FloatSlider(min=-3, max=3, step=0.1, value=0, description="Intercept")

# Display interactive plot
display(widgets.interactive(plot_classifier, slope=slope_slider, intercept=intercept_slider))

```



10 Distribution of Student Heights

For visualizing the student heights using matplotlib library, we performed the following steps:-

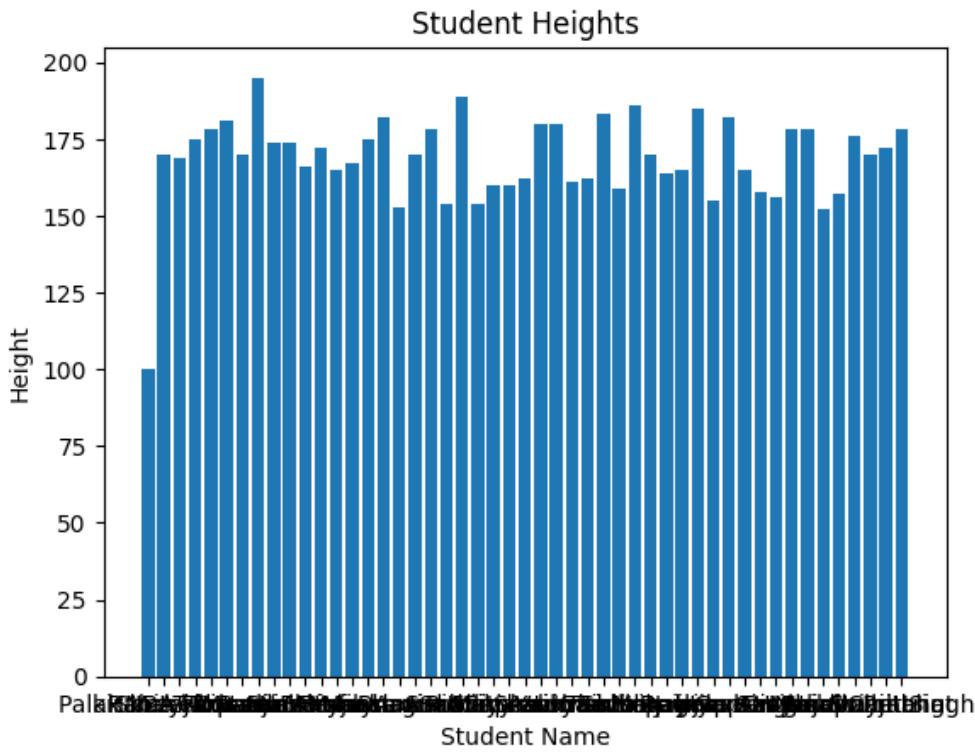
- Uploaded the student-height.xlsx file to colab(upload option in file icon on left pane).
- when creating pandas dataframe(df), used, pd.read_excel('student-height.xlsx', sheet_name='Sheet1')
- Plotted bar chart using the 2 columns of the dataframe.

```

import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_excel('student-height.xlsx', sheet_name='Sheet1')
plt.bar(df['Student Name'], df['Height'])
plt.xlabel("Student Name")
plt.ylabel("Height")
plt.title("Student Heights")
plt.show()

```



Some problems with this graph:

- The x labels(student names) are all mashed together
 - The heights in this bar chart are for each and every student, maybe there's a better way to visualize it.

We could use normal distribution instead or, even histogram. But it comes at a cost which is that we wouldn't be able to figure out a specific student's height by just looking at the plot in that case.

So, there's always drawbacks and other things we should keep in mind before choosing a plot for best visualization.

This will be covered more in later sessions.

11 Best Practices

- Keep plots simple and focused
 - Use appropriate plot types for your data
 - Label axes and include titles
 - Use color effectively but not excessively
 - Consider colorblind-friendly palettes
 - Ensure text is readable (font size, contrast)
 - Use meaningful scales and units

Seeing Data Differently

From Mumbai Trains to AI Brains



Figure 1: Mumbai Local Train Map

1 Why Visualization Matters

Lost in Mumbai?

Once upon a bustling morning in Mumbai, a young traveller named Asha arrived in the city, brimming with excitement and curiosity. As she stepped into the vibrant chaos of the local station, she was greeted by a surprising dilemma: two very different guides to navigate the sprawling 1000+ km rail network.

In one hand, she held a thick, 10-page document crammed with station names, connections, and schedules—a maze of text that seemed to promise confusion at every turn. On the other, she saw a single, colourful map, where clear lines, intuitive symbols, and neatly labelled routes turned complexity into a visual masterpiece.

Fascinated, Asha chose the map. With a few moments of study, the overwhelming network transformed into a series of simple, interconnected paths. The once intimidating maze of data became a friendly guide, leading her effortlessly to her destination. This experience eased her journey and sparked a realization: the art of visualization can turn chaos into clarity, making even the most complex information accessible and engaging.

Which would you choose? This real-world example shows how visualization turns chaos into clarity.

2 From Problems to Solutions

2.1 The Data Challenge

Why raw numbers fail us:

- **Overload:** Mumbai's rail network spans 150+ stations which is impossible to memorize
- **Blind Spots:** Text can't show spatial relationships between stations
- **Time Sink:** Analysts spend hours explaining what a map shows instantly

2.2 Visualization Superpowers

Case Studies

1. Mumbai Rail Map:

- Textual: 10+ pages with 95% redundancy
- Visual: Single A3 sheet

2. AI Sentiment Analysis:

- Numbers: "Model A scored 0.87, Model B 0.63"
- Visual: Side-by-side color-coded bars (green=good, red=bad)

3 Become a Visualization Wizard

3.1 Tools of the Trade

Listing 1: Sample Code for Line Plot

```

1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import numpy as np
4
5 # Generate sample data (Use Real Data If available)
6 dates = pd.date_range(start='2023-01-01', end='2023-01-31')
7 steps = np.random.randint(3000, 12000, size=len(dates))
8 data = pd.DataFrame({'Date': dates, 'Steps': steps})
9
10 # Create line plot
11 plt.figure(figsize=(12, 6))
12 plt.plot(data['Date'], data['Steps'], marker='o', linestyle='--', color='blue')
13 plt.title('Daily Step Count - January 2023', fontsize=16)
14 plt.xlabel('Date', fontsize=12)
15 plt.ylabel('Steps', fontsize=12)
16 plt.grid(True, linestyle='--', alpha=0.7)
17 plt.xticks(rotation=45)
18 plt.tight_layout()
19 plt.show()
```

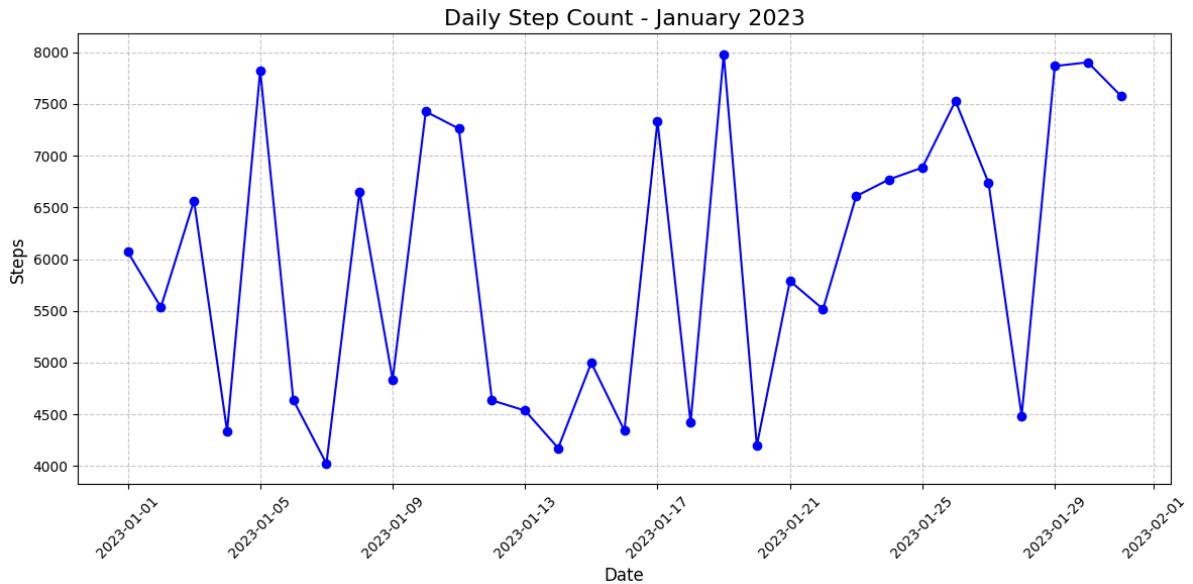


Figure 2: A line plot showing daily step patterns. Notice the weekend drops!

3.2 Code Explanation

Importing Libraries

- **matplotlib.pyplot**: Used for plotting the line graph.
- **pandas**: Handles and manipulates data.
- **numpy**: Generates random step counts.

Generating Sample Data

- The date range is generated as:

```
dates = pd.date_range(start='2023-01-01', end='2023-01-31')
```

which creates a series of dates from January 1 to January 31, 2023.

- The step count values are randomly chosen in the range:

```
steps = np.random.randint(3000, 12000, size=len(dates))
```

where each step count lies between 3,000 and 12,000.

- A Pandas DataFrame is created as:

```
data = pd.DataFrame({'Date': dates, 'Steps': steps})
```

containing two columns: "Date" and "Steps".

Creating the Line Plot

- The figure size is set to:

```
plt.figure(figsize=(12, 6))
```

- The line plot is created using:

```
plt.plot(data['Date'], data['Steps'], marker='o', linestyle='-', color='blue')
```

where:

- x -axis represents the dates.
- y -axis represents the step count.
- Circular markers (o) are used for individual points.
- A solid line ($-$) is drawn between points.

- Labels and titles are set as follows:

```
plt.title('Daily Step Count - January 2023', fontsize=16)
```

```
plt.xlabel('Date', fontsize=12), plt.ylabel('Steps', fontsize=12)
```

- A grid is enabled for better visualization:

```
plt.grid(True, linestyle='-', alpha=0.7)
```

- X-axis labels are rotated:

```
plt.xticks(rotation=45)
```

- Finally, the layout is adjusted and the plot is displayed:

```
plt.tight_layout()
```

```
plt.show()
```

3.3 Advanced Techniques

- Histogram (Data Distribution):** A histogram groups numerical data into bins, offering a clear view of the data's distribution and highlighting patterns like skewness or the presence of multiple modes.
- Scatter Plot (Relationships):** A scatter plot visualizes the relationship between two variables, making it easier to identify correlations, clusters, or outliers.
- Box Plot (Summary and Outliers):** A box plot provides a statistical summary of data through quartiles and reveals potential outliers, offering insight into data variability.
- Bar Chart (Categorical Comparisons):** A bar chart is ideal for comparing quantities across different categories, enabling quick visual comparisons.

4 Why AI Needs Visualization

- **Enhanced Clarity:** Visualizations break down complex datasets into understandable segments.
- **Faster Insights:** Patterns and trends are easier to identify visually.
- **Improved Decision-Making:** Clear visuals support better analysis and informed decisions.

Real-World AI Example

Sentiment Analysis: An AI model analyzed 10,000 product reviews. The visualization below compares two approaches:

- **Text-Only Report:** 15-page document with scores
- **Visual Dashboard:** Interactive map showing regional sentiment clusters

Result: Stakeholders using the visual report made decisions 4x faster!

5 Key Takeaways

Data visualization bridges the gap between raw data and actionable insights. By mastering these techniques, you empower yourself to communicate complex ideas clearly and make data-driven decisions effectively. A well-crafted visualization not only enhances understanding but also sparks curiosity and engagement.

Key Takeaways

- Visual representations simplify complex data and highlight important trends.
- Hands-on practice, such as coding with Python, is crucial for mastering visualization techniques.
- Always consider the context and audience when designing your visualizations.
- Effective visualizations can transform overwhelming datasets into clear, actionable insights.
- Know Your Audience: Tourists need maps, engineers need schematics
- Start Simple: Basic line plots → histograms → interactive dashboards
- Color Wisely: Use palettes like `viridis` for accessibility
- Test Early: Show drafts to colleagues - if they're confused, iterate!

Minor in AI – Batch 04

03 March 2025

Title: Mastering Data Visualization with Matplotlib: From Basics to Stunning Plots

Topics:

- Customize axes, titles, and labels to enhance readability
- Work with Different Plot Types
- Create bar charts (plt.bar()), scatter plots (plt.scatter()), and histograms (plt.hist()).
- Differentiate between various chart types and their appropriate use cases.

Note:

Below are codes that we types together. For others, refer to the collab link.

Why that is for daily_steps CSV file, a line plot not a good idea?

```
import pandas as pd
import matplotlib.pyplot as plt

# Read CSV file
df = pd.read_csv("daily_steps.csv")

# Create the line plot
plt.plot(df['Person_ID'], df['Daily_Steps'])

# Add labels and title
plt.xlabel("Person ID")
plt.ylabel("Daily Steps")
plt.title("Daily Steps of Person")

# Display the plot
plt.show()
```

Instead we can go for a histogram.

```
import pandas as pd
import matplotlib.pyplot as plt

# Read CSV file
df = pd.read_csv("daily_steps.csv")
```

```

# Plot histogram
plt.hist(df["Daily_Steps"], bins=10, edgecolor="black")
plt.xlabel("Daily Steps")
plt.ylabel("Frequency")
plt.title("Distribution of Daily Steps Among People")
plt.show()

```

Understanding the customizations

```

import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y1 = [2, 4, 6, 8, 10]  # Line 1
y2 = [1, 3, 5, 7, 9]  # Line 2

# Set figure size
plt.figure(figsize=(8, 5))

# Plot with different styles
plt.plot(x, y1, color="blue", marker="o", markersize=8,
         linestyle="--", linewidth=2, label="Dashed Blue")
plt.plot(x, y2, color="red", marker="s", markersize=10,
         linestyle="-.", linewidth=2, label="Dash-dot Red")

# Labels and title
plt.xlabel("X-axis Label", fontsize=12, color="darkblue")
plt.ylabel("Y-axis Label", fontsize=12, color="darkred")
plt.title("Matplotlib Customization Demo", fontsize=14,
          fontweight="bold")

# Grid and legend
plt.grid(True, linestyle="--", alpha=0.5)
plt.legend()

# Show plot
plt.show()

```

Scatter Plots:

```

import pandas as pd
import matplotlib.pyplot as plt

# Sample Data
df = pd.read_csv("daily_steps.csv")

```

```

df["Calories_Burned"] = df["Daily_Steps"] * 0.04 # Assuming
0.04 calories per step

# Scatter Plot
plt.scatter(df["Daily_Steps"], df["Calories_Burned"])
plt.xlabel("Daily Steps")
plt.ylabel("Calories Burned")
plt.title("Daily Steps vs. Calories Burned")
plt.show()

```

Difference Table: (Source: ChatGPT)

Difference Between Line Plot, Histogram, and Scatter Plot

Chart Type	Purpose	When to Use?	Example Use Cases
Line Plot	Shows trends over time	When tracking changes over continuous data (usually time)	Stock prices, temperature changes, heart rate monitoring
Histogram	Shows the distribution of data	When analyzing the frequency of values within a dataset	Exam score distribution, age distribution, website traffic per hour
Scatter Plot	Shows relationships between two variables	When analyzing correlations or patterns between two numerical variables	Study time vs. exam score, height vs. weight, advertising spend vs. sales

Other Case Study used: (Interpreting the Mumbai local train lines)



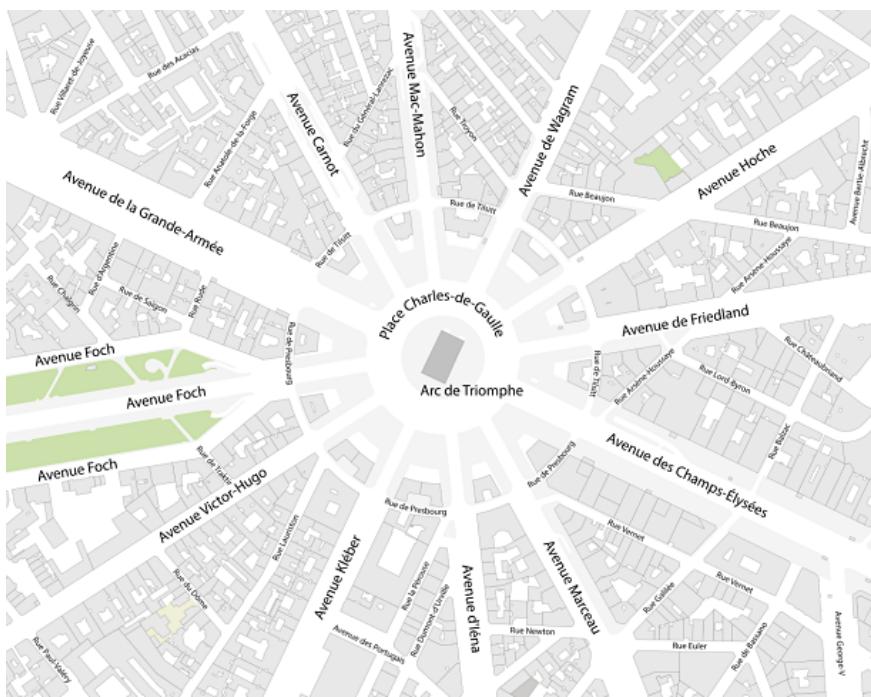
Basics to Stunning Plots in Matplotlib

Minor in AI, IIT Ropar

4th March, 2025

1 Design : We enforce or let it emerge?

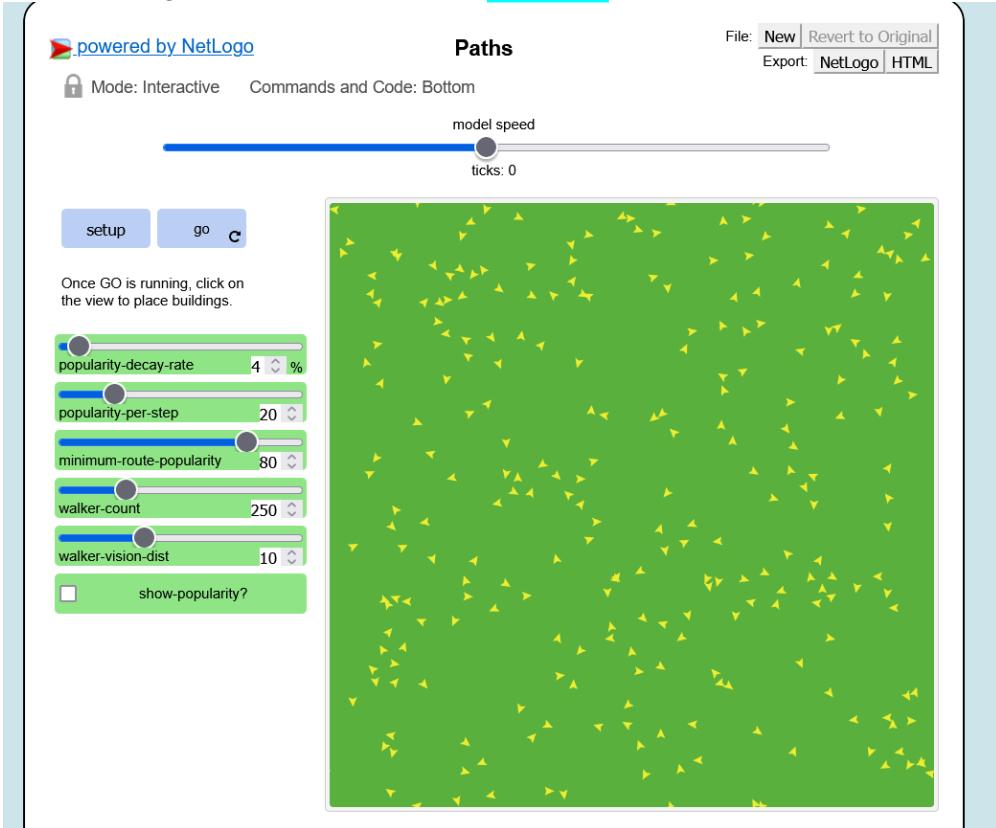
1.1 Place Charles de Gaulle, 12 avenues



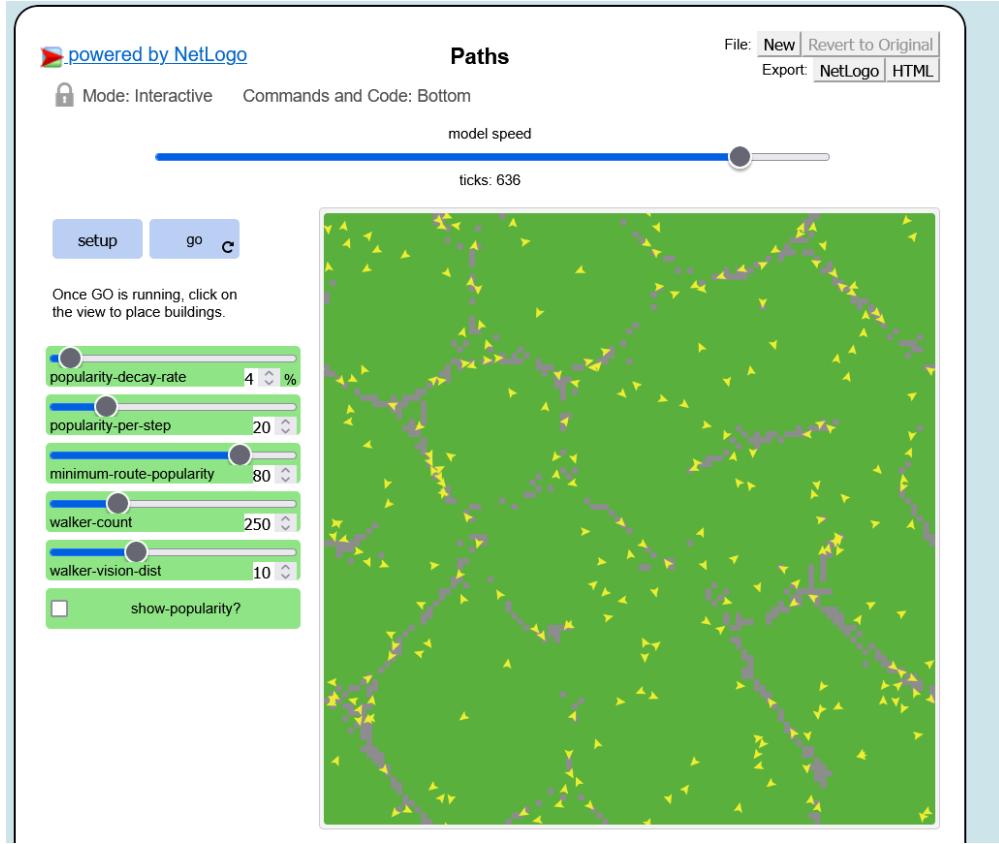
Place Charles de Gaulle is a large road junction in Paris, France, the meeting point of twelve straight avenues, its an example of complex design enforced in city planning.
There are designs that we enforce on people and, sometimes designs happens by themselves in nature.

1.2 Design emerging itself

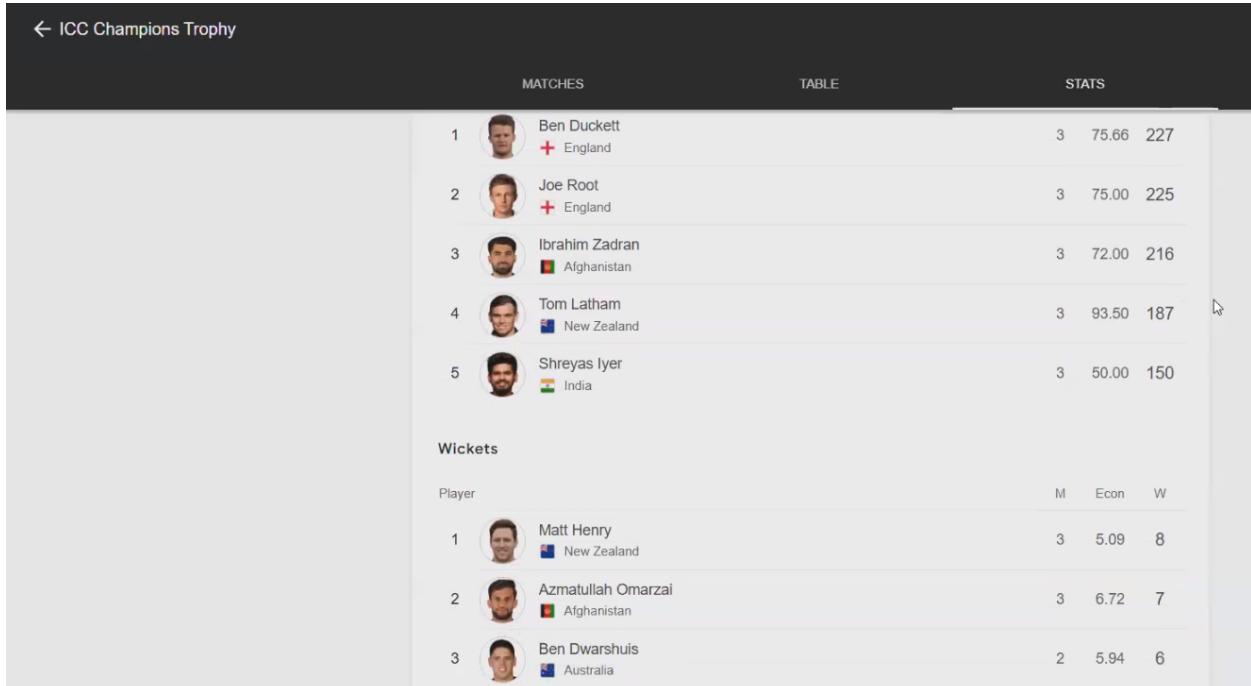
Check out NetLogo Website for simulations. [Click here](#)



After running the simulation by clicking on 1st setup and then on go button. As the agents/people walk in an area. The larger the number of people walk through a certain path or more frequent a certain path is being traversed upon then, its getting highlighted by pink colour in below pic.



2 We don't need plots at all places



Here, the stats can be shown through line graph or, histogram or, other visualizations. But they have chosen to display in this format.

Why?

Because generally if a person wants to check out he would want to see player-wise stats. That's why when visualizing the data we should always know our audience so, that we can serve them the visuals better.

3 Daily Steps vs. Calories burned

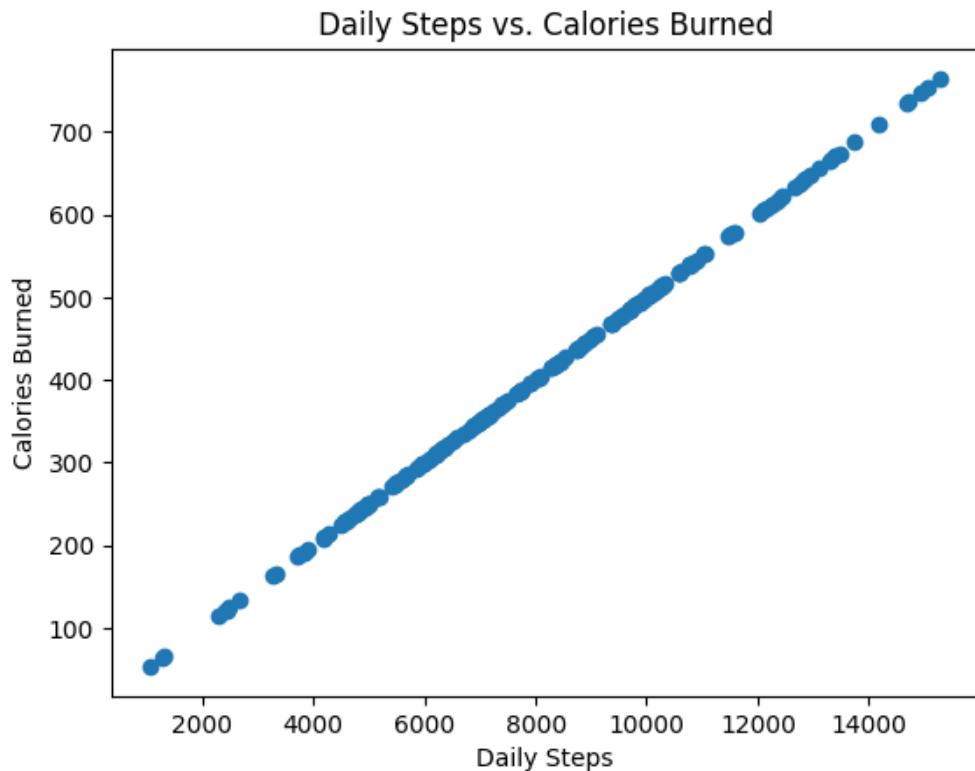
Continuing from last class. We now add another column(attribute) named Calories Burned in the dataframe and drAw a scatter plot to find out the relationship b/w Daily steps and Calories burned.

```
# Calories burned with step counts

import pandas as pd
import matplotlib.pyplot as plt

# Sample Data
df = pd.read_csv("daily_steps.csv")
df['Calories_Burned'] = df['Daily_Steps'] * 0.05

# Plotting
plt.scatter(df['Daily_Steps'], df['Calories_Burned'])
plt.xlabel("Daily Steps")
plt.ylabel("Calories Burned")
plt.title("Daily Steps vs. Calories Burned")
plt.show()
```



In above scatter plot we can see that the graph is linear with positive inclination. i.e. As the Daily steps increase so, does the calories burned. Therefore, There is **positive correlation** b/w Daily Steps and Calories burned.

4 Subplots

We can use subplots to plot multiple subplots in a plot. Play around with the below code to get more familiar.

```
# Program for Subplots
import matplotlib.pyplot as plt

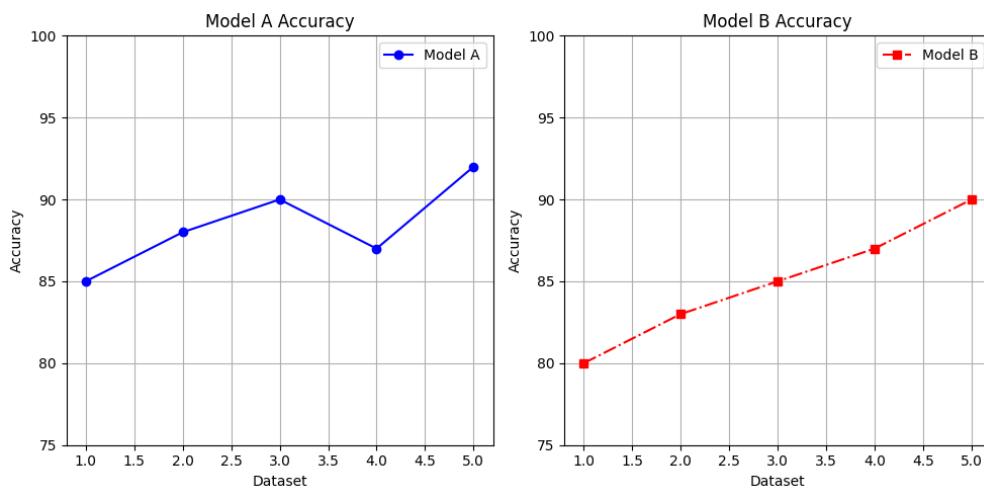
plt.figure(figsize=(10, 5))
datasets = [1, 2, 3, 4, 5]

acc_model_a = [85, 88, 90, 87, 92]
acc_model_b = [80, 83, 85, 87, 90]

plt.subplot(1, 2, 1)
plt.plot(datasets, acc_model_a, marker='o', linestyle='--', color='blue', label='Model A')
plt.xlabel('Dataset')
plt.ylabel('Accuracy')
plt.title('Model A Accuracy')
plt.ylim(75, 100)
plt.grid()
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(datasets, acc_model_b, marker='s', linestyle='-.', color='red', label='Model B')
plt.xlabel('Dataset')
plt.ylabel('Accuracy')
plt.title('Model B Accuracy')
plt.ylim(75, 100)
plt.grid()
plt.legend()

plt.tight_layout()
plt.show()
```



5 Meme Accuracy

If we have a model that can recognise if a given image/content is meme or not. Then we can plot a line graph for depicting meme recognition accuracy for each training cycle as below.

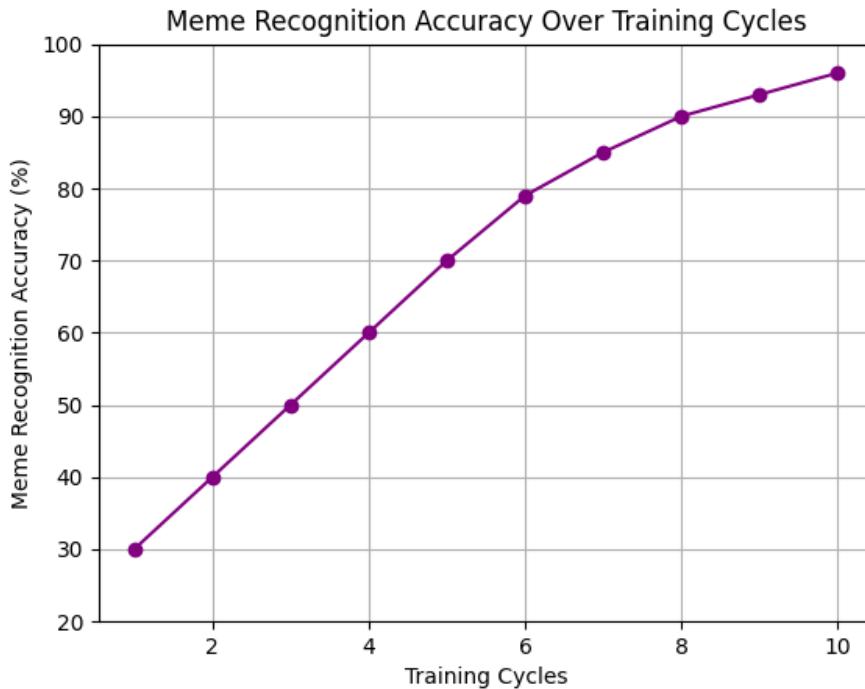
```
# Meme Accuracy
import matplotlib.pyplot as plt

training_cycles = list(range(1, 11))
meme_recognition = [30, 40, 50, 60, 70, 79, 85, 90, 93, 96]

plt.plot(training_cycles, meme_recognition, marker='o', color = 'purple', linestyle = '--')
plt.title('Meme Recognition Accuracy Over Training Cycles')
plt.xlabel('Training Cycles')
plt.ylabel('Meme Recognition Accuracy (%)')
plt.ylim(20, 100)
plt.grid(True)

plt.savefig("meme_accuracy.png", dpi = 100)

plt.show()
```



The graph is depicting an upward trend in meme recognition accuracy as the number of training cycles increases.

6 Curves

`linspace()` function in numpy can be used to create sine and cos plots. Try creating cos plot by examining how sine plot is created in below code.

```

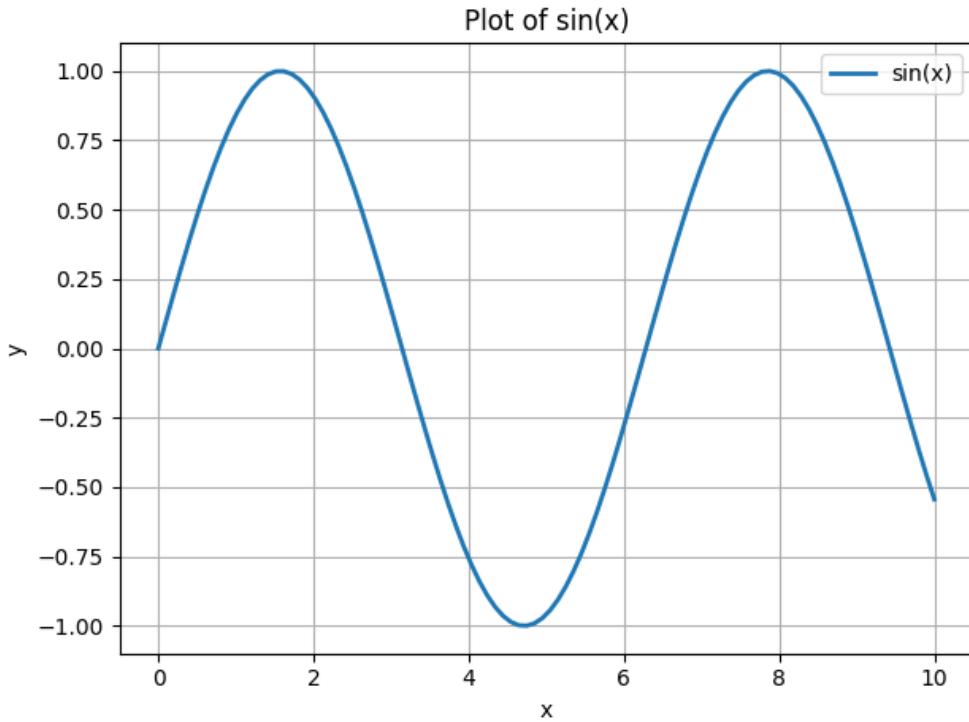
# High resolution and optimizations
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y, label='sin(x)', linewidth=2)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Plot of sin(x)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("sine_plot.pdf", dpi=300)

plt.show()

```



7 AI Learning Journey

Here we are simulating study sessions and different emotions the AI can feel like frustration, confidence, study speed, brain overload like a human.

```

import matplotlib.pyplot as plt
import numpy as np

# Simulated "mood swings" over 20 study sessions
sessions = np.arange(1, 21)
frustration = np.exp(-sessions/5) + np.random.uniform(0, 0.1, len(sessions))

```

```

confidence = 1 - frustration
study_speed = np.exp(-sessions/10) * 0.1
brain_overload = np.abs(np.sin(sessions / 3)) * 0.5 + np.random.uniform(0, 0.05, len(sessions))

# Create 4 subplots
fig, axes = plt.subplots(2, 2, figsize=(10, 8))
fig.suptitle("AI Learning Journey", fontsize=14)

# Frustration Level
axes[0, 0].plot(sessions, frustration, 'r-o', label="Frustration")
axes[0, 0].set_title("How Frustrated AI Feels")
axes[0, 0].set_xlabel("Study Sessions")
axes[0, 0].set_ylabel("Frustration Level")
axes[0, 0].legend()

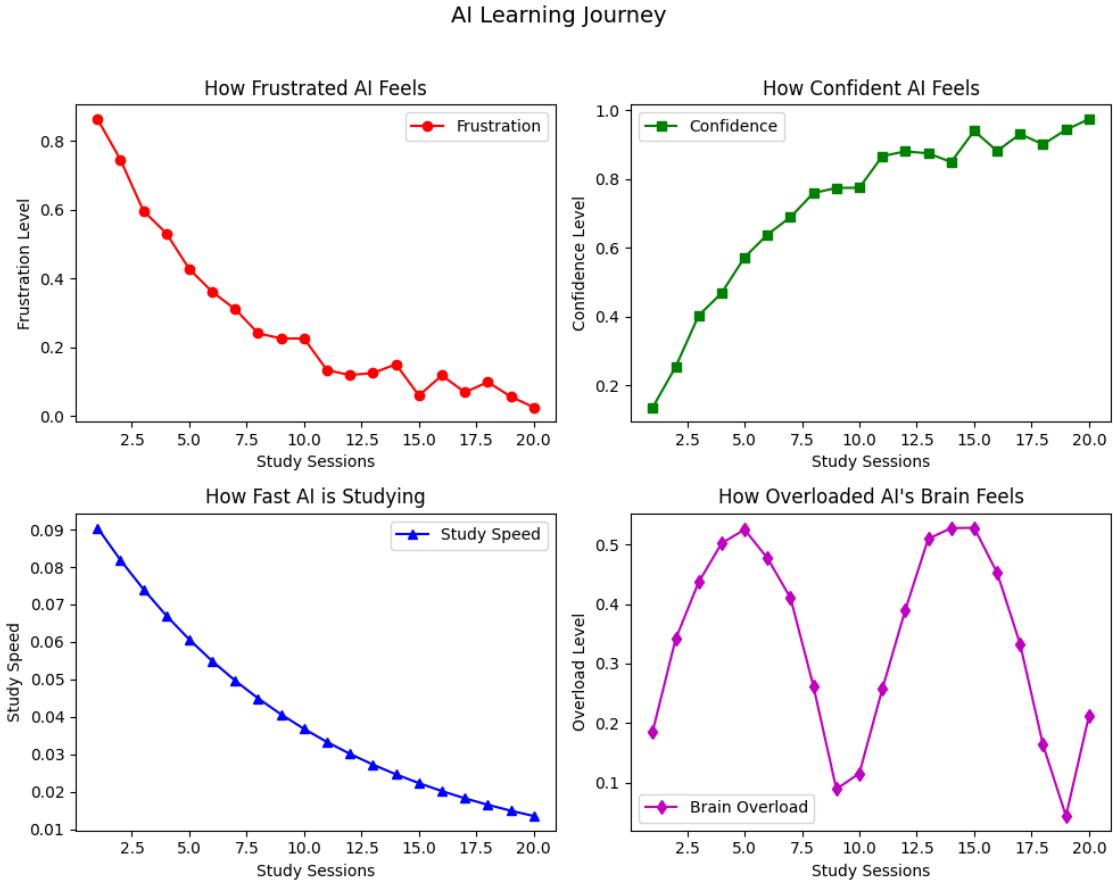
# Confidence Level
axes[0, 1].plot(sessions, confidence, 'g-s', label="Confidence")
axes[0, 1].set_title("How Confident AI Feels")
axes[0, 1].set_xlabel("Study Sessions")
axes[0, 1].set_ylabel("Confidence Level")
axes[0, 1].legend()

# Study Speed
axes[1, 0].plot(sessions, study_speed, 'b-^', label="Study Speed")
axes[1, 0].set_title("How Fast AI is Studying")
axes[1, 0].set_xlabel("Study Sessions")
axes[1, 0].set_ylabel("Study Speed")
axes[1, 0].legend()

# Brain Overload
axes[1, 1].plot(sessions, brain_overload, 'm-d', label="Brain Overload")
axes[1, 1].set_title("How Overloaded AI's Brain Feels")
axes[1, 1].set_xlabel("Study Sessions")
axes[1, 1].set_ylabel("Overload Level")
axes[1, 1].legend()

# Adjust layout
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()

```



8 Correlation doesn't mean Causation

8.1 Ice Cream Sales and Shark Attacks

An Intern was asked to analyze data on ice cream sales and shark attacks over the summer months. He noticed that as ice cream sales go up, shark attacks also increase! To prove his point, he made a line plot connecting the two trends.

"Look! The lines move together! That must mean eating ice cream causes sharks to attack!" He is wrong to conclude that. In this case what we are observing is the as one variable increase another is also increasing, i.e. there is a positive correlation. BUT! that just means for the given data they are positively correlated it DOES NOT mean one variable is causing the other variable to increase or decrease with itself. // The possible reason for both lines to be increasing together is that the season would be summer therefore, people are more likely to eat ice cream and also more likely to go to open waters where sharks are thus, increase in shark attacks.

```
# prompt: An Intern was asked to analyze data on ice cream sales and shark attacks over the summer month
```

```
import matplotlib.pyplot as plt
import numpy as np

# Simulate ice cream sales data
ice_cream_sales = np.random.randint(50, 200, size=10) # 10 data points

# Simulate shark attack data (correlated with ice cream sales)
shark_attacks = ice_cream_sales * 0.1 + np.random.randint(0, 5, size=10) # Introduce some noise
```

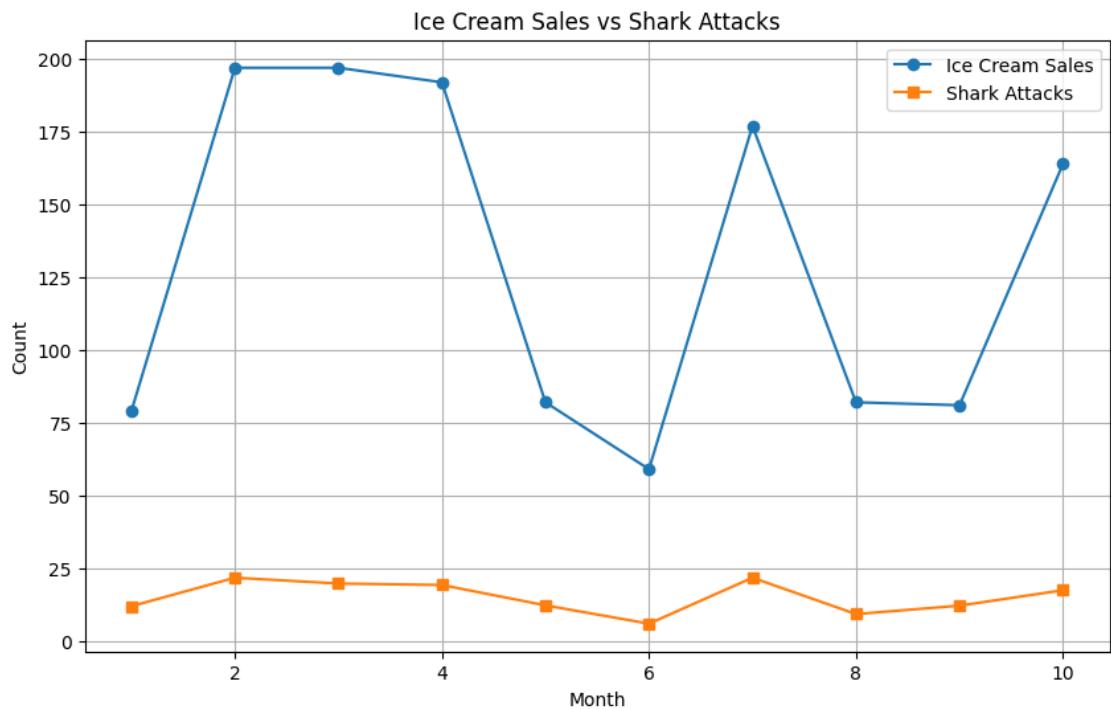
```

# Months
months = range(1, 11)

plt.figure(figsize=(10, 6))
plt.plot(months, ice_cream_sales, label='Ice Cream Sales', marker='o', linestyle='--')
plt.plot(months, shark_attacks, label='Shark Attacks', marker='s', linestyle='--')

plt.xlabel("Month")
plt.ylabel("Count")
plt.title("Ice Cream Sales vs Shark Attacks")
plt.legend()
plt.grid(True)
plt.show()

```



8.2 Concert Ticket Price

Intern was asked to analyze how concert ticket prices impact the number of people attending a concert for different events. He decided to use a histogram to show the relationship. "I made a histogram showing how many concerts had different ticket price ranges! That should explain everything, right?"

Think about this. It will be covered in next session.

Mastering Data Visualization with Seaborn: Case Studies, Categorical Data, and Practical Applications for AI Engineers

Minor In AI, IIT Ropar
5th March, 2025

Welcome!

Welcome, aspiring AI engineers, to the fascinating world of data visualization! In this book, we'll embark on a journey to master the art of transforming raw data into insightful visuals using Seaborn, a powerful Python library. We'll focus on practical applications particularly relevant to your future careers.



Figure 1: Hehe, penguin (look forward for penguin data)

Setting the Stage - Categorical vs. Continuous Data

Imagine you're building an AI model to understand user behavior on an online streaming platform (like Netflix, Hulu, etc). You have a lot of data. What do you do with it? You have *watch time*, *movie genre*, *subscription type*, and *ratings*.

Before we dive into creating visualizations, it's crucial to understand the different *types* of data we'll be working with. This will determine the most appropriate and effective ways to represent it visually.

Specifically, let's consider these distinctions:

- **Categorical Data:** Think of data that falls into distinct categories or groups. These categories are usually limited in number. In our streaming platform example, *movie genre* (action, comedy, drama, thriller) and *subscription type* (free, standard, premium) are examples of categorical data. The number of options are fixed, or at least finite.
- **Continuous Data:** This type of data can take on any value within a given range. It's often numerical and can have a very large (or infinite) number of possible values. *Watch time* (120 minutes, 98.3 minutes, etc.) and *ratings* (0.1 to 10) are prime examples of continuous data in our streaming scenario. There's theoretically a large number of values that the data points can take.

Now, let's ask ourselves a question: What about *years*? For instance, if we're looking at movies released between 1990 and 1995, does "year" become categorical, or continuous?

It Depends!

The key here is *context*.

- If you're only considering movies from 2000 to 2005, then "year" effectively becomes categorical. You have only five distinct categories.
- But if you're analyzing movies spanning from the 1800s to the present day, "year" transforms into continuous data because of the large range of possible values.

Understanding this distinction between categorical and continuous data is crucial because it affects which visualizations are most appropriate and interpretable. For example, using a scatter plot on categorical data often won't provide meaningful insights.

Introducing Seaborn - Visualizing the Penguin Data

Let's switch gears and consider a different dataset: Penguins.

Researchers are often interested in the physical characteristics of penguins, such as the length and depth of their bills (beaks). These measurements can help identify different species of penguins.

To explore this data visually, we'll use Seaborn, a Python library built on top of Matplotlib. Seaborn provides a higher-level interface for creating informative and aesthetically pleasing statistical graphics. It builds on top of Matplotlib, which is a more basic data plotting library.

seaborn comes pre-installed with many basic datasets for learning and playing around with.

Here's how to get started:

```
1 import matplotlib.pyplot as plt
2 import seaborn as sns
3
4 # Load the penguin dataset
5 penguins = sns.load_dataset("penguins")
6
7 # Create a scatter plot
8 sns.scatterplot(data=penguins, x="bill_length_mm", y="bill_depth_mm", hue="species",
9                  style="species")
10
11 plt.show()
```

Listing 1: Creating a scatter plot with the Penguins dataset

Let's break down this code:

1. Import Libraries:

- `import matplotlib.pyplot as plt`: Imports the Matplotlib library and assigns it the alias `plt`. This library provides the foundation for creating plots and visualizations in Python.
- `import seaborn as sns`: Imports the Seaborn library and assigns it the alias `sns`. Seaborn builds on top of Matplotlib and provides a higher-level interface for creating statistical graphics.

2. Load Dataset:

- `penguins = sns.load_dataset("penguins")`: Loads the "penguins" dataset from Seaborn's built-in datasets. This dataset contains information about different species of penguins and their physical characteristics.

3. Create Scatter Plot:

- `sns.scatterplot(data=penguins, x="bill_length_mm", y="bill_depth_mm", hue="species", style="species")`: Creates a scatter plot using Seaborn's `scatterplot` function.
 - `data=penguins`: Specifies the dataset to use for the plot. In this case, it's the "penguins" dataset.
 - `x="bill_length_mm"`: Specifies the column to use for the x-axis of the plot. Here, it's the "bill_length_mm" column, representing the bill length of the penguins.

- `y="bill_depth_mm"`: Specifies the column to use for the y-axis of the plot. It's the `"bill_depth_mm"` column, representing the bill depth of the penguins.
- `hue="species"`: Specifies the column to use for coloring the points in the plot. The points will be colored differently based on the species of the penguins, allowing us to distinguish between them visually.
- `style="species"`: Specifies the column to use for varying the style of the points in the plot. In this case, the style (e.g., shape) of the points will also vary based on the species of the penguins.

4. Display Plot:

- `plt.show()`: Displays the plot that has been created using Matplotlib. This function opens a window or displays the plot inline, depending on the environment (e.g., Jupyter Notebook).

This code generates a scatter plot where each point represents a penguin. The x-axis shows the bill length, the y-axis shows the bill depth, and the color and shape of the points differentiate the penguin species.

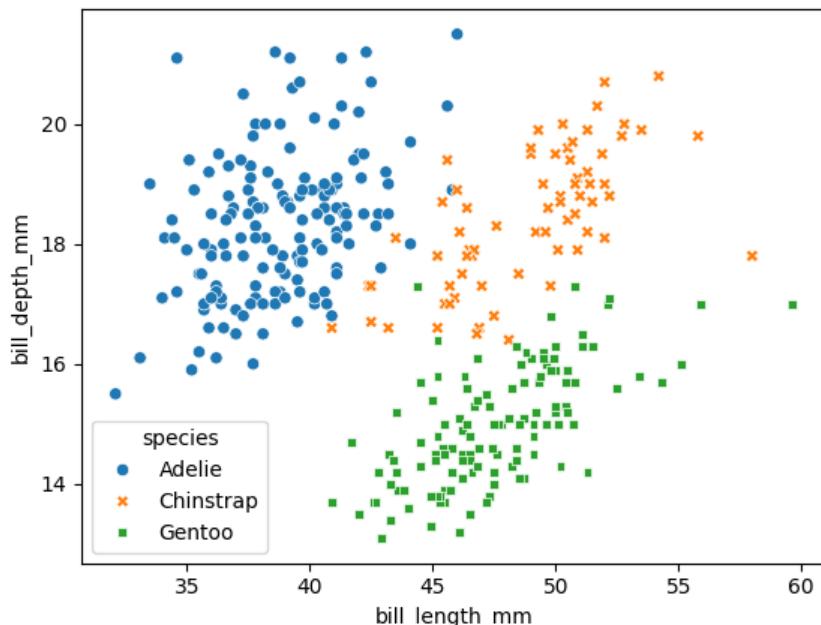


Figure 2: A scatter plot of penguin bill length vs. bill depth, colored by species.

Notice the `hue` and `style` parameters. These are where the *categorical* nature of "species" comes into play. Seaborn uses different colors (hue) and marker styles to visually distinguish the different penguin species.

- **What Happens If You Use Continuous Data for hue?**

If you were to accidentally set `hue` to a continuous variable like "year" (assuming such a column existed), Seaborn would likely issue a warning. The library understands that `hue` is best suited for categorical data to create distinct visual groupings.

Exploring More Seaborn Datasets and Plot Types

Seaborn isn't just limited to penguins! It comes with several built-in datasets that allow you to explore different visualization techniques.

Let's dive into the "Titanic" dataset and see how we can use Seaborn to gain insights into the passengers and their survival rates.

```

1 import seaborn as sns
2 import matplotlib.pyplot as plt
3 import pandas as pd
4
5 # Load Titanic dataset
6 titanic = sns.load_dataset("titanic")
7
8 # Set Seaborn theme
9 sns.set_theme(style="whitegrid")
10
11 # Create a figure with subplots
12 fig, axes = plt.subplots(2, 2, figsize=(12, 8))
13
14 # 1. Bar Plot: Survival Rate by Class
15 sns.barplot(data=titanic, x="class", y="survived", hue="class", estimator=lambda x: sum(
16     x) / len(x), palette="Blues", legend=False, ax=axes[0, 0])
16 axes[0, 0].set_title("Survival Rate by Class")
17
18 # 2. Box Plot: Age Distribution by Class
19 sns.boxplot(data=titanic, x="class", y="age", hue="class", palette="Set2", legend=False,
20             ax=axes[1, 0])
20 axes[1, 0].set_title("Age Distribution by Class")
21
22 # 3. Count Plot: Number of Passengers by Embark Town
23 sns.countplot(data=titanic, x="embark_town", hue="embark_town", palette="coolwarm",
24                 legend=False, ax=axes[0, 1])
24 axes[0, 1].set_title("Number of Passengers by Embark Town")
25
26 # 4. Violin Plot: Fare Distribution by Class
27 sns.violinplot(data=titanic, x="class", y="fare", hue="class", palette="muted", legend=
28                  False, ax=axes[1, 1])
28 axes[1, 1].set_title("Fare Distribution by Class")
29
30 # Adjust layout and display
31 plt.tight_layout()
31 plt.show()

```

Listing 2: Visualizing the Titanic dataset with multiple plot types

In this example, we're creating four different types of plots:

1. **Bar Plot:** Shows the survival rate for each passenger class.
2. **Box Plot:** Displays the distribution of ages within each passenger class. Box plots are useful for understanding the median, quartiles, and outliers in your data.
3. **Count Plot:** Visualizes the number of passengers who embarked from each town.
4. **Violin Plot:** Similar to a box plot, but provides a more detailed view of the data distribution. Violin plots are a combination of box plots and kernel density estimation (KDE) plots, displaying the probability density of the data at different values.

```

1 import seaborn as sns
2 import matplotlib.pyplot as plt
3
4 # Load the Flights dataset
5 flights = sns.load_dataset("flights")
6
7 # Set Seaborn theme
8 sns.set_theme(style="whitegrid")
9
10 # Create a figure with subplots
11 fig, axes = plt.subplots(2, 2, figsize=(14, 10))
12
13 # 1. Line Plot: Yearly Trend of Airline Passengers
14 sns.lineplot(data=flights, x="year", y="passengers", estimator="sum", errorbar=None,
15               marker="o", ax=axes[0, 0])
15 axes[0, 0].set_title("Total Passengers Per Year")
16
17 # 2. Box Plot: Monthly Passenger Distribution Over Years
18 sns.boxplot(data=flights, x="month", y="passengers", hue="month", palette="coolwarm",
18             legend=False, ax=axes[0, 1])

```

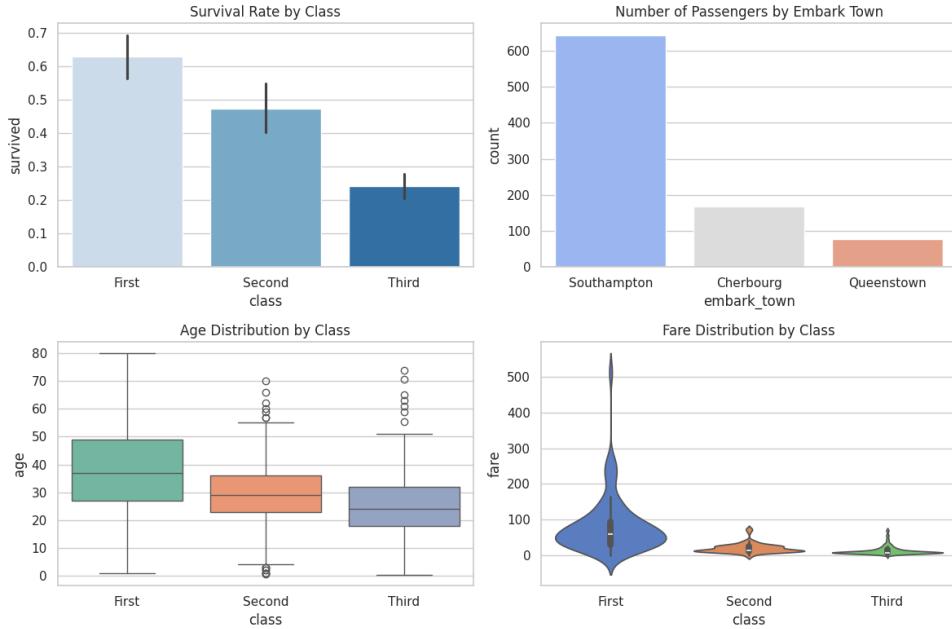


Figure 3: Multiple plots visualizing the Titanic dataset, including bar plot, box plot, count plot, and violin plot.

```

19 axes[0, 1].set_title("Passenger Distribution by Month")
20
21 # 3. Bar Plot: Average Monthly Passenger Count
22 sns.barplot(data=flights, x="month", y="passengers", hue="month", estimator="mean",
23               palette="Blues", legend=False, ax=axes[1, 0])
23 axes[1, 0].set_title("Average Monthly Passengers")
24
25 # 4. Heatmap: Monthly Passenger Count Over Years
26 flights_pivot = flights.pivot(index="month", columns="year", values="passengers")
27 sns.heatmap(flights_pivot, annot=True, fmt="d", cmap="YlGnBu", linewidths=0.5, ax=axes
28 [1, 1])
28 axes[1, 1].set_title("Heatmap of Monthly Passengers")
29
30 # Adjust layout and display
31 plt.tight_layout()
32 plt.show()

```

Listing 3: Visualizing the Flights dataset with multiple plot types

This example uses the "flights" dataset to illustrate how the distribution of air passengers has evolved over time. It creates four different plots:

1. **Line Plot:** Yearly Trend of Airline Passengers
2. **Box Plot:** Passenger Distribution by Month
3. **Bar Plot:** Average Monthly Passengers
4. **Heatmap:** Monthly Passenger Count Over Years

Heatmaps - Unveiling Correlations

Let's focus specifically on heatmaps because they're incredibly valuable for identifying relationships between variables.

A heatmap uses color intensity to represent the values in a matrix. In our flight data example, we can create a heatmap to visualize the number of passengers for each month and year.

```

1 import seaborn as sns
2 import matplotlib.pyplot as plt
3 import pandas as pd

```

```

4
5 # Load dataset
6 flights = sns.load_dataset("flights")
7
8 # Create a pivot table
9 flights_pivot = flights.pivot(index="month", columns="year", values="passengers")
10
11 # Create a heatmap
12 sns.heatmap(flights_pivot, annot=True, fmt="d", cmap="YlGnBu", linewidths=0.5)
13
14 # Add labels
15 plt.xlabel("Year")
16 plt.ylabel("Month")
17 plt.title("Number of Passengers (Flights Dataset)")
18
19 plt.show()

```

Listing 4: Creating a heatmap of the Flights dataset

Key points about heatmaps:

- `annot=True`: Displays the values in each cell of the heatmap.
- `fmt="d"`: Specifies the format of the values (in this case, integers).
- `cmap="YlGnBu"`: Sets the color palette.
 - The color palette has to do with the colormap of choice for the heatmap
 - Colormap has to do with what colors you want to use to represent the data points in the heatmap
 - In this case we are using the color palette 'YlGnBu'
- Lighter colors represent lower passenger counts, while darker colors represent higher passenger counts.

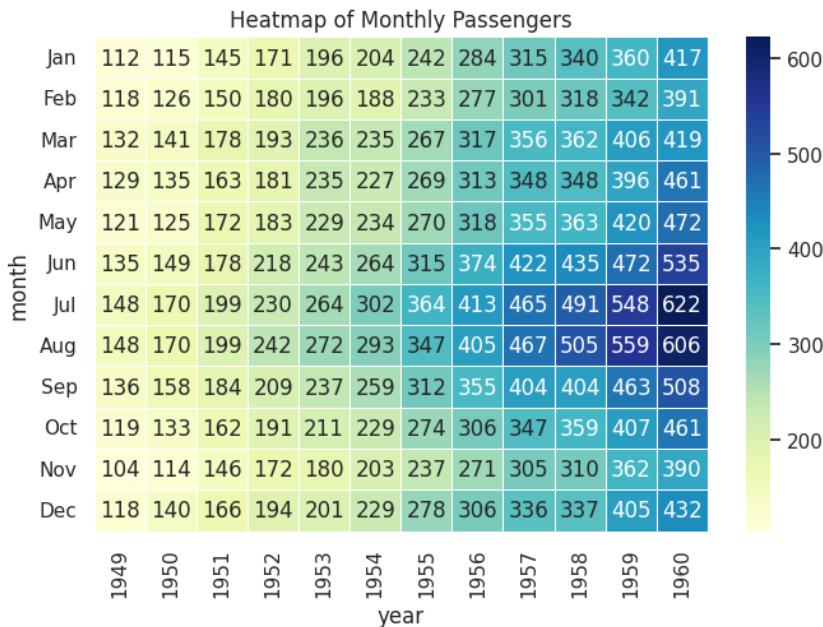


Figure 4: Heatmap visualizing monthly passenger counts over the years.

What if your Heatmap looks blank?

If your heatmap appears empty, the most likely reason is that your pivot table (the input to the heatmap) has missing values or incorrect indexing. Double-check that your data is properly structured before creating the heatmap.

We've only scratched the surface of Seaborn's capabilities in this brief introduction. As you continue your journey to mastering data visualization, remember to explore the documentation, experiment with different plot types, and always consider the type of data you're working with. With practice, you'll be able to create insightful visuals that drive data-driven decisions in your AI projects.

Happy visualizing!

Minor in AI

Class Notes

March 05

Title: Seaborn-Advanced and attractive visualizations

Syllabus:

- Customize plots to make them more informative and appealing
- Present data visually to convey insights.

Seaborn is a Python data visualization library built on top of Matplotlib, designed to make statistical graphics more attractive and informative. It was created by Michael Waskom and first released in 2012. Seaborn is middle name of Michael. Seaborn simplifies complex visualizations and integrates well with Pandas data structures, making it ideal for exploratory data analysis.

Seaborn provides various plot types, including:

- Relational plots (scatterplot, lineplot)
- Categorical plots (boxplot, violinplot, barplot, swarmplot)
- Distribution plots (histplot, kdeplot)
- Regression plots (regplot, residplot)
- Matrix plots (heatmap, clustermap)

It is widely used in data science and statistical analysis.

Activities discussed:

- Categorical data and continuous data
- Penguin types and bill in data science

Seaborn already has data loaded with penguin, titanic, flights etc.

Sample Programs:

Note: refer collab and video for more

Program for scatter plot on penguins

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load built-in dataset
penguins = sns.load_dataset("penguins")

# Create scatter plot
sns.scatterplot(data=penguins, x="bill_length_mm", y="bill_depth_mm",
hue="species", style="species")

# Show the plot
plt.show()
```

Program on flights data:

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Load dataset
flights = sns.load_dataset("flights")

# Pivot table
flights_pivot = flights.pivot(index="month", columns="year",
values="passengers")

# Plot heatmap
plt.figure(figsize=(10, 6))
sns.heatmap(flights_pivot, annot=True, fmt="d", cmap="coolwarm")
```

```
# Labels
plt.xlabel("Year")
plt.ylabel("Month")
plt.title("Number of Passengers (Flights Dataset)")

plt.show()
```

Seaborn Demystified: Visual Data Analysis for Actionable Intelligence

Minor in AI, IIT Ropar

6th March, 2025

Welcome, future data explorers! This notes is your guide to mastering Seaborn, a powerful Python library for creating insightful and visually appealing data visualizations. We'll go beyond just the syntax, focusing on how to extract meaningful insights from your data and use those insights to make strategic decisions.

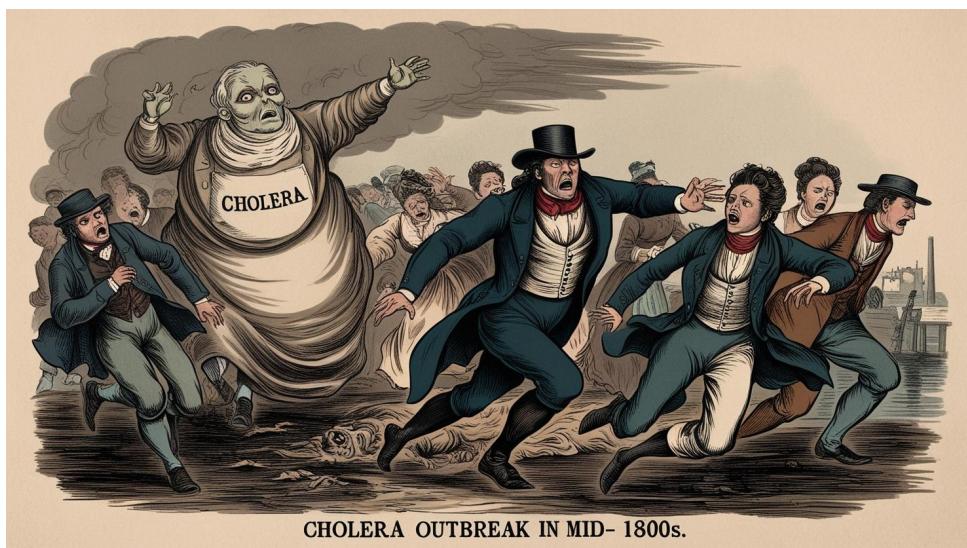


Figure 1: Scary scary Cholera back in the day!

1 The Power of Seeing: The John Snow Story

Imagine a bustling London in the mid-1800s. A terrifying outbreak of cholera grips the city, leaving residents panicked and desperate. Doctors and officials are baffled, unable to pinpoint the source of the deadly disease. Meetings are held, theories are debated, but the deaths continue.

Enter Dr. John Snow. He wasn't content with just discussing the problem. He decided to *visualize* it. He meticulously plotted each cholera death on a map of London. He then marked the locations of the water pumps throughout the city.

Suddenly, a pattern emerged. A dense cluster of deaths centered around a particular water pump on Broad Street. Dr. Snow hypothesized that the pump was contaminated and responsible for the outbreak. He convinced the authorities to shut down the pump. The result? The cholera outbreak slowed dramatically.

This story demonstrates the incredible power of visualization. Sometimes, a simple graph can reveal insights that are invisible in raw data. It can transform confusion into clarity, and ultimately, lead to life-saving decisions. This is the power of Seaborn, and this is what we'll be learning to harness in this book.

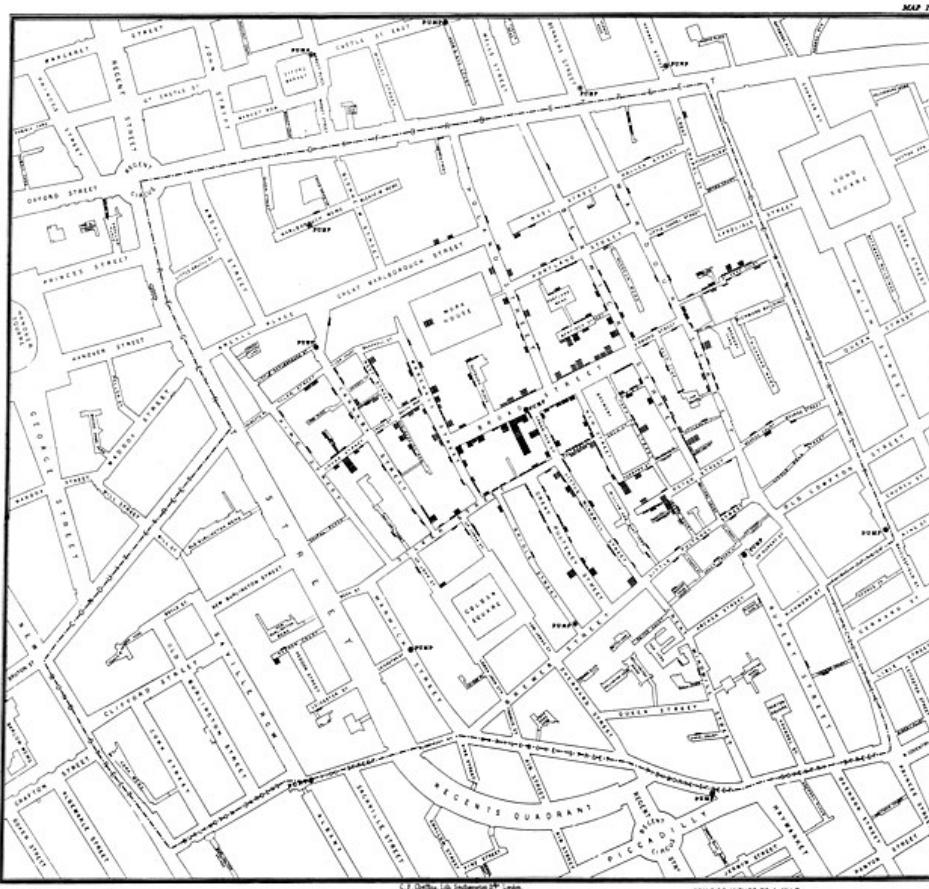


Figure 2: John Snow’s cholera map showing clusters of deaths around the Broad Street pump

2 Setting the Stage: Python, Pandas, and Seaborn

Before we dive into the visual wonders of Seaborn, let’s make sure our stage is set. You’ll need a basic understanding of Python and the Pandas library.

- **Python:** The programming language we’ll be using. If you’re new to Python, there are countless online resources to get you started.
- **Pandas:** A powerful library for data manipulation and analysis. Pandas provides data structures like DataFrames that make working with tabular data a breeze.
- **Seaborn:** Our star player! Seaborn is built on top of Matplotlib and provides a high-level interface for creating informative and aesthetically pleasing statistical graphics.

3 Count Plots: Tallying the Votes

Let’s start with a simple yet effective visualization: the count plot. Imagine you’ve conducted a survey asking people about their favorite programming language. You have a list of responses, and you want to quickly see which languages are the most popular. That’s where a count plot comes in.

Here’s how you can create a count plot using Seaborn:

```

1 import seaborn as sns
2 import matplotlib.pyplot as plt
3 import pandas as pd
4
5 # Create a simple dataset
6 data = pd.DataFrame({

```

```

7     "Language": ["C", "Python", "Python", "Java", "JS", "C", "C++", "JS", "Python", "R"]
8 }
9
10 # Set Seaborn theme
11 sns.set_theme(style="whitegrid")
12
13 # Create countplot
14 sns.countplot(data=data, x="Language", hue="Language", palette="coolwarm",
15                 legend=False)
16
17 # Add title
18 plt.title("Favorite Programming Languages")
19
20 # Show plot
21 plt.show()

```

Listing 1: Creating a Count Plot with Seaborn

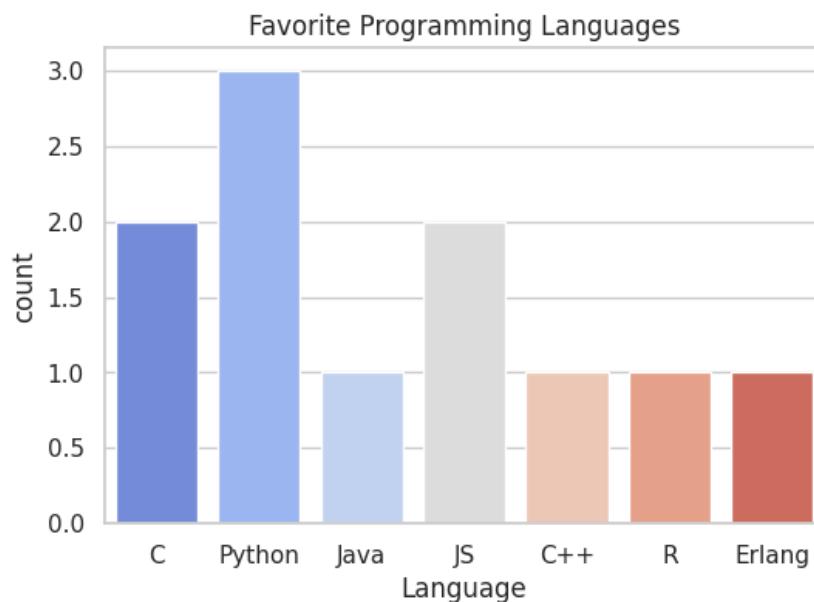


Figure 3: A count plot showing the frequency of different programming languages

Code Breakdown:

- We import the necessary libraries: `seaborn`, `matplotlib.pyplot`, and `pandas`.
- We create a Pandas DataFrame with a column called "Language" containing the survey responses.
- `sns.set_theme(style="whitegrid")` sets a visually appealing theme for our plot.
- `sns.countplot(data=data, x="Language")` creates the count plot. We specify the DataFrame (`data`) and the column we want to count (`Language`).
- `plt.title("Favorite Programming Languages")` adds a title to our plot.
- `plt.show()` displays the plot.

A count plot essentially counts the occurrences of each category in a column and displays them as bars. Think of it like a histogram for categorical data.

4 Scatter Plots and Kernel Density Estimation (KDE): Unveiling Relationships

Now, let's move on to a more sophisticated visualization: the scatter plot. Scatter plots are perfect for exploring the relationship between two numerical variables. Imagine you're studying penguins and want to see if there's a relationship between their bill length and bill depth.

```
1 import seaborn as sns
2 import matplotlib.pyplot as plt
3 import pandas as pd
4
5 # Load dataset
6 penguins = sns.load_dataset("penguins")
7
8 # Drop NaN values
9 penguins = penguins.dropna()
10
11 # Set figure size
12 plt.figure(figsize=(10, 6))
13
14 # Create a scatter plot with KDE overlay
15 sns.kdeplot(
16     data=penguins,
17     x="bill_length_mm",
18     y="bill_depth_mm",
19     hue="species",
20     fill=True,
21     alpha=0.4,
22 )
23 sns.scatterplot(
24     data=penguins,
25     x="bill_length_mm",
26     y="bill_depth_mm",
27     hue="species",
28     edgecolor="black"
29 )
30
31 # Titles and labels
32 plt.title("Penguin Bill Dimensions: KDE and Scatter Plot", fontsize=14)
33 plt.xlabel("Bill Length (mm)", fontsize=12)
34 plt.ylabel("Bill Depth (mm)", fontsize=12)
35 plt.legend(title="Species")
36 plt.grid(True)
37
38 # Show plot
39 plt.show()
```

Listing 2: Creating a Scatter Plot with KDE Overlay

Insights and Strategic Decision-Making:

Now, this is where things get interesting. Looking at the scatter plot, you might observe:

- **Clustering:** Penguins of the same species tend to cluster together. This suggests that bill dimensions can be used to distinguish between species.
- **Overlapping:** There is some overlap between the species, especially in the middle. This means that relying solely on bill length and depth might not be enough to accurately classify all penguins. You might need to consider other features, such as flipper length or body mass.
- **Outliers:** There may be a few outliers – penguins that fall far from the main clusters. These outliers could be due to measurement errors, misidentification, or simply represent unusual individuals. Further investigation might be needed to understand these outliers.

By visualizing the data, you gain valuable insights that can inform your research and decision-making.

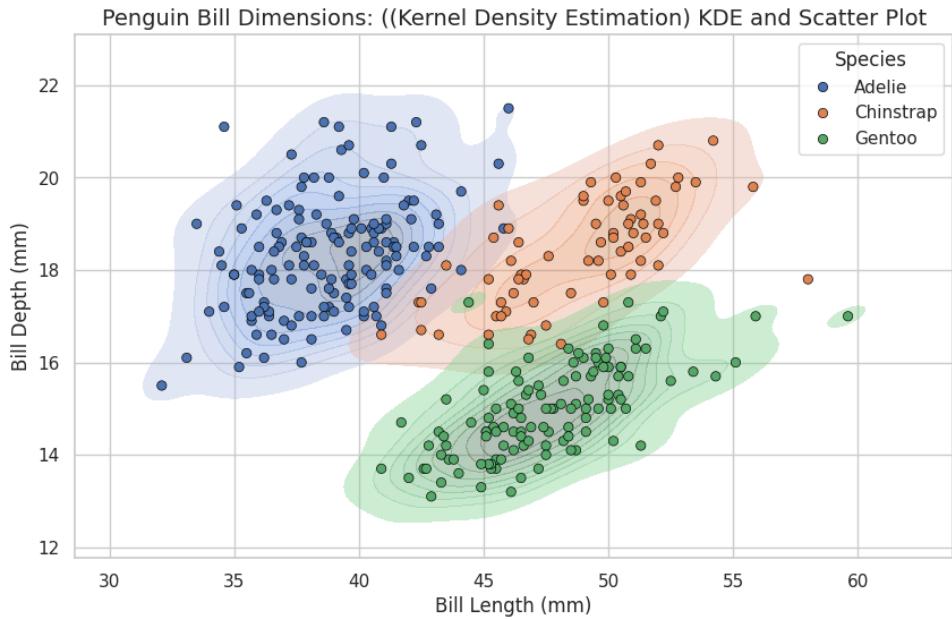


Figure 4: A scatter plot of penguin bill length vs. bill depth, with KDE overlay

5 Regression Plots: Spotting Trends

Let's explore trend analysis with regression plots. Let's use the example of customer age versus total spending.

```

1 import seaborn as sns
2 import matplotlib.pyplot as plt
3 import pandas as pd
4 import numpy as np
5
6 # Generate sample customer sales data
7 np.random.seed(42)
8 data = pd.DataFrame({
9     "Customer_Age": np.random.randint(18, 65, 200),
10    "Total_Spend": np.random.uniform(100, 5000, 200)
11 })
12
13
14 plt.figure(figsize=(10, 6))
15
16 # Scatter plot with trend line
17 sns.regplot(data=data, x="Customer_Age", y="Total_Spend", scatter_kws={'alpha': 0.6}, line_kws={'color': 'red'})
18
19 plt.title("Customer Age vs Total Spending (with Trend Line)", fontsize=14)
20 plt.xlabel("Customer Age", fontsize=12)
21 plt.ylabel("Total Spend (in $)", fontsize=12)
22
23 plt.show()

```

Listing 3: Creating a Regression Plot

In the example plot generated, The trend line is flat, suggesting that there is no correlation between customer age and the total amount spent.

6 Bar Plots: Payment preference

Let's explore analyzing payment methods in the customer payment preferences example:

```
1 import seaborn as sns
2 import matplotlib.pyplot as plt
3 import pandas as pd
4 import numpy as np
5
6 # Generate sample payment method data
7 np.random.seed(42)
8 payment_methods = ["Credit Card", "Debit Card", "UPI", "Cash", "Net Banking", "Wallet"]
9 transactions = np.random.randint(100, 1000, len(payment_methods))
10
11 data = pd.DataFrame({
12     "Payment_Method": payment_methods,
13     "Number_of_Transactions": transactions
14 })
15
16 plt.figure(figsize=(10, 6))
17
18 # Fix: Assign 'hue' to the x variable and disable the legend
19 sns.barplot(data=data, x="Payment_Method", y="Number_of_Transactions",
20             hue="Payment_Method", palette="coolwarm", legend=False)
21
22 # Customize plot
23 plt.title("Customer Payment Preferences", fontsize=14)
24 plt.xlabel("Payment Method", fontsize=12)
25 plt.ylabel("Number of Transactions", fontsize=12)
26
27 # Rotate x-axis labels for readability
28 plt.xticks(rotation=20)
29
30 # Show plot
31 plt.show()
```

Listing 4: Creating a Bar Plot for Payment Preferences

Strategic Decision-Making:

When opening a market you should consider the top three payment modes. From the dataset provided and the plot, UPI, Debit Card and Cash emerge as the top three payments.

7 Pair Plots: The Big Picture

Now, let's unleash the power of the pair plot. This plot is your go-to tool when you want to explore the relationships between multiple numerical variables in your dataset. It creates a matrix of scatter plots, showing the relationship between each pair of variables.

```
1 import seaborn as sns
2 import matplotlib.pyplot as plt
3 import pandas as pd
4 import numpy as np
5
6 # Generate sample employee performance data
7 np.random.seed(42)
8 num_employees = 100
9
10 data = pd.DataFrame({
11     "Years_of_Experience": np.random.randint(1, 20, num_employees),
12     "Salary": np.random.randint(30, 150, num_employees), # Salary in $1000s
13     "Projects_Completed": np.random.randint(1, 50, num_employees),
```

```

14     "Work_Hours_per_Week": np.random.randint(30, 60, num_employees)
15 )
16
17
18 # Pair plot to visualize relationships between numerical variables
19 sns.pairplot(data, diag_kind="kde")
20
21 # Show plot
22 plt.show()

```

Listing 5: Creating a Pair Plot

In pair plots, all relationships are scattered in a plot. The data is scattered across the plot, and insights can be gathered from the plots. In the diagonal elements, you are free to pick the kind of data you want. You can go for KDE, histogram, based on what you want to see.

8 Swarm Plots

To look at the salary distributions across the department, swarm plots could be really helpful:

```

1 import seaborn as sns
2 import matplotlib.pyplot as plt
3 import pandas as pd
4 import numpy as np
5
6 # Set seed for reproducibility
7 np.random.seed(42)
8 num_employees = 150
9
10 # Define departments
11 departments = ["HR", "Engineering", "Marketing", "Sales", "Finance"]
12
13 # Generate synthetic salary data
14 data = pd.DataFrame({
15     "Department": np.random.choice(departments, num_employees),
16     "Salary": np.concatenate([
17         np.random.randint(40, 80, 30),    # HR salaries
18         np.random.randint(70, 150, 40),   # Engineering salaries
19         np.random.randint(50, 100, 30),   # Marketing salaries
20         np.random.randint(45, 90, 25),   # Sales salaries
21         np.random.randint(60, 120, 25)   # Finance salaries
22     ])
23 })
24
25 # Create the swarm plot with the corrected syntax
26 plt.figure(figsize=(10, 6))
27 sns.swarmplot(data=data, x="Department", y="Salary", hue="Department",
28                 palette="coolwarm", legend=False)
29
30 # Add labels and title
31 plt.xlabel("Department")
32 plt.ylabel("Salary ($1000s)")
33 plt.title("Swarm Plot of Salaries by Department")
34
35 # Show the plot
36 plt.show()

```

Listing 6: Creating a Swarm Plot for Salary Distribution

9 Beyond the Basics: Customization

Seaborn offers endless possibilities for customization. You can control colors, fonts, labels, titles, and much more to create visualizations that are both informative and visually appealing. With custom charts, you can leverage different customizations. The customization charts code is available in the shared collab link.

10 Visualizing Your Way to Success

Seaborn is more than just a plotting library. It's a powerful tool for exploring data, uncovering hidden patterns, and making strategic decisions. By mastering Seaborn, you'll be able to transform raw data into actionable insights and communicate your findings effectively. So, dive in, experiment, and unleash the power of visualization!

Minor in AI

Class Notes

March 06

Title: Seaborn-Advanced and attractive visualizations

Syllabus:

- Customize plots to make them more informative and appealing
- Present data visually to convey insights.

Activities discussed:

- John Snow's work on cholera outbreak in London: importance of visualization

Seaborn, a Python data visualization library, simplifies statistical plotting, making it easier to uncover patterns, trends, and relationships in data. With built-in themes and color palettes, it enhances readability and aesthetics. Seaborn's functions, such as sns.pairplot() and sns.heatmap(), allow users to explore correlations, distributions, and categorical relationships effectively. It integrates well with Pandas, enabling direct visualization from DataFrames. Through violin plots, box plots, and regression plots, Seaborn helps in identifying outliers, trends, and deviations. Its ease of customization and statistical plotting capabilities make it a powerful tool for drawing meaningful insights and supporting data-driven decision-making.

Refer to collab for all the programs.

The AI Detective: Finding Hidden Patterns in Data with Mean, Median & More

Minor In AI, IIT Ropar

10th Feb, 2025

Welcome, future AI enthusiasts! This module is your guide to understanding some fundamental statistical concepts that form the backbone of data analysis and decision-making in the world of Artificial Intelligence. We'll journey from the vast plains of Africa to the inner workings of Python, uncovering how these seemingly simple ideas can unlock powerful insights from data.

1 The Great Migration and the Curious Case of Numbers

Imagine witnessing the Great Migration of the Masai Mara in Kenya. Millions of wildebeest, zebras, and other animals embark on a perilous journey in search of greener pastures. It's a breathtaking spectacle, a dance of life and survival played out on a grand scale.



Figure 1: The magnificent Great Migration in the Masai Mara, Kenya, showing thousands of wildebeest crossing the plains. Photo credit: [Matt Scobel on 500px](#)

When faced with this immense movement, what questions come to mind? You might wonder:

- **How many** animals participate in this migration each year?
- **What types** of animals make the journey?
- **When** is the peak migration season?
- **Why** do they migrate at all?

These questions are all about data. Data isn't just numbers in a spreadsheet; it's a representation of the real world. And to understand the real world, we need to analyze data.

Think about tracking the number of animals. Are they all moving in a single massive herd, or are they breaking up into smaller groups? How long does the migration take? Gathering this data allows us to understand the rhythm and scale of this natural phenomenon.

But what if we only had a small sample of data about the ratings about the documentary about the migration? Maybe only a few viewers have provided some data? This is where some important concepts come in:

2 Unveiling Central Tendencies: Mean, Median, and Mode

Let's start with a simple example. Suppose we asked ten people to rate a documentary about the Masai Mara migration on a scale of 1 to 5, where 5 is the best. Here are the ratings we received:

4, 4, 4, 4, 5, 5, 5, 1, 4

From just glancing at this data, we can tell that people generally liked the documentary. But how can we summarize this information more precisely?

2.1 The Average Storyteller: Mean

One way is to calculate the *mean*, also known as the average. To find the mean, we add up all the ratings and divide by the number of ratings:

$$(4 + 4 + 4 + 4 + 4 + 5 + 5 + 5 + 1 + 4) / 10 = 3.8$$

So, the *mean* rating is 3.8. In Python, we can use the `numpy` library to calculate the mean:

```
1 import numpy as np
2
3 ratings = [4, 4, 4, 4, 4, 5, 5, 5, 1, 4]
4 mean_rating = np.mean(ratings)
5 print(f"The mean rating is: {mean_rating}") # Output: The mean rating is: 3.8
```

Listing 1: Calculating the mean using NumPy

2.2 The Middle Ground: Median

Another way to describe the “center” of our data is to find the *median*. The median is the middle value when the data is sorted. First, let's sort the ratings:

1, 4, 4, 4, 4, 4, 5, 5, 5, 5

Since we have an even number of ratings (10), the median is the average of the two middle values (4 and 4).

$$(4 + 4) / 2 = 4$$

So, the *median* rating is 4. Let's calculate it in python as well!

```
1 import numpy as np
2
3 ratings = [4, 4, 4, 4, 4, 5, 5, 5, 1, 4]
4 median_rating = np.median(ratings)
5 print(f"The median rating is: {median_rating}") #The median rating is: 4.0
```

Listing 2: Finding the median using NumPy

2.3 The Popular Choice: Mode

Finally, the *mode* is the value that appears most often in the data. In our example, the rating of 4 appears five times, which is more than any other rating. Therefore, the *mode* is 4. Again, let's calculate it in python:

```
1 import scipy.stats as stats
2 ratings = [4, 4, 4, 4, 4, 5, 5, 5, 1, 4]
3 mode_rating = stats.mode(ratings)
4 print(f"The mode rating is: {mode_rating[0]}") #The mode rating is: [4]
```

Listing 3: Determining the mode using SciPy

So, in our migration data example, one might also be curious about what the median animal count looks like during October. The mean and mode might also be insightful in their own way.

3 Visualizing Distributions: Histograms

Histograms are powerful tools for visualizing the distribution of data. They show us how frequently different values occur within a dataset.

Imagine plotting the heights of all the students in a class. A histogram would show how many students fall into different height ranges. Most students might cluster around the average height, with fewer students being very tall or very short. This is a *normal distribution*.

Now, consider the Masai Mara migration. Let's say we track the daily number of animals crossing a particular river. A histogram of this data might show a peak during the main migration season, with fewer crossings at the beginning and end.

Here's an example of how to create a histogram in Python using the `seaborn` library:

```

1 import seaborn as sns
2 import matplotlib.pyplot as plt
3 import numpy as np
4
5 # Generate synthetic data with one peak (normal distribution)
6 data = np.random.normal(loc=50, scale=10, size=500) # Mean=50, Std Dev=10
7
8 # Create the histogram using seaborn
9 sns.histplot(data, bins=30, kde=True, color="royalblue")
10
11 # Add labels and title
12 plt.xlabel("Values")
13 plt.ylabel("Frequency")
14 plt.title("Histogram")
15
16 # Show the plot
17 plt.show()

```

Listing 4: Creating a histogram with Seaborn

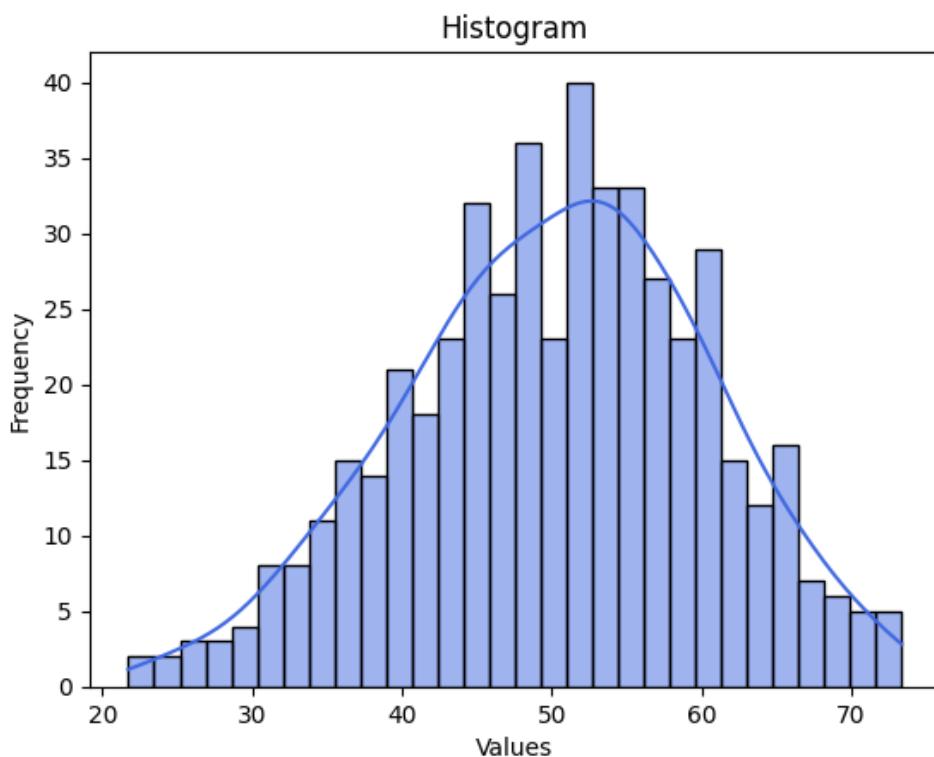


Figure 2: Histogram showing the daily count of animals crossing a river during the Great Migration.

You might also encounter distributions where data is not symmetrical, or there might be a few peaks

in the dataset. A visualization with multiple distribution would look like the code below.

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 # Unimodal distribution
5 np.random.seed(0)
6 data_unimodal = np.random.normal(loc=0, scale=1, size=1000)
7 plt.figure(figsize=(10, 4))
8 plt.hist(data_unimodal, bins=30, edgecolor='black')
9 plt.title('Unimodal Distribution')
10 plt.xlabel('Value')
11 plt.ylabel('Frequency')
12 plt.show()
13
14
15 # Bimodal distribution
16 np.random.seed(1)
17 data_bimodal1 = np.random.normal(loc=-2, scale=1, size=500)
18 data_bimodal2 = np.random.normal(loc=2, scale=1, size=500)
19 data_bimodal = np.concatenate((data_bimodal1, data_bimodal2))
20 plt.figure(figsize=(10, 4))
21 plt.hist(data_bimodal, bins=30, edgecolor='black')
22 plt.title('Bimodal Distribution')
23 plt.xlabel('Value')
24 plt.ylabel('Frequency')
25 plt.show()
26
27
28 # Multimodal distribution (3 modes)
29 np.random.seed(2)
30 data_multimodal1 = np.random.normal(loc=-3, scale=0.8, size=300)
31 data_multimodal2 = np.random.normal(loc=0, scale=1.2, size=400)
32 data_multimodal3 = np.random.normal(loc=3, scale=0.9, size=300)
33 data_multimodal = np.concatenate((data_multimodal1, data_multimodal2,
   data_multimodal3))
34 plt.figure(figsize=(10, 4))
35 plt.hist(data_multimodal, bins=30, edgecolor='black')
36 plt.title('Multimodal Distribution (3 modes)')
37 plt.xlabel('Value')
38 plt.ylabel('Frequency')
39 plt.show()
```

Listing 5: Creating various types of distributions with Matplotlib

4 Box Plots: A Deeper Dive into Data Distribution

Box plots, also known as box-and-whisker plots, offer a concise way to visualize the distribution of data, highlighting key statistics like quartiles and outliers.

4.1 Unpacking the Box Plot

A box plot consists of:

- **A box:** The box represents the interquartile range (IQR), which contains the middle 50% of the data. The left edge of the box corresponds to the 25th percentile (Q1), and the right edge corresponds to the 75th percentile (Q3).
- **A line inside the box:** This line represents the median (Q2), the middle value of the dataset.
- **Whiskers:** The whiskers extend from the edges of the box to the farthest data points within a certain range (typically 1.5 times the IQR).

- **Outliers:** Data points beyond the whiskers are considered outliers and are plotted individually as dots or circles.

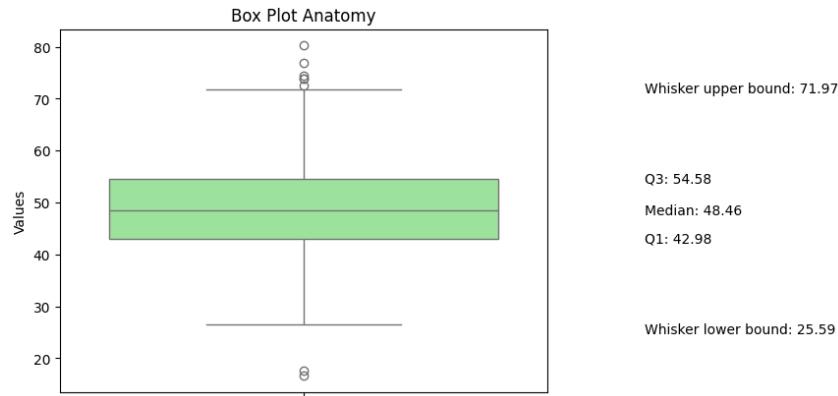


Figure 3: Detailed anatomy of a box plot showing quartiles, median, whiskers, and outliers.

4.2 Creating Box Plots in Python

Here's how to create a basic box plot using Seaborn:

```

1 #basic Box Plot
2 import matplotlib.pyplot as plt
3 import numpy as np
4 data = np.random.randn(100) * 10 + 50
5 plt.boxplot(data)
6 plt.show()

```

Listing 6: Creating a basic box plot with Matplotlib

4.3 Interpreting Box Plots

The beauty of box plots lies in their ability to quickly convey information about the spread, center, and skewness of data.

- **Spread:** A wide box indicates high variability in the middle 50% of the data, while a narrow box suggests low variability.
- **Skewness:** If the median is closer to one edge of the box, the data is skewed. A median closer to the left edge indicates right skewness (positive skew), and a median closer to the right edge indicates left skewness (negative skew).
- **Outliers:** The presence of outliers can indicate unusual or erroneous data points that warrant further investigation.

Let's consider different scenarios to see how box plots help with different datasets

4.4 Case Study: Comparing Student Scores

Imagine comparing the scores of two different classes on the same test. A box plot for each class could reveal:

- Which class has a higher median score.
- Which class has a wider range of scores (more variability).
- Whether either class has any outliers (students who performed exceptionally well or poorly).

By looking at these aspects, we can draw conclusions about the performance of the two classes and determine what additional insights can be extracted.

Statistical Hypothesis Testing

Minor in AI, IIT Ropar

1 Why Traditional Statistics Fail: The Threshold Paradox

The Silent Café Paradox

Story: In a small town, four friend groups face a dilemma - should they try the new café? Each member has an "action threshold":

- Radhika (0): "I'll go immediately - I love new places!"
- Aryan (1): "I'll go if at least 1 friend joins"
- Neha (2): "I need 2 friends to go first"
- Raj (3): "Only if 3 friends commit"

The Plot Twist:

- **Group 2:** [Aryan, Aryan, Neha, Neha, Raj]
Seems promising: Average threshold = 1.8
Reality: Café remains empty - no initiators!
- **Group 4:** [Radhika, Aryan, Raj, Raj, Raj]
Looks average: Mean threshold = 2
Reality: Only Radhika and Aryan go initially - others stay home!

The Statistical Mystery

- **Surface Analysis:** Group 4's mean (2) matches population average
- **Hidden Truth:** Thresholds create *activation chains*
 → No 0-threshold = No ignition (Group 2)
 → Majority need critical mass = System stalls (Group 4)

Key Insight: “The First Mover Crisis”

- **Trendsetters (0-threshold):** The spark for social change.
 Early adopters (Threshold 0) drive group behavior.
- **Statistical Deception:**
 $\text{Mean } (\mu) = \frac{\sum x_i}{n}$ ignores activation sequences.
 Median ignores threshold dependencies.
 Traditional statistics (mean/median/mode) can be misleading
- **Solution Path:** Requires **network analysis + hypothesis testing**
Threshold analysis instead of **basic statistics**
 “Do groups with 0-threshold members have significantly different activation rates?”

2 Hypothesis Testing: The Data Scientist's Toolkit

Problem Statement

”Do Company A and Company B have significantly different salaries?”

Traditional approach: Compare means

Better approach: Statistical hypothesis testing

2.1 Core Concepts

In hypothesis testing, several foundational ideas guide the analysis. The **null hypothesis** (H_0) is the default assumption that there is “no significant difference” between the groups or variables under study, meaning any observed variation is attributed to random chance rather than a true effect. In contrast, the **alternative hypothesis** (H_1) asserts that a “significant difference exists,” indicating that the observed data reflect a genuine effect or difference between the groups.

A critical part of this analysis is the **p-value**, which represents the probability of obtaining the observed results, or even more extreme ones, if the null hypothesis were true. This measure helps determine whether the evidence is strong enough to reject H_0 in favor of H_1 . To make this decision, a **significance level** (α) is set, commonly at 0.05, signifying a 5% risk of a false positive. If the p-value falls below this threshold, the observed difference is considered statistically significant, leading to the rejection of the null hypothesis.

- **Null Hypothesis (H_0):** “No significant difference”
- **Alternative Hypothesis (H_1):** “Significant difference exists”
- **p-value:** Probability of observing results if H_0 is true
- **Significance Level (α):** 0.05 (5% risk of false positive)

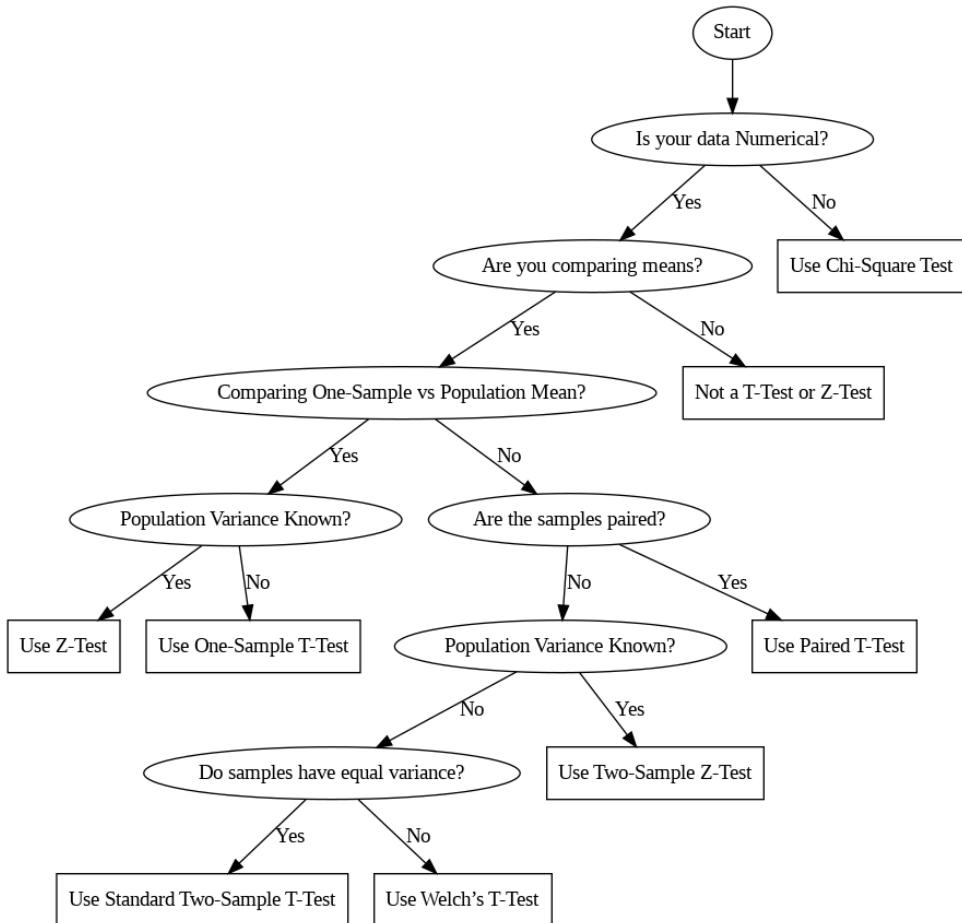
3 Choosing the Right Statistical Test

Test Selection Framework

1. Check variance equality (Levene’s Test)
2. Small variance → Use T-test
3. Large variance → Use Welch’s Test
4. Categorical data → Chi-square Test

3.1 Levene’s Test: Variance Checker

To determine whether two groups have equal variances, we use Levene’s test. A high p-value (typically above 0.05) suggests that the variances are not significantly different, while a low p-value indicates that they are.



```

1 from scipy.stats import levene # Importing Levene's test from scipy.stats
2
3 # Sample salary data for two groups
4 salaries_A = [55,60,62,58,63,59,61,60,57,64,62,56,59,58,65]
5 salaries_B = [52,55,57,53,58,54,56,55,51,59,57,50,54,53,60]
6
7 # Performing Levene's test to check variance equality
8 levene_stat, p_value = levene(salaries_A, salaries_B)
9 print(f"Variance Check: p={p_value:.4f}")
  
```

3.2 Independent T-Test

An independent t-test compares the means of two groups under the assumption that their variances are equal. If the p-value is below a chosen significance level (e.g., 0.05), we reject the null hypothesis and conclude that there is a significant difference between the two means.

```

1 from scipy.stats import ttest_ind # Importing t-test function
2
3 # Performing an independent t-test assuming equal variance
4 t_stat, p_value = ttest_ind(salaries_A, salaries_B, equal_var=True)
5 print(f"T-Test Results: t={t_stat:.2f}, p={p_value:.4f}")
  
```

3.3 Welch's T-Test (Unequal Variance)

Welch's t-test is similar to the independent t-test but does not assume equal variances. It is useful when the two groups have significantly different variances.

```
1 # Performing Welch's t-test assuming unequal variances
2 t_stat, p_value = ttest_ind(salaries_A, salaries_B, equal_var=False)
3 print(f"Welch's T-Test Results: t={t_stat:.2f}, p={p_value:.4f}")
```

3.4 Chi-Square Test (Categorical Data)

The chi-square test is used to determine if there is a significant association between categorical variables. The test compares observed frequencies with expected frequencies under the assumption of independence.

```
1 from scipy.stats import chi2_contingency # Importing chi-square test
2 # function
3
4 # Observed frequency table (Male, Female) for different brands
5 observed = [[30,40], # Apple
6             [50,30], # Samsung
7             [20,30]] # OnePlus
8
9 # Performing chi-square test
10 chi2_stat, p_value, dof, expected = chi2_contingency(observed)
11 print(f"Chi-Square Test Results: X2={chi2_stat:.2f}, p={p_value:.4f}")
```

4 Real-World Case Studies

4.1 Case 1: Student Sleep Hours (Z-Test)

A Z-test is useful when comparing a sample mean to a known population mean, assuming the population standard deviation is known. In this case, we analyze student sleep hours to test if the mean sleep duration differs significantly from 8 hours.

```
1 import scipy.stats as stats # Importing statistical functions from
2 # SciPy
3 import numpy as np # Importing NumPy for array operations
4
5 # Generating sample data with a normal distribution (mean=7.5, std dev
6 # =1.5, sample size=40)
7 sample_data = np.random.normal(loc=7.5, scale=1.5, size=40)
8
9 # Computing the Z-score manually
10 z_score = (np.mean(sample_data) - 8.0) / (1.5 / np.sqrt(40)) # Standard
# error adjustment
11
12 # Calculating the p-value for a one-tailed test
13 p_value = stats.norm.cdf(z_score)
14 print(f"Z-Test Results: Z={z_score:.2f}, p={p_value:.4f}")
```

4.2 Case 2: Battery Life Claim (Two-Tailed Z-Test)

A two-tailed Z-test is used when we want to check whether a sample mean is significantly different (higher or lower) than a claimed population mean. This is useful for verifying

manufacturer claims about product performance.

```
1 # Calculating the p-value for a two-tailed test
2 p_value = 2 * (1 - stats.norm.cdf(abs(z_score))) # Multiply by 2 for
   two-tailed significance
3 print(f"Two-Tailed Z-Test p-value: {p_value:.4f}")
```

5 Conclusion: Essential Insights

Cheat Sheet

- Always check variance before choosing T-test/Welch
- $p < 0.05 \rightarrow$ Reject null hypothesis
- Threshold analysis needs special handling
- Type I Error: False positive (5% risk)
- Type II Error: False negative

Remember!

- Z-test: Large samples ($n > 30$)
- T-test: Small samples with normal distribution
- Chi-square: Categorical frequency analysis
- Levene's Test: Gatekeeper for variance checks

6 The Human Spark in Numbers

The Unseen Pulse of Groups

Behind every statistic lies a human story. Our café tale reveals a beautiful truth:
Change begins with just one brave soul.

- Averages tell us *where* groups stand, but miss *how* they move
- The magic isn't in the middle – it's in those who dare to start
- Next time you see a trend, ask: "Who lit the first candle?"

This isn't just about numbers – it's about understanding the invisible threads that connect us all. Whether launching a product, spreading ideas, or building communities, remember: **the right spark can ignite what perfect averages never could.**

The Art and Science of Algorithm Evaluation: Understanding Time Complexity and Asymptotic Analysis

Minor In AI, IIT Ropar

12th March, 2025

Contents

1 Your Favorite Ice Cream and the Soul of an Algorithm	2
2 Losing Your Keys and the Linear Search Algorithm	2
3 Why Counting Steps Isn't Enough	3
4 Finding the Basic Operation: The Heart of Algorithm Analysis	4
5 Best Case, Worst Case, and Average Case Analysis	4
6 Asymptotic Notation: Big O, Omega, and Theta	4
7 What Does 'n' Really Mean? The Order of Growth	5

1 Your Favorite Ice Cream and the Soul of an Algorithm

Imagine you have a favorite ice cream shop, or a juice center, or maybe a place that sells your favorite evening snacks. What makes you like it so much? Is it the taste? The ambiance? The friendly staff? You might have a feeling, a sense of *why* you love it. You might even rank it in your mind. “On a scale of 1 to 10, this place is a 9!”

Now, let’s switch gears. Imagine you’ve written a computer program—an algorithm—to solve a problem. Can you simply say, “This algorithm is good because it looks nice, has beautiful loops, and feels right”? Unfortunately, no. The computing world demands more. It asks for *quantifiable* proof. It wants to know *how well* your algorithm performs. This is where the art and science of algorithm evaluation comes in.

Just like you might try to understand why you love your favorite ice cream, we need tools to understand the qualities of our algorithms. We need ways to measure them, to compare them, and to improve them. That’s what this book is all about. We’ll learn how to analyze algorithms, not based on subjective feelings, but on mathematical principles.



Figure 1: Happily eating ice cream vs. Coding lol

2 Losing Your Keys and the Linear Search Algorithm

Let’s say you’ve lost your keys. You decide to search for them. You might check your coat pocket, then the shelf, then the drawer, and so on. You have a sequence of places you check. This methodical approach is similar to a simple search algorithm called a *linear search*.

A linear search is a straightforward way to find a specific item (like your keys) within a list of items (like possible hiding places). It works by checking each item in the list one by one, until you find the item you’re looking for.

Here’s how we can represent a linear search in Python:

```

1 def linear_search(arr, key):
2     """
3         Searches for a key in an array using a linear search.
4         Args:
5             arr: The array to search.
6             key: The key to search for.
7         Returns:

```

```

8     The index of the key if found, otherwise -1.
9 """
10    i = 0 #iterator
11    n = len(arr) #length of the array
12    while i < n: #iterating through the array
13        if arr[i] != key: # comparing each element with the key
14            i = i + 1
15        else:
16            return i
17    return -1

```

Listing 1: Linear Search Algorithm Implementation

Explanation:

- The `linear_search` function takes two arguments: the array `arr` to search and the `key` you're looking for.
- It initializes an index `i` to 0 and gets the length of the array `n`.
- It then enters a `while` loop that continues as long as `i` is less than `n` (meaning we haven't reached the end of the array).
- Inside the loop, it compares the element at the current index `arr[i]` with the `key`.
- If they are not equal, it increments `i` to check the next element.
- If they are equal, it means we've found the `key`, so the function returns the index `i`.
- If the loop completes without finding the `key`, it means the `key` is not in the array, so the function returns -1.

Example:

```

1 my_array = [10, 15, 34, 78, 22, 91]
2 search_key = 78
3 index = linear_search(my_array, search_key)
4 print(f"The element {search_key} is found at index: {index}") # Output: 3
5 search_key = 100
6 index = linear_search(my_array, search_key)
7 print(f"The element {search_key} is found at index: {index}") # Output: -1

```

Listing 2: Example Usage of Linear Search

3 Why Counting Steps Isn't Enough

Now that we have a linear search algorithm, how do we determine if it's any good? One initial thought might be to count the *number of steps* the algorithm takes. We could assign a “cost” to each line of code and add them up.

However, this approach, called *step counting*, has some serious flaws:

- **Hardware Dependency:** The time it takes to execute a single line of code can vary greatly depending on the computer's processor, memory, and other hardware components.
- **Compiler Optimizations:** The compiler (the program that translates your code into machine instructions) can optimize the code in various ways, making step counting inaccurate.
- **Input Sensitivity:** The number of steps can depend heavily on the specific input to the algorithm. In our `linear_search` example, if the `key` is the first element in the array, the algorithm finds it quickly. If the `key` is the last element or not in the array at all, it takes much longer.

Because of these issues, step counting is not a reliable way to compare the efficiency of algorithms. We need a more abstract and hardware-independent measure.

4 Finding the Basic Operation: The Heart of Algorithm Analysis

Instead of counting every single step, we focus on the *most important* step. This is called the *basic operation*. The basic operation is the operation that contributes the most to the algorithm's running time. It's the operation that is executed the most frequently.

In our `linear_search` example, the basic operation is the *comparison* between the array element `arr[i]` and the `key`: `arr[i] != key`. This comparison is performed in each iteration of the `while` loop.

By focusing on the basic operation, we can analyze the algorithm's efficiency in a more meaningful way. We're not concerned with the exact time each line of code takes to execute, but rather with *how many times* the most crucial operation is performed.

5 Best Case, Worst Case, and Average Case Analysis

The number of times the basic operation is executed can vary depending on the input. This leads to three important scenarios:

- **Best Case:** The best case occurs when the `key` is the first element in the array. In this case, the basic operation (the comparison) is executed only *once*.
- **Worst Case:** The worst case occurs when the `key` is the last element in the array or not in the array at all. In this case, the basic operation is executed n times, where n is the number of elements in the array.
- **Average Case:** The average case assumes that the `key` is equally likely to be at any position in the array. On average, the basic operation will be executed $n/2$ times.

```

1 # Code from Chapter 2 repeated to illustrate the best case, worst case & average case
2 def linear_search(arr, key):
3     """
4     Searches for a key in an array using a linear search.
5     Args:
6         arr: The array to search.
7         key: The key to search for.
8     Returns:
9         The index of the key if found, otherwise -1.
10    """
11    i = 0 #iterator
12    n = len(arr) #length of the array
13    while i < n: #iterating through the array
14        if arr[i] != key: # comparing each element with the key
15            i = i + 1
16        else:
17            return i
18    return -1

```

Listing 3: Linear Search Algorithm (repeated for analysis)

6 Asymptotic Notation: Big O, Omega, and Theta

To express the efficiency of algorithms in a standard way, we use *asymptotic notation*. Asymptotic notation describes how the running time of an algorithm grows as the input size increases.

There are three main types of asymptotic notation:

- **Big O Notation (O):** Big O notation describes the *worst-case* running time of an algorithm. It provides an *upper bound* on the growth rate. In our `linear_search` example, the worst-case running time is $O(n)$, which means that the running time grows linearly with the input size.
- **Omega Notation (Ω):** Omega notation describes the *best-case* running time of an algorithm. It provides a *lower bound* on the growth rate. In our `linear_search` example, the best-case running time is $\Omega(1)$, which means that the running time is constant regardless of the input size.

- **Theta Notation (Θ):** Theta notation describes the *average-case* running time of an algorithm when the best and worst case are the same. It provides a *tight bound* on the growth rate.

Think of it this way:

- **Big O (O):** “The algorithm will *never* take longer than this.” (Worst-case scenario)
- **Omega (Ω):** “The algorithm will *always* take at least this long.” (Best-case scenario)
- **Theta (Θ):** “The algorithm will *usually* take this long.” (Average-case scenario)

Back to Linear Search

Therefore we can represent our Linear search with all three notations:

- **Big O:** $O(n)$
- **Omega:** $\Omega(1)$
- **Theta:** Can’t use Theta notation, because the Best and Worst case are different.

7 What Does ‘n’ Really Mean? The Order of Growth

The ‘n’ in $O(n)$, $\Omega(n)$, or $\Theta(n)$ represents the *size of the input*. But what does that *really* mean? It’s about understanding the *order of growth*.

Let’s revisit our `linear_search` example. Suppose we perform the search with a varying number of keys: 10, 20, 50, 100, 1000, and 10000.

In the best case, the number of comparisons is always 1. If we plot this on a graph, we get a horizontal line. The time taken is constant, regardless of the input size.

In the worst case, the number of comparisons is equal to the number of keys. If we plot this on a graph, we get a straight line sloping upwards. The time taken grows linearly with the input size.

That’s what $O(n)$ means: as the input size increases, the running time increases linearly.

If an algorithm had a running time of $O(n^2)$ (quadratic), the time taken would grow much faster as the input size increased.

The order of growth helps us compare algorithms and choose the most efficient one for a particular task.

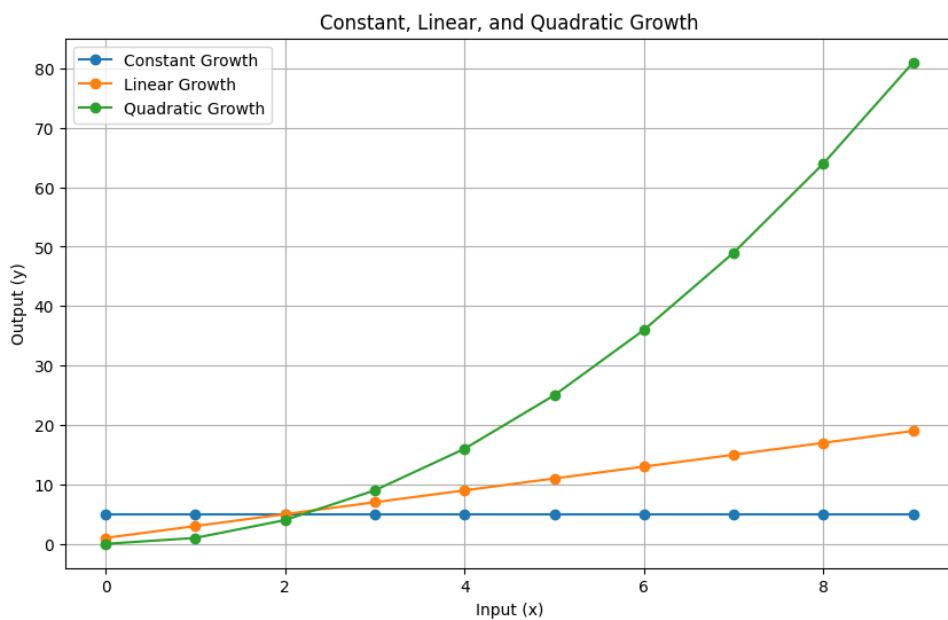


Figure 2: Graphs illustrating constant, linear, and quadratic growth.

Understanding time complexity and asymptotic analysis is crucial for becoming a skilled programmer and AI developer. By learning to evaluate algorithms, you can write code that is not only correct but also efficient and scalable. This knowledge will empower you to tackle complex problems and build innovative solutions.

The 100 Slide AI: A Playbook for AI Application in the Real World

1 Decoding Images: A Sneak Peek into AI in Agriculture

Imagine a farmer standing in a vast field of corn. They want to know how well their crops are growing and whether they need more water or fertilizer. Traditionally, this meant physically inspecting the field, a time-consuming and labor-intensive task. But what if technology could offer a helping hand?

Artificial intelligence, specifically through techniques like image analysis, can revolutionize agriculture. By analyzing images taken from drones or satellites, AI algorithms can automatically assess the health of plants, identify areas affected by disease, and estimate yield potential.

This book will equip you with the foundational knowledge to understand and even build such applications. We'll dive into the core concepts of AI, focusing on practical applications and using Python and Google Colab, two powerful and accessible tools. Get ready to transform raw data into actionable insights!



Figure 1: AI and Robotics in agriculture.

2 Your AI Toolkit: Key Concepts and Foundations

This section introduces the essential building blocks of AI, focusing on concepts directly applicable to real-world problems.

2.1 The Machine Learning Landscape

Machine learning is at the heart of many AI applications. It allows computers to learn from data without explicit programming. There are primarily three types of machine learning:

- **Supervised Learning:** This involves training a model on a dataset where the desired output is already known. Think of it like teaching a child by showing them examples and telling them what each example is.
 - **Classification:** Predicting which category something belongs to. For example, determining if an image contains a cat or a dog. In the agriculture example, classifying a plant image as either corn or rice.
 - **Regression:** Predicting a continuous value. For example, predicting a student’s exam score based on their study habits.
- **Unsupervised Learning:** This involves training a model on a dataset without any labels. The model must discover patterns and structures on its own.
 - **Clustering:** Grouping similar data points together. For example, grouping customers based on their purchasing behavior. In the context of agriculture, this could be grouping areas of a field based on their satellite imagery characteristics.
 - **Dimensionality Reduction:** Reducing the number of variables (dimensions) in a dataset while preserving its essential information.
- **Reinforcement Learning:** This involves training an “agent” to make decisions in an environment to maximize a reward. Think of it like training a dog with treats.

2.2 Ethics in AI

As AI becomes more pervasive, it’s crucial to consider its ethical implications. This includes issues like fairness, transparency, and accountability.



Figure 2: Ethics in AI.

3 Vectors and Vector Spaces: The Language of AI

3.1 What are Vectors?

At the very core of AI and Machine Learning lies the concept of vectors. They are fundamental for representing and processing data in a way that computers can understand. Think of a vector as an arrow pointing in a specific direction with a certain length. In mathematical terms, a vector is an ordered list of numbers.

Any type of data, be it images, videos, text, or numerical data, can be represented as a vector.

- **Images:** An image can be broken down into pixels, and each pixel's color value (red, green, blue) can be represented as a number. All these numbers arranged in a specific order form a vector.
- **Videos:** A video is essentially a sequence of images. Therefore, each frame can be represented as a vector, and the entire video becomes a sequence of vectors.
- **Text:** Text can be converted into numerical representations using techniques like word embeddings. Each word is assigned a vector based on its meaning and context.
- **Tabular Data:** In a spreadsheet, each row can be considered a vector, where each element in the vector corresponds to a value in a specific column.

3.2 Features: The Building Blocks of AI

You'll often hear the terms "vectors" and "features" used interchangeably in machine learning. A feature is a measurable property or characteristic of a phenomenon being observed. Features *are* vectors.

3.3 From Vectors to Matrices

While a vector is a one-dimensional array of numbers, a matrix is a two-dimensional array. A matrix can be thought of as a collection of vectors arranged in rows and columns. This allows for efficient representation and manipulation of data.

4 Dimensionality Reduction: Simplifying Complexity

4.1 The Curse of Dimensionality

Imagine trying to analyze a dataset with thousands of variables. It would be incredibly complex and computationally expensive. This is known as the “curse of dimensionality.” Dimensionality reduction techniques help to overcome this challenge by reducing the number of variables while preserving essential information.

4.2 Principal Component Analysis (PCA): A Powerful Tool

Principal Component Analysis (PCA) is a widely used dimensionality reduction technique. The basic idea behind PCA is to find the “principal components” of the data, which are the directions along which the data varies the most. By projecting the data onto these principal components, we can reduce the dimensionality while retaining most of the variance.

- **Visualizing Variance:** Imagine a cloud of data points scattered in three-dimensional space. PCA aims to find the line along which the points are the most spread out (maximum variance). This line is the first principal component. The second principal component is perpendicular to the first and captures the next highest amount of variance.
- **Projection:** The original data points are then projected onto these principal components, creating a lower-dimensional representation. This is similar to shining a light on the data and capturing its shadow on a lower-dimensional surface.

4.3 Feature Selection vs. Dimensionality Reduction

It’s crucial to understand the difference between feature selection and dimensionality reduction.

- **Feature Selection:** This involves selecting a subset of the original features based on their relevance to the task. For example, if you have 10 features, you might select the 3 most important ones.
- **Dimensionality Reduction (PCA):** This involves creating new features that are combinations of the original features. These new features are the principal components. PCA chooses two “directions of maximum variance” in the data to project the data onto. So PCA reduces the number of parameters by essentially looking at the “shadow” of your parameters onto the new “directions of maximum variance”.

4.4 Practical Application: Image Analysis

Consider a satellite image of a farmland. The image may contain many different spectral bands, each representing a different wavelength of light. These spectral bands can be thought of as features. By applying PCA, we can reduce the number of bands while preserving the most important information, such as the vegetation index. This simplified representation can then be used for tasks like crop classification and monitoring.

5 Next Steps: Clustering and Beyond

This book has laid the groundwork for your AI journey. In the next section, we will explore clustering techniques, another powerful tool for unsupervised learning. Armed with these concepts, you'll be well-equipped to tackle a wide range of AI applications in various domains. Remember, the world of AI is vast and constantly evolving. Continue exploring, experimenting, and building!

Bubble Sort

Minor in AI, IIT Ropar

17th March, 2025

1 Introduction



Imagine you're tasked with organizing a vast library of books by author. Traditionally, this would be a labor-intensive task, but what if technology could simplify the process? This case study explores how the bubble sort algorithm can be used to efficiently sort data, making it a foundational tool in programming.

2 Background: Why Bubble Sort Matters

Bubble sort is a simple yet educational algorithm that has become famous due to its ease of understanding and implementation. Despite being inefficient compared to other sorting algorithms, studying bubble sort helps in grasping worst-case scenarios and improving algorithm design.

3 How Bubble Sort Works: A Step-by-Step Guide

Let's dive into a practical example to illustrate how bubble sort works:

Example Array: 64, 34, 25, 90, 12

Step-by-Step Process

Pass 1

1. **Compare 64 and 34:** Since $64 > 34$, swap them. - Array becomes: [34, 64, 25, 90, 12]

2. **Compare 64 and 25:** Since $64 > 25$, swap them. - Array becomes: [34, 25, 64, 90, 12]
3. **Compare 64 and 90:** Since $64 < 90$, no swap. - Array remains: [34, 25, 64, 90, 12]
4. **Compare 90 and 12:** Since $90 > 12$, swap them. - Array becomes: [34, 25, 64, 12, 90]

Pass 2

1. **Compare 34 and 25:** Since $34 > 25$, swap them. - Array becomes: [25, 34, 64, 12, 90]
2. **Compare 34 and 64:** Since $34 < 64$, no swap. - Array remains: [25, 34, 64, 12, 90]
3. **Compare 64 and 12:** Since $64 > 12$, swap them. - Array becomes: [25, 34, 12, 64, 90]
4. **Compare 64 and 90:** Since $64 < 90$, no swap. - Array remains: [25, 34, 12, 64, 90]

Pass 3

1. **Compare 25 and 34:** Since $25 < 34$, no swap. - Array remains: [25, 34, 12, 64, 90]
2. **Compare 34 and 12:** Since $34 > 12$, swap them. - Array becomes: [25, 12, 34, 64, 90]
3. **Compare 34 and 64:** Since $34 < 64$, no swap. - Array remains: [25, 12, 34, 64, 90]
4. **Compare 64 and 90:** Since $64 < 90$, no swap. - Array remains: [25, 12, 34, 64, 90]

Pass 4

1. **Compare 25 and 12:** Since $25 > 12$, swap them. - Array becomes: [12, 25, 34, 64, 90]
2. **Compare 25 and 34:** Since $25 < 34$, no swap. - Array remains: [12, 25, 34, 64, 90]
3. **Compare 34 and 64:** Since $34 < 64$, no swap. - Array remains: [12, 25, 34, 64, 90]
4. **Compare 64 and 90:** Since $64 < 90$, no swap. - Array remains: [12, 25, 34, 64, 90]

Pass 5

1. **Compare 12 and 25:** Since $12 < 25$, no swap. - Array remains: [12, 25, 34, 64, 90]
2. **Compare 25 and 34:** Since $25 < 34$, no swap. - Array remains: [12, 25, 34, 64, 90]
3. **Compare 34 and 64:** Since $34 < 64$, no swap. - Array remains: [12, 25, 34, 64, 90]
4. **Compare 64 and 90:** Since $64 < 90$, no swap. - Array remains: [12, 25, 34, 64, 90]

Final Sorted Array

[12, 25, 34, 64, 90]

4 Bubble Sort Code

```
def bubble_sort(array):
    n = len(array)
    for i in range(n):
        # Start looking at each item of the list one by one,
        # comparing it with its adjacent value
        for j in range(0, n-i-1):
            # If we find an element that is greater than its adjacent element
            if array[j] > array[j+1]:
                # Swap them
                array[j], array[j+1] = array[j+1], array[j]
```

```

    if array[j] > array[j+1]:
        # Swap them
        array[j], array[j+1] = array[j+1], array[j]
return array

arr = [64, 34, 25, 90, 12]
print("Before sorting:-", arr)
arr = bubble_sort(arr)
print("After sorting:-", arr)
# Output
# Before sorting: [64, 34, 25, 90, 12]
# After sorting: [12, 25, 34, 64, 90]

```

The above code works exactly as the step by step process mentioned above.

5 Optimization: Making Bubble Sort More Efficient

- Basic Optimization: Introduce a flag to stop passes if no swaps occur, reducing unnecessary iterations.
- Impact: Reduces comparisons but maintains $O(n^2)$ time complexity in the worst case.

```

def bubble_sort_modified(array):
    n = len(array)
    for i in range(n):
        # Create a flag that will allow the function to terminate early
        # if there's nothing left to sort
        swapped = False
        # Start looking at each item of the list one by one,
        # comparing it with its adjacent value
        for j in range(0, n-i-1):
            # If we find an element that is greater than its adjacent element
            if array[j] > array[j+1]:
                # Swap them
                array[j], array[j+1] = array[j+1], array[j]
                # Set the flag to True so we'll loop again after this iteration
                swapped = True
        # If no two elements were swapped in inner loop, the list is sorted
        if not swapped:
            break
    return array

arr = [64, 34, 25, 90, 12]
print("Before sorting:-", arr)
arr = bubble_sort_modified(arr)
print("After sorting:-", arr)
# Output
# Before sorting: [64, 34, 25, 90, 12]
# After sorting: [12, 25, 34, 64, 90]

```

In the above code just a little modification to the original bubble sort is done. Here, we have added a swapped variable, if for a pass no swaps were taken place then swapped variable has value False, which in turn would lead to breaking out of the loop and algorithm won't iterate to further as the array has become sorted.

6 Case Analysis: Understanding Performance

- Best Case: Not applicable traditionally, but with optimization, it can be linear if the array is already sorted.
- Worst Case: $O(n^2)$ when the array is reverse sorted.
- Average Case: Generally considered the same as the worst case due to the consistent number of comparisons.

7 Comparison with Other Algorithms

- Selection Sort: Similar time complexity but reduces the number of swaps.
- Python's Sorting Algorithm (Timsort): A hybrid of merge sort and insertion sort, offering better efficiency.

8 Key Takeaways:

- Importance of Intuition: Understanding how algorithms work is crucial for effective programming.
- Efficiency Analysis : Focus on reducing the number of basic operations to improve algorithm efficiency.
- Practical vs. Theoretical Efficiency : There can be a difference between theoretical analysis and real-world performance due to hardware and compiler optimizations.

9 Future Directions: Building on Bubble Sort

- Improving Efficiency: Techniques like divide and conquer can lead to more efficient algorithms.
- Real-World Applications: Understanding algorithmic efficiency is critical for handling large datasets in AI and ML projects.

10 Conclusion

Bubble sort is a foundational algorithm that, despite its inefficiency, provides valuable insights into the principles of sorting and algorithm design. By understanding how bubble sort works and its limitations, developers can better appreciate the importance of efficient algorithms in real-world applications.

Minor in AI

More on Algorithms II

17 March Notes

Programs used:

```
def bubble_sort(num):
    n = len(num)
    for i in range(n-1):
        for j in range(n-i-1):
            if num[j] > num[j+1]:
                num[j], num[j+1] = num[j+1], num[j]
    return num
# Example usage
num = [1, 2, 3, 4, 5, 6, 7, 8]
sorted_list = bubble_sort(num)
print("Sorted array:", sorted_list)
```

```
def bubble_sort(num):
    n = len(num)
    comparisons = 0 # Initialize comparison count
    for i in range(n-1):
        for j in range(n-i-1):
            comparisons += 1
            if num[j] > num[j+1]:
                num[j], num[j+1] = num[j+1], num[j]
    print("Number of comparisons:", comparisons)
    return num

# Example usage
num = num = [1, 2, 3, 4, 5, 6, 7, 8]
sorted_list = bubble_sort(num)
print("Sorted array:", sorted_list)
```

```

def bubble_sort(num):
    n = len(num)
    comparisons = 0 # Initialize comparison count
    for i in range(n - 1):
        swapped = False # Flag to track swapping
        for j in range(n - i - 1):
            comparisons += 1
            if num[j] > num[j + 1]:
                num[j], num[j + 1] = num[j + 1], num[j]
                swapped = True # A swap occurred
        if not swapped:
            break
    print("Number of comparisons:", comparisons)
    return num

# Example usage
num = [1, 2, 3, 4, 5, 6, 7, 8]
sorted_list = bubble_sort(num)
print("Sorted array:", sorted_list)

```

Selection Sort

```

def selection_sort(num):
    n = len(num)
    for i in range(n):
        min_index = i
        for j in range(i + 1, n):
            if num[j] < num[min_index]:
                min_index = j
        num[i], num[min_index] = num[min_index], num[i]
    return num
# Example usage
num = [64, 25, 12, 22, 11]
sorted_list = selection_sort(num)
print("Sorted array:", sorted_list)

```

Minor in AI

Hypothesis Testing with Z-Test

1 The Chocolate Factory Dilemma

Real-World Motivation

A renowned chocolate factory markets its bars as "precisely 50g." During routine quality checks, you randomly select 10 bars and find an average weight of 52g with a standard deviation of 2g. **Is this 2g difference meaningful**, or could it occur by random chance in a properly calibrated production line?

Why This Matters:

- **Cost Implications:** Halting production for adjustments costs \$20,000/hour
- **Regulatory Compliance:** Products must stay within $\pm 3\text{g}$ of claimed weight
- **Brand Trust:** Consistent weight maintains customer loyalty

Hypothesis Testing Approach:

1. **Define Thresholds:** Establish a 95% confidence level (5% error tolerance)
2. **Quantify Uncertainty:** Calculate how much variation is expected from random sampling
3. **Statistical Proof:** Determine if 52g is statistically different from 50g using the Z-test

Key Insight: A 2g difference in a small sample ($n=10$) carries more uncertainty than in larger samples. The Z-test helps distinguish *meaningful discrepancies* from *random fluctuations* using probability theory.

2 ABC of Z-Testing

2.1 Statistical Hypotheses

Every test begins with two mutually exclusive claims:

- **Null Hypothesis (H_0):** Assumes no effect/difference
(Factory claim: $\mu = 50\text{g}$)
- **Alternative Hypothesis (H_1):** Challenges the status quo
(Actual mean $\neq 50\text{g}$)

The Z-test mathematically compares these using sample data. A key requirement is either:

- Large sample size ($n \geq 30$), or
- Known population standard deviation

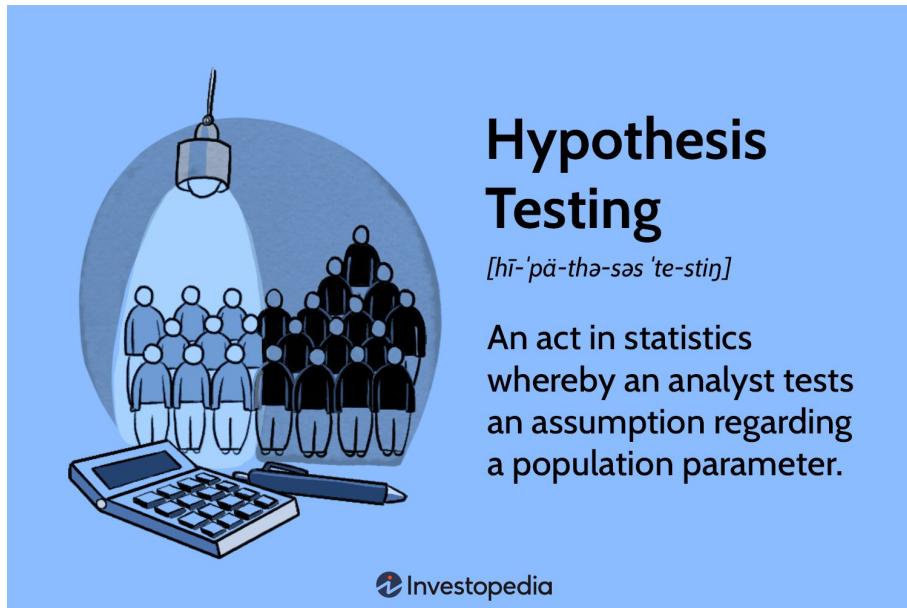


Figure 1: Source: Investopedia, *Hypothesis testing*.

2.2 The Z-Score Formula

Central Equation

$$Z = \frac{\bar{X} - \mu}{\sigma/\sqrt{n}}$$

Where:

- \bar{X} : Observed sample mean (52g)
- μ : Hypothesized population mean (50g)
- σ : Standard deviation (2g)
- n : Number of observations (10)

This Z-score represents how many standard errors the sample mean deviates from the claimed mean. Larger absolute values indicate stronger evidence against H_0 .

2.3 Decision Framework

Hypothesis testing follows a structured workflow:

1. **Define Significance Level (α)**: Typically 5% (0.05)
2. **Calculate Critical Value**: $Z = \pm 1.96$ for 95% confidence
3. **Compute Test Statistic**: Z-score from sample data
4. **Compare and Conclude**:
 - Reject H_0 if $|Z| >$ Critical Value
 - Fail to reject H_0 otherwise

3 Getting Hands-In

3.1 Visualizing Critical Regions

The following Python code visualizes critical regions in a normal distribution using the SciPy and Matplotlib libraries. It plots the sampling distribution and highlights the rejection zones based on a 95% confidence level.

Listing 1: Distribution Plotting in Python

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.stats import norm
4
5 # Parameters
6 population_mean = 50
7 sample_std = 2
8 sample_size = 10
9
10 # Calculations
11 standard_error = sample_std / np.sqrt(sample_size)
12 critical_z = 1.96 # 95% confidence level
13 lower_bound = population_mean - critical_z * standard_error
14 upper_bound = population_mean + critical_z * standard_error
15
16 # Visualization
17 x = np.linspace(46, 54, 500)
18 y = norm.pdf(x, population_mean, standard_error)
19 plt.figure(figsize=(10,4))
20 plt.plot(x, y, label='Sampling Distribution')
21 plt.axvline(lower_bound, color='red', linestyle='--',
22              label='Rejection Boundary')
23 plt.axvline(upper_bound, color='red', linestyle='--')
24 plt.fill_between(x, y, where=(x < lower_bound)|(x > upper_bound),
25                   color='orange', alpha=0.3, label='Rejection Zone')
26 plt.legend()
27 plt.title('Normal Distribution with 95% Confidence Interval')
28 plt.xlabel('Sample Mean (grams)')
29 plt.ylabel('Probability Density')
30 plt.show()
```

3.1.1 Code Explanation

This code snippet visualizes a normal distribution along with critical regions by computing confidence intervals and plotting them. Below is a breakdown of each section:

- **Importing Libraries:** The necessary libraries such as NumPy, Matplotlib, and SciPy are imported for numerical computation and visualization.
- **Defining Parameters:**
 - $\mu = 50$ (Population Mean)
 - $\sigma = 2$ (Sample Standard Deviation)
 - $n = 10$ (Sample Size)

- **Computing Standard Error:**

$$SE = \frac{\sigma}{\sqrt{n}}$$

This gives an estimate of the variability of the sample mean.

- **Determining Critical Values:** Using the standard normal table, the critical z-score for a 95% confidence level is 1.96. The lower and upper rejection boundaries are calculated as:

$$\text{Lower Bound} = \mu - Z_{critical} \times SE$$

$$\text{Upper Bound} = \mu + Z_{critical} \times SE$$

- **Plotting the Distribution:**

- The probability density function (PDF) of the normal distribution is plotted.
- Vertical dashed red lines indicate the rejection boundaries.
- The rejection regions (where the sample mean is unlikely to fall) are shaded in orange.
- **Displaying the Plot:** The final plot provides a visual representation of the sampling distribution and critical regions, helping in hypothesis testing.

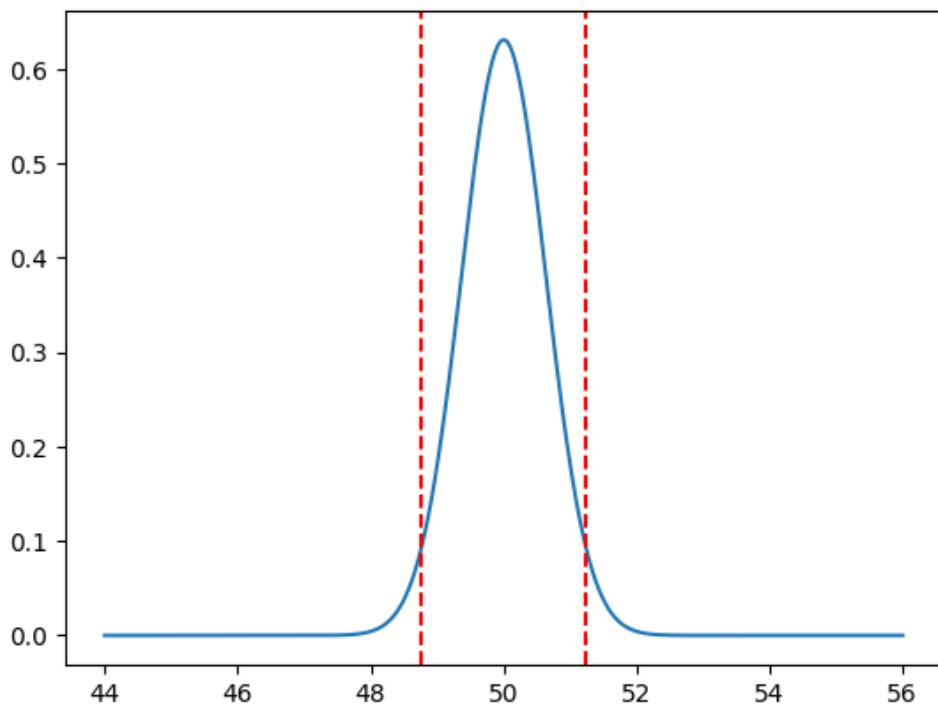


Figure 2: Source: Investopedia, Visual guide to decision boundaries: Sample means beyond red lines (51.24g/48.76g) would reject H_0

3.2 Automated Hypothesis Testing

The following Python code performs a Z-test for hypothesis testing. It calculates the Z-score and compares it with a critical value to determine whether to reject the null hypothesis (H_0).

Listing 2: Python Z-Test Code

```
1 from scipy.stats import norm
2
3 def z_test(pop_mean, sample_mean, sample_std, n, alpha=0.05):
4     # Standard error calculation
5     se = sample_std / (n ** 0.5)
6
7     # Z-score computation
8     z = (sample_mean - pop_mean) / se
9
10    # Critical value (two-tailed)
11    critical_z = norm.ppf(1 - alpha/2)
12
13    # Decision logic
14    if abs(z) > critical_z:
15        print(f"Reject H0 (Z = {z:.2f}, Critical Z = +/-{critical_z:.2f} )")
16    else:
17        print(f"Fail to reject H0 (Z = {z:.2f})")
18
19    return z
20
21 # Chocolate factory test
22 z_test(pop_mean=50, sample_mean=52, sample_std=2, n=10)
```

3.2.1 Code Explanation

This function automates hypothesis testing using the Z-test, a statistical test to determine if a sample mean significantly differs from a population mean.

- **Importing SciPy's Norm:** The function uses `norm.ppf()` from SciPy to compute the critical Z-value.

- **Function Parameters:**

- `pop_mean`: Population mean (μ)
- `sample_mean`: Sample mean (\bar{x})
- `sample_std`: Sample standard deviation (s)
- `n`: Sample size
- `alpha`: Significance level (default = 0.05 for a 95% confidence level)

- **Standard Error Calculation:**

$$SE = \frac{s}{\sqrt{n}}$$

The standard error measures the variability of the sample mean.

- **Z-Score Calculation:**

$$Z = \frac{\bar{x} - \mu}{SE}$$

The Z-score measures how many standard errors the sample mean is away from the population mean.

- **Critical Value Calculation:** The critical value for a two-tailed test at 95% confidence is obtained using:

$$Z_{\alpha/2} = \text{norm.ppf}(1 - \alpha/2)$$

This represents the threshold beyond which the null hypothesis is rejected.

- **Decision Rule:**

- If $|Z| > Z_{\alpha/2}$, reject H_0 (significant difference).
- Otherwise, fail to reject H_0 (no significant difference).

- **Example Application:** The function is tested with a chocolate factory scenario where:

- $\mu = 50$ (Expected weight of a chocolate bar)
- $\bar{x} = 52$ (Observed sample mean)
- $s = 2$ (Sample standard deviation)
- $n = 10$ (Sample size)

The function determines if the observed sample mean significantly differs from the expected weight.

4 Real-World Applications

Case Studies

1. Pharmaceutical Quality Control

A drug manufacturer claims tablets contain 100mg active ingredient. 50 tablets average 98mg ($SD=5\text{mg}$).

Z-test reveals $Z = -2.83$ ($p=0.0047$) - reject H_0 , indicating underdosing.

2. E-Commerce A/B Testing

Website A: 12% conversion rate (10,000 visitors)

Website B: 13% conversion rate (10,000 visitors)

$Z = 5.0$ ($p < 0.0001$) - statistically significant improvement.

3. Clinical Research

New treatment reduces recovery time from 14 to 13 days ($SD=2$ days, $n=100$).

$Z = -5.0$ shows strong evidence for treatment efficacy.

5 Key Insights

Essential Takeaways

- **Confidence vs Significance:** 95% confidence level corresponds to 5% significance ($\alpha = 0.05$).
- **Sample Size Sensitivity:** Larger samples increase test power but require smaller differences to reject H_0 .
- **Error Types:**
 - Type I: False positive (rejecting true H_0)
 - Type II: False negative (failing to reject false H_0)
- **Assumptions:** Normally distributed data or large sample (Central Limit Theorem).

References

- **Figure 1 Source:** Investopedia¹
- **Figure 2 Source:** Investopedia²
- **Z-Test Implementation:** [Colab Notebook Link](#)

¹Retrieved from <https://www.investopedia.com/terms/h/hypothesistesting.asp>.

²Retrieved from <https://www.investopedia.com/terms/h/hypothesistesting.asp>.

Foundations of Python for AI: Revisiting Data Types, Conversions, and Operators Through a Case Study

Minor In AI, IIT Ropar

19th March, 2025

Welcome to the wonderful world of Python and Artificial Intelligence!

This module is designed for beginners, aiming to gently introduce you to the foundational concepts of Python programming that are crucial for your journey into AI. We'll explore data types, conversions, and operators, not just with abstract definitions, but through an engaging case study that will make learning fun and intuitive.

1 The Alien Message

Imagine this: An alien is trying to communicate with us. Not with advanced technology or complex equations, but with a series of seemingly random numbers: **73, 80, 76**.

What could these numbers possibly mean? Is it a secret code? A countdown to something? Maybe it's directions to their home planet!

This, my friend, is our first AI challenge. We will use Python to "decode" this message, revealing the hidden meaning and, along the way, solidify our understanding of Python's basic building blocks.



Figure 1: "Random fact: Gnorts, Mr. Alien" backwards is "Neil Armstrong"!!

2 Understanding Data Types

Before we crack the alien's code, we need to understand how Python organizes information. These are called **data types**. Think of them as different containers for storing different kinds of things.

2.1 Integers (int)

Integers are whole numbers, like -2, 0, 73, 80, or 76. They're perfect for representing counts, quantities, or any value that doesn't require decimals.

In Python, we can assign these numbers to variables:

```
1 m1 = 73
2 m2 = 80
3 m3 = 76
```

Here, `m1`, `m2`, and `m3` are variables holding integer values.

2.2 Booleans (bool)

Booleans represent truth values: `True` or `False`. They're essential for decision-making in programs. Imagine asking a question: "Is the number greater than 50?" The answer will always be either `True` or `False`.

```
1 is_greater = m1 > 50 # is_greater will be True
2 print(is_greater)
```

A fascinating thing about booleans is that Python understands how to work with them and numbers as well. Let's see what happens when we convert our numbers to booleans.

```
1 print(bool(73))
2 print(bool(0))
3 print(bool(-1))
```

All the numbers convert to `True` when converted to boolean, except for 0, which results in `False`.

2.3 Floats (float)

Floats are numbers with decimal points, like 3.14, -2.5, or 73.0. They're useful for representing measurements, percentages, or any value that requires precision.

2.4 Strings (str)

Strings are sequences of characters, enclosed in single quotes (') or double quotes (""). They are used to represent text, names, sentences, or any combination of characters.

```
1 name = "Alien"
2 message = 'Hello, Earth!'
```

One peculiar thing about an empty string is that it acts as `False` when converted to boolean.

```
1 print(bool("alien"))
2 print(bool(""))
```

3 Type Conversion

Sometimes, we need to change the data type of a value. This is called **type conversion**. Python provides built-in functions for this.

- `int()`: Converts a value to an integer.
- `float()`: Converts a value to a float.

- `str()`: Converts a value to a string.
- `bool()`: Converts a value to a boolean.
- `chr()`: Converts an integer to its corresponding character (based on ASCII).

Back to our alien message! Let's see what happens if we convert our numbers to Booleans:

```
1 print(bool(m1)) # True
2 print(bool(m2)) # True
3 print(bool(m3)) # True
```

True, True, True... Doesn't seem to reveal much, does it? Let's try converting them to floats:

```
1 print(float(m1)) # 73.0
2 print(float(m2)) # 80.0
3 print(float(m3)) # 76.0
```

Still not helpful! This is where `chr()` comes in. `chr()` converts an integer to a corresponding character. There are codes (also known as ASCII codes) for every character that's used on the computer. Let's take a look at what `chr` does.

```
1 print(chr(m1))
2 print(chr(m2))
3 print(chr(m3))
```

The output will be:

Output

```
I
P
L
```

Aha! Suddenly, the alien message is clear. The numbers 73, 80, and 76 represent the letters I, P, and L. Could the aliens be trying to tell us something? Maybe they are into *IPL*.

4 Operators

Operators are symbols that perform operations on values. They are essential for manipulating data and performing calculations.

4.1 Arithmetic Operators

These are the basic operators for performing mathematical calculations:

- `+` (Addition)
- `-` (Subtraction)
- `*` (Multiplication)
- `/` (Division)
- `//` (Floor Division - returns the integer part of the division)
- `**` (Exponentiation - raises a number to a power)
- `%` (Modulo - returns the remainder of a division)

```
1 a = 10
2 b = 3
3
4 print(a + b)    # 13
5 print(a / b)    # 3.3333333333333335
6 print(a // b)   # 3
7 print(a ** b)   # 1000
```

4.2 String Concatenation

The + operator can also be used to join strings together. This is called string concatenation.

```
1 str1 = "Hello"
2 str2 = "Alien"
3 combined_string = str1 + " " + str2  # "Hello Alien"
4 print(combined_string)
```

One thing to remember here is that you cannot add a string and a number directly, and it can result in an error if you're not careful. The other way of concatenating a string is as follows:

```
1 num = 2024
2 print("Alien " + str(num))
```

4.3 String Repetition

The * operator can be used to repeat a string multiple times.

```
1 alien_repeat = "Alien" * 5  # "AlienAlienAlienAlienAlien"
2 print(alien_repeat)
```

4.4 The join() method

The *join()* method of the string class can be very powerful to concatenate iterables such as lists, tuples and sets. See the examples below.

```
1 str_list = ["hello", "alien"]
2 print(" ".join(str_list))
3
4 str_tuple = ("hello", "alien")
5 print(" ".join(str_tuple))
```

5 Back to the Message - String Concatenation

Let's say we want to combine the IPL. How can we do that?

```
1 m1 = chr(73)
2 m2 = chr(80)
3 m3 = chr(76)
4
5 alien_message = m1 + m2 + m3
6 print(alien_message) # Output: IPL
```

The message is still IPL, but this time we can confirm the characters are concatenated.

6

Congratulations! You've taken the first steps in your Python for AI journey. We learned about data types, conversions, and operators, and used them to decode a secret message from an alien! This is just the beginning. As you continue your journey, you'll discover even more powerful tools and techniques for solving complex AI problems. Remember to have fun and stay curious. There's a whole universe of knowledge waiting to be explored!


```

# 2. int() - Convert to integer
print("Using int():")
print(int(num_str))           # str to int → 100
print(int(float_num))         # float to int (truncates)
print(int(True))              # bool to int → 1
print(int(False))             # bool to int → 0
print()

# 3. bool() - Convert to boolean
print("Using bool():")
print(bool(zero))             # int 0 to bool → False
print(bool(10))               # int 10 to bool → True
print(bool(""))                # empty string to bool → False
print(bool(non_empty_string)) # non-empty string to bool → T
print()

# Additional Example: Typecasting during arithmetic
print("Arithmetic Example with Typecasting:")
result = int("20") + 5
print("int('20') + 5 =", result)

```

```

# Arithmetic Operations on int and float

# Declare int and float variables
a = 10          # int
b = 3           # int
x = 5.5         # float
y = 2.0         # float

print("Arithmetic Operations on int:")
print(a, "+", b, "=", a + b)      # Addition
print(a, "-", b, "=", a - b)      # Subtraction
print(a, "*", b, "=", a * b)      # Multiplication
print(a, "/", b, "=", a / b)       # Division (float result)
print(a, "**", b, "=", a ** b)     # Exponentiation
print(a, "//", b, "=", a // b)     # Floor Division
print()

print("Arithmetic Operations on float:")
print(x, "+", y, "=", x + y)      # Addition
print(x, "-", y, "=", x - y)      # Subtraction
print(x, "*", y, "=", x * y)      # Multiplication
print(x, "/", y, "=", x / y)       # Division
print(x, "**", y, "=", x ** y)     # Exponentiation
print(x, "//", y, "=", x // y)     # Floor Division
print()

```

```
print("Mixed int and float operations:")
print(a, "+", x, "=", a + x)          # int + float
print(b, "*", y, "=", b * y)          # int * float
print(a, "/", y, "=", a / y)          # int / float
print(x, "**", b, "=", x ** b)        # float ** int
```

```
# Define strings
str1 = "Hello"
str2 = "World"

# Method 1: Using + operator
result1 = str1 + " " + str2
print("Using + operator:", result1)

# Method 2: Using join()
result2 = " ".join([str1, str2])
print("Using join():", result2)

# Method 3: Using * operator for repetition
repeat_str = str1 * 3
print("Using * operator for repetition:", repeat_str)

# Method 4: Concatenating with numbers (after type
# conversion)
num = 2025
result3 = str1 + " " + str(num)
print("Concatenating string and number:", result3)
```

Minor in AI

Decision Making in Programming



1 From Rides to Rules

Imagine planning a thrilling day at an amusement park where every decision is critical to ensure safety and maximum enjoyment. The park, with its dazzling rides and vibrant atmosphere, sets strict rules—age limits, height restrictions, and exclusive privileges for VIP members—to manage the crowd efficiently. Today, we dive into how these real-life conditions inspire decision-making in programming. By drawing parallels between an amusement park's entry rules and code logic, we unveil how thoughtful programming transforms complex decision processes into clear, efficient, and engaging code.

Practical Side of Conditionals

The amusement park example is not just a classroom exercise; it represents everyday challenges where multiple conditions must be evaluated swiftly and accurately. As we explore these concepts, think about how the same logic applies to systems you encounter daily—be it securing online transactions or managing smart home devices.

2 Deep Dive into Decision Making

At the heart of our discussion is the challenge of determining whether someone meets all the criteria to enter the park. This involves setting up rules such as checking if a visitor meets the minimum age, ensuring they have the required height (unless they are a VIP), and verifying there are no health issues that could make a ride unsafe.

The solution is to create a Python function—aptly named `can_enter`—that takes parameters like age, height, VIP status, and health condition. The function then uses a sequence of `if-else` statements to evaluate each condition in a logical, and efficient order. This not only simulates the decision process but also teaches us the importance of writing code that is both intuitive and optimized.

Understanding the can_enter Function

The Python function `can_enter` determines whether a person is allowed to enter a park based on multiple conditions. These include the visitor's age, height, VIP status, and any existing health issues. The function returns `True` only if all the entry criteria are satisfied.

Function Definition

```
1 def can_enter(age, height, is_vip, has_health_issue):
2     if age < 12:
3         return False
4     if height < 140 and not is_vip:
5         return False
6     if has_health_issue:
7         return False
8     return True
9
10 print(can_enter(15, 145, False, False)) # Expected output: True
```

Listing 1: Python function to evaluate park entry conditions

Parameter Descriptions

- `age` – The age of the visitor (in years).
- `height` – The height of the visitor (in centimeters).
- `is_vip` – A boolean indicating whether the visitor has VIP status.
- `has_health_issue` – A boolean indicating whether the visitor has any health issues.

Logical Flow of the Function

1. **Age Check:** The function first checks if the visitor's age is less than 12. If so, they are immediately disqualified.

`if age < 12 ⇒ return False`

2. **Height Check (VIP Exception):** Next, it verifies if the visitor is at least 140 cm tall. However, if the visitor is a VIP, the height requirement is waived.

`if height < 140 and not is_vip ⇒ return False`

3. **Health Check:** The function then ensures the visitor does not have any health issues. If `has_health_issue` is `True`, entry is denied.

`if has_health_issue ⇒ return False`

4. **All Conditions Met:** If none of the above checks fail, the visitor is allowed to enter.

`return True`

Example Execution

```
can_enter(15, 145, False, False)
```

- Age = 15, $15 \geq 12 \Rightarrow$ Pass
- Height = 145, $145 \geq 140 \Rightarrow$ Pass (VIP not needed)
- No health issues \Rightarrow Pass

Result: True — The person is allowed to enter the park.

This function ensures a clear and readable structure, with early exits (`return False`) improving efficiency by avoiding unnecessary checks once a condition fails.

3 Expanding Horizons with Random Data and Visualization

After mastering decision-making with conditional statements, we expand into the realm of data manipulation and visualization. Using Python's `random` library, we generate a list of random numbers to simulate data points—a concept fundamental in machine learning and statistical analysis.

This exercise isn't just about generating numbers; it is about categorizing data into meaningful groups. By separating the generated scores into high and low counts, we lay the groundwork for visual representation using the popular `matplotlib` library. The creation of a scatter plot provides a visual understanding of the distribution of these scores, reinforcing the connection between raw data and its analytical representation.

The Code in Action

Consider the following snippet:

```
1 import random
2 import matplotlib.pyplot as plt
3
4 # Generate a list of 50 random scores between 0 and 100
5 random_scores = [random.randint(0, 100) for _ in range(50)]
6
7 # Separate the scores into high (>=50) and low (<50)
8 high_scores = [score for score in random_scores if score >= 50]
9 low_scores = [score for score in random_scores if score < 50]
10
11 # Plot the scores using a scatter plot for visual analysis
12 plt.scatter(range(len(random_scores)), random_scores, c='blue', label='Scores')
13 plt.xlabel('Index')
14 plt.ylabel('Score')
15 plt.title('Scatter Plot of Random Scores')
16 plt.legend()
17 plt.show()
```

Listing 2: Generating random scores and visualizing them

Explanation of the Code in Action

This Python code demonstrates how to combine random number generation, conditional filtering, and data visualization using a scatter plot. The objective is to simulate and visually analyze a set of randomly generated scores.

1. Importing Required Libraries

- `random` – Used to generate random integers.
- `matplotlib.pyplot` – A plotting library used for visualizing data.

2. Generating Random Scores

```
random_scores = [random.randint(0, 100) for _ in range(50)]
```

This line creates a list of 50 random integers between 0 and 100, simulating test scores or performance values.

3. Categorizing the Scores

```
high_scores = [score for score in random_scores if score >= 50]
low_scores = [score for score in random_scores if score < 50]
```

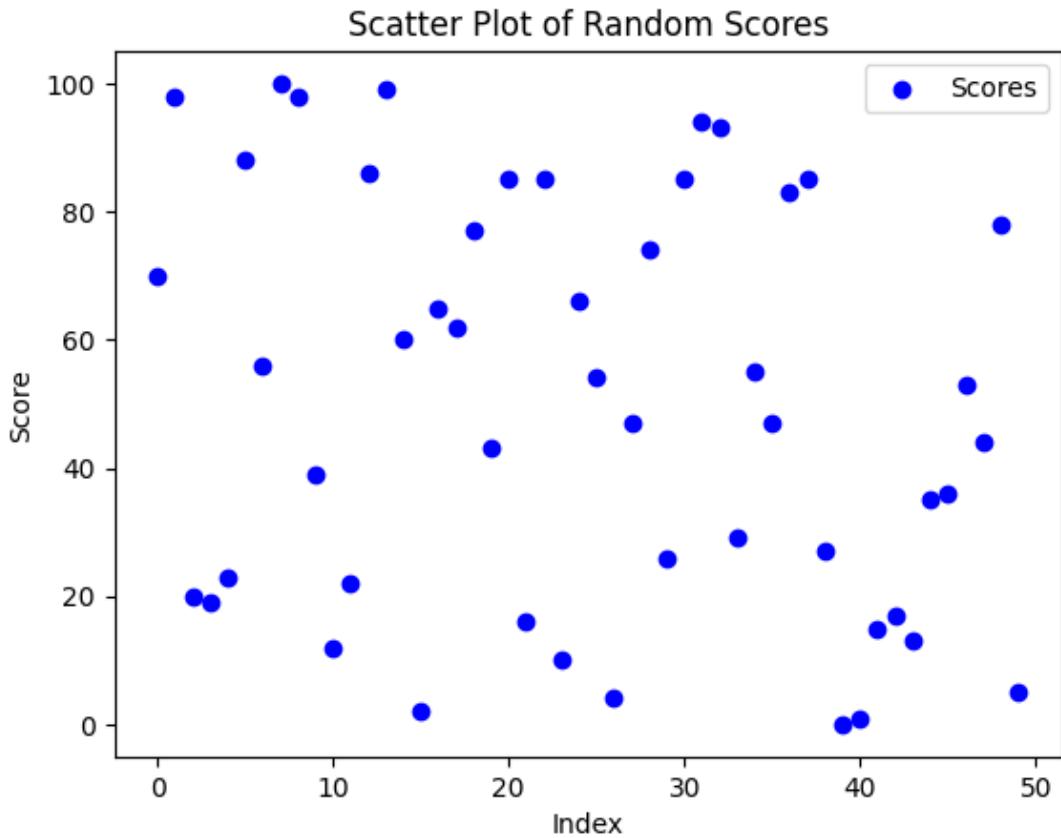
- Scores ≥ 50 are considered **high**.
- Scores < 50 are considered **low**.

These lines use list comprehensions to filter and separate the scores based on the threshold value of 50.

4. Visualizing with a Scatter Plot

```
plt.scatter(range(len(random_scores)), random_scores, c='blue', label='Scores')
plt.xlabel('Index')
plt.ylabel('Score')
plt.title('Scatter - Plot - of - Random - Scores')
plt.legend()
plt.show()
```

- `plt.scatter(...)` creates a scatter plot with:
 - The x-axis representing the index (position) of each score in the list.
 - The y-axis representing the score value.
- `plt.xlabel`, `plt.ylabel`, and `plt.title` add context to the plot.
- `plt.legend()` adds a legend to identify the data series.
- `plt.show()` displays the generated plot.



Conclusion

This code block demonstrates how decision-making (splitting high and low scores) and visualization (scatter plot) can help uncover patterns in random data. While numbers alone may seem abstract, visual representation helps tell a more meaningful story.

This exercise showcases how the combination of decision-making logic and data visualization can transform abstract numbers into a story that is both insightful and visually appealing.

4 Conclusion

A journey through the essentials of decision-making in programming, inspired by the dynamic environment of an amusement park. We discovered that by structuring code logically and using clear, meaningful variable names, we can mirror real-world decision processes in our programs.

Furthermore, the exploration of random data generation and visualization demonstrated how programming bridges the gap between raw data and its interpretation, an essential skill in the realm of artificial intelligence. Remember, the true beauty of programming lies in its ability to simplify complexity and create elegant solutions that are as efficient as they are understandable.

Key Takeaways

Through today's engaging exercises, we learned:

- How to implement decision-making logic using conditional statements in Python.
- The importance of ordering conditions to create efficient, readable code.
- Practical techniques for generating and categorizing random data.
- The power of data visualization in making sense of numerical information.

Embrace these insights as the foundation for more advanced explorations in AI and programming.

Minor in AI

20 March 2025

Revision - Mastering Input and Decision-Making in Python: From Basics to Logic

```
# Variables
# age (int)
# height (int, in cm)
# is_accompanied (bool)
# is_vip (bool)
# has_heart_condition (bool)
# is_banned (bool)

def can_enter_ride():
    if has_heart_condition:
        print("Entry Denied: Health risk due to heart condition.")
    elif is_vip and not is_banned:
        print("Entry Allowed: VIP access granted.")
    elif is_vip and is_banned:
        print("Entry Denied: VIP banned from rides.")
    elif age >= 12 and height >= 140:
        if 12 <= age <= 15 and not is_accompanied:
            print("Entry Denied: Ages 12-15 must be accompanied.")
        else:
            print("Entry Allowed: Meet age and height requirements.")
    else:
        print("Entry Denied: Does not meet age or height
requirements.")

age = 13
height = 145
is_accompanied = True
is_vip = False
has_heart_condition = True
is_banned = False
can_enter_ride()
```

```

import random
import matplotlib.pyplot as plt
# Initialize counters and totals
low_count = 0
low_total = 0
average_count = 0
average_total = 0
high_count = 0
high_total = 0
scores = []

# Generate and process 100 random numbers
for i in range(100):
    num = random.randint(1, 100)
    score = num // 10
    scores.append(score)

    if score <= 4:
        low_count += 1
        low_total += score
    elif score <= 7:
        average_count += 1
        average_total += score
    else:
        high_count += 1
        high_total += score

if low_count > 0:
    low_mean = low_total / low_count
else:
    low_mean = 0

if average_count > 0:
    average_mean = average_total / average_count
else:
    average_mean = 0

if high_count > 0:
    high_mean = high_total / high_count
else:
    high_mean = 0

# Simple prints
print("Low Bucket (0-4):", low_count, "scores, Mean =", round(low_mean, 2))
print("Average Bucket (5-7):", average_count, "scores, Mean =", round(average_mean, 2))
print("High Bucket (8-10):", high_count, "scores, Mean =", round(high_mean, 2))

plt.scatter(range(100), scores, color='blue')
plt.title("Scatter Plot of Scores")
plt.show()

```

Python Data Structures: A Practical Revision

Functions, Lists, and Tuples for Robotics and Machine Learning Engineers

Minor in AI, IIT Ropar
21st March, 2025

Welcome, budding AI and Machine Learning Engineers! This document is your friendly guide to mastering essential Python data structures – functions, lists, and tuples – crucial building blocks for your AI journey. We'll learn through hands-on examples and practical exercises.

1 Functions - Your Code's Superpowers!

Imagine you are working in the AI industry and you are asked to develop autonomous shuttles that would identify different routes, and for each route it would have to perform calculations for the distance, traffic and time it would take to reach the other end. You might need to repeat these calculations multiple times. Instead of writing the same set of instructions again and again, wouldn't it be great if you could bundle them into a single unit, like a mini-program within your program? That's precisely what a function allows you to do!

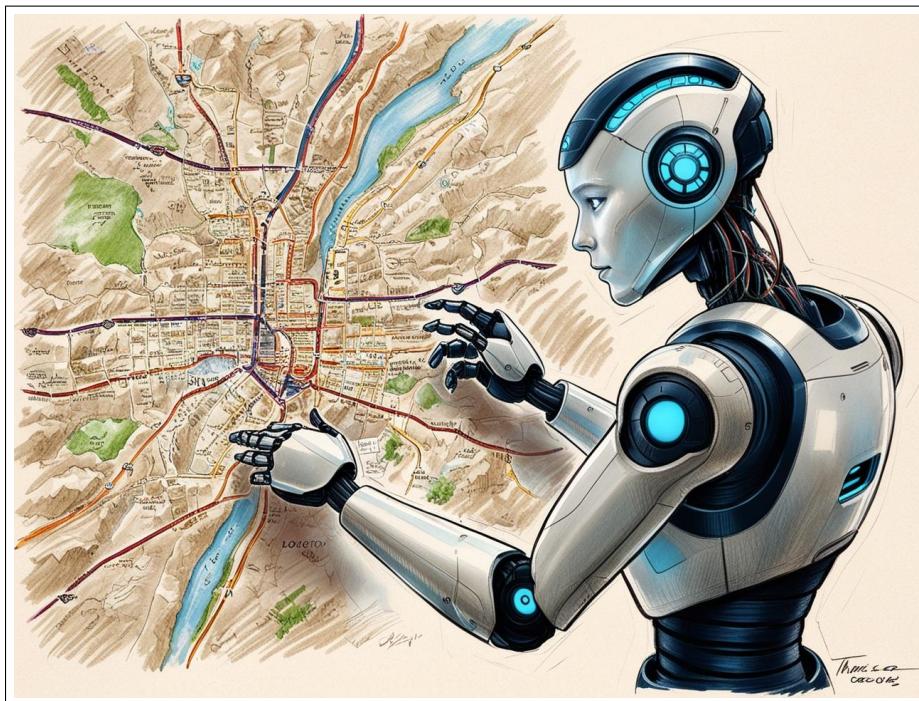


Figure 1: Building AI agents is a goal within your reach!

1.1 What is a Function?

Think of a function as a recipe. It takes ingredients (inputs), performs a series of steps (code), and delivers a dish (output).

- **A function is a block of organized, reusable code that performs a specific task.** This prevents redundancy, improves code readability, and makes your programs more manageable.

1.2 Defining a Function

In Python, we use the `def` keyword to define a function, followed by the function name, parentheses `()`, and a colon `:`. The code block within the function is indented.

```

1 def greet(): #Definition of the function
2     print("Hello, world!") #Main body of the function

```

1.3 Calling a Function

Defining a function only creates it. To actually execute the code inside, you need to *call* the function by simply writing its name followed by parentheses:

```

1 greet() # Calling the function
2 #Output: Hello, world!

```

1.4 Function Arguments

Functions can accept inputs called *arguments*. These arguments are specified within the parentheses during the function definition.

```

1 def greet(name):
2     print("Hello, ", name)
3
4 greet("Alice") #Output: Hello, Alice
5 greet("Bob") #Output: Hello, Bob

```

In this example, `name` is an argument to the function `greet`. When we call the function with `greet("Alice")`, the value "Alice" is passed as the argument.

```

1 def add_numbers(number1, number2):
2     print(number1 + number2)
3
4 add_numbers(5, 6) #Output: 11
5 add_numbers(10, 20) #Output: 30

```

- **Parameters** are the variables listed inside the parentheses in the function definition.
- **Arguments** are the actual values passed to the function when it is called.

1.5 The return Statement

Functions can also *return* a value. The `return` statement exits the function and passes back a value to the caller.

```

1 def square(number):
2     result = number * number
3     return result
4
5 output = square(5)
6 print(output) #Output: 25

```

Important points to remember

1. You are not able to access the variable `result` outside the function, which means that the scope of this variable is local.
2. The `return` statement has to be the last line of the function.

1.6 The pass Statement

The `pass` statement is a placeholder. It does nothing. You can use it when you need a function definition but don't have any code to put inside yet.

```

1 def my_function():
2     pass # Placeholder, function does nothing
3
4 my_function() # the function will not return any error

```

1.7 Built-in Functions

Python provides many built-in functions that are always available. Examples include `print()`, `len()`, `sqrt()`, `pow()`, and many more.

```

1 import math
2
3 print(math.sqrt(16)) #Output: 4.0
4 print(pow(2, 3)) #Output: 8

```

Note

Some functions require you to import the relevant module (like `math` for `sqrt`).

1.8 Default Arguments

You can specify default values for function arguments. If the caller doesn't provide a value for that argument, the default value is used.

```

1 def greet(name, message="Good morning!"):
2     print(name + ", " + message)
3
4 greet("Alice") #Output: Alice, Good morning!
5 greet("Bob", "Good evening!") #Output: Bob, Good evening!

```

1.9 Recursive Functions

A *recursive function* is a function that calls itself. This can be useful for solving problems that can be broken down into smaller, self-similar subproblems.

Example: Factorial

The factorial of a number n (denoted as $n!$) is the product of all integers from 1 to n . For example, $5! = 5 * 4 * 3 * 2 * 1 = 120$.

```

1 def factorial(number):
2     if number == 1:
3         return 1
4     else:
5         return number * factorial(number - 1)
6
7 print(factorial(6)) #Output: 720

```

Explanation:

- The function checks if the number is 1. If so, it returns 1 (base case).

- Otherwise, it returns the number multiplied by the factorial of the number minus 1. This recursive call breaks the problem down into smaller subproblems until it reaches the base case.

2 Lists - Your Versatile Data Container!

Imagine you are working in an AI project that involves collecting and analyzing data about a group of students. You need to store information like their names, ages, and courses they are enrolled in. Using separate variables for each piece of information would be cumbersome. That's where *lists* come in handy!

2.1 What is a List?

- A list is an ordered, mutable (changeable) collection of items. It allows you to store multiple values within a single variable.

2.2 Creating a List

Lists are created using square brackets [], with items separated by commas.

```
1 ages = [25, 28, 42, 30]
2 names = ["Alice", "Bob", "Charlie"]
```

You can also use the `list()` constructor:

```
1 fruits = list(("apple", "banana", "cherry")) # note the double round-brackets
```

Lists can contain items of different data types:

```
1 student_data = ["Jack", 20, "Computer Science", [90, 95, 85]]
```

To create an empty list, simply use empty square brackets:

```
1 empty_list = []
```

2.3 List Characteristics

- **Ordered:** The items have a defined order, and that order is maintained.
- **Mutable:** You can change, add, or remove items after the list is created.
- **Allows Duplicates:** Lists can contain multiple items with the same value.

2.4 Accessing List Items

Each item in a list has an index, starting from 0. You can access items using their index within square brackets.

```
1 languages = ["Python", "Swift", "C++"]
2 print(languages[0]) #Output: Python
3 print(languages[1]) #Output: Swift
4 print(languages[2]) #Output: C++
```

You can also use negative indexing to access items from the end of the list:

```
1 print(languages[-1]) #Output: C++ (last item)
2 print(languages[-2]) #Output: Swift (second to last item)
```

2.5 Slicing Lists

You can extract a portion of a list using *slicing*. Slicing uses the colon operator : to specify a range of indices.

```
1 my_list = [1, 2, 3, 4, 5, 6, 7, 8]
2 print(my_list[2:5]) #Output: [3, 4, 5] (items from index 2 up to, but not including,
   index 5)
```

- `my_list[start:end]` extracts items from index `start` up to (but not including) index `end`.
- If `start` is omitted, it defaults to 0 (the beginning of the list).
- If `end` is omitted, it defaults to the end of the list.

2.6 Checking for Item Existence

You can use the `in` keyword to check if an item exists in a list.

```
1 fruits = ["apple", "banana", "orange"]
2 print("apple" in fruits) #Output: True
3 print("mango" in fruits) #Output: False
```

2.7 Changing List Items

You can change the value of a specific item by assigning a new value to its index.

```
1 colors = ["red", "green", "blue"]
2 colors[0] = "purple"
3 print(colors) #Output: ['purple', 'green', 'blue']
```

You can also change a range of values using slicing:

```
1 colors = ["red", "black", "green", "white"]
2 colors[2:4] = ["blue", "grey"]
3 print(colors) #Output: ['red', 'black', 'blue', 'grey']
```

2.8 Looping Through a List

You can iterate over the items in a list using a `for` loop:

```
1 fruits = ["apple", "banana", "orange"]
2 for fruit in fruits:
3     print(fruit)
```

2.9 Sorting Lists

You can sort a list using the `sorted()` function (which creates a new sorted list) or the `sort()` method (which sorts the list in place).

```
1 numbers = [5, 1, 4, 3]
2 sorted_numbers = sorted(numbers)
3 print(sorted_numbers) #Output: [1, 3, 4, 5]
4
5 numbers.sort() #Sorts in place
```

```
6 print(numbers) #Output: [1, 3, 4, 5]
```

You can also sort in descending order:

```
1 numbers = [5, 1, 4, 3]
2 numbers.sort(reverse=True)
3 print(numbers) #Output: [5, 4, 3, 1]
```

2.10 List Methods

Python provides several built-in methods for working with lists. Here are a few examples:

- `append(item)`: Adds an item to the end of the list.
- `extend(iterable)`: Appends elements from an iterable (e.g., another list) to the end of the list.
- `insert(index, item)`: Inserts an item at a specific index.
- `remove(item)`: Removes the first occurrence of an item.
- `pop(index)`: Removes and returns the item at a specific index (or the last item if no index is specified).
- `clear()`: Removes all items from the list.
- `index(item)`: Returns the index of the first occurrence of an item.
- `count(item)`: Returns the number of times an item appears in the list.

3 Tuples - Your Immutable Data Container!

Imagine you are working on an AI model that requires you to store a set of fixed coordinates for a particular location. These coordinates should not be accidentally changed during the program's execution. This is where *tuples* become extremely useful.

3.1 What is a Tuple?

- A tuple is an ordered, immutable (unchangeable) collection of items. Tuples are similar to lists, but they cannot be modified after creation.

3.2 Creating a Tuple

Tuples are created using parentheses (), with items separated by commas.

```
1 coordinates = (10, 20)
2 names = ("Alice", "Bob", "Charlie")
```

You can also use the `tuple()` constructor:

```
1 fruits = tuple(["apple", "banana", "cherry"]) # note the use of parenthesis
```

Tuples can contain items of different data types:

```
1 mixed_tuple = ("Jack", 20, "Computer Science")
```

Important Point to Remember: If you want to create a tuple with a single item, you need to add a comma after the item.

```
1 single_item_tuple = ("hello",) #Output: ('hello',)
2 not_a_tuple = ("hello") #Output: 'hello'
```

3.3 Tuple Characteristics

- **Ordered:** The items have a defined order, and that order is maintained.
- **Immutable:** You cannot change, add, or remove items after the tuple is created.
- **Allows Duplicates:** Tuples can contain multiple items with the same value.

3.4 Accessing Tuple Items

You can access items in a tuple using their index, just like lists.

```
1 coordinates = (10, 20)
2 print(coordinates[0]) #Output: 10
3 print(coordinates[1]) #Output: 20
```

You can also use negative indexing to access items from the end of the tuple:

```
1 print(coordinates[-1]) #Output: 20 (last item)
```

3.5 Slicing Tuples

You can extract a portion of a tuple using slicing, just like lists.

```
1 my_tuple = (1, 2, 3, 4, 5, 6, 7, 8)
2 print(my_tuple[2:5]) #Output: (3, 4, 5)
```

3.6 Deleting a Tuple

You can delete an entire tuple using the `del` keyword.

```
1 animals = ("dog", "cat", "bird")
2 del animals
3 # print(animals) will cause an error because 'animals' no longer exists
```

3.7 Updating a Tuple (Workaround)

Since tuples are immutable, you cannot directly modify their items. However, you can use a workaround to achieve a similar effect:

1. Convert the tuple to a list.
2. Modify the list.
3. Convert the list back to a tuple.

```
1 fruits = ("apple", "banana", "cherry")
2 fruits_list = list(fruits) #Converting the tuple into a list
3 fruits_list.append("orange") #Appending the list
4 fruits = tuple(fruits_list) #Converting the list back into a tuple
5
6 print(fruits)
```

3.8 Looping Through a Tuple

You can iterate over the items in a tuple using a `for` loop, just like lists.

```
1 fruits = ("apple", "banana", "cherry")
2 for fruit in fruits:
3     print(fruit)
```

3.9 Joining Tuples

You can join two tuples together using the `+` operator.

```
1 tuple1 = ("a", "b", "c")
2 tuple2 = (1, 2, 3)
3 tuple3 = tuple1 + tuple2
4 print(tuple3) #Output: ('a', 'b', 'c', 1, 2, 3)
```

3.10 Tuple Methods

Tuples have only frequently used two built-in methods:

- `count(item)`: Returns the number of times an item appears in the tuple.

- `index(item)`: Returns the index of the first occurrence of an item.
-

Congratulations! You've successfully navigated the world of functions, lists, and tuples in Python. These fundamental data structures are essential tools for any aspiring robotics and machine learning engineer. Keep practicing, experimenting, and exploring, and you'll be well on your way to building amazing AI-powered applications.

Object-Oriented Programming in Python: Demystifying Classes and Objects

A Practical Music Playlist Case Study

Minor In AI, IIT Ropar
24th March, 2025



1 The Evolution of Data Management in Programming

Programming isn't just about storing data; it's about organizing and interacting with it efficiently. Let's explore how our approach to managing data evolves:

1.1 Traditional Approach: Simple Lists

Initially, programmers use simple data structures like lists to store information:

```
1 # A naive approach to storing song information
2 favorite_songs = ["Not Like Us - Kendrick Lamar",
3                     "Kodak - Seedymau",
4                     "Wavy - We end current Hojila"]
5
6 # Iterating and playing songs
7 for song in favorite_songs:
8     print("Now playing:", song)
```

Limitations of this Approach:

- No clear separation between song attributes
- Difficult to add more details like duration, genre, or release year
- Limited ability to perform song-specific actions

2 Object-Oriented Programming: A Paradigm Shift

OOP introduces a revolutionary way of thinking about code. Instead of viewing data as passive information, we treat it as active, self-contained entities with their own properties and behaviors.

2.1 Classes: The Blueprint of Objects

A class is like a blueprint that defines the structure and behavior of objects:

```
1 class Song:
2     def __init__(self, title, artist, duration):
3         # Constructor method: Initializes object attributes
4         self.title = title      # Public attribute
5         self.artist = artist    # Public attribute
6         self.duration = duration # Public attribute
7
8     # Method to represent object's behavior
9     def play(self):
10        print(f"Now playing: {self.title} by {self.artist}")
```

Key Concepts:

- `__init__` is a special method called when creating a new object
- `self` refers to the instance being created
- Methods define actions that objects can perform

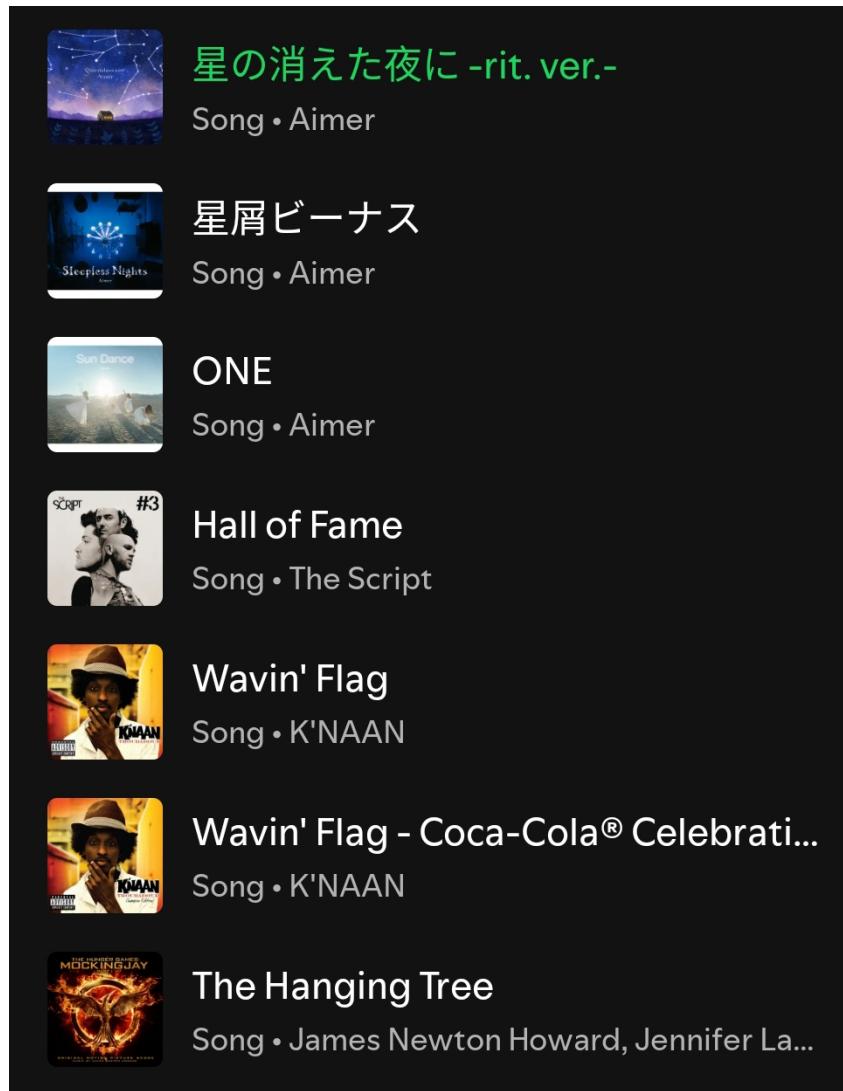


Figure 1: A list of songs!

3 Creating and Using Objects

Objects are specific instances of a class, each with its own unique data:

```

1 # Creating song objects
2 song1 = Song("Not Like Us", "Kendrick Lamar", 180)
3 song2 = Song("Kodak", "Seedymau", 200)
4 song3 = Song("Wavy", "We end current Hojila", 220)
5
6 # Accessing object attributes
7 print(song1.title)      # Output: Not Like Us
8 print(song2.artist)     # Output: Seedymau
9
10 # Invoking object methods
11 song1.play()    # Plays the specific song

```

Object Characteristics:

- Each object is a unique instance of the class

- Objects can have the same structure but different data
- Methods can be called directly on individual objects

4 Encapsulation: Data Protection and Control

Encapsulation is about protecting an object's internal state and providing controlled access:

```

1 class Song:
2     def __init__(self, title, artist, duration):
3         self.title = title
4         self.artist = artist
5         self.__duration = duration # Private attribute
6
7     # Getter method
8     def get_duration(self):
9         return self.__duration
10
11    # Setter method with validation
12    def set_duration(self, duration):
13        if duration > 0:
14            self.__duration = duration
15        else:
16            print("Invalid duration value.")

```

Encapsulation Benefits:

- Prevents direct manipulation of sensitive data
- Allows implementing validation logic
- Provides a clean interface for interacting with objects

5 Building Complex Systems: The Playlist Class

Combining multiple classes to create more complex applications:

```

1 class Playlist:
2     def __init__(self):
3         self.songs = [] # List to store Song objects
4
5     def add_song(self, song):
6         self.songs.append(song)
7
8     def play_all(self):
9         for song in self.songs:
10             song.play()
11
12 # Creating and using a playlist
13 my_playlist = Playlist()
14 my_playlist.add_song(song1)

```

```
15 my_playlist.add_song(song2)
16 my_playlist.play_all()
```

Advanced OOP Concepts Demonstrated:

- Composition (Playlist contains Song objects)
- Separation of concerns
- Modular and extensible design

6 The Power of Object-Oriented Programming

- **Abstraction:** Simplify complex systems by modeling real-world entities
- **Modularity:** Break down problems into manageable, independent components
- **Reusability:** Create flexible, adaptable code structures
- **Maintainability:** Easier to understand, modify, and extend code

Remember: OOP is not just a programming technique, it's a way of thinking about software design!

Minor in AI

Title: Revision: OOP in Action

LO: OOP in Action: Case Studies on Encapsulation, Inheritance, and Object Design

24 March 2025

Case Study: Olympics 2024

Link: <https://www.olympics.com/en/olympic-games/paris-2024>

Observations:

Think of the Olympic Games. You have different people taking part, different sports, and different countries involved.

- You can think of each athlete, each sporting event, and each country as a separate type of thing with its own details and actions.
- Some athletes specialize in certain sports. For example, a swimmer, a runner, and a gymnast are all athletes, but they perform in different ways.
- Certain details about an athlete, like their health or training data, are kept private and are not shown to everyone.
- When it's time to compete, each sport has its own rules and ways of playing, so the way athletes take part in different events can vary.

This is similar to how programmers design systems—by thinking of real-world things, giving them properties and actions, keeping some details hidden, and allowing different behaviors depending on the situation.

Case Study: Music Play List App

What is Object Design?

Object design is about thinking in terms of real-world objects and modeling them in code. Each object has attributes (data) and methods (actions). This makes code organized, reusable, and easy to understand.

Without Object Design (Procedural Code)

```
songs = ["Song A", "Song B", "Song C"]  
  
for song in songs:  
    print("Playing:", song)
```

Problem:

We can't store artist, duration, or add behaviors. Limited flexibility.

With Object Design (Using Classes and Objects)

```
class Song:  
    def __init__(self, title, artist):  
        self.title = title  
        self.artist = artist  
  
    def play(self):  
        print("Playing:", self.title, "by", self.artist)  
  
playlist = [Song("Song A", "Artist X"), Song("Song B", "Artist Y")]  
  
for song in playlist:  
    song.play()
```

Explanation:

- Song is an object with title and artist.
- play is a method that prints the song.
- Objects make it easier to manage and scale the app.

What is self?

- self refers to the current object.
- It allows you to access or update the object's own data inside methods.

Example:

In self.title, it means "this song's title."

__init__ Function

- This is a special function that is called automatically when you create a new object from a class.
- It sets up the object with the details you give it.
- In this case, when you create a song, you provide its title and artist, and the function stores them for later use.

Notes on self:

- `self` is a name used inside a class to refer to the current object.
- Think of it like saying "me" when you talk about yourself. In the same way, `self.title` means "my title" — the title that belongs to this object.
- Without `self`, Python thinks you're talking about a temporary variable that disappears after the function ends.

Encapsulation:

What is Encapsulation?

Encapsulation means hiding data inside the object and only allowing access through methods. This helps protect data and control how it's used.

```
class Song:
    def __init__(self, title, artist, duration):
        self.title = title
        self.artist = artist
        self.duration = duration # in seconds

song = Song("Song A", "Artist X", 180)
song.duration = -50 # Invalid, but allowed
print(song.duration)
```

Problem:

Anyone can change duration to a negative number. No protection.

With Encapsulation (Using Private Variable)

```
class Song:
    def __init__(self, title, artist, duration):
        self.title = title
        self.artist = artist
        self.duration = duration

song = Song("Song A", "Artist X", 180)
print(song.title)
print(song.duration)
```

```

class Song:
    def __init__(self, title, artist, duration):
        self.title = title
        self.artist = artist
        self.__duration = duration # Private

    def get_duration(self):
        return self.__duration

    def set_duration(self, duration):
        if duration > 0:
            self.__duration = duration
        else:
            print("Invalid duration")

song = Song("Song A", "Artist X", 180)
song.set_duration(-50) # Invalid duration
print(song.title)
print(song.get_duration())

```

Encapsulation means hiding details that don't need to be shown to everyone.

Just like:

- You use a remote control without knowing exactly how it works inside.
- A car has an engine, but you just use the steering wheel and pedals — you don't need to see the engine while driving.

Similarly in programming, some data or functions are meant to be kept private (hidden inside), and some can be used outside.

How do we hide data in Python?

We mark it private by adding two underscores __ in front of it.

What is Inheritance?

Inheritance lets you create new classes based on existing ones. You can reuse and extend the behavior of a base class. This reduces code duplication.

```

class Song:
    def __init__(self, title, artist):
        self.title = title
        self.artist = artist

class Podcast:
    def __init__(self, title, host):
        self.title = title
        self.host = host

```

Problem:

Both classes have title. Code is repeated.

```

class AudioContent:
    def __init__(self, title):
        self.title = title

class Song(AudioContent):
    def __init__(self, title, artist):
        super().__init__(title)
        self.artist = artist

    def play(self):
        print("Playing song:", self.title, "by", self.artist)

class Podcast(AudioContent):
    def __init__(self, title, host):
        super().__init__(title)
        self.host = host

    def play(self):
        print("Playing podcast:", self.title, "hosted by",
self.host)

song = Song("Song A", "Artist X")
podcast = Podcast("Episode 1", "Host Y")

song.play()
podcast.play()

```

What is super()?

- super() is used to call the parent class (also called base class) from the child class.
- It helps you reuse code from the parent class without rewriting it

Object Oriented Programming

Welcome to the exciting world of algorithms and data structures! This book is designed to gently introduce you to fundamental concepts like searching, sorting, and algorithm complexity, all while using the familiar and accessible environment of Google Colab with Python.

We'll start with a practical example to build intuition before diving into the technical details. Let's imagine you're building a music app...

1 Building a Music App: A Hierarchical Approach

Imagine you're creating your own music app. You want to organize your music collection effectively. Initially, you might think of simply listing all your songs in a long, single list. But that quickly becomes unwieldy as your collection grows.

Then your manager recommends that you could also integrate a podcast feature, so you think to yourself "oh no, more mess".

This is where the power of relationships and hierarchy comes in. You realize that songs and podcasts, while distinct, share common characteristics: they both have a title, a duration, and maybe even a release year. This commonality suggests a hierarchical structure:

- **Audio (Base):** A general category encompassing both songs and podcasts, sharing common attributes like title and duration.
- **Song (Derived):** A specific type of audio with attributes like artist.
- **Podcast (Derived):** Another specific type of audio with attributes like host.

This organization mirrors real-world hierarchies like the animal kingdom or organizational structures. Just as a manager was once an employee, a song and a podcast both begin as a common type of audio that share certain basic features.

This is the core concept of **Object-Oriented Programming (OOP)**, where we model relationships between different entities in our code.

Now, let's translate this into Python code within Google Colab.

2 Object-Oriented Programming (OOP) Fundamentals



Figure 1: A hierarchical structure of the music app showing Audio, Song, and Podcast classes.

OOP is a programming paradigm that revolves around the concept of “objects,” which combine data (attributes) and actions (methods).

- **Class:** A blueprint or template for creating objects. In our music app example, `Song`, and `Podcast` are classes.
- **Object:** An instance of a class. A specific song with a title and artist is an object.
- **Attributes:** Data associated with an object. Examples: `title`, `artist`, `host`, `duration`.
- **Methods:** Functions that an object can perform. (We’ll add some later!)

Encapsulation: Encapsulation is the bundling of data (attributes) and methods that operate on that data into a single unit, or class.

Let’s look at some code:

```
1 class Song:
2     def __init__(self, title, artist):
3         self.title = title
4         self.artist = artist
5
6 my_song = Song("Bones", "Imagine Dragons")
7 print(my_song.title) # Output: Bones
```

Here, `Song` is a class, `my_song` is an object of the `Song` class, and `title` and `artist` are attributes. The `__init__` method is a special method called a constructor. It’s automatically called when a new object is created and initializes the object’s attributes.

3 Inheritance: Building Hierarchies

Inheritance is a powerful feature of OOP that allows you to create new classes (derived classes) based on existing classes (base classes). The derived class inherits attributes and methods from the base class, promoting code reuse and establishing relationships.

Let's create our audio hierarchy:

```
1 class Audio:
2     def __init__(self, title):
3         self.title = title
4
5 class Song(Audio): # Song inherits from Audio
6     def __init__(self, title, artist):
7         super().__init__(title) # Call the parent class's
8             constructor
9         self.artist = artist
10
11 class Podcast(Audio): # Podcast inherits from Audio
12     def __init__(self, title, host):
13         super().__init__(title)
14         self.host = host
15
16 my_song = Song("Bones", "Imagine Dragons")
17 my_podcast = Podcast("Mind in AI", "HS")
18
19 print(my_song.title) # Output: Bones
20 print(my_podcast.host) # Output: HS
```

- We define a base class, `Audio`, with a `title` attribute.
- The `Song` and `Podcast` classes inherit from `Audio` using the syntax `class Song(Audio):`.
- `super().__init__(title)` calls the constructor of the base class (`Audio`) to initialize the `title` attribute. This avoids redundant code.
- Now, both `Song` and `Podcast` automatically have the `title` attribute, along with their specific attributes (`artist` and `host`, respectively).

4 Protected and Private Members

Sometimes, you want to control the visibility of attributes and methods within a class. Python offers mechanisms for this:

- **Protected Members:** Attributes or methods prefixed with a single underscore (`_`). They are intended to be internal to the class and its subclasses but can still be accessed from outside.

- **Private Members:** Attributes or methods prefixed with a double underscore (_). They are meant to be strictly internal to the class and are more difficult to access from outside.

```

1  class Song(Audio):
2      def __init__(self, title, artist):
3          super().__init__(title)
4          self.artist = artist
5          self._duration = 3 # Protected member
6
7  class Podcast(Audio):
8      def __init__(self, title, host):
9          super().__init__(title)
10         self.host = host
11         self.__rating = 5 # Private member
12
13     def get_rating(self):
14         return self.__rating # Access private member
15         through a getter method
16
17 my_song = Song("Bones", "Imagine Dragons")
18 my_podcast = Podcast("Mind in AI", "HS")
19
20 print(my_song._duration) # Accessing a protected member
21     (generally discouraged)
22 print(my_podcast.get_rating()) # Accessing a private member
23     through a getter method

```

Trying to directly access `my_podcast.__rating` will result in an error. This is because of **name mangling**. Python internally renames private attributes to prevent accidental access from outside the class.

Name Mangling: To truly hide the name, Python uses name mangling to create a function called `_className_attributeName`

The `get_rating` method provides a controlled way to access the private `__rating` attribute. These methods are often called “getters” (for retrieving values) and “setters” (for modifying values). They encapsulate the internal workings of the class and provide an interface for interacting with it.

5 Sorting: Putting Things in Order

Now, let’s move on to sorting. Sorting is the process of arranging elements in a specific order (e.g., ascending or descending).

Imagine you’re a gambler who’s trying to arrange a deck of cards in his hands. You are dealt with a card from the deck, and you have to arrange the cards one by one to the correct place. The way you arrange the cards is exactly the thought process of insertion sort.

Let’s focus on **Insertion Sort**, a simple and intuitive sorting algorithm.

Insertion Sort Intuition:

The way you sort cards that you pick from the deck is exactly how Insertion sort works. You maintain a sorted subarray and iteratively insert elements from the unsorted portion into their correct positions within the sorted subarray.

Here is the general idea of an Insertion sort Algorithm:

1. Start with the second element in the array. This element is what will be compared with the previously seen array.
2. If the previous element is smaller than the value that is selected, that means it's at the right place, so continue to the next value.
3. If the previous value is greater than the currently selected value, use a temporary variable to copy the current value. Shift the greater element up to the next element. Keep doing this till you reach a smaller value, and copy the variable at the next element.

5.1 Best, Average, and Worst Cases

Table 1: Complexities and stability of some sorting algorithms

Name of the algorithm	Average case time complexity	Worst case time complexity	Stable?
Bubble sort	$\Theta(n^2)$	$O(n^2)$	Yes
Selection sort	$\Theta(n^2)$	$O(n^2)$	No
Insertion sort	$\Theta(n^2)$	$O(n^2)$	Yes
Merge sort	$\Theta(n \log_2 n)$	$O(n \log_2 n)$	Yes
Quick sort	$\Theta(n \log_2 n)$	$O(n^2)$	No
Bucket sort	$\Theta(d(n+k))$	$O(n^2)$	Yes
Heap sort	$\Theta(n \log_2 n)$	$O(n \log_2 n)$	No

When analyzing sorting algorithms, we consider different scenarios:

- **Best Case:** The array is already sorted. In insertion sort, the number of comparisons is minimal (linear time complexity).
- **Average Case:** The array is randomly ordered. Insertion sort performs reasonably well (quadratic time complexity).
- **Worst Case:** The array is sorted in reverse order. Insertion sort requires the maximum number of comparisons and shifts (quadratic time complexity).

Insertion sort's best case performance is linear, while bubble sort still takes quadratic time. Therefore, if it is already sorted, Insertion sort is the better method to use.

6 Conclusion

This book has provided a gentle introduction to fundamental programming concepts such as Object Oriented Programming, hierarchical design, Encapsulation,

and sorting algorithms. We used the real world example of building a music app to contextualize the code that you would be writing. By experimenting with code examples and building your own projects, you'll solidify your understanding and gain valuable problem-solving skills. Happy coding!

Revision on Vectors, Matrices, Numpy, Pandas

26/03/2025
Minor in AI

1 Topics

1.1 Object-Oriented Programming (OOP)

- Basic concepts and revision of OOP principles

1.2 Vectors and Matrices

- Vector operations (addition, scaling, dot product)
- Matrix operations (multiplication, transposition, etc.)

1.3 NumPy Library

- Array creation and manipulation
- Reshaping arrays and flattening matrices
- Element-wise operations
- Calculating statistics (mean, standard deviation)
- Generating random arrays and matrices
- Cumulative sum operations
- Creating identity matrices
- Finding maximum and minimum values within arrays

1.4 Pandas Library and DataFrame Operations

- Creating and managing DataFrames
- Data types handling (categorical, numerical)
- Basic statistics (mean, median, mode, standard deviation)

- Data filtering, sorting, and grouping
- Handling missing values
- Reading from and writing to CSV files
- Introduction to panel data concepts

2 Reference Links

2.1 Vector and Matrices (Operations)

<https://students.masaischool.com/lectures/98627?tab=notes>

2.2 Numpy

<https://students.masaischool.com/lectures/99856?tab=notes>

2.3 Pandas

<https://students.masaischool.com/lectures/101140?tab=notes>

<https://students.masaischool.com/lectures/101141?tab=notes>

Colab Notebook

https://colab.research.google.com/drive/19mLcIUuKwFRugEE10Fh9iV-RbQv_2b7U#scrollTo=xJ4iwwNaCcy0

Mastering Markov Chains: Bridging the Gap Between Theory and Real-World Applications in Human Behavior

Minor In AI, IIT Ropar
27th March, 2025

Welcome, aspiring AI explorers! This module is designed to guide you on a fascinating journey into the world of Markov Chains, a powerful tool for understanding and modeling sequential decision-making, especially in scenarios where human behavior plays a central role. We'll start with the core ideas, build your intuition, and then translate that understanding into practical Python code. Don't worry if you're new to Python or AI – we'll take it one step at a time.

The Instagram Scroll: A Day in the Life of a Markov Chain

Imagine you're scrolling through Instagram. What happens? You see a post, maybe you like it, comment on it, or keep scrolling. Sometimes you get bored and close the app. This seemingly random behavior can actually be modeled using a Markov Chain.

Think about it: what you do *right now* is influenced by what you are doing *right now*. For instance, if you are currently scrolling, there's a pretty good chance you'll keep scrolling. But, eventually, something might catch your eye, leading you to like and comment. Or, you might realize you have an exam to study for, and you close the app.



*Gen Z+ Doomscrolling!

Traditional machine learning often focuses on historical data. "You've liked 100 cat videos in the past, so we'll show you more cat videos!" But what if today you're feeling more like dogs? Or what if you're simply tired of cute animals and want to look at landscapes? Markov Chains allow us to consider that *current state* of mind. This is especially relevant to model the uncertainty associated with real-time behavior.

Let's say, for simplicity, there are three possible actions:

1. **Scrolling**
2. **Liking/Commenting**
3. **Closing the App**

A Markov Chain would model the *probabilities* of transitioning between these states. For example:

- If you are currently **Scrolling**, there's a 70% chance you'll continue **Scrolling**, a 20% chance you'll **Like/Comment**, and a 10% chance you'll **Close the App**.

- If you are currently **Liking/Commenting**, there's a 30% chance you'll stay in **Liking/Commenting**, a 50% chance you'll go back to **Scrolling**, and a 20% chance you'll **Close the App**.
- If you have **Closed the App**, there's an 80% chance you'll start **Scrolling** again, a 15% chance you'll go straight to **Liking/Commenting**, and only a 5% chance you'll keep the app closed.

These probabilities tell us how likely you are to move from one state (activity) to another. The key idea is that your next action *only* depends on your current action, not on the entire history of your Instagram usage. That's the defining characteristic of a Markov Chain!

This simple example illustrates the core concept. Now, let's delve into the theory and see how we can implement this using Python.

Building Your First Markov Chain in Python

Now, let's see how to represent the above Instagram transitions using a Markov Chain in Python using the NumPy library.

```
import numpy as np

# Create a transition matrix
transition_matrix = np.array([
    [0.7, 0.2, 0.1], # Probabilities from Scrolling to Scrolling, Liking/Commenting, Closing
    [0.5, 0.3, 0.2], # Probabilities from Liking/Commenting to Scrolling, Liking/Commenting, Closing
    [0.8, 0.15, 0.05] # Probabilities from Closing to Scrolling, Liking/Commenting, Closing
])

# Possible states
states = ["Scrolling", "Liking/Commenting", "Closing"]

# Initial state
current_state = "Scrolling"
print(f"You are currently {current_state}")

# Simulate the next state based on the transition matrix
current_state_index = states.index(current_state) #Find index for the current state
next_state_index = np.random.choice(len(states), p=transition_matrix[current_state_index]) #Return a random index
next_state = states[next_state_index] # Map index back to a state
print(f"You will transition to {next_state}")
```

Code Breakdown:

1. `import numpy as np`: Imports the NumPy library, essential for numerical operations, especially for working with arrays.
2. `transition_matrix = np.array(...)`: This is the heart of our Markov Chain. It's a 2D NumPy array representing the probabilities of transitioning between states. Each row represents the current state, and each column represents the possible next states. The values in the matrix are the probabilities.
 - The **first row** ([0.7, 0.2, 0.1]) represents the probabilities *when you are currently scrolling*. 70% chance of continuing to scroll, 20% chance of liking/commenting, and 10% chance of closing the app.
 - The **second row** ([0.5, 0.3, 0.2]) represents the probabilities *when you are currently liking/commenting*. 50% chance of going back to scrolling, 30% chance of continuing to like/comment, and 20% chance of closing the app.
 - The **third row** ([0.8, 0.15, 0.05]) represents the probabilities *when you have closed the app*. 80% chance of starting scrolling again, 15% chance of going straight to liking/commenting, and only a 5% chance of keeping the app closed.

Important: Notice that the probabilities in each row *must add up to 1*. This ensures that all possible transitions from a given state are accounted for.

3. `states = ["Scrolling", "Liking/Commenting", "Closing"]`: A simple list of the possible states we can be in.
4. `current_state = "Scrolling"`: Here, we manually specify the starting state.
5. `current_state_index = states.index(current_state)`: Gets the index of the current state from the list of possible states.
6. `next_state_index = np.random.choice(len(states), p=transition_matrix[current_state_index])`: This is where the "magic" happens. The next step is randomly selected using probabilities from the transition matrix based on the current state.
 - `np.random.choice(len(states), ...)`: Chooses a random index from the range of states in the chain.
 - `p=transition_matrix[current_state_index]`: Takes the appropriate row from the transition matrix (based on the current state) for the list of probabilities for the next state.
7. `next_state = states[next_state_index]`: Maps the index of the next state back to the name of the next state.
8. `print(...)`: Prints the next state

When you run this code, it will randomly choose the next state based on the probabilities defined in the `transition_matrix`. Each time you run it, you might get a different result!

Understanding the Transition Matrix

Let's break down the transition matrix in more detail:

```
[0.7, 0.2, 0.1], # Probabilities from Scrolling
[0.5, 0.3, 0.2], # Probabilities from Liking/Commenting
[0.8, 0.15, 0.05] # Probabilities from Closing
```

- **Rows:** Each row represents the *current state*. The first row is "Scrolling", the second is "Liking/Commenting", and the third is "Closing".
- **Columns:** Each column represents the *possible next states*. The first column is "Scrolling", the second is "Liking/Commenting", and the third is "Closing".
- **Values:** The values at each intersection represent the *probability* of moving from the row's current state to the column's next state.
 - `transition_matrix[0][0]` (0.7): The probability of going from "Scrolling" to "Scrolling" (staying in the same state).
 - `transition_matrix[0][1]` (0.2): The probability of going from "Scrolling" to "Liking/Commenting".
 - `transition_matrix[2][0]` (0.8): The probability of going from "Closing" back to "Scrolling".

Understanding how to read and interpret the transition matrix is crucial to grasping the behavior of the Markov Chain.

From States to Simulation

Now that we understand the basics, let's create a simulation to observe the Markov Chain in action over a longer period:

```
import numpy as np

# Transition Matrix (same as before)
transition_matrix = np.array([
    [0.7, 0.2, 0.1],
    [0.5, 0.3, 0.2],
    [0.8, 0.15, 0.05]
```

```

])
# States (same as before)
states = ["Scrolling", "Liking/Commenting", "Closing"]

# Initial State
current_state = "Scrolling"

# Number of steps to simulate
num_steps = 15

# Simulation Loop
print(f"Starting simulation from {current_state}:")
for i in range(num_steps):
    current_state_index = states.index(current_state) #Find index for the current state
    next_state_index = np.random.choice(len(states), p=transition_matrix[current_state_index]) #Return index of next state
    current_state = states[next_state_index] # Map index back to a state
    print(f"Step {i+1}: You are now {current_state}")

```

Key Improvements:

- **Simulation Loop:** The `for` loop simulates the Markov Chain for a specified number of steps (`num_steps = 15`).
- **State Update:** In each step, the `current_state` is updated to the `next_state`, reflecting the transition based on the probabilities.
- **Output:** The code now prints the state at each step of the simulation, giving us a sequence of states representing the user's "journey" through Instagram.

Run this code, and you'll see a different sequence of Instagram activities each time, based on the defined probabilities! This showcases the power of Markov Chains in simulating real-world, probabilistic behavior.

Beyond Instagram: Applications of Markov Chains

While our Instagram example is illustrative, Markov Chains have a wide range of applications:

- **Speech Recognition:** Predicting the next word in a sentence.
- **Financial Modeling:** Predicting stock prices or market movements.
- **Genetics:** Modeling the evolution of genes.
- **Recommendation Systems:** Suggesting products or content based on user behavior.
- **Weather Forecasting:** Predicting future weather conditions.

The core principle remains the same: understanding the probabilities of transitioning between different states allows us to model and predict sequential events.

Balancing Past Data and Current State

As we've seen, Markov Chains emphasize the importance of the *current state* in determining the next action. However, it's crucial to remember that traditional AI models rely heavily on *historical data*. The best approach often involves finding a balance between these two perspectives.

Consider our Instagram example again. While the current state (scrolling, liking, closing) strongly influences the next action, historical data could provide valuable insights into user preferences. For instance, if a user consistently likes posts from a particular account, the probability of liking a future post from that account might be higher.

By combining historical data with the principles of Markov Chains, we can create more accurate and personalized models of human behavior.

Congratulations! You've taken your first steps into the fascinating world of Markov Chains. You've learned the core concepts, seen how to implement them in Python, and explored a range of real-world applications.

Remember, this is just the beginning. As you continue your journey, explore more advanced topics such as Hidden Markov Models, reinforcement learning, and the application of Markov Chains to more complex problems. The possibilities are endless!

Minor in AI

Revision

28 March, 2025

Topics Revised

1. **File handling and exception:** Introduction to file operations (reading, writing, appending) and handling exceptions using try-except blocks.
2. **Introduction to statistics:** Understanding the distinction between descriptive and inferential statistics.
3. **Data types and variables:** Covering structured vs. unstructured data, cross-sectional vs. time series data, and univariate vs. multivariate data.
4. **Population and sample:** Understanding the importance of representative sampling.
5. **Measures of central tendency**
6. **Measures of dispersion (variability)**
7. **Data distribution concepts:** Including frequency, relative frequency, and cumulative frequency.
8. **Graphical representation techniques:**
 - Histograms
 - Box plots
 - Scatter plots
9. **Practical application examples:** Utilizing Python libraries such as NumPy and Pandas for computing these statistics and visualizing data.

Related Lectures

- **File Handling and exception:**
 - [File handling and exceptions Lecture](#)
- **Pandas:**
 - [Pandas Lecture 1](#)
 - [Pandas Lecture 2](#)
- **Descriptive Statistics:**
 - [Lecture Link](#)
- **Plotting:**
 - [Plotting Lecture 1](#)
 - [Plotting Lecture 2](#)
 - [Plotting Lecture 3](#)