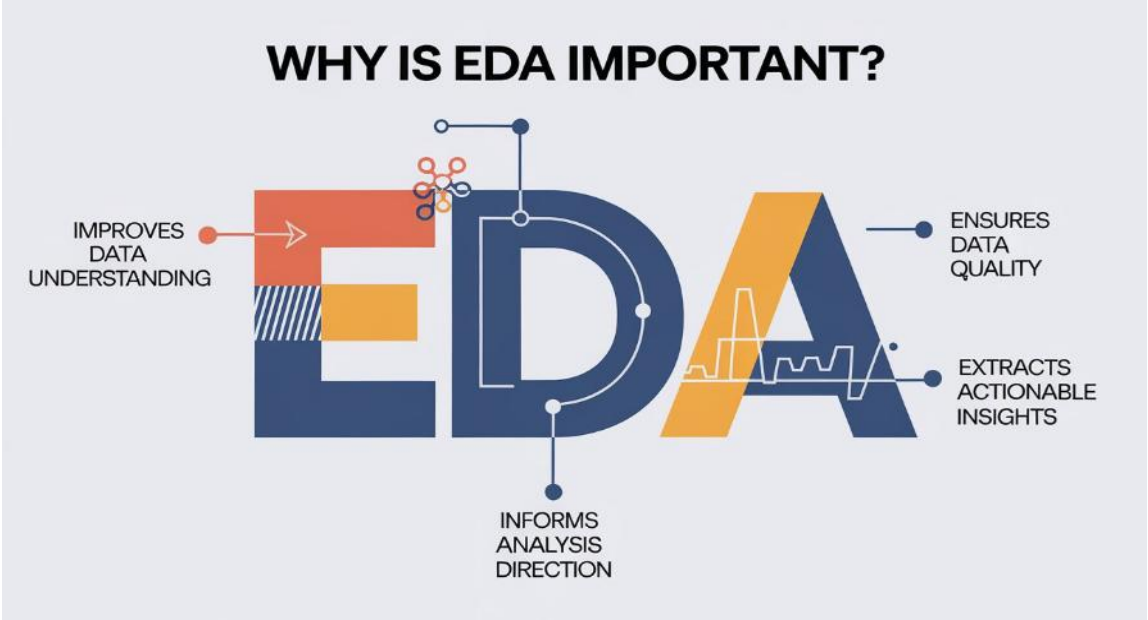
INTRODUCTION TO EDA

UNDERSTANDING EDA

- **Exploratory data analysis (EDA)** is the key process of investigating and interpreting the datasets and summarizing the important conclusions from the data using visual techniques.
- EDA helps to understand what the data can reveal beyond the formal modeling or hypothesis testing.
- It's the first step to making sense of raw data.

WHY IS EDA IMPORTANT?

- EDA is essential because it lays the groundwork for effective data analysis
- by improving understanding, ensuring data quality, guiding analytical techniques, and extracting valuable insights that shape deeper investigations.



WHY IS EDA IMPORTANT?

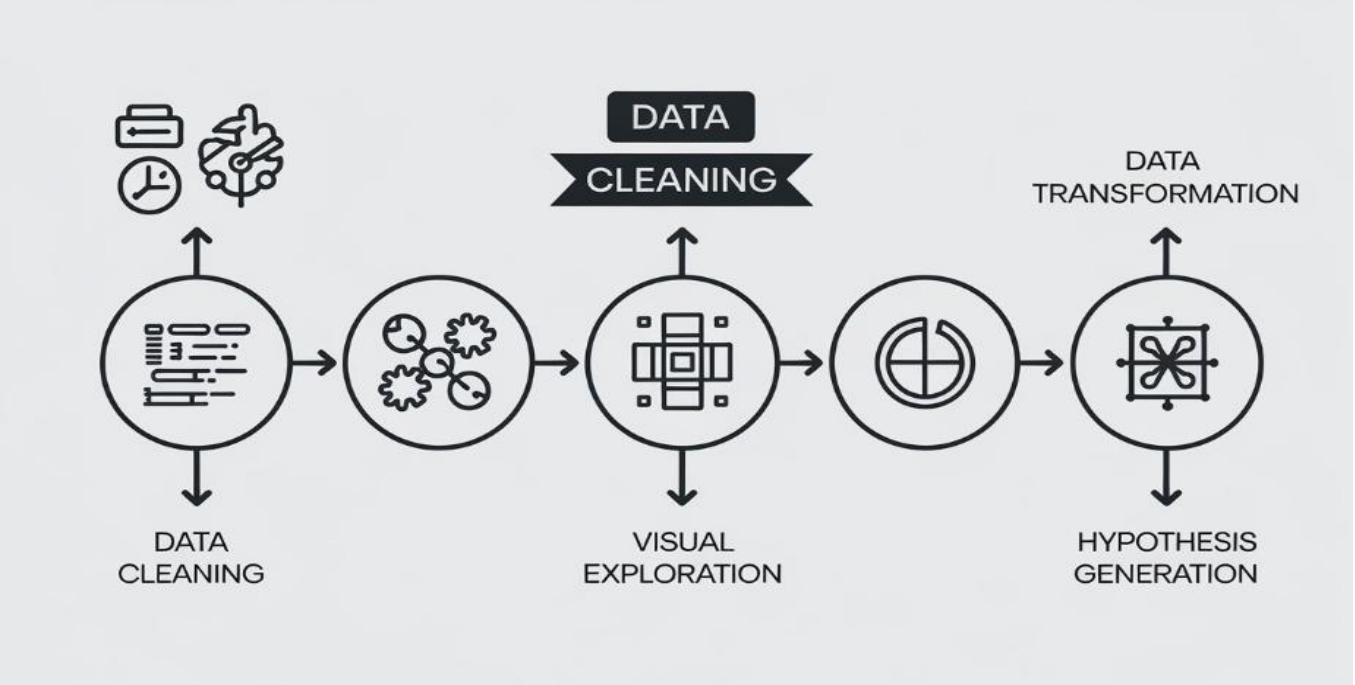
- **Improves Data Understanding:**
  - EDA enables a deep understanding of the dataset's structure
  - Finding trends
  - finding relationships
  - It gives you an idea of what the data represents.
- **Ensures Data Quality:**
  - EDA helps in identifying missing values, outliers, and data inconsistencies
  - It ensure the accuracy and reliability of the dataset.
  - Detecting and handling these issues early prevents misleading results in later analyses.

WHY IS EDA IMPORTANT?

- **Informs Analysis Direction:**
  - EDA provides suitable models or techniques for further analysis.
  - Like regression, classification, or clustering
  - EDA sets the stage for effective modelling decisions.
- **Extracts Actionable Insights:**
  - EDA uncovers meaningful trends and anomalies
  - enabling you to form hypotheses and identify areas
  - These insights helps you for
    - business decisions
    - strategy changes
    - further exploratory research.

KEY STEPS IN EDA

- The EDA process follows a structured approach—collecting, cleaning, transforming, visualizing, and summarizing the data.
- Each step is vital to ensure the dataset is ready for deeper analysis and insights generation.



- **Data Collection:**
  - Start by gathering the dataset from various sources like CSV files, Excel sheets, or databases.
  - Ensuring the data is comprehensive and relevant is crucial for meaningful analysis.
- **Data Cleaning:**
  - Address missing values, fix incorrect data types, and handle outliers.
  - It ensures data quality, making sure your analysis is based on accurate and reliable data.
- **Data Transformation:**
  - Apply transformations like scaling numerical features, encoding categorical variables, or normalizing data as needed.
  - It prepare the data for efficient analysis and model performance.

- **Visual Exploration:**
  - Use visualizations such as histograms, scatter plots, box plots, and heatmaps to explore the distribution of variables, spot trends, and understand relationships within the data.
- **Summarization & Hypothesis Generation:**
  - Summarize key findings from EDA.
  - Use these insights to create hypotheses for deeper analysis or guide further exploration.
  - It sets the direction for advanced analysis or modelling.

TOOLS FOR EDA

- The following libraries and tools mentioned below form the backbone of EDA, enabling data manipulation, statistical analysis, and the creation of insightful visualizations to explore and understand your data.

- **Pandas:**
  - A powerful library for data manipulation and generating summary statistics.
  - It makes tasks like filtering, grouping, and reshaping data simple and intuitive, allowing for quick exploratory insights.

- **Matplotlib/Seaborn:**
  - Essential libraries for creating visualizations.
  - Matplotlib provides flexibility to create a wide range of charts
  - while Seaborn simplifies the process by offering visually appealing and statistical plots like heatmaps, histograms, and box plots.

- **NumPy:**
  - The foundation for efficient numerical operations in Python.
  - It enables fast computation on arrays and matrices,
  - making it essential for handling large datasets and performing calculations.

- **SciPy:**
  - A library built for advanced statistical analysis.
  - It offers functions for hypothesis testing, probability distributions, and other mathematical tools that are critical for deeper statistical insights during EDA.

WHAT IS PANDAS?

Pandas is an open-source library that is made mainly for working with relational or labeled data.

It provides various data structures and operations for manipulating numerical data and time series.

This library is built on top of the NumPy library.

Pandas is fast and it has high performance & productivity for users.

WHY PANDAS?

- Pandas allows us to analyze big data and make conclusions based on statistical theories.
- Pandas can clean messy data sets, and make them readable and relevant.
- Relevant data is very important in data science.

HOW TO USE

- Pandas Installation:
- pip install pandas
- Or
- Py –m pip instal l pandas

```
#Series
import pandas as pd
a=[11,12,2,34,56,78,89,90]
myvar= pd.Series(a)
print(myvar)
```

```
#check pandas version
import pandas as pd
print(pd.__version__)
```

```
#access element in Series
print(myvar[3])
#by default 0-7 index numbers given as a label
```

CREATE OWN LABELS

```
#let's create own labels
import pandas as pd
a=[1,7,4]
myvar=pd.Series(a,index=["a","b","c"])
print(myvar)
print(myvar["b"])
```

DICTIONARY IN SERIES

```
import pandas as pd
studentObject= {
                    "id":1,
                    "name":"sonam",
                    "email":"sonam@gmail.com"
                } #dictionary object in Series
s1= pd.Series(studentObject)
print(s1) #see id,name and email become labels
print("Id: ",s1["id"])
print("Name: ",s1["name"])
print("Email: ",s1["email"])
```

WHAT IS DATAFRAME?

- A **DataFrame** is a 2D tabular data structure in Pandas, similar to an Excel sheet or SQL table.
- It allows us to **store, manipulate, and analyze** structured data.

```python
import pandas as pd  # Importing pandas

# Creating a simple dataset using a dictionary
data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35],
    "City": ["New York", "Los Angeles", "Chicago"]
}

# Converting dictionary into a pandas DataFrame
df = pd.DataFrame(data)

# Displaying the DataFrame
print(df)
```

UNDERSTANDING IN DETAILS

```
#dataframes
import pandas as pd
data={
    "name":["alex","bob","catty","david"],
    "age" : [30,23,34,28]
}

myvar= pd.DataFrame(data)
print(myvar)
#you want to get the row then use loc attribute
print("First Row")
print(myvar.loc[0])
#get first 2 rows
print(myvar.loc[[0,1]])
```

UNDERSTANDING IN DETAILS

```
#dataframes
import pandas as pd
data={
    "name":["alex","bob","catty","david"],
    "age" : [30,23,34,28]
}

myvar= pd.DataFrame(data, index=["u1","u2","u3","u4"])
print(myvar)
#you want to get the row then use loc attribute
print("First Row")
print(myvar.loc["u3"])
```

UNDERSTANDING IN DETAILS

```
import pandas as pd
data={
    "name":["alex","bob","catty","david"],
    "age" : [30,23,34,28]
}

myvar= pd.DataFrame(data, columns=['name','age'])
print(myvar)
```

CREATING DF FROM LIST OF LISTS

```
import pandas as pd

data = [
    ["Alice", 25, "New York"],
    ["Bob", 30, "Los Angeles"],
    ["Charlie", 35, "Chicago"],
    ["David", 40, "Houston"]
]

df = pd.DataFrame(data, columns=["Name", "Age", "City"])
print(df)
```

DATA FROM EXTERNAL RESOURCES

```
import pandas as pd

df = pd.read_csv("data.csv")  # Reads data from a CSV file
print(df.head())  # Displays first 5 rows

#inport data from excel
df = pd.read_excel('data.xlsx', sheet_name='Sheet1')
print(df.head())  # Displays first 5 rows
```

DATA FROM EXTERNAL RESOURCES

```
import pandas as pd

df = pd.read_csv("data.csv")  # Reads data from a CSV file
print(df.head())  # Displays first 5 rows

#inport data from excel
df = pd.read_excel('data.xlsx', sheet_name='Sheet1')
print(df.head())  # Displays first 5 rows
```

FUNCTIONS

- **print(df)** : first 5 and last 5 rows
- **print(df.to_string())** : convert them into string to see all
- **print(df.head(2))**: First 2 rows
- **print(df.tail(2))** : Last 2 rows
- **print(df.shape)** : No of rows & columns
- **print(df.info())**: Summary of DataFrame
- **print(df.describe())**: Summary statistics (numerical columns)
- **print(df["Name"])**:Select a single column
- **print(df[["Name", "Age"]])**:Select multiple columns

ACTIVITY

- **Analyzing Student Performance Data using Pandas:**
- Create Dummy csv file which can have Student_ID,Name,Math,Science,English,Total,Grade from mokaroo.com
- Load the CSV file into pandas.
- **df = pd.read_csv("student_performance.csv")**
- Display the first few rows
- **print(df.head())**
- Fill Missing Data (if any)
- **df.fillna({'Total': 0}, inplace=True)**
- **(**Modifies the DataFrame directly, without returning a new one)

- Calculate total marks by summing Math, Science, and English scores
- **df['Total'] = df[['Math', 'Science', 'English']].sum(axis=1)**
- **print(df[['Name', 'Total']])**
- Write a function for assigning grades and then call that function to apply on field
- **df['Grade'] = df['Total'].apply(assign_grade)**
- Display with grades
- **print(df[['Name', 'Total', 'Grade']])**

```python
# Function to assign grades
def assign_grade(total):
    if total >= 250:
        return 'A'
    elif total >= 200:
        return 'B'
    elif total >= 150:
        return 'C'
    else:
        return 'D'
```

- Filter Students with Grade A:
- **grade_a_students =**
  df[df['Grade'] == 'A']
- **print(grade_a_students)**
- Save Data to new CSV file:
- **df.to_csv("data_new.csv", index=False)**
- print("data saved successfully!")

ASSIGNMENT

- **Analyzing Sales Data Using Pandas:**
- You are given a CSV file containing sales data for an online store. The task is to clean the data, perform calculations, and extract insights.
- Steps:
  1. Create the Sample CSV File
  2. Load the Data into Pandas
  3. Fill Missing Data and Clean the Dataset
  4. Calculate the Total Amount for Each Order
  5. Find the Most Sold Product (you can group by product, take quantity sum and find max (idxmax())
  6. Find Total Sales per Category(group by category and sum total_amount)
  7. Find Orders Above $500
  8. Save the Processed Data to a New CSV File

WHAT IS NUMPY?

- **NumPy (Numerical Python)** is a powerful Python library for numerical computing,
- It is especially useful for working with arrays, matrices, and mathematical operations.
- Why Numpy?
  - Faster than Python lists (due to optimized C code)
  - Supports multi-dimensional arrays
  - Provides mathematical & statistical functions
  - Useful for Machine Learning & Data Science

HOW TO USE

- Numpy Installation:
- pip install numpy

```
import numpy as np  # Import NumPy

# Creating a 1D NumPy array
arr = np.array([10, 20, 30, 40, 50])

# Displaying the array and its type
print("NumPy Array:", arr)
print("Type:", type(arr))  # Check the type
```

```python
import numpy as np  # Import NumPy

## Creating Array
arr1 = np.array([1, 2, 3, 4])  # 1D array
arr2 = np.array([[1, 2, 3], [4, 5, 6]])  # 2D array
print(arr1, arr2)

## Checking Shape and Size
print(arr2.shape)  # (rows, columns)
print(arr2.size)   # Total elements
```

```
## Generate Arrays
print(np.zeros((2,3)))   # 2x3 matrix filled with 0s
print(np.ones((3,3)))    # 3x3 matrix filled with 1s
print(np.arange(1, 10, 2))  # [1, 3, 5, 7, 9] (like range())

## Performing Some Operations
a = np.array([5, 10, 15])
b = np.array([2, 4, 6])

print(a + b)  # Element-wise addition
print(a * b)  # Element-wise multiplication
print(a ** 2)  # Squaring elements
```

Customer Purchase Analysis (Total & Average Spend)

```python
import numpy as np

# Purchase amounts (5 customers, 3 different purchases)
purchases = np.array([
    [200, 150, 100],
    [300, 250, 400],
    [100, 200, 150],
    [500, 600, 550],
    [50, 80, 100]
])

# Total spend per customer
total_spend = purchases.sum(axis=1)

# Average spend per customer
average_spend = purchases.mean(axis=1)

print("Total Spend:", total_spend)
print("Average Spend:", average_spend)
print(max(average_spend))
```

Solve the system of equations

```python
# 2x + 3y = 8
# 4x + y = 10
import numpy as np

# Coefficients of variables
A = np.array([[2, 3], [4, 1]])

# Constants on the right-hand side
B = np.array([8, 10])

# Solve for x and y
solution = np.linalg.solve(A, B)

print("Solution (x, y):", solution)
```

ASSIGNMENT

- Basic Operations with NumPy
- You are given an array of sales revenue for 10 days. Your task is to perform basic operations using NumPy to analyze the data.
    1. Import NumPy and Create an Array
    2. Find the total sales, average sales, and highest/lowest sales
    3. Print all
    4. Find which days had sales above the average and print
    5. Apply a 10% Increase to All Sales and Store in a New Array
    6. Sort Sales in Ascending Order

METHODS OF DATA COLLECTION

- It refers to the various techniques used to gather data for analysis like
  - Surveys
  - observational studies
  - web scraping

HOW TO GATHER DATA?

- **Surveys and Questionnaires:**
  - Data collected through structured forms, ideal for gathering user opinions or customer feedback.
- **Observation:**
  - Capturing real-time data by monitoring behaviors, events, or systems, often used in research or scientific studies.
- **Transactional Data:**
  - Data generated by systems or software from sales, purchases, or user interactions.
- **Web Scraping:**
  - Automatically extracting data from websites, useful for gathering large datasets from public or semi-structured sources.
- **Sensor Data:**
  - Real-time data collected from sensors, such as temperature or traffic monitoring systems.

DATA CLEANING

- Handling missing values
- Removing duplicates
- Data type conversions
- Outlier detection and treatment

Improve data quality by handling missing values, removing duplicates, converting data types, and treating outliers to ensure accurate and meaningful insights

HANDLING MISSING VALUES

- Missing values are common in datasets and occur when no data is available for certain entries or fields.
- In EDA, handling missing values is crucial because they can skew results, reduce model performance, or cause errors in analysis.
- **Methods to Handle Missing Values:**
  - **Identifying Missing Values:** Use functions like **df.isnull()** or **df.info()** to detect missing or null values in the dataset.
  - **Removing Missing Values:**
  - Drop Rows or Columns: Remove rows or columns with missing values using **df.dropna()**
  - This is useful when missing data is minimal and doesn't impact the analysis.

HANDLING MISSING VALUES

- **Imputation:**
- **Mean/Median/Mode Imputation:**
  - Replace missing numerical values with the mean, median, or mode of the column using **df.fillna()**
- **Forward/Backward Fill:**
  - Use methods like forward fill **(ffill)** or backward fill **(bfill)** to propagate the last or next valid observation.
- **Flagging Missing Values:**
  - Create a new indicator column to flag rows where data is missing.
  - This can be useful in certain types of analysis where missing data may carry information.

WHAT ARE DUPLICATES?

Rows in a dataset containing identical or nearly identical values across all selected columns.

Duplicates can occur due to errors in data entry, multiple imports, or redundancy in data sources.

If not addressed, duplicates can distort analysis by over-representing certain data points, leading to incorrect conclusions.

HANDLING DUPLICATES

- Handling duplicates in EDA involves identifying, analyzing, and removing or treating them as necessary.
- **Identifying Duplicates**:
  - Using functions like **df.duplicated()** to check for repeated rows in the dataset.
- **Removing Duplicates:**
  - Applying **df.drop_duplicates()** to remove duplicates, either globally or based on specific columns, while retaining the first or last occurrence.

DATA TYPE CONVERSIONS

- It ensure that each variable in a dataset is stored in the most appropriate format for analysis.
- Incorrect data types can lead to errors, inefficient processing, or inaccurate results during data manipulation, transformation, and analysis.
- **Steps for Converting Data Types:**
  - Identify Incorrect Data Types: Check data types using df.dtypes.
  - Convert to Appropriate Types:
  - Convert Strings to Numeric: df['column'].astype(int)
  - Convert Strings to Datetime: pd.to_datetime(df['column'])

EXAMPLE

```
data = {'Date': ['2021-01-01', '2022-05-14'], 'Price': ['1000', '2500']}
df = pd.DataFrame(data)


# Convert Date column to datetime and Price column to integer
df['Date'] = pd.to_datetime(df['Date'])
df['Price'] = df['Price'].astype(int)


print(df.dtypes)
```

WHAT ARE OUTLIERS

- Outliers are data points that are significantly different from the majority of the data.
- They can occur due to variability in measurement or experimental errors.
- Outliers can distort analysis, leading to incorrect conclusions.
- **Techniques:**
  - Z-score Method
  - IQR (Interquartile Range) Method

TECHNIQUES

- **Z-score Method**: Calculate Z-scores to identify values that are a certain number of standard deviations from the mean.
- Formula: Z = (X - mean) / std_dev
- **IQR (Interquartile Range) Method**: Identify outliers using the 1.5*IQR rule. Values outside this range are considered outliers.
- IQR = Q3 - Q1
- **Outlier Treatment:**
- Remove Outliers: If they are erroneous, simply remove them.
- Cap/Floor Outliers: Limit extreme values by capping or flooring them at a threshold.
- Transformation: Apply log transformations or other methods to minimize the effect of outliers.

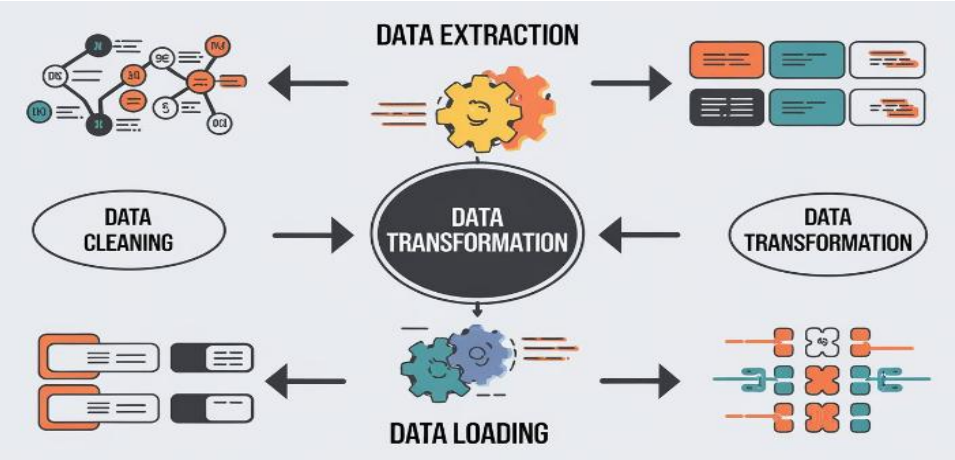
EXAMPLE

- Calculate **Q1 (25th percentile)** and **Q3 (75th percentile)**.
- Compute the **IQR = Q3 - Q1**.
- Define lower and upper bounds:
  - **Lower Bound** = Q1 - 1.5 * IQR
  - **Upper Bound** = Q3 + 1.5 * IQR
- Identify data points that fall outside these bounds as **outliers**.

```
import numpy as np
# Sample data with an outlier
data = np.array([10, 12, 14, 15, 18, 21, 25, 100])

# Calculate Q1, Q3, and IQR
Q1 = np.percentile(data, 25)
Q3 = np.percentile(data, 75)
IQR = Q3 - Q1

# Define the bounds for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Identify outliers
outliers = data[(data < lower_bound) | (data > upper_bound)]

print("Outliers:", outliers)
```

DATA TRANSFORMATION

- Data Transformation is the process of converting raw data into a format suitable for analysis.
- It involves adjusting, modifying, or creating new features to ensure the data aligns with model requirements and improves the effectiveness of data analysis and machine learning models.



KEY TECHNIQUES

- **Creating Interaction Terms:**
- Generate features by combining multiple columns.
- For example, multiplying or concatenating two features may reveal more complex relationships between them.
- df['Interaction'] = df['Feature1'] * df['Feature2']
- **Binning or Discretization:**
- Convert continuous variables into categorical bins, such as segmenting age groups into young, middle-aged, and senior.
- df['Age_Bin'] = pd.cut(df['Age'], bins=[0, 18, 35, 50, 80], labels=['Child', 'Youth', 'Adult', 'Senior'])

DATA VISUALIZATION

- Data Visualization is the graphical representation of information and data.
- It enables analysts and data scientists to visualize patterns, trends, and correlations that might be difficult to identify in raw data form.
- **Why is it Important?**
  - Humans process visual information much more efficiently than text or numbers.
  - Allows for intuitive insights into large datasets and assists in data-driven decision-making.

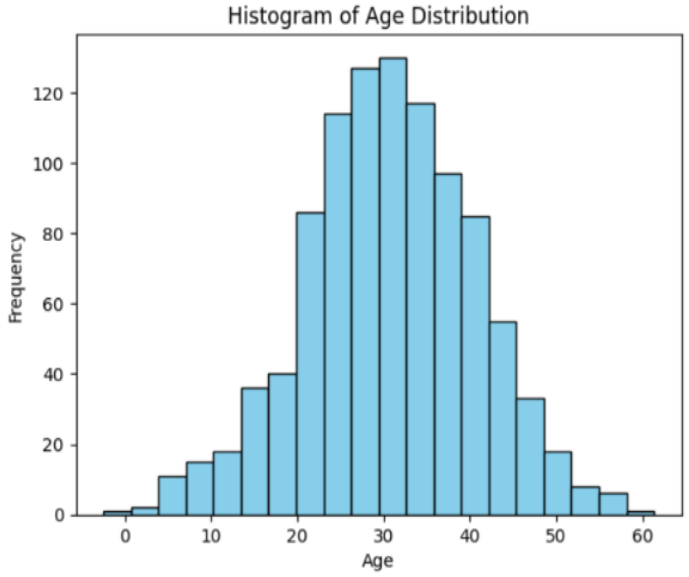
IMPORTANCE OF DATA VISUALIZATION

VISUALIZATION TYPES

- **Histograms:**
- A histogram shows the distribution of a single continuous variable by dividing the data into bins (intervals) and counting the number of observations in each bin.
- Use Case: Visualizing the frequency distribution of numerical data.
- Helps With: Understanding the shape of the data distribution (normal, skewed, etc.), identifying modes, and spotting outliers.

EXAMPLE

- Let's Suppose a dataset containing the ages of 1000 people, and you want to see the distribution of ages.
- In this histogram, each bar represents the frequency of people within a certain age range (bins).
- If most bars are centered around the age of 30, it suggests that the majority of people in the dataset are in that age range.
- Histograms are particularly useful for identifying skewness (e.g. if the data is right- or left-skewed) and for detecting outliers in the distribution of the data.

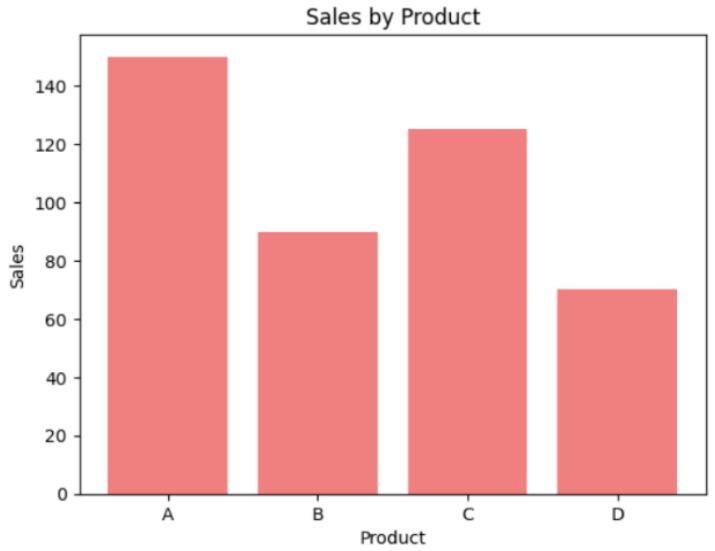


BAR CHARTS

- Bar charts display categorical data with rectangular bars representing the frequency or value of each category.
- The height or length of the bars corresponds to the value it represents.
- **Use Case:**
  - Visualizing counts or comparisons across different categories.
- **Helps With:**
  - Comparing different groups (e.g., sales in different regions, customer preferences by product category).

EXAMPLE

- Imagine you have sales data for four different products (A, B, C, and D) in a store, and you want to compare which product had the highest sales.
- A bar chart will clearly show the comparison between these products.
- The bar chart presents a straightforward comparison between the sales of different products.
- Each bar represents the sales value for a particular product, allowing you to see which product performed best (in this case, Product A) and which performed the least (Product D).
- Bar charts are ideal for comparing values across different categories.



SCATTER PLOT

- A scatter plot visualizes the relationship between two continuous variables by plotting individual data points.
- Each point represents one observation with its position determined by its values for two variables.
- **Use Case:**
  - Understanding the correlation between two numerical variables.
- **Helps With:**
  - Identifying relationships (positive, negative, or none), spotting outliers, and understanding trends

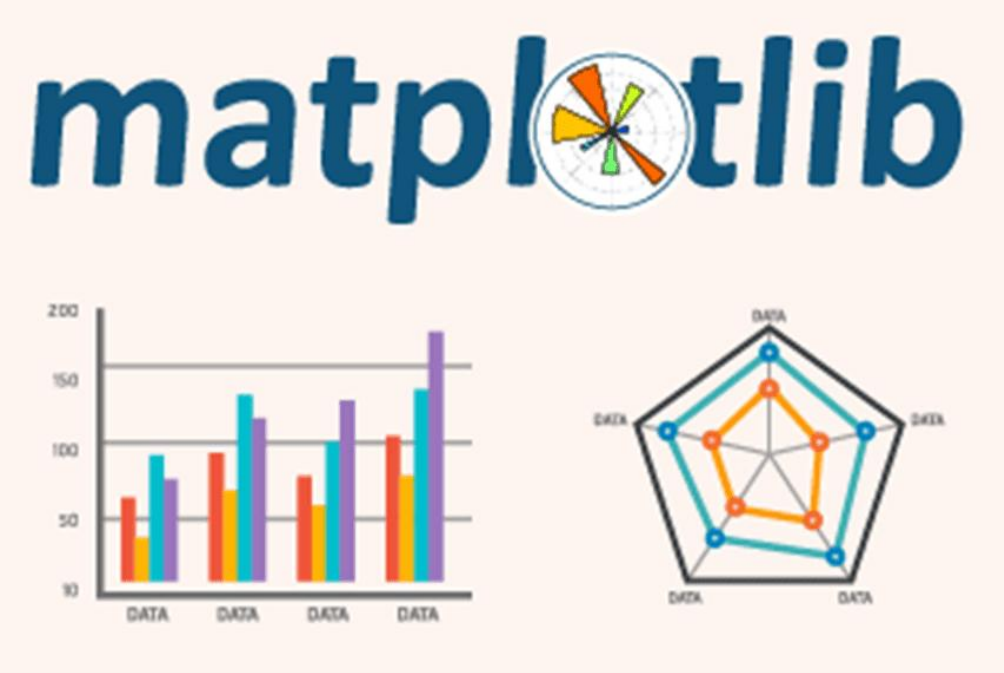
EXAMPLE

- You have data showing the number of hours students studied and their corresponding test scores.
- A scatter plot can help you see whether there is a relationship between hours studied and test scores (e.g., do students who study more score higher?).
- The scatter plot shows how test scores change as the number of hours studied increases.
- In this case, there is a positive correlation (as hours studied increase, test scores also increase).
- Scatter plots are useful for identifying relationships between two variables and detecting potential outliers that don't follow the overall trend.

VISUALIZATION TYPES

- **Box Plot:**
  - Displays the distribution of data through five summary statistics: minimum, first quartile, median, third quartile, and maximum. Useful for identifying outliers and comparing distributions.
- **Heatmap:**
  - Provides a color-coded matrix that shows correlations between multiple variables. It's particularly helpful for quickly spotting strong positive or negative correlations.
- **Line Plot:**
  - Often used to visualize data trends over time. Ideal for time-series data, showing how a variable evolves.

VISUALIZATION LIBRARIES



MATPLOTLIB

- Matplotlib is a powerful library in Python for data visualization
- Installation:
  - **pip install matplotlib**
- Import:
  - **import matplotlib.pyplot as plt**

CREATING SIMPLE LINE CHART

```
import matplotlib.pyplot as plt

# Create a figure with a specific size
# (width=10 inches, height=6 inches)
plt.figure(figsize=(10, 6))

# Your plotting code goes here
plt.plot([1, 2, 3], [4, 5, 6])

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

SOME MORE CONFIGURATION

```
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

# Blue line with circle markers
plt.plot(x, y, marker='o', linestyle='-', color='b')

plt.xlabel("X-axis Label")
plt.ylabel("Y-axis Label")
plt.title("Simple Line Plot")
plt.grid(True)
plt.show()
```

SCATTER PLOT

```
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

# Red asterisk markers
plt.scatter(x, y, color='r', marker='*')
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Scatter Plot")
plt.show()
```

CREATING BAR CHART

```
categories = ['A', 'B', 'C', 'D']
values = [5, 7, 3, 8]

plt.bar(categories, values, color='green')
plt.xlabel("Categories")
plt.ylabel("Values")
plt.title("Bar Chart Example")
plt.show()
```

CREATING HISTOGRAM

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

# Generate 1000 random numbers
data = np.random.randn(1000)
plt.hist(data, bins=30, color='purple', edgecolor='black')
plt.xlabel("Bins")
plt.ylabel("Frequency")
plt.title("Histogram Example")
plt.show()
```

CREATING PIE CHART

```
sizes = [20, 30, 25, 25]
labels = ['Python', 'Java', 'C++', 'JavaScript']
colors = ['blue', 'red', 'green', 'yellow']

plt.pie(sizes, labels=labels, colors=colors, autopct='%1.1f%%')
plt.title("Programming Language Popularity")
plt.show()
```

SEABORN

- Seaborn is a Python data visualization library built on top of Matplotlib.
- It provides a high-level interface for creating attractive and informative statistical graphics.
- Installing Seaborn:
  - pip install seaborn
- Importing Seaborn
  - import seaborn as sns
  - import matplotlib.pyplot as plt

MATPLOTLIB VS SEABORN

Feature	Matplotlib	Seaborn
Customization	Requires manual styling	Has built-in themes and color palettes
Data Handling	Works with lists and arrays	Works seamlessly with Pandas DataFrames
Plot Types	Basic plots (line, scatter, bar)	Advanced statistical plots (histograms, boxplots, violin plots, heatmaps)
Default Styling	Minimal styling	Elegant and visually appealing by default
Built-in Functions	Limited	Provides statistical analysis capabilities

EXAMPLE

Conclusion:

Use Matplotlib when you need fine-grained control over plots (customization, multiple subplots, animations).

Use Seaborn when you need elegant, statistical visualizations with minimal code.

```python
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Sample Data
data = pd.DataFrame({
    "x": [1, 2, 3, 4, 5],
    "y": [10, 20, 15, 25, 30]
})

sns.lineplot(x="x", y="y", data=data)
plt.title("Seaborn Line Plot")
plt.show()
```

LET'S DO VISUALIZATION

- Install Required Libraries:
- pip install numpy matplotlib
- Import Libraries and Create Sales Data

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

# Given sales revenue for 10 days (in dollars)
sales = np.array([1200, 1500, 1100, 1800, 2500, 1700, 1400, 1600, 1300, 1900])
# Adjust Days from 1 to 10
days = np.arange(1, 11)

print("Sales Data:", sales)
```

BASIC ANALYSIS

```
total_sales = np.sum(sales)
average_sales = np.mean(sales)
highest_sales = np.max(sales)
lowest_sales = np.min(sales)

print(f"Total Sales: {total_sales}")
print(f"Average Sales: {average_sales:.2f}")
print(f"Highest Sales: {highest_sales}")
print(f"Lowest Sales: {lowest_sales}")
```

LINE CHART - SALES TREND OVER DAYS

```
plt.figure(figsize=(8, 5))
plt.plot(days, sales, marker='o', linestyle='-', color='b', label="Sales")

plt.axhline(average_sales, color='r', linestyle='--', label="Average Sales")

plt.xlabel("Days")
plt.ylabel("Sales Revenue ($)")
plt.title("Sales Trend Over 10 Days")
plt.legend()
plt.grid(True)
plt.show()
```

BAR CHART - SALES COMPARISON BY DAY

```
plt.figure(figsize=(8, 5))
plt.bar(days, sales, color='c', label="Sales")

plt.xlabel("Days")
plt.ylabel("Sales Revenue ($)")
plt.title("Sales per Day")
plt.legend()
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

PIE CHART - SALES CONTRIBUTION PER DAY

```
plt.figure(figsize=(6, 6))
plt.pie(sales, labels=days,colors=plt.cm.Paired.colors,autopct='%1.1f%%')

plt.title("Sales Contribution Per Day")
plt.show()
```

ASSIGNMENT

- **Employee Salary Analysis & Visualization:**
- Mindsprint wants to analyze the salary distribution of its employees across different departments.
- The dataset contains details such as Employee ID, Name, Age, Department, Salary, and Experience in years.
- use **Pandas** for analysis and **Matplotlib** for visualization.

TASKS

1. Display Basic Information About the Dataset
2. Find the Average Salary in Each Department
3. Find the Highest and Lowest Salary in the Company
4. Find the Employee with the Maximum Salary
5. Count the Number of Employees in Each Department
6. Plot a Bar Chart Showing the Number of Employees in Each Department
7. Plot a Pie Chart Showing Salary Distribution by Department
8. Scatter Plot Showing Salary vs Experience
9. Find Employees Who Earn More Than the Average Salary
10. Sort Employees Based on Salary in Descending Order