OHM SRI SAIRAM

Data Wrangling and Exploratory Data Analysis

Prepared by

N.Siva

Assistant Professor AIML Department

Annamacharya University Rajampet.

Title of the Course : **Data Wrangling and Exploratory Data Analysis**
Category : **PC**
Year : **III**
Semester : **I**
Course Code : **24AAIM51T**
Branch/es : **AI&ML**

Lecture Hours	Tutorial Hours	Practice Hours	Credits	Total Hours
3	0	0	3	52

Course Objectives:

1. Understand the fundamentals of EDA and data wrangling using Python for structured data.
2. Apply data cleaning and pre-processing techniques to prepare high-quality datasets.
3. Extract and manage data from Excel, PDFs, and databases using Python and SQL.
4. Analyse and visualize data to uncover patterns and insights.
5. Perform real-world EDA and collect data through web scraping for machine learning.

Course Outcomes:

At the end of the course, the student will be able to

1. Demonstrate understanding of EDA and perform data wrangling on structured datasets.
2. Apply data preprocessing techniques to handle missing values, duplicates, and outliers.
3. Extract, process, and integrate data from Excel files, PDFs, and relational databases.
4. Analyze data using statistical methods and visualize insights through various plots.
5. Conduct end-to-end EDA on real-world datasets and perform web scraping for data collection.

Unit 1 Introduction to EDA and Data Wrangling **10**

Introduction to Data Science and EDA, Importance of EDA in Data Science Life Cycle, What Is Data Wrangling, Importance of Data Wrangling, Tasks of Data Wrangling, Data Wrangling Tools, Introduction to Python for Data Wrangling, Python Basics for Data Wrangling, Handling Structured Data: CSV, JSON, and XML Formats, Data Meant to Be Read by Machines.

Unit 2 Data Wrangling and Preprocessing **10**

Handling Missing Data (mean, median, drop, and interpolation), Dealing with Duplicates, Removing duplicates and Fuzzy Matching, Outliers, and Anomalies, Encoding Categorical Variables (Label, One-hot), Data Transformation: Scaling, Normalization, Binning, Data Types Conversion and Data Type Casting, Measures of central tendency and disperency.

Unit 3 Working with Excel Files, PDFs, and Databases 11

Installing Python Packages for Data Wrangling, Parsing Excel Files, Programmatic Approaches to PDF Parsing, Converting PDF to Text (pdfminer), Acquiring and Storing Data, Introduction to Databases for Data Wrangling, Relational Databases: MySQL, Data Exploration: Table functions and Joining Datasets.

Unit 4 Exploratory Data Analysis **10**

Visualization with Matplotlib and Seaborn**,** Customizing Plots: Titles, Legends, Labels and Themes Distribution Plots: Histograms, Boxplots, KDE, Bar Charts, Count Plots, Pie Charts, Scatter Plots, Pair Plots, Heatmaps, Correlation and Covariance Analysis, Advanced Visuals: Violin Plots, Strip Plots, Swarm Plots, Multivariate Visualization and Subplots

Unit 5 EDA Case Studies and Web Scraping **11**

Step-by-step EDA on Sample Datasets (Titanic, Iris, Sales, etc.), Feature Engineering Techniques in EDA, EDA Report Generation using Python Notebooks, Preparing Data for Machine Learning Models, Web Scraping: What to Scrape and How, Analyzing and Parsing Web Pages with LXML and XPath, Advanced Web Scraping Using Selenium.

Textbook:

1. Jake VanderPlas, Python Data Science Handbook: Essential Tools for Working with Data, O'Reilly, 2016.

Reference Books:

1. Jacqueline Kazil and Katharine Jarmul, Data Wrangling with Python: Tips and Tools to Make Your Life Easier, O'Reilly Media, Sebastopol, CA, USA, 2016
2. Web Scraping with Python: Collecting More Data from the Modern Web" – Ryan Mitchell, O'Reilly Media.
3. Wes McKinney, Python for Data Analysis, 2nd Edition, O'Reilly, 2018.
4. Allen B. Downey, Think Stats: Probability and Statistics for Programmers, O'Reilly, 2014.

CO-PO Mapping:

Course Outcomes	Engineering Knowledge	Problem Analysis	Design/Development of solutions	Conduct investigations of complex problems	Engineering tool usage	The engineer and the world	Ethics	Individual and Collaborative Team work	Communication	Project management and finance	Life-long learning	PSO1	PSO2	PSO3
24AAIM51T.1	3	2	1	1	3	-	-	-	-	-	2	3	2	1
24AAIM51T.2	2	3	2	2	3	-	-	-	-	-	2	3	3	2
24AAIM51T.3	2	3	2	3	2	-	-	-	-	-	1	3	3	2
24AAIM51T.4	1	2	2	2	3	-	-	-	2	-	1	2	3	2
24AAIM51T.5	2	3	3	3	3	-	-	-	2	1	2	3	3	3

Unit 1 Introduction to EDA and Data Wrangling **10**

Introduction to Data Science and EDA, Importance of EDA in Data Science Life Cycle, What Is Data Wrangling, Importance of Data Wrangling, Tasks of Data Wrangling, Data Wrangling Tools, Introduction to Python for Data Wrangling, Python Basics for Data Wrangling, Handling Structured Data: CSV, JSON, and XML Formats, Data Meant to Be Read by Machines.

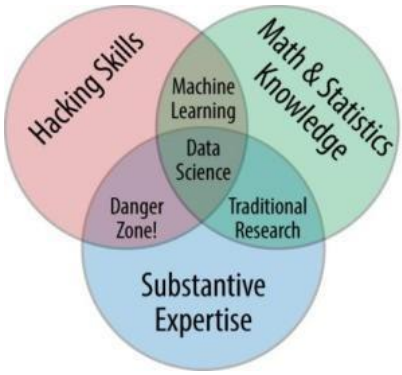
Introduction to Data Science

Data Science is a multidisciplinary field that combines statistical methods, programming skills, and domain knowledge to extract meaningful insights and knowledge from structured and unstructured data. It is at the heart of modern technological advancements, empowering industries to make data–driven decisions and innovate rapidly.

What is Data Science?

Data Science involves the process of collecting, processing, analyzing, and interpreting data to uncover patterns and support decision–making. It draws techniques from fields such as:

- **Statistics** - for understanding data distribution and relationships
- **Computer Science** - for programming and data engineering
- **Mathematics** - for algorithm design and modeling
- **Domain Expertise** - for applying results meaningfully in specific industries



Key Components of Data Science

1. **Data Collection:** Gathering data from various sources such as databases, web APIs, sensors, or spreadsheets.
2. **Data Cleaning and Preprocessing**: Handling missing values, removing noise, and converting raw data into usable formats.
3. **Exploratory Data Analysis (EDA):** Visualizing and summarizing the main characteristics of data using graphs and statistics.
4. **Data Modeling**: Applying algorithms (e.g., regression, classification, clustering) to find patterns or make predictions.
5. **Model Evaluation**: Assessing the performance of models using metrics like accuracy,

precision, recall, or RMSE.

a. **Deployment and Visualization**: Presenting the results in interactive dashboards or embedding models in applications for real-time usage.

Common Tools and Technologies

- **Programming Languages**: Python, R, SQL
- **Libraries/Frameworks**: Pandas, NumPy, Scikit-learn, TensorFlow, Matplotlib, Seaborn
- **Databases**: MySQL, PostgreSQL, MongoDB
- **Big Data Tools**: Hadoop, Spark
- **Cloud Platforms**: AWS, Azure, Google Cloud
- **Visualization Tools**: Tableau, Power BI, Plotly

Applications of Data Science

Data Science is transforming numerous sectors, such as:

- **Healthcare**: Predicting disease, personalized treatments
- **Finance**: Fraud detection, credit risk analysis
- **Retail**: Customer segmentation, recommendation engines
- **Transportation**: Route optimization, predictive maintenance
- **Agriculture**: Crop yield prediction, smart farming
- **Entertainment**: Content recommendations, trend analysis

Data Scientist Roles

- **Data Analyst** - Analyzes and visualizes data for business insights.
- **Machine Learning Engineer** - Builds predictive models and AI systems.
- **Data Engineer** - Designs and maintains data infrastructure and pipelines.
- **Business Intelligence Analyst** - Focuses on KPIs and business metrics.
- **Data Scientist** - A hybrid role that spans all aspects from data wrangling to decision-making.

Data Science Life Cycle

The **Data Science Life Cycle** is a structured approach that outlines the major steps involved in solving a data-driven problem using data science methodologies. Just like the software development life cycle (SDLC), the data science life cycle ensures that data projects are systematic, repeatable, and goal-oriented.

1. Problem Definition

This is the **most critical phase**, as a poorly defined problem leads to poor outcomes.

Key §uestions:

- What is the business problem?

- What are the success criteria?
- What value will the solution bring?

Example:
For a bank, the goal might be **predicting loan default** to reduce risk.

2. Data Collection

Once the problem is defined, the next step is to **gather relevant data. Sources:**

- Databases (MySQL, MongoDB)
- APIs (Twitter API, Weather API)
- Web scraping
- Public datasets (Kaggle, UCI, data.gov.in)
- IoT devices or mobile applications

Activities:

- Identify relevant data sources
- Connect and extract data
- Store in a data warehouse or data lake

3. Data Cleaning and Preprocessing

Raw data is often messy, incomplete, or inconsistent. Cleaning it ensures the quality of analysis.

Tasks Involved:

- Handling **missing values** (imputation or removal)
- Treating **outliers**
- Removing **duplicates**
- Converting **data types**
- Encoding **categorical variables Tools:**

- Python (Pandas, NumPy)
- R
- SQL

4. Exploratory Data Analysis (EDA)

EDA helps in understanding the underlying patterns in the data and forms the basis for modeling decisions.

Activities:

- Summary statistics (mean, median, mode, etc.)
- Visualization (histograms, scatter plots, box plots)
- Correlation analysis
- Univariate, bivariate, and multivariate analysis

Goals:

- Spot trends and outliers
- Form hypotheses
- Identify data transformations needed

5. Feature Engineering and Selection

This involves creating new input variables (features) and selecting the most relevant ones for modeling.

Techniques:

- Binning, scaling, normalizing
- Creating interaction terms
- Removing highly correlated variables
- Using feature importance scores

Why It's Important:

Good features can dramatically improve the accuracy of models.

6. Model Building

This is the core stage where machine learning or statistical models are trained to learn from the data.

Model Types:

- **Supervised Learning**: Regression, Classification
- **Unsupervised Learning**: Clustering, Dimensionality Reduction
- **Reinforcement Learning**: Agent-based decision making

Process:

- Split data into training and testing sets
- Choose algorithms (e.g., decision trees, SVM, neural networks)
- Train the model on the training data

7. Model Evaluation

The model is assessed for its performance using various metrics, depending on the problem type.

Common Metrics:

- Accuracy, Precision, Recall, F1-score (for classification)
- RMSE, MAE, R^2 (for regression)
- Confusion matrix, ROC-AUC curve

Techniques:

- Cross-validation
- Hyperparameter tuning
- Model comparison

8. Model Deployment

Once a model performs well, it's deployed to make real-time or batch predictions.

Deployment Methods:

- REST APIs
- Web applications (Flask, Django)
- Cloud platforms (AWS SageMaker, GCP AI Platform)

Considerations:

- Scalability
- Monitoring and feedback loops
- Version control

9. Monitoring and Maintenance

After deployment, the model must be continuously monitored for **data drift**, **performance degradation**, and **user feedback**.

Activities:

- Retraining with new data

- Logging predictions
- Alerting when thresholds are breached

Introduction to EDA (Exploratory Data Analysis)

Exploratory Data Analysis (EDA) is the initial and critical phase in any data science or machine learning project. It involves analyzing datasets to summarize their main characteristics, often using visual methods. The goal of EDA is to understand the structure, patterns, and anomalies in the data before applying any modeling techniques.

Importance of EDA in the Data Science Life Cycle

Exploratory Data Analysis (EDA) is a **foundational and essential phase** in the Data Science Life Cycle. It serves as the bridge between raw data and actionable insights. While data collection and cleaning prepare the data, EDA allows data scientists to **understand** the data, **ask the right questions**, and **guide modeling decisions**.

1. Understanding the Data

EDA provides a **clear picture** of the dataset's structure, content, and quality. It answers questions like:

- What variables are present?
- What are the distributions?
- Are there missing or erroneous values?

2. Detecting Data §uality Issues

EDA helps in identifying:

- **Missing values**
- **Outliers**
- **Inconsistent data types**
- **Duplicate records**

This ensures that further steps are not built on faulty data.

3. Identifying Patterns and Trends

With EDA, you can discover:

- Relationships between variables (e.g., age and salary)
- Groupings or clusters
- Seasonal or time-based trends

These insights inform **feature selection** and **business understanding**.

4. Feature Engineering Guidance

EDA shows which variables are:

- Most relevant
- Redundant

- Non−informative

This helps in creating or transforming variables to improve model performance.

5. Improves Model Accuracy

A good understanding of data through EDA leads to:

- Better preprocessing
- Informed algorithm selection
- Better tuning of model parameters All of which enhance **predictive accuracy**.

6. Assumptions Testing

Many models assume certain data characteristics (like linearity, normal distribution). EDA helps:

- Validate those assumptions
- Decide whether transformation (e.g., log−scaling) is needed

7. Visual Storytelling

EDA uses graphs and plots to:

- Communicate findings to non−technical stakeholders
- Create compelling narratives around data
- Support business decisions visually

1. What is Data Wrangling?

- **Definition**: Data wrangling (also called data munging) is the process of cleaning, organizing, and transforming raw data into a structured and usable format for analysis.
- **Objective**: To prepare data so it is suitable for machine learning, analytics, and decision-making.
- **Example**: Converting unstructured survey responses (text) into a structured tabular format (Excel/CSV).

2. Steps in Data Wrangling

(a) Cleaning

- **Meaning**: Removing errors, missing values, duplicates, or inconsistencies in data.
- **Example**: A dataset of students has some missing ages and duplicate records.

Code Example:

```
import pandas as pd #

Sample raw data

data = {'Name': ['John', 'Alice', 'Bob', 'Alice'], 'Age':
```

```
    [20, None, 22, None],
    'Marks': [85, 90, None, 90]}

df = pd.DataFrame(data) print("Original
Data:\n", df)
# Handling missing values
df['Age'].fillna(df['Age'].mean(), inplace=True)  # replace missing Age with mean
df.drop_duplicates(inplace=True)              # remove duplicates
df['Marks'].fillna(0, inplace=True)          # replace missing Marks with 0

print("\nCleaned Data:\n", df)
```

(b) Structuring

- **Meaning**: Converting raw/unorganized data into tabular or structured formats like CSV, JSON, or SQL.
- **Example**: Taking JSON API data and converting it into a Pandas DataFrame.

Code Example:

```
import pandas as pd
import json

# JSON Data
json_data = ''' [
 {"Name":"John", "Age":20, "Marks":85},
 {"Name":"Alice", "Age":22, "Marks":90}
] '''
# Convert JSON → DataFrame
df = pd.read_json(json_data)
```

```
print("Structured Data:\n", df)
```

(c) Enriching

- **Meaning**: Adding external or derived information to improve dataset quality.
- **Example**: Adding a new column "Pass/Fail" based on marks.

Code Example:

```
df['Result'] = df['Marks'].apply(lambda x: 'Pass' if x >= 50 else 'Fail')

print("Enriched Data:\n", df)
```

(d) Validating

- **Meaning**: Ensuring data accuracy and consistency (checking ranges, formats, or rules).
- **Example**: Student ages must be >0 and <100.

Code Example:

```
# Check for invalid ages

invalid_ages = df[(df['Age'] <= 0) | (df['Age'] > 100)]

print("Invalid Ages:\n", invalid_ages)


# Remove invalid data

df = df[(df['Age'] > 0) C (df['Age'] < 100)]

print("Validated Data:\n", df)
```

(e) Transforming

- **Meaning**: Converting data into the required format (scaling, normalization, encoding).
- **Example**: Normalize marks to a 0–1 range.

Code Example:

```
from sklearn.preprocessing import MinMaxScaler
```

```
scaler = MinMaxScaler()

df['Normalized_Marks'] =

scaler.fit_transform(df[['Marks']]) print("Transformed

Data:\n", df)
```

. Importance of Data Wrangling

1. Improves Data Quality

Data quality refers to the **accuracy, consistency, completeness, and reliability** of a dataset. Raw data often contains **missing values, duplicate entries, outliers, and inconsistent formats**. Poor data quality leads to wrong analysis, biased machine learning models, and poor business decisions. Data wrangling improves data quality by applying systematic **cleaning, validating, and correcting techniques**.

For example, in a **healthcare dataset**, patient records may have missing ages, wrong gender entries, or duplicate rows. Wrangling ensures that ages are filled with averages or medians, gender is standardized (e.g., "M/F" instead of "Male/Female"), and duplicate patient entries are removed.

Code Example (Improving Data Quality):

```
import pandas as pd

# Sample raw dataset
data = {'PatientID':[1,2,2,3],
    'Age':[25, None, None, 150],
    'Gender':['M','Female','Female','
M']} df = pd.DataFrame(data)
print("Original Data:\n", df)

# Handle missing values
df['Age'].fillna(df['Age'].mean(),
```

```
inplace=True)

# Remove invalid ages (>120 years
unrealistic) df = df[(df['Age'] > 0) C (df['Age']
< 120)]

# Standardize gender
df['Gender'] = df['Gender'].replace({'Female':'F', 'Male':'M'})

# Remove duplicates
df = df.drop_duplicates()

print("\nImproved Data Quality:\n",

df)
```

After wrangling, the dataset is **clean, consistent, and reliable**, ensuring that healthcare decisions or ML predictions (like disease risk) are accurate.

2. Saves Time

In real-world projects, analysts and data scientists spend **60-80% of their time cleaning and preparing data** before they can analyze or build models. Manual cleaning of large datasets (e.g., 1 million records in Excel) is **time-consuming and error-prone**. Data wrangling automates repetitive tasks like **removing duplicates, filling missing values, merging datasets, and applying transformations**, saving huge amounts of time.

For example, in a **retail dataset** with thousands of sales transactions, manually finding duplicates or missing product prices could take days. With Python's Pandas, the same task can be done in **a few lines of code**.

Code Example (Saving Time with Automation):

```
import pandas as pd

# Sample raw sales data
sales = {'OrderID':[101,102,102,103],
    'Product':['Shoes','Shirt','Shirt','Watch'],
    'Price':[500, None, None, 2000]}
df = pd.DataFrame(sales)
print("Original Data:\n", df)
```

```
# Automating wrangling
df['Price'].fillna(df['Price'].mean(), inplace=True) # Fill missing
prices df = df.drop_duplicates()        # Remove duplicates

print("\nWrangled Data (Time-Saving):\n", df)
```

What could take **hours of manual corrections** is completed in **seconds** using wrangling tools. This allows data scientists to **spend more time on insights, modeling, and decision-making** rather than fixing messy data.

3. Enhance Decision Making

Data wrangling enhances decision-making by converting noisy, incomplete, or inconsistent data into a reliable form that managers, analysts, or AI systems can interpret correctly. Decisions made on raw, unprocessed data often lead to **incorrect business strategies** because patterns may be hidden in the noise. Wrangling ensures that the data is **clean, structured, and enriched** with relevant features, which makes analysis more meaningful. For example, imagine an **e-commerce company** analyzing customer purchases. Raw sales data might contain missing transaction values or duplicate records. If management makes decisions based on such flawed data, they may overestimate revenue or underestimate customer churn. But after wrangling (removing duplicates, filling missing values, and categorizing customers by age group), the business gets **clearer insights** such as "young customers buy more electronics, while middle-aged customers buy more home products."

Code Example:

```
import pandas as pd

# Sample data with missing values
data = {'CustomerID': [1, 2, 3, 3],
    'Purchase': [200, None, 300,
300]} df = pd.DataFrame(data)

# Cleaning step: fill missing and remove duplicates
df['Purchase'].fillna(df['Purchase'].mean(),
inplace=True) df = df.drop_duplicates()

print("Wrangled Data:\n", df)
print("Average Purchase:", df['Purchase'].mean())
```

With wrangled data, decision-makers rely on **accurate KPIs (Key Performance**

Indicators), leading to better product strategies, customer targeting, and marketing decisions.

4. Supports Machine Learning

Machine learning models heavily rely on **high-quality input data**. Raw datasets often contain categorical variables in string form, missing values, inconsistent scales, or outliers. If this messy data is fed directly into ML models, the results are **biased, inaccurate, or unusable**. Data wrangling supports ML by preparing the dataset through **feature engineering, scaling, encoding, and normalization.**

For example, a dataset predicting student performance may include **categorical columns (like gender, school name), numerical columns (like marks, age), and missing values**. A model like Logistic Regression or Random Forest cannot directly handle missing values or categorical strings. Wrangling ensures that all missing values are filled, categorical data is encoded (One-hot encoding or Label encoding), and numerical values are normalized.

Code Example:

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
# Raw data
df = pd.DataFrame({'Name':['John','Alice','Bob'],
        'Gender':['M','F','M'],
        'Marks':[85, None, 90]})

# Handle missing values
df['Marks'].fillna(df['Marks'].mean(), inplace=True)

# Encode categorical values
encoder = LabelEncoder()
df['Gender'] = encoder.fit_transform(df['Gender'])

# Normalize marks
scaler =
MinMaxScaler()
```

```
df[['Marks']] = scaler.fit_transform(df[['Marks']])

print("Wrangled Data for ML:\n", df)
```

With wrangled data, ML models learn **better patterns** and provide **more accurate predictions**, such as predicting which students may fail or which customers are likely to churn.

3. Integrates Diverse Sources

In today's digital era, data comes from **multiple sources**: relational databases (MySQL, PostgreSQL), NoSQL databases (MongoDB, Cassandra), APIs (Twitter, weather, stock data), flat files (CSV, Excel), and unstructured data (PDFs, logs, images). To extract insights, these sources must be **merged into a single, consistent dataset**. Data wrangling acts as the glue that integrates these diverse sources by cleaning, aligning formats, and merging them seamlessly.

For example, consider a **bank** that wants to evaluate loan risks. Data may come from:

- Customer details in **Excel** files.
- Transaction history in **MySQL database**.
- Credit score from a **third-party API**.
- Social media data from **JSON APIs**.

Without wrangling, this information remains scattered. By integrating them, the bank can build a **comprehensive customer profile** and decide whether to approve or reject a loan.

Code Example:

```
# Import customer data from Excel
df_customers =
pd.read_excel("customers.xlsx")

# Import transaction history from CSV
df_transactions = pd.read_csv("transactions.csv")

# Merge on CustomerID
df_merged = pd.merge(df_customers, df_transactions, on="CustomerID")
print("Integrated Dataset:\n", df_merged.head())
```

By integrating multiple data sources, organizations get a **360-degree view of customers, markets, or processes**, leading to smarter decisions, personalized recommendations, and better risk management.

Tasks of Data Wrangling

Data Wrangling involves multiple tasks that help convert raw, unstructured or messy data into clean and structured format suitable for analysis or Machine Learning.

1. Data Collection
2. Data Cleaning
3. Data Transformation
4. Data Integration
5. Data Reduction
6. Data Validation
7. Data Enrichment

1. Data Collection

Data collection is the **first step** of data wrangling, where data is gathered from various sources like **CSV, Excel, databases (MySQL, PostgreSQL), APIs, JSON files, web scraping, or IoT sensors**. Real-world data rarely exists in a single location, so it must be aggregated from multiple platforms. The challenge is that sources may use different formats (e.g., JSON vs. CSV) or

inconsistent schemas (e.g., "CustomerID" vs. "Cust_ID"). A data wrangler must ensure all required information is available for analysis. For example, a retail business may collect **sales data** from a CSV, **customer details** from Excel, and **product reviews** from a web API. Collecting structured and unstructured data is the foundation for further wrangling.

Python Example:

```
import pandas as pd

# Collecting data from different
sources df_csv =
pd.read_csv("sales.csv")
df_excel =
pd.read_excel("customers.xlsx") df_json
= pd.read_json("feedback.json")

print("CSV Sample:\n", df_csv.head())
print("Excel Sample:\n", df_excel.head())
print("JSON Sample:\n", df_json.head())
```

2. Data Cleaning

Data cleaning ensures that data is **accurate, complete, and consistent** by handling missing

values, duplicates, outliers, and incorrect entries. Raw datasets often contain **human errors, sensor glitches, or incomplete records**. Cleaning involves steps like **filling missing values with averages/medians, removing duplicate records, correcting inconsistent formats (e.g., M/F vs Male/Female), and detecting outliers**. For example, in a student dataset, some ages may be missing or recorded as 200, which is invalid. Cleaning ensures that the dataset is usable for analysis and models.

Python Example:

```
df = pd.DataFrame({'Name':['John','Alice','Bob','Alice'],
                   'Age':[20, None, 200, None],
                   'Marks':[85, 90, None, 90]})

# Fill missing values
df['Age'].fillna(df['Age'].mean(), inplace=True)
df['Marks'].fillna(df['Marks'].median(),
inplace=True)

# Remove duplicates
df.drop_duplicates(inplace=True
)

# Remove outliers (Age > 100 invalid)
df = df[(df['Age'] > 0) & (df['Age'] < 100)]
print("Cleaned Data:\n", df)
```

3. Data Transformation

Data transformation converts data into the **desired format or structure**. It involves **normalization, scaling, encoding categorical values, type conversions, feature engineering, and aggregations**. This ensures that data is **machine-readable** and suitable for ML models. For example, sales amounts can be normalized to a scale of 0–1, or gender (M/F) can be converted into numerical codes. Transformation also includes creating new features, such as "Total_Price = Quantity × Price". Without transformation, ML models may not work properly because they require **numerical and standardized inputs**.

Python Example:

```
from sklearn.preprocessing import MinMaxScaler,

LabelEncoder df =

pd.DataFrame({'Name':['John','Alice','Bob'],
                   'Gender':['M','F','M'],
                   'Salary':[20000, 35000, 50000]})

# Encoding categorical values
encoder = LabelEncoder()
df['Gender'] = encoder.fit_transform(df['Gender'])

# Normalization of Salary
```

```
scaler = MinMaxScaler()
df[['Salary']] = scaler.fit_transform(df[['Salary']])

# Feature Engineering
df['Bonus'] = df['Salary'] *
0.10 print("Transformed
Data:\n", df)
```

4. Data Integration

Data integration combines data from **multiple sources into a unified dataset**. Businesses often store information across different systems — customer details in Excel, transactions in SQL, feedback in JSON. Integration ensures these datasets are merged using common keys (e.g., CustomerID). This is essential for **360-degree views of customers, operations, or markets**. Integration challenges include **schema mismatches, duplicate records, and inconsistent formats**. For example, a bank integrates customer demographics with transaction history to assess loan eligibility.

Python Example:

```
df_customers = pd.DataFrame({'CustomerID':[1,2,3],
'Name':['John','Alice','Bob']})
df_sales = pd.DataFrame({'CustomerID':[1,2,2,3],
'Purchase':[200,300,400,500]})

# Merge on CustomerID
df_merged = pd.merge(df_customers, df_sales, on="CustomerID", how="inner")
print("Integrated Data:\n", df_merged)
```

5. Data Reduction

Data reduction minimizes dataset size while retaining essential information. Large datasets are costly to store and process. Reduction involves **feature selection (dropping irrelevant columns), dimensionality reduction (PCA), sampling, and aggregation**. For example, in a customer dataset, attributes like “Middle Name” or “Full Address” may not be necessary for prediction tasks. Similarly, sampling 10,000 rows from a dataset of 1 million speeds up training without losing major patterns.

Python Example:

```
df =

pd.read_csv("bigdata.csv") #

Drop irrelevant columns
df_reduced = df.drop(columns=['Address','Phone'])

# Take a random sample of 1000 rows
df_sample = df_reduced.sample(n=1000,
```

```
random_state=42) print("Reduced Data Shape:",

df_sample.shape)
```

6. Data Validation

Data validation ensures that data is **correct, meaningful, and within valid ranges**. Validation checks include **range checks (Age > 0), type checks (Age must be integer), format checks (email format), and uniqueness constraints (unique CustomerID)**. Without validation, incorrect data could lead to **misleading analysis and biased ML models**. For example, validating that salaries are non-negative prevents financial reports from showing impossible values.

Python Example:

```
df = pd.DataFrame({'Name':['John','Alice'],
                   'Age':[25,-5],
                   'Email':['john@mail.com','alice(at)mail.com']})

# Range validation
df = df[(df['Age'] > 0) & (df['Age'] < 100)]

# Email format validation (simple
check) df =
df[df['Email'].str.contains("@")]

print("Validated Data:\n", df)
```

7. Data Enrichment

Data enrichment improves datasets by **adding new information** from internal or external sources. It enhances context and value of the data. For example, adding **geographical location** (state, city) to customer addresses, or using **external APIs** to add weather conditions to sales data. Enrichment also includes **derived features** (e.g., Total Sales = Quantity × Price). This step ensures datasets are not only clean but also **more informative** for analysis.

Python Example:

```
df = pd.DataFrame({'Customer':['John','Alice'],
                   'Purchase':[200,300]})

# Adding derived column
df['Discounted_Price'] = df['Purchase'] * 0.90  # 10% discount

# Enrich with external info (example: adding customer segment)
df['Segment'] = ['Premium','Standard']
```

```
print("Enriched Data:\n", df)
```

Data Wrangling Tools

Data wrangling can be performed using different tools ranging from **programming languages**, **ETL tools**, **big data frameworks**, to **spreadsheet applications**. Each has its own strengths and is suitable for specific use cases.

1. Programming Languages

Python (Pandas, NumPy, OpenPyXL)

Python is the most popular programming language for data wrangling because of its simplicity and extensive libraries. **Pandas** is widely used for handling structured data (tables), providing functions to read/write CSV, Excel, JSON, and databases. **NumPy** is essential for numerical operations, arrays, and matrix computations, which form the backbone of machine learning preprocessing. **OpenPyXL** is used specifically for reading and writing Excel files. Together, these libraries allow users to clean, transform, integrate, and validate data with just a few lines of code.

Example:

```
import pandas as pd
df = pd.read_csv("sales.csv")
df['Revenue'] = df['Price'] * df['Quantity']
print(df.head())
```

Used for **ETL, cleaning messy CSVs, ML preprocessing, and automation**.

R (dplyr, tidyr)

R is another powerful programming language, especially for statistical computing and data wrangling. The **dplyr** package provides functions for filtering, grouping, and summarizing data efficiently. The **tidyr** package focuses on reshaping and tidying data, e.g., converting wide datasets into long formats. R is often chosen by statisticians and researchers because of its built-in statistical functions and visualization capabilities.

Example: A researcher analyzing survey data may use `dplyr` to group responses by gender and calculate average satisfaction scores.
Used in **academia, healthcare research, and statistical analysis**.

2. ETL Tools (Extract, Transform, Load)

Talend

Talend is an open-source ETL tool that provides a visual interface for integrating and transforming data. It allows users to drag and drop components to design data pipelines without heavy coding. Talend supports many data sources, including relational databases, cloud storage, and APIs. It is widely used in enterprises for large-scale **data migration, cleansing, and integration**.

Example: A bank may use Talend to extract customer data from Oracle, transform it by removing duplicates, and load it into a PostgreSQL database for analytics.

Apache NiFi

Apache NiFi is a data integration tool designed for **real-time data flows**. It supports the automation of data movement between systems using a simple, drag-and-drop web interface. NiFi is great for **streaming data wrangling** because it handles continuous flows from IoT devices, sensors, or social media feeds.

Example: A logistics company may use NiFi to collect live GPS data from trucks, filter out irrelevant points, and feed it into a dashboard for monitoring delivery performance.

Informatica

Informatica is a leading commercial ETL tool widely used in large organizations. It provides advanced features for **data cleansing, profiling, and governance**. It can handle huge volumes of structured and unstructured data and integrates well with enterprise systems. Informatica is preferred where **data quality and compliance** (e.g., banking, healthcare) are critical.

Example: A healthcare provider uses Informatica to merge patient data from multiple hospitals, ensuring consistency and compliance with regulations like HIPAA.

3. Big Data Tools

Apache Spark

Apache Spark is a powerful big data processing framework that supports large-scale data wrangling and transformation. Unlike Pandas (which runs on a single machine), Spark can handle **distributed data processing across clusters**. It provides APIs for Python (PySpark), R, Scala, and Java. Spark is highly efficient for handling **petabytes of data**, making it useful in machine learning pipelines and real-time analytics.

Example (PySpark):

```
from pyspark.sql import SparkSession
spark =
```

```
SparkSession.builder.appName("Wrangling").getOrCreate() df =
spark.read.csv("bigdata.csv", header=True, inferSchema=True)
df.show(5)
```

Used in **big data environments like e-commerce logs, financial transactions, and IoT streams**.

Hadoop

Apache Hadoop is another big data tool that allows storage and processing of massive datasets using a **distributed file system (HDFS)**. Hadoop works in batch mode and is more suitable for data storage and historical processing compared to real-time Spark. It integrates with tools like Hive (SQL on Hadoop) for wrangling large-scale structured datasets.

Example: A telecom company may use Hadoop to store **years of call records** and then process them to identify call drop patterns and improve service quality.

4. Spreadsheets

Microsoft Excel

Excel is one of the oldest and most widely used tools for data wrangling, especially in small-scale business and academic settings. It allows users to perform basic cleaning (remove duplicates, fill blanks), transformation (formulas, pivot tables), and integration (importing CSV, SQL

connections). While not suitable for big data, Excel is extremely useful for quick analysis and visualization.

Example: A sales manager may use Excel to clean a monthly sales report by removing duplicates and summarizing totals by region using pivot tables.

Google Sheets

Google Sheets provides similar functionality to Excel but is cloud-based and supports **collaboration in real time**. It is often used for lightweight data wrangling tasks in startups or small businesses. Google Sheets supports importing data from APIs, CSVs, and Google Forms, making it easy to collect and process data collaboratively.

Example: A teacher may collect student test results via Google Forms, automatically stored in Sheets, and use formulas to calculate averages and grades.

Additional Tools to Mention

- **OpenRefine** → Specialized for cleaning messy, unstructured data.
- **Power BI / Tableau Prep** → Visualization tools with built-in wrangling.
- **Alteryx** → A drag-and-drop tool for advanced wrangling and analytics.
- **KNIME** → Open-source analytics platform with strong ETL support.

Introduction to Python for Data Wrangling

What is Python?

Python is a high-level, general-purpose programming language that is widely used in data science, artificial intelligence, and machine learning. It is known for its simple syntax, readability, and large ecosystem of libraries, which make it especially suitable for handling data. Python is an interpreted language, meaning code can be written and executed interactively, which is very useful when exploring or wrangling datasets. Because it is cross-platform and open-source, Python has become the language of choice for researchers, data analysts, and engineers who work with both structured and unstructured data.

Why Use Python for Data Wrangling?

Data wrangling is the process of cleaning, transforming, and preparing raw data for analysis. Python is highly suitable for this because it combines ease of use with powerful tools. Unlike traditional spreadsheet-based approaches, Python can handle extremely large datasets, automate repetitive cleaning tasks, and integrate seamlessly with databases, APIs, and big data frameworks. Its libraries allow quick handling of missing values, removal of duplicates, and reshaping of data with just a few lines of code. Another major reason for using Python is its community support — there are countless tutorials, packages, and forums, which means that almost every wrangling challenge has a known Python solution.

Key Libraries for Data Wrangling in Python

Several Python libraries are designed specifically for data wrangling tasks:

- **Pandas**: Provides DataFrame and Series objects for working with tabular data, making it easy to filter, group, merge, and reshape datasets.
- **NumPy**: Supports fast numerical computations, array manipulation, and mathematical operations that form the backbone of data wrangling.
- **OpenPyXL / xlrd**: Used for reading and writing Excel files, which are still common in many organizations.
- **json** and **xml.etree.ElementTree**: Handle JSON and XML parsing, useful for semi-structured data.

- **Matplotlib / Seaborn**: Primarily for visualization, but often used in wrangling to explore distributions and detect outliers.

Together, these libraries form a robust toolkit that allows analysts to move smoothly from raw, messy data to clean, structured datasets ready for analysis or machine learning.

Example using Pandas:

```
import pandas as pd
df = pd.read_csv("sales.csv")         # Load data
df.dropna(inplace=True)               # Clean missing
values df['Revenue'] = df['Price'] * df['Qty']  #
Transform print(df.head())
```

Basic Workflow of Data Wrangling in Python

The workflow of data wrangling in Python usually follows a systematic sequence of steps. First is **data collection**, where raw data is imported from sources like CSV files, Excel sheets, APIs, or databases. Next comes **data cleaning**, where missing values are filled, duplicates removed, and inconsistent formats corrected. After that is **data transformation**, where data types are converted, numerical features are scaled or normalized, and categorical variables are encoded into numeric form. In many projects, **data integration** is required, which involves merging multiple datasets into one consistent dataset. Then comes **validation**, where the wrangled data is checked for accuracy, completeness, and logical consistency. Finally, the clean dataset is **stored or exported** back into files or databases for further analysis and visualization.

Mini Example Workflow:

```
# 1. Collect
df = pd.read_csv("employees.csv")

# 2. Clean
df.drop_duplicates(inplace=True
)
df['Salary'].fillna(df['Salary'].mean(), inplace=True)

# 3. Transform
df['Bonus'] = df['Salary'] * 0.10

# 4. Validate
print(df.info())

# 5. Store
df.to_csv("cleaned_employees.csv", index=False)
```

This workflow demonstrates how Python, with its ecosystem of libraries, supports the entire wrangling pipeline — from ingestion to transformation and storage — making it one of the most effective tools for data wrangling today.

3. Python Basics for Data Wrangling

Python has become the most widely used programming language for data wrangling because of its simplicity, readability, and powerful ecosystem of libraries. In data wrangling, Python allows us to load data from multiple sources, clean inconsistencies, transform it into usable forms, validate the correctness, and prepare it for analysis or machine learning. Libraries like **Pandas** handle structured tabular data, **NumPy** speeds up numerical operations, **Matplotlib/Seaborn** allow for visualization, and packages like **OpenPyXL** or **xlrd** support Excel file handling. Additionally,

Python's built-in `json` and `xml.etree.ElementTree` modules enable parsing semi-structured formats like JSON and XML. Before we dive into libraries, we must understand Python's fundamental programming concepts, since they form the foundation of every wrangling workflow.

1. *Variables and Data Types*

In Python, variables are used to store data values, and unlike some languages, you don't need to declare their type explicitly. The type is inferred when you assign a value. Python supports several built-in data types such as integers (`int`), floating-point numbers (`float`), strings (`str`), 20ooleans (`bool`), and special objects like `None`. Complex data types include lists, dictionaries, tuples, and sets. Understanding these is crucial for wrangling, as different sources may store numeric IDs as text or numbers, and you may need to convert between them. For example, a CSV file might load ages as strings that you must cast into integers before analysis.

Example Code:

```
age = 25        # integer
name = "Alice"  # string
height = 5.7    # float
is_student = True  #
boolean print(type(age),
type(name))
```

2. *Lists*

Lists are ordered, mutable collections in Python, making them ideal for holding rows of data

or temporary values extracted during parsing. Lists can store elements of mixed types, but in data wrangling we usually prefer uniformity (all numbers, all strings). You can add new items with `append()`, remove with `remove()` or `pop()`, and sort them with `sort()`. They are also heavily used with list comprehensions, which let you create new lists from existing ones in a clean and efficient manner. Lists are the stepping stone to pandas DataFrames, since many raw datasets first come as lists of records before being converted to structured formats.

Example Code:

```
scores = [50, 60, 70]
scores.append(80)
scaled = [s / 100 for s in scores]
print(scaled)
```

3. *Dictionaries*

Dictionaries are Python's implementation of key–value pairs, similar to JSON objects. They are crucial for wrangling semi-structured data, as APIs and logs often return data in nested dictionary form. Dictionaries allow you to look up values quickly by key, update entries, or add new keys. For example, when parsing an API response about a user, you might store `{ "name": "Alice", "age": 22, "city": "Hyderabad" }`. This structure is easily converted into a pandas DataFrame column later. Dictionaries are also used for mappings, such as converting categorical codes into labels.

Example Code:

```
student = {"name": "Alice", "age": 22, "marks": [80, 90, 95]}
student["average"] = sum(student["marks"]) / len(student["marks"])
print(student)
```

4. *Control Flow (if, for, while)*

Data wrangling often requires conditional logic. The `if` statement allows branching—processing records differently depending on values. For loops are useful for iterating through rows or files, while while loops repeat until a condition is met. Together, they allow dynamic handling of data anomalies, validation, or filtering. For example, you might write a loop that goes through every row and skips those where a critical field is missing. Control flow structures ensure that you can handle exceptions and customize wrangling logic to your dataset.

Example Code:

```
records = [{"age": 20}, {"age": None}, {"age": 35}]
cleaned = []
for r in records:
    if r["age"] is None:
```

```
        continue
    if 0 < r["age"] < 100:
        cleaned.append(r)
print(cleaned)
```

5. *Functions*

Functions are reusable blocks of code that help modularize the wrangling process. Instead of repeating the same cleaning logic in multiple places, you can define a function once and call it wherever needed. Functions accept parameters, can return results, and make the pipeline easier to

maintain. For example, a function might remove currency symbols from prices and convert them into floats. Wrangling often requires such utility functions, especially when cleaning messy string fields or validating input ranges.

Example Code:

```
def clean_price(value: str) -> float:
    return float(value.replace("₹", "").replace(",", "").strip())

print(clean_price("₹12,345.50"))
```

6. *File Handling*

Since most raw data is stored in files, Python's file handling capabilities are essential. You can open files using `open()` in different modes like read (`"r"`), write (`"w"`), or append (`"a"`). Using a `with` statement ensures files are closed properly after use. For text files, you can read line by line, which is helpful when dealing with large log files. For structured files like CSV or JSON, you can use specialized modules or libraries such as `csv`, `json`, or pandas. Proper file handling ensures smooth integration of external datasets into your wrangling pipeline.

Example Code:

```
with open("data.txt", "r") as f:
    lines = [line.strip() for line in f]
print(lines[:5])
```

7. *Importing Libraries*

Libraries extend Python's capabilities beyond its standard functions. In data wrangling, importing the right library saves time and effort. For example, `import pandas as pd` is a convention that allows easy access to the pandas toolkit. You can import entire modules or specific functions, and libraries like pandas, NumPy, and re (regular expressions) are standard

in wrangling projects. Good practice is to keep imports at the top of your script for readability and maintainability.

Example Code:

```
import pandas as pd
import numpy as np
print(pd.__version__, np.__version__)
```

8. *Reading CSV using Pandas*

CSV is one of the most common formats for datasets. Pandas' `read_csv()` makes loading them effortless, automatically recognizing column names, handling missing values, and parsing dates if specified. After loading, you can inspect with `head()`, check data types with `info()`, and quickly begin cleaning. This makes CSVs perfect entry points into wrangling pipelines.

Example Code:

```
df = pd.read_csv("sales.csv", parse_dates=["order_date"])
print(df.head())
```

9. *Working with JSON Data*

JSON is widely used for semi-structured data, especially from APIs. In Python, the `json` module helps load JSON strings or files into dictionaries and lists, while `pandas.json_normalize()` can flatten nested JSON into DataFrames. JSON parsing is essential in modern wrangling, as real-time data often comes in this format.

Example Code:

```
import json
from pandas import json_normalize

raw =
'{"id":1,"user":{"name":"Alice","city":"Hyderabad"},"marks":[85,90]}' obj
= json.loads(raw)
flat = json_normalize(obj)
print(flat)
```

10. *Regular Expressions (re module)*

Regular expressions allow powerful pattern matching and text cleaning. In wrangling, they are indispensable for extracting phone numbers, standardizing email formats, or cleaning unwanted characters. Python's `re` module provides functions like `search`, `findall`, and `sub`. With regex, messy text fields can be quickly standardized and validated, which is often necessary in survey data, logs, or web-scraped content.

Example Code:

```
import re
text = "Contact: +91-98765-43210, Email: user@example.com"
phone = re.search(r"\+91-\d{5}-\d{5}", text)
print(phone.group())
```

11. *NumPy Basics*

NumPy is the foundation for numerical computations in Python. Its arrays (`ndarray`) are faster and more memory-efficient than Python lists. NumPy provides vectorized operations, meaning you can perform calculations across entire arrays without explicit loops. In wrangling, NumPy is often used for scaling, filtering, handling missing values, or preparing numerical arrays for machine learning. For instance, you can replace NaN values, compute averages, or normalize columns efficiently.

Example Code:

```
import numpy as np
arr = np.array([10, 20, 30, np.nan])
print("Mean (ignoring NaN):",
np.nanmean(arr))
```

Handling Structured Data: CSV, JSON, and XML Formats

1. CSV (Comma-Separated Values)

CSV is one of the most common formats for storing **tabular data** in rows and columns, where each row represents a record and each column represents a field. Fields are typically separated by commas, though variations exist using tabs or semicolons. CSVs are simple, lightweight, and supported by almost every data tool, from Excel to databases. They are ideal for structured datasets such as student grades, employee records, or sales transactions. However, they are limited when it comes to representing **nested or hierarchical data**. In Python, the **pandas** library makes working with CSVs very easy, allowing quick loading, cleaning, and

transformation of data. For example, you can load a student dataset, drop missing values, and compute averages in just a few lines of code.

Example CSV File:

```
Name,Age,Marks
John,20,85
Alice,22,90
```

Python Code:

```
import pandas as pd

# Load CSV into DataFrame
df = pd.read_csv("students.csv")

# Display first few rows
print(df.head())

# Calculate average marks
print("Average Marks:", df["Marks"].mean())
```

2. JSON (JavaScript Object Notation)

JSON is a widely used format for representing **structured and semi-structured data**. It stores information as **key–value pairs** and can represent **nested or hierarchical structures**, which makes it more flexible than CSV. JSON is commonly used in web APIs, configuration files, and applications where data has relationships or embedded objects. For example, a JSON file may represent a student's details along with their subjects in an array. Python's built-in **json** module can easily load JSON into dictionaries and lists, and pandas provides `json_normalize()` to flatten nested data into a table. JSON is preferred when dealing with **web scraping, REST APIs, and NoSQL databases**.

Example JSON File:

```
{
  "Name":
  "John",
  "Age": 20,
  "Marks": 85,
  "Subjects": ["Math", "Science"]
}
```

Python Code:

```
import json

# Load JSON file
with open("data.json") as
    f: data =
    json.load(f)

# Access fields
print("Student Name:", data["Name"])
```

```
print("Subjects:", data["Subjects"])
```

3. XML (Extensible Markup Language)

XML is a markup language that represents data using **tags** in a hierarchical **tree structure**. It is highly flexible and supports both human readability and machine parsing. Each element in XML is wrapped in a start and end tag, making it self-descriptive. XML is often used in **web services (SOAP), configuration files, and older enterprise systems**. Compared to JSON, XML is more verbose, but it allows attributes, namespaces, and schema validation, which are useful in enterprise applications. Python's `xml.etree.ElementTree` module allows easy parsing of XML documents. You can navigate through child tags, extract text, and convert it into usable formats for analysis. Although JSON has become more popular, XML is still critical in many industries like healthcare, finance, and government systems.

Example XML File:

```
<Student>
   <Name>John</Name>
   <Age>20</Age>
   <Marks>85</Marks>
</Student>
```

Python Code:

```
import xml.etree.ElementTree as ET

# Parse XML file
tree =
ET.parse("student.xml") root
= tree.getroot()

# Print tags and values
for child in root:
    print(child.tag, ":", child.text)
```

4. Data Meant to Be Read by Machines

Machine-readable data refers to information that is stored in a structured and standardized format, allowing computers to automatically parse, interpret, and process it without requiring manual intervention. Unlike human-readable data (like handwritten notes, printed tables, or scanned PDFs), machine-readable formats follow strict rules for how information is represented. This structure enables programs to extract values, manipulate data, and integrate it into larger workflows such as business reporting, analytics, or AI/ML pipelines.

A **machine-readable format** encodes both the values and the structure of the data. For example, in a CSV file, rows and columns are explicitly separated by delimiters such as

commas, while in JSON and XML, key–value pairs and tags provide hierarchical organization. Because of this predictability, computer programs can load such files directly into memory structures (like Python's lists, dictionaries, or pandas DataFrames) and start working with them immediately.

Examples of Machine-Readable Formats

CSV (Comma-Separated Values) – Stores tabular data, where each line represents a row and values are separated by commas. This is one of the simplest machine-readable formats.

Example CSV:

```
Name,Age,Mar
ks
John,20,85
Alice,22,90
```

Python Code:

```
import pandas as pd
df = pd.read_csv("students.csv")
print(df.head())
```

JSON (JavaScript Object Notation) – Stores structured data as nested key–value pairs. Widely used in APIs and web applications.

Example JSON:

```
{ "Name": "John", "Age": 20, "Marks": 85 }
```

Python Code:

```
import json
data = '{"Name": "John", "Age": 20, "Marks": 85}'
obj = json.loads(data)
print("Student Name:", obj["Name"])
```

XML (Extensible Markup Language) – Represents data in a tree-like structure with tags. Often used in enterprise applications and government systems.

Example XML:

```
<Student>
  <Name>John</Name>
   <Age>20</Age>
   <Marks>85</Marks>
</Student>
```

Python Code:

```
import xml.etree.ElementTree as
ET tree =
ET.parse("student.xml") root =
tree.getroot()
for child in root:
    print(child.tag,
    child.text)
```

Databases (SQL and NoSQL) – SQL databases like MySQL and PostgreSQL store data in tables, while NoSQL databases like MongoDB store it in flexible document formats (often JSON-like). These are inherently machine-readable because queries can directly fetch structured data.

Non-Machine-Readable Data

Some data formats, although human-readable, are difficult for machines to interpret automatically:

- **PDF documents**: Data is presented for human eyes, but not structured for easy parsing.
- **Scanned images**: They contain pixels, not structured values.
- **Handwritten notes**: Require OCR (Optical Character Recognition) and often still need manual cleaning.

Extracting structured information from these sources often requires **additional processing tools** such as `pdfminer` (for PDFs) or OCR libraries like `Tesseract` (for scanned text).

Importance of Machine-Readable Data

1. **Automation** – When data is machine-readable, workflows can be automated, reducing the need for human intervention. For example, a Python script can read a CSV file of sales transactions daily and update dashboards without manual input.
2. **Error Reduction** – Manual data entry is prone to mistakes. Machine-readable formats standardize data capture, ensuring consistency and reducing human error. For example, JSON returned from an API will always follow the same structure, unlike free-text reports.
3. **Integration with AI/ML Pipelines** – High-quality, structured, machine-readable data is a prerequisite for training and deploying AI/ML models. For instance, a dataset stored in CSV or a database can be directly loaded into pandas, cleaned, and fed into a scikit-learn model.

Example: Automating AI Input with Machine-Readable Data

```
import pandas as pd
from sklearn.linear_model import LinearRegression
```

```
# Load machine-readable CSV dataset df =
pd.read_csv("sales.csv")

# Clean missing values
df.dropna(inplace=True)

# Train a simple model
X = df[["Ad_Spend"]]   # independent variable y =
df["Revenue"]          # dependent variable

model = LinearRegression().fit(X, y) print("Model
Coefficient:", model.coef_)
```

Here, the dataset stored in CSV format can be directly processed by a machine learning model without manual intervention — which is possible **only because it is machine-readable**.

Unit 2 Data Wrangling and Preprocessing

Handling Missing Data (mean, median, drop, and interpolation), Dealing with Duplicates, Removing duplicates and Fuzzy Matching, Outliers, and Anomalies, Encoding Categorical Variables (Label, One-hot), Data Transformation: Scaling, Normalization, Binning, Data Types Conversion and Data Type Casting, Measures of central tendency and disperency.

Handling Missing Data (mean, median, drop, interpolation)

Handling missing data is a crucial step in data preprocessing, as missing values can negatively impact the performance of machine learning models. To ensure the quality of analysis and model performance, we handle missing data using different strategies:

1. **Drop the data** (if the impact is minimal)
2. **Replace with Mean**
3. **Replace with Median**
4. **Interpolate** (predict missing value using surrounding data)

Sample DataFrame

We'll create a simple pandas DataFrame with missing values.

```
import pandas as
pd import numpy
```

```
as np # Sample
data
data = {
    'Name': ['Aarav', 'Bhavya', 'Chirag', 'Divya', 'Esha'],
    'Age': [25, np.nan, 34, np.nan, 29],
    'Monthly_Income': [45000, 52000, np.nan, 48000, np.nan]
}

df = pd.DataFrame(data)
print("Original Data:\n", df)
```

#Output

Name Age
Monthly_Income 0 Aarav
25.0 45000.0
1 Bhavya NaN 52000.0
2 Chirag s4.0
Na
N s Divya NaN
480
00.0
4 Esha 29.0 NaN

1. Dropping Missing Data

Drop Rows with Any Missing Value: Removes rows where any column has a missing value.

```
df_dropped_rows = df.dropna()
print("After Dropping Rows:\n", df_dropped_rows)
```

#Output

After Dropping Rows:
Name Age
Monthly_Income o Aarav
25.o 45ooo.o

Use When:

- Dataset is large
- Missing data is very minimal

2. Fill Missing Data with Mean

```
df_mean = df.copy()
```

```
df_mean['Age'] = df_mean['Age'].fillna(df_mean['Age'].mean())
df_mean['Monthly_Income'] =
df_mean['Monthly_Income'].fillna(df_mean['Monthly_Income'].mean())
print("Filled with Mean:\n", df_mean)
```

#Output

Filled with Mean:

	Name	Age	Monthly_Income
o	Aarav	25.000000	45000.000000
1	Bhavya	29.333333	52000.000000
2	Chirag	34.000000	48333.333333
3	Divya	29.333333	48000.000000
4	Esha	29.000000	48333.333333

Mean Age = (25 + s4 + 29) / s = 29.ss
Mean Monthly_Income = (45000 + 52000 + 48000) / s = 48sss.ss

Use When:

- Data is **normally distributed** (symmetrical)
- No extreme outliers

3. Fill Missing Data with Median

```
df_median = df.copy()
# Replace missing values with the median (no inplace=True)
df_median['Age'] = df_median['Age'].fillna(df_median['Age'].median())
df_median['Monthly_Income'] =
df_median['Monthly_Income'].fillna(df_median['Monthly_Income'].median(
)) print("Filled with Median:\n", df_median)
```

#Output

Filled with Median:

	Name	Age	Monthly_Income
o	Aarav	25.o	45ooo.o
1	Bhavya	29.o	52ooo.o
2	Chirag	34.o	48ooo.o
3	Divya	29.o	48ooo.o
4	Esha	29.o	48ooo.o

Explanation:

- Median Age = 29
- Median Income = 48000

How to calculate Median

The median is the middle point in a dataset—half of the data points are smaller than the median and half of the data points are larger.

To find the median:

- Arrange the data points from smallest to largest.
- If the number of data points is odd, the median is the middle data point in the list.
- If the number of data points is even, the median is the average of the two middle data points in the list.

Use When Data is **skewed and y**ou want to avoid outlier influence.

4. Fill Missing Data using Interpolation

```
df_interp = df.copy()
# Interpolate missing values (linear by default)
df_interp['Age'] = df_interp['Age'].interpolate()
df_interp['Monthly_Income'] = df_interp['Monthly_Income'].interpolate()
print("Filled with Interpolation:\n", df_interp)
```

#Output
Filled with Interpolation:

	Name	Age	Monthly_Income
o	Aarav	25.o	45ooo.o
1	Bhavya	29.5	52ooo.o
2	Chirag	34.o	5oooo.o
3	Divya	31.5	48ooo.o
4	Esha	29.o	48ooo.o

Explanation:

Interpolates values **linearly** using available values above and below the missing data.

Use When:

- Data has a natural order (like time-series, sequences)
- Values are continuous

Step-by-Step Interpolation Math

We use the **linear interpolation formula**:

$$y = y_1 + \frac{(x - x_1)}{(x_2 - x_1)} \cdot (y_2 - y_1)$$

Dealing with Duplicates

In data analysis, duplicate data can skew your results. Pandas provides useful functions like duplicated() and drop_duplicates() to handle such issues.

Pandas functions for handling duplicates

Purpose	Code Example
Detect duplicate rows	df.duplicated()
Remove duplicate rows	df.drop_duplicates()
Remove duplicates from specific col	df.drop_duplicates(subset='Name')
Count duplicate values	df{'Name').value_counts()
Show all duplicate rows	df{df.duplicated('Name', keep=False))
Mark duplicates in column	df{'is_duplicate') = df.duplicated(...)

Creating a Sample DataFrame with Duplicates

```
import pandas as pd
data = {
    'EmpID': [201, 202, 203, 201, 204, 205, 202],
    'Name': ['Amit', 'Priya', 'Ravi', 'Amit', 'Neha', 'Kiran', 'Priya'],
    'Department': ['HR', 'Finance', 'IT', 'HR', 'Finance', 'IT', 'Finance']
}
```

```
df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)
```

#Output

	EmpID	Name	Department
0	201	Amit	HR
1	202	Priya	Finance
2	20s	Ravi	IT
s	201	Amit	HR
4	204	Neha	Finance
5	205	Kiran	IT
a	202	Priya	Finance

1. Detecting Duplicate

Rows Syntax :

```
df.duplicated()
```

Example

```
print(df.duplicated())
```

#Output:

0 False

1 False

2 False

s True

4 False

5 False

a True

Rows s and a are duplicates of 0 and 1 respectively.

2. *Removing Duplicate Rows*

Syntax: `df.drop_duplicates()`

Example

```
df_unique = df.drop_duplicates()

print(df_unique)
```

#Ouput

	EmpID	Name	Department
0	201	Amit	HR
1	202	Priya	Finance
2	20s	Ravi	IT
4	204	Neha	Finance
5	205	Kiran	IT

3. *Removing Duplicates Based on Specific Column*

Syntax: `df.drop_duplicates(subset='Column Name')`

Example: Drop duplicate Names

```
df_unique_names = df.drop_duplicates(subset='Name')

print(df_unique_names)
```

#Output

	EmpID	Name	Department
0	201	Amit	HR
1	202	Priya	Finance
2	20s	Ravi	IT
4	204	Neha	Finance
5	205	Kiran	IT

4. Find All Duplicate Names (Keep=False)

```
duplicates = df[df.duplicated(subset='Name',
```

```
keep=False)] print(duplicates)
```

#Output

EmpID Name Department

0 201 Amit HR

1 202 Priya Finance

s 201 Amit HR

a 202 Priya Finance

5. *Count How Many Times Each Name Appears*

```
print(df['Name'].value_counts())
```

#Output

Amit 2
Priya 2
Ravi 1
Neha 1
Kiran 1

6. Mark Duplicates in a New Column

```
df['is_duplicate'] = df.duplicated(subset=['EmpID', 'Name'], keep=False)
print(df)
```

#Output

EmpID	Name	Department	is_duplicate
o	2o1	Amit	HR	True
1	2o2	Priya	Finance	True
2	2o3	Ravi	IT	False
3	2o1	Amit	HR	True
4	2o4	Neha	Finance	False
5	2o5	Kiran	IT	False
6	2o2	Priya	Finance	True

Problem Statement:

A company tracks employee visits to its branches. Some employees may have visited the same branch more than once. Here handle duplicates in pandas.

Dealing with Duplicates in Pandas

```
import pandas as pd
# Step 1:Create sample DataFrame with Indian employee visit data
data = {
    'EmpID': [301, 302, 303, 301, 304, 305, 302],
    'EmployeeName': ['Lakshmi', 'Manoj', 'Anjali', 'Lakshmi', 'Rohit',
  'Divya', 'Manoj'],
    'Branch': ['Hyderabad', 'Chennai', 'Bangalore', 'Hyderabad', 'Chennai', 'Mumbai',
  'Chennai']
}
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)
# Step 2: Identify duplicate visits (same EmpID &
Branch) # Use duplicated() with subset on both EmpID
and Branch
df['Is_Duplicate_Visit'] = df.duplicated(subset=['EmpID', 'Branch'], keep=False)
print("\nQ Step 2 - Duplicates Identified (EmpID + Branch):\n", df)
# Step 3: Remove duplicate visits — keep only the first occurrence

df_unique_visits = df.drop_duplicates(subset=['EmpID', 'Branch'],
keep='first') print("\n Step 3 - Unique Visits (Removed Duplicates):\n",
df_unique_visits) # Step 4:Count how many times each branch was visited
branch_counts = df['Branch'].value_counts()
print("\n Step 4 - Branch Visit Counts:\n", branch_counts)
# Step 5: Mark all duplicate employee entries regardless of branch
df['Is_Duplicate_Emp'] = df.duplicated(subset='EmpID', keep=False)
print("\n Step 5 - Duplicate Employee Records (Based on EmpID):\n", df)
```

Dealing with Outliers

Outliers are data points that differ significantly from other observations. They can:

- Indicate variability in data.
- Be due to measurement error or data entry issues.
- Skew the statistical analysis if not handled.

Tools from pandas & numpy for Detecting Outliers

We often combine pandas with numpy or scipy for detecting outliers using:

1. **Statistical methods** (IQR, Z−Score)
2. **Visual methods** (Boxplots, Histograms)

Detecting Outliers using I§R (Interquartile Range)

The IQR is the range between the 25th percentile (Q1) and the c5th percentile (Qs).

Formula:

IQR = Qs – Q1
Lower bound = Q1 – 1.5 *
IQR Upper bound = Qs + 1.5
* IQR

Here **Outliers** are any data point below the lower bound or above the upper bound.

```
#Program
import pandas as pd
data = {'score': [55, 60, 61, 62, 63, 64, 1000, 65, 66, 67]}
df = pd.DataFrame(data)
Q1 =
df['score'].quantile(0.25) Q3
= df['score'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
outliers = df[(df['score'] < lower_bound) | (df['score'] > upper_bound)]
print("Outliers:
")
print(outliers)
```

```
#Output
Outliers:
 score
6   1000
```

2. Using Z-Score (Standard Deviation Method)

Z-Score tells us how many standard deviations a value is from the mean.

Formula:

Z = (X - mean) / standard deviation

Threshold: Common cut–off(|Z|) is ±s. Values beyond ±s are considered outliers.

If |Z| > 3 → Outlier (can also use threshold 2.5 or 2)

```
import pandas as pd
import numpy as np
# Create a sample dataset
data = {'score': [55, 60, 61, 62, 63, 64, 1000, 65, 66, 67]}
df = pd.DataFrame(data)
print(df)
#Calculate Z-Scores
mean = df['score'].mean()
std = df['score'].std()
```

```
df['z_score'] = (df['score'] - mean) / std
print(df)
#Filter Outliers (|Z| > 2.5)
z_outliers = df[df['z_score'].abs() > 2.5]
print(z_outliers)

#Output
    score
0      55
1      60
2      61
3      62
4      63
5      64
6    1000
7      65
8      66
9      67
```

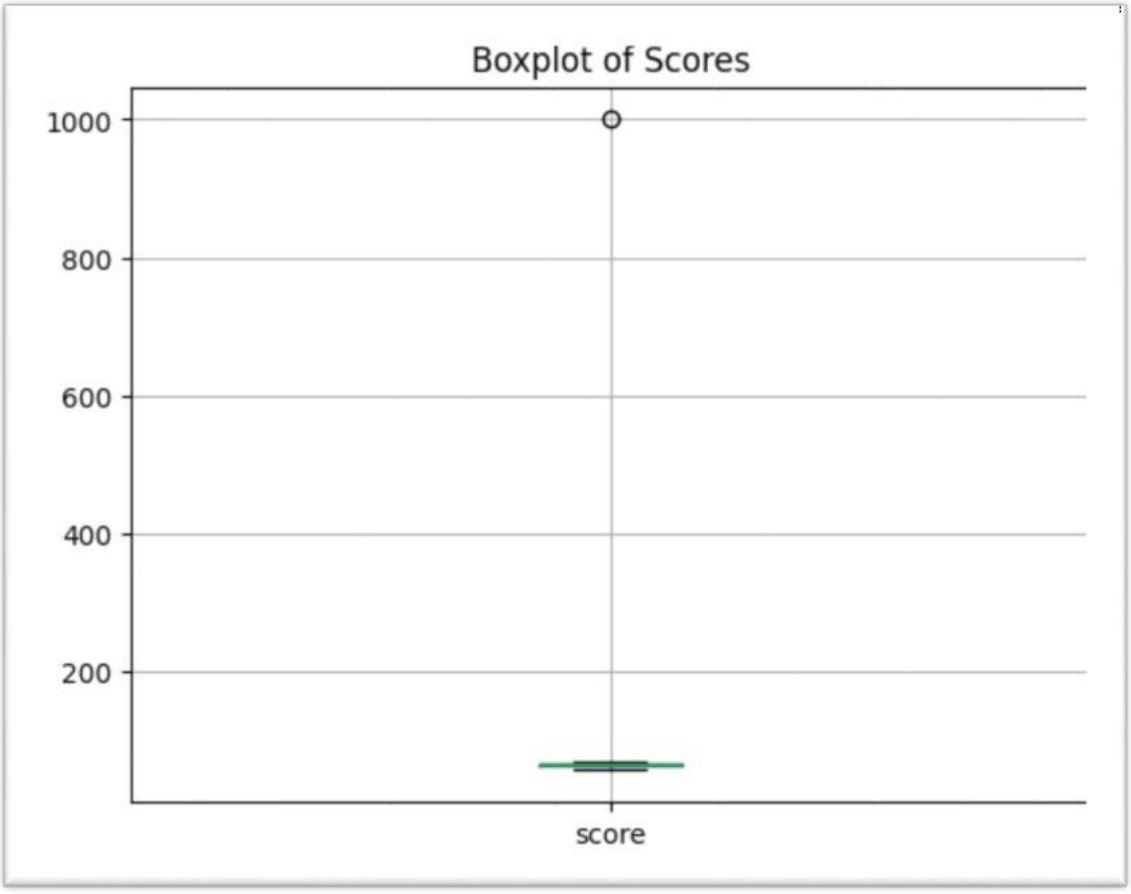
```
    score   z_score
0      55 -0.341692
1      60 -0.324827
2      61 -0.321454
3      62 -0.318080
4      63 -0.314707
5      64 -0.311334
6    1000  2.845859
7      65 -0.307961
8      66 -0.304588
9      67 -0.301215
   score    z_score
6   1000   2.845859
```

3. **Using Boxplots (Visualization)**

Boxplots show outliers as dots outside the whiskers.

```
import matplotlib.pyplot as
plt # Create a sample dataset
data = {'score': [55, 60, 61, 62, 63, 64, 1000, 65, 66, 67]}
df = pd.DataFrame(data)
print(df)
```

```
df.boxplot(column='score')
plt.title("Boxplot of
Scores") plt.show()

#Output
    score
0      55
1      60
2      61
3      62
4      63
5      64
6    1000
7      65
8      66
9      67
```



Dealing with Outliers

Once detected, you can choose to either **remove**, **cap**, **transform**, or **replace** the outliers.

1. Removing Outliers (Using IQR)

If outliers are likely due to error or noise, just remove them.

#Program

```
import pandas as pd
data = {'score': [55, 60, 61, 62, 63, 64, 1000, 65, 66, 67]}
df = pd.DataFrame(data)
Q1 =
df['score'].quantile(0.25) Q3
= df['score'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 *
IQR
df_no_outliers = df[(df['score'] >= lower_bound) & (df['score'] <=
upper_bound)]
print(df_no_outliers)
```

```
#Output
   score

0     55
1     60
2     61
3     62
4     63
5     64
7     65
8     66
9     67
```

2: **Capping Outliers (Winsorizing)**

Capping, or **winsorizing**, is a technique used to **limit extreme values** in a dataset by setting them to a specified percentile value rather than removing them.

Purpose:

- Keep **all rows** in the dataset (unlike removing outliers).
- Reduce the **influence** of extreme values on statistical models.
- Maintain the **data structure** and **row count**.

How Does It Work?

Instead of removing outliers beyond a threshold, we **cap** them at a chosen **percentile limit**, like the 5th and 95th percentiles. This retains all data points but reduces the influence of outliers.

Syntax: DataFrame.clip(lower=lower_cap, upper=**upper_cap**)

Where lower_cap=5th percentile and upper_cap= 95th percentile.

```
import pandas as pd
data = {'score': [55, 60, 61, 62, 63, 64, 1000, 65, 66, 67]}
df = pd.DataFrame(data)
lower_cap = df['score'].quantile(0.05)
upper_cap = df['score'].quantile(0.95)
df['score_capped'] = df['score'].clip(lower=lower_cap, upper=upper_cap)
print(df[['score', 'score_capped']])

#Output
    score  score_capped
0      55         57.25
1      60         60.00
2      61         61.00
3      62         62.00
4      63         63.00
5      64         64.00
6    1000        580.15
7      65         65.00
8      66         66.00
9      67         67.00
```

3: **Replacing Outlier with Median**

Instead of **removing** or **capping**, we **replace outlier values** (detected using Z−Score, IQR, etc.) with the **median** of the data. The median is **resistant to outliers**, making it a good choice for substitution.

```
#Program
import pandas as pd
import numpy as np
#Create a sample
dataset
data = {'score': [55, 60, 61, 62, 63, 64, 1000, 65, 66, 67]}
df = pd.DataFrame(data)
```

```
print(df)
#Calculate Z-Scores
mean = df['score'].mean()
std = df['score'].std()
df['z_score'] = (df['score'] - mean) / std
print(df)
#Find median
median_val = df['score'].median()
#Replace outliers (Z <= 2.5 or Z < -2.5) with median
df['score_replaced'] = df['score'].where(df['z_score'].abs() < =2.5,
median_val) print(df[['score', 'z_score', 'score_replaced']])
```

```
#Output
    score

0      55
1      60
2      61
3      62
4      63
5      64
6    1000
7      65
8      66
9      67
    score    z_score
0      55 -0.341692
1      60 -0.324827
2      61 -0.321454
3      62 -0.318080
4      63 -0.314707
5      64 -0.311334
6    1000  2.845859
7      65 -0.307961
8      66 -0.304588
9      67 -0.301215
    score    z_score  score_replaced
0      55 -0.341692            55.0
```

```
1      60 -0.324827            60.0
2      61 -0.321454            61.0
3      62 -0.318080            62.0
4      63 -0.314707            63.0
5      64 -0.311334            64.0
6    1000  2.845859            63.5
7      65 -0.307961            65.0
8      66 -0.304588            66.0
9      67 -0.301215            67.0
```

4. Replacing Outlier with Log Transformation

Log transformation compresses **large values** and expands **small ones**, helping reduce the **impact of outliers** and **skewness**.

Formula: log(x)

Note: Apply only to **positive** values (non–zero, non–negative).

```
#Program
import pandas as
pd import numpy
as np




# Create a sample dataset
data = {'score': [55, 60, 61, 62, 63, 64, 1000, 65, 66, 67]}
df = pd.DataFrame(data)
print(df)
# Apply log transformation
df['log_score'] =
np.log(df['score'])
print(df[['score', 'log_score']])


#Output
    score
0      55
1      60
2      61
3      62
4      63
5      64
6    1000
7      65
8      66
9      67
   score  log_score
0      55   4.007333
1      60   4.094345
2      61   4.110874
3      62   4.127134
4      63   4.143135
```

```
5      64   4.158883
6    1000   6.907755
7      65   4.174387
8      66   4.189655
9      67   4.204693
```

Dealing with Anomalies

Anomalies (also known as outliers or unusual data points) are data values that deviate significantly from other observations. These can occur due to measurement errors, data entry issues, or natural variability. Handling them correctly is important for robust data analysis.

Types of Anomalies

1. Global outliers - Entirely different from the rest of the dataset.
2. Contextual Anomalies: Unusual in specific context (e.g., temperature high in winter).
3. Collective Anomalies: A group of related data points is anomalous.

Causes of Anomalies

- Data entry or measurement errors.
- Sensor malfunctions or logging issues.
- Natural rare events (e.g., fraud transactions).
- Sudden changes in patterns or trends.

Techniques to Detect Anomalies

1. Statistical Methods

- Z−score: Values beyond ±s standard deviations are considered anomalies.
- IQR (Interquartile Range): Data outside Q1−1.5×IQR and Qs+1.5×IQR are flagged as outliers.

2. Visualization

- Box Plots: Highlight extreme values.
- Scatter Plots: Show unusual data points.
- Histograms: Reveal unusual distributions.

3. Machine Learning Approaches

- Clustering−based (e.g., DBSCAN, K−Means): Points far from clusters are anomalies.
- Isolation Forests: Isolate anomalies by random partitioning.
- Auto encoders: Use reconstruction error to detect anomalies in high−dimensional data.

Dealing with Anomalies

1. Remove Anomalies

- Suitable when anomalies are due to data entry errors or irrelevant noise.
- Example: Dropping values with Z−score > s.

2. Cap/Floor Values (Winsorizing)

- Replace extreme values with nearest acceptable threshold.

3. Transformation

- Apply log, square root, or Box−Cox transformations to reduce the effect of outliers.

4. Imputation

- Replace anomalies with mean, median, or model−based predictions.

5. Keep Anomalies

- If anomalies are valid rare events (like fraud detection), retain them for analysis.

Encoding Categorical Variables

Encoding categorical variables is a process of converting non-numeric (categorical) data into numerical format so that machine learning algorithms can interpret and process them effectively. Many algorithms (e.g., linear regression, neural networks, SVM) work only with numerical values, hence categorical variables must be encoded.

Types of Categorical Variables

1. **Nominal Variables**: Categories without any order (e.g., color: red, blue, green).
2. **Ordinal Variables**: Categories with a defined order or ranking (e.g., low, medium, high).

Common Encoding Techniques

1. Label Encoding

- Converts each category into a unique integer.
- Suitable for ordinal variables.

#Program

```
from sklearn.preprocessing import LabelEncoder
import pandas as pd
data = pd.DataFrame({'Color': ['Red', 'Blue', 'Green', 'Red']})
encoder = LabelEncoder()
```

```
data['Color_encoded'] = encoder.fit_transform(data['Color'])
print(data)
```

```
#Output
   Color   Color_encoded
0   Red               2
1  Blue               0
2 Green               1
3   Red               2
```

2. One-Hot Encoding

- Creates a new binary column for each category.
- Suitable for nominal variables.

#Program

```
import pandas as pd
data = pd.DataFrame({'Color': ['Red', 'Blue', 'Green',
'Red']}) encoded = pd.get_dummies(data, columns=['Color'])
print(encoded)
```

```
#Output
   Color_Blue  Color_Green  Color_Red
0           0            0          1
1           1            0          0
2           0            1          0
3           0            0          1
```

3. Ordinal Encoding

- Assigns ordered integers to categories based on their rank.
- Vou define the order manually.

#Program

```
import pandas as pd
from sklearn.preprocessing import OrdinalEncoder
data = pd.DataFrame({'Size': ['Small', 'Medium', 'Large', 'Medium']})
encoder = OrdinalEncoder(categories=[['Small', 'Medium', 'Large']])
data['Size_encoded'] = encoder.fit_transform(data[['Size']])
print(data)

#Output
     Size  Size_encoded
0   Small           0.0
1  Medium           1.0
2   Large           2.0
3  Medium           1.0
```

4. Binary Encoding

- Combines hashing and one−hot encoding to reduce dimensionality.
- Converts categories into binary code.
- Useful when you have many categories.

#Program

```
import category_encoders as
ce import pandas as pd
data = pd.DataFrame({'City': ['Paris', 'London', 'Berlin', 'Paris']})
encoder = ce.BinaryEncoder(cols=['City'])
encoded = encoder.fit_transform(data)
print(encoded)
```

5. Target / Mean Encoding

- Replaces each category with the mean of the target variable for that category.
- Useful in supervised learning.

```
#Program
import pandas as pd
data =
pd.DataFrame({
                'City': ['Paris', 'London', 'Berlin', 'Paris'],
                'Sales': [100, 200, 150, 120] })
target_mean = data.groupby('City')['Sales'].mean()
data['City_encoded'] = data['City'].map(target_mean)
print(data)

#Output
    City  Sales  City_encoded
0   Paris    100         110.0
```

```
1   London     200          200.0
2   Berlin     150          150.0
3    Paris     120          110.0
```

Choosing the Right Technique

- **Nominal variables: One-hot encoding or binary encoding.**
- **Ordinal variables: Label or ordinal encoding.**
- **High-cardinality categorical variables: Binary or target encoding.**

Data Transformations

Data transformations are essential steps in data preprocessing, helping to make datasets suitable for analysis or machine learning models.

Three common techniques are Scaling, Binning, and Normalization.

1. *Scaling*

Scaling adjusts the range of features so they can be compared or used in algorithms that are sensitive to feature magnitude (e.g., k–NN, gradient descent, SVM).

Why use Scaling?

- Features with large ranges can dominate those with small ranges.
- Speeds up convergence in optimization algorithms.
- Ensures uniformity for distance–based algorithms.

Common Types:

- **Min-Max Scaling (Rescaling)**
  Brings data into a fixed range, usually {0,1).

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

- **Standardization (Z-score scaling)**
  Centers data around mean 0 with standard deviation 1.

$$x_{scaled} = \frac{x - \mu}{\sigma}$$

#Program

```
from sklearn.preprocessing import MinMaxScaler, StandardScaler
import numpy as np
data = np.array([[10], [20], [30], [40], [50]])
# Min-Max Scaling
minmax =
MinMaxScaler()
```

```
print(minmax.fit_transform(data
)) # Standardization
standard = StandardScaler()
print(standard.fit_transform(data))

#Outpu
      t
  [[0.

     ]
[0.25]
[0.5 ]
[0.75]
[1.  ]]
          [[-
  1.41421356]
[-0.70710678]
[ 0.        ]
[ 0.70710678]
    [ 1.41421356]]
```

2. *Binning (Discretization)*

Binning converts continuous values into categorical bins or intervals.

Why use Binning?

- Reduces noise and variability.
- Makes models easier to interpret.
- Useful in histogram creation and feature engineering.

Types:

- Equal-width Binning: Divides range into equal-sized intervals.
- Equal-frequency Binning: Each bin has approximately the same number of samples.
- Custom Binning: Based on domain knowledge.

```
#Program
import pandas as pd
data = [5, 7, 12, 18, 24, 30, 35, 42]
bins = [0, 10, 20, 30, 50]
labels = ['Low', 'Medium', 'High', 'Very High']
binned_data = pd.cut(data, bins=bins, labels=labels)
print(binned_data)

#Output
['Low', 'Low', 'Medium', 'Medium', 'High', 'High', 'Very High', 'Very High']
Categories (4, object): ['Low' < 'Medium' < 'High' < 'Very High']
```

3. *Normalization*

Normalization scales individual samples so that the entire vector or row has a unit norm. It's often used for text data, clustering, and algorithms where direction of data matters more than magnitude. **Why use Normalization?**

- Useful in distance-based models (k-NN, cosine similarity).
- Ensures fair contribution of each feature.

Common Method:

- L2 Normalization:

$$x_{norm} = \frac{x}{||x||_2}$$

- **L1 Normalization:**

$$x_{norm} = \frac{x}{||x||_1}$$

```
#Program
from sklearn.preprocessing import
Normalizer import numpy as np
data = np.array([[3, 4], [1, 2], [2,
2]]) norm = Normalizer(norm='l2')
print(norm.fit_transform(data))
```

#Output

```
[[0.6        0.8       ]
 [0.4472136  0.89442719]
   [0.70710678 0.70710678]]
```

Data type conversions and Data type casting

In pandas, data type conversions and casting are crucial for preparing data, ensuring consistency, and optimizing memory. Conversion changes a Series or DataFrame column to types like int, float, string, datetime, or categorical, often needed when importing data stored as text. Casting further optimizes types, such as downcasting floata4 to floats2 or converting integers to categories to save memory.

Method	Purpose	Example
astype(dtype)	**Convert Series/DataFrame column to a specified type**	**df('A') = df('A').astype(float)**
pd.to_numeric()	**Convert values to numeric, with error handling**	**pd.to_numeric(df('col'), errors='coerce')**
pd.to_datetime()	**Convert values to datetime format**	**df('date') = pd.to_datetime(df('date'))**
pd.to_timedelta()	**Convert values to timedelta objects**	**df('duration') = pd.to_timedelta(df('duration'))**
Converting to categorical	**Optimize repeated values for memory and speed**	**df('category') = df('category').astype('category')**

#Program

```
import pandas as pd
# Sample DataFrame with mixed
types data = {
    'id': ['1', '2', '3'],
    'amount': ['100.5', '200.0', '300.25'],
    'date': ['2025-08-01', '2025-08-02', '2025-08-03'],
    'status': ['paid', 'pending', 'paid']
}
df = pd.DataFrame(data)
print("Before Conversion:")
```

```
print(df.dtypes)
print(df)
# 1. Convert 'id' to integer
df['id'] = df['id'].astype(int)
# 2. Convert 'amount' to float and downcast to save memory
df['amount'] = pd.to_numeric(df['amount'],
downcast='float') # 3. Convert 'date' to datetime
df['date'] =
pd.to_datetime(df['date']) # 4.
Convert 'status' to categorical
df['status'] =
df['status'].astype('category')
print("\nAfter Conversion and Casting:")
print(df.dtypes)
print(df)
```

#Output

```
Before Conversion:
id         object
amount     object
date       object
status
           obj
ect dtype:
object
  id  amount         date    status
0  1  100.5   2025-08-01      paid
1  2  200.0   2025-08-02   pending
2  3  300.25 2025-08-03      paid
```

After Conversion and

```
Casting: id   int64
amount     float32
date
         datetime64[ns]
status   category
dtype: object
   id  amount       date   status
0   1   100.5 2025-08-01     paid
1   2   200.0 2025-08-02  pending
2   3   300.2 2025-08-03     paid
```

Unit 3 Working with Excel Files, PDFs, and Databases

Installing Python Packages for Data Wrangling, Parsing Excel Files, Programmatic Approaches to PDF Parsing, Converting PDF to Text (pdfminer), Acquiring and Storing Data, Introduction to Databases for Data Wrangling, Relational Databases: MySQL, Data Exploration: Table functions and Joining Datasets.

Data Wrangling: Installing Python Packages, Excel Parsing, PDF Parsing, Data Acquisition and Storage

These topics form the foundation of **Data Wrangling and Exploratory Data Analysis (EDA)**. Data wrangling involves collecting, cleaning, transforming, and storing data so that it can be analyzed effectively.

1. Installing Python Packages for Data Wrangling

What is a Python Package?

A **Python package** is a collection of modules that provide additional functionality. Instead of writing every function from scratch, we install packages developed by the Python community.

Common Data Wrangling Packages

Package	Purpose
NumPy	Numerical computations
Pandas	Data manipulation and analysis
Matplotlib	Data visualization
Seaborn	Statistical visualization
OpenPyXL	Reading/Writing Excel files
xlrd	Reading older Excel (.xls) files
pdfminer.six	Extracting text from PDF
PyPDF2	Reading PDF documents
Tabula-py	Extracting tables from PDFs
BeautifulSoup	Web scraping
Requests	Downloading web data
SQLAlchemy	Database connectivity

Installing Packages

Python packages are installed using **pip**.

Install a single package

pip install pandas

Install multiple packages
pip install numpy pandas matplotlib seaborn

Install Excel libraries
pip install openpyxl
pip install xlrd

Install PDF libraries
pip install pdfminer.six
pip install PyPDF2

Install Web Scraping Libraries
pip install requests
pip install beautifulsoup4

Install Database Libraries
pip install sqlalchemy
pip install pymysql

Checking Installed Packages
pip list
Example
```
Package      Version
------------ -------
numpy        2.2.1
pandas       2.3.0
matplotlib   3.10.0
```

Updating Packages
pip install --upgrade pandas

Uninstalling Packages
pip uninstall pandas

Importing Packages
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Advantages
- Saves development time
- Ready-made functions
- Community support
- Highly optimized
- Easy installation

2. Parsing Excel Files
What is Excel Parsing?
Excel parsing means **reading Excel data into Python for processing and analysis.**
Excel files usually have extensions
- .xlsx
- .xls

The most popular library is **Pandas** with **OpenPyXL**.

Reading Excel File
Suppose we have
students.xlsx

Code
```
import pandas as pd

df = pd.read_excel("students.xlsx")

print(df)
```
Output
```
Name    Marks
John     80
David    75
Sara     90
```

Reading Specific Sheet
```
df = pd.read_excel("students.xlsx", sheet_name="Sheet2")
```

Reading Multiple Sheets
```
excel = pd.ExcelFile("students.xlsx")

print(excel.sheet_names)
```
Output
```
['Sheet1', 'Sheet2']
```

Reading All Sheets
```
all_sheets = pd.read_excel("students.xlsx", sheet_name=None)

print(all_sheets.keys())
```

Reading Selected Columns
```
df = pd.read_excel("students.xlsx", usecols=["Name","Marks"])
```

Reading Specific Rows
```
df = pd.read_excel("students.xlsx", nrows=5)
```

Writing Excel File
```
df.to_excel("output.xlsx", index=False)
```

Advantages
- Easy data import
- Multiple sheet support
- Fast processing
- Supports filtering

3. Programmatic Approaches to PDF Parsing
What is PDF Parsing?
PDF Parsing means extracting information from PDF files automatically using programs.
PDFs may contain
- Text
- Tables
- Images
- Charts
- Forms

Types of PDF Parsing
1. Text Extraction
Extracts plain text.
Example
Student Name

John
Marks: 90

2. Table Extraction
Extracts structured tables.
Example

Name	Marks
John	90

3. OCR Parsing
Used when PDF contains scanned images.
Libraries
- Tesseract OCR
- EasyOCR

4. Metadata Extraction
Extracts
- Author
- Creation Date
- Title

Common Libraries

Library	Purpose
pdfminer.six	Text extraction
PyPDF2	Read PDF
pdfplumber	Tables
Tabula-py	Tables
Camelot	Tables
OCR	Image PDFs

Example using PyPDF2
```
from PyPDF2 import PdfReader

reader = PdfReader("sample.pdf")

page = reader.pages[0]

text = page.extract_text()

print(text)
```

4. Converting PDF to Text using pdfminer
What is pdfminer?
pdfminer.six is a Python library used to extract text while preserving layout information.
Installation
```
pip install pdfminer.six
```

Basic Example
```
from pdfminer.high_level import extract_text

text = extract_text("sample.pdf")

print(text)
```
Output
Annual Report

Company Sales
Revenue increased by 25%.

Save Extracted Text

```
from pdfminer.high_level import extract_text

text = extract_text("sample.pdf")

with open("output.txt","w",encoding="utf-8") as file:
    file.write(text)
```

Reading Multiple PDFs

```
import os
from pdfminer.high_level import extract_text

folder = "pdfs"

for file in os.listdir(folder):

    if file.endswith(".pdf"):

        text = extract_text(os.path.join(folder,file))

        print(text)
```

Advantages of pdfminer

- Accurate text extraction
- Unicode support
- Layout preservation
- Works with multi-page PDFs
- Free and open source

Limitations

- Cannot directly extract tables
- Poor results on scanned PDFs
- OCR required for image-based PDFs

5. Acquiring Data

What is Data Acquisition?

Data acquisition is the process of **collecting data from various sources** for analysis.

Sources of Data

1. CSV Files

```
df = pd.read_csv("students.csv")
```

2. Excel Files

```
df = pd.read_excel("students.xlsx")
```

3. JSON Files

```
df = pd.read_json("students.json")
```

4. SQL Databases

```
import sqlite3
import pandas as pd

conn = sqlite3.connect("college.db")

df = pd.read_sql("SELECT * FROM students", conn)
```

5. APIs

```
import requests

response = requests.get("https://api.example.com/data")
```

```
print(response.json())
```

6. Web Scraping

```
import requests
from bs4 import BeautifulSoup

html = requests.get("https://example.com").text

soup = BeautifulSoup(html,"html.parser")

print(soup.title.text)
```

7. IoT Sensors

Example

- Temperature sensor
- GPS sensor
- Camera

8. Cloud Storage

Examples

- Google Drive
- AWS S3
- Azure Blob Storage

Data Acquisition Workflow

Data Sources
↓
Collect Data
↓
Validate Data
↓
Clean Data
↓
Transform Data
↓
Store Data
↓
Analysis

6. Storing Data

What is Data Storage?

Data storage is the process of saving collected data for future retrieval and analysis.

Types of Storage

1. CSV File

```
df.to_csv("students.csv", index=False)
```

2. Excel File

```
df.to_excel("students.xlsx", index=False)
```

3. JSON File

```
df.to_json("students.json")
```

4. SQL Database

```
import sqlite3
```

```
conn = sqlite3.connect("college.db")

df.to_sql("students", conn, if_exists="replace")
```

5. Cloud Storage

Examples

- AWS S3
- Google Cloud Storage
- Microsoft Azure

Advantages of Proper Data Storage

- Easy retrieval
- Improved security
- Backup and recovery
- Data sharing
- Scalability
- Better data management

Overall Data Wrangling Pipeline

Raw Data Sources
(CSV, Excel, PDF, APIs, Databases, Websites)

Install Required Python Packages

Acquire/Collect Data

Parse Excel Files / Parse PDF Files

Convert PDF to Text

Clean and Transform Data

Store Data (CSV/Excel/SQL/Cloud)

Exploratory Data Analysis (EDA) & Visualization

Important Interview and Exam Points

1. **Data Wrangling** is the process of collecting, cleaning, transforming, and preparing raw data for analysis.
2. **Pandas** is the primary Python library for data manipulation, while **NumPy** supports numerical computations.
3. **OpenPyXL** is commonly used to read and write .xlsx Excel files.
4. **Excel parsing** involves reading worksheets, selecting rows/columns, and importing data into a DataFrame.
5. **PDF parsing** extracts information such as text, tables, or metadata from PDF documents.
6. **pdfminer.six** is widely used for extracting text from text-based PDFs while preserving layout.
7. **Scanned PDFs** require **Optical Character Recognition (OCR)** tools because they contain images rather than selectable text.
8. **Data acquisition** collects data from sources such as CSV files, Excel files, databases, APIs, websites, sensors, and cloud storage.
9. **Common storage formats** include CSV, Excel, JSON, SQL databases, and cloud storage platforms.

10. A typical **data wrangling workflow** is: **Acquire Data → Parse Files → Clean → Transform → Store → Analyze**.

Introduction to Databases for Data Wrangling, Relational Databases (MySQL), Data Exploration, Table Functions, and Joining Datasets

Databases are one of the most important sources of data in **Data Wrangling and Exploratory Data Analysis (EDA)**. Large organizations such as banks, hospitals, universities, e-commerce companies, and government agencies store their data in databases rather than in Excel or CSV files because databases provide secure, organized, and efficient data management.

1. Introduction to Databases for Data Wrangling

What is a Database?

A **database** is an organized collection of related data that is stored electronically and can be easily accessed, managed, updated, and retrieved.

Unlike Excel files, databases can efficiently store millions of records while ensuring data integrity and minimizing duplication.

Definition

Database: A structured collection of data stored electronically and managed using a Database Management System (DBMS).

Example

Consider a college management system.

Student Table

Student_ID	Name	Department
101	Ravi	CSE
102	Priya	AI
103	John	ECE

Marks Table

Student_ID	Subject	Marks
101	Python	90
102	Database	85
103	AI	95

These tables are stored separately but are connected using the **Student_ID**.

Why Databases are Used in Data Wrangling

Data wrangling involves collecting, cleaning, transforming, and preparing data.

Databases provide:

- Centralized storage
- Faster querying
- Easy filtering

- Data consistency
- Reduced redundancy
- Multi-user access
- Secure storage
- Efficient data retrieval

Database Management System (DBMS)
A **DBMS** is software used to create, manage, and manipulate databases.
Examples include:
- MySQL
- Oracle
- PostgreSQL
- Microsoft SQL Server
- SQLite
- MariaDB

Components of a Database
1. Tables
A table stores related information.
Example:

RollNo	Name	Marks
101	Rahul	90
102	Anita	95

2. Rows (Records)
Each row represents one record.
Example:
|101|Rahul|90|

3. Columns (Fields)
Each column stores one type of information.
Example:
- RollNo
- Name
- Marks

4. Primary Key
A **Primary Key** uniquely identifies every row.
Example:
Student_ID

Student_ID	Name
101	Rahul
102	Anita

No duplicate Student_ID values are allowed.

5. Foreign Key
A **Foreign Key** connects two tables.
Example
Student Table

Student_ID	Name
101	Rahul

Marks Table

Student_ID	Subject
101	Python

Student_ID in the Marks table is a Foreign Key.

Database Architecture
Users
|

SQL Queries
|
DBMS (MySQL)
|
Database
|
Tables

Advantages of Databases
- Efficient storage
- Security
- Backup and recovery
- Concurrent access
- High performance
- Data consistency
- Reduced redundancy

2. Relational Databases
What is a Relational Database?
A **Relational Database** stores data in multiple related tables.
Relationships are created using
- Primary Key
- Foreign Key

Example
Employee Table

EmpID	Name
1	Alice
2	Bob

Salary Table

EmpID	Salary
1	50000
2	60000

EmpID links both tables.

Features of Relational Databases
- Data stored in tables
- Supports SQL
- Relationships between tables
- Data integrity
- ACID properties
- Normalization

Popular Relational Databases

Database	Organization
MySQL	Oracle
PostgreSQL	Open Source
SQLite	Open Source
Oracle Database	Oracle
SQL Server	Microsoft

3. MySQL
What is MySQL?
MySQL is an **open-source Relational Database Management System (RDBMS)** that stores data in tables and allows users to interact with the data using **Structured Query Language (SQL)**.
It is one of the world's most popular databases for web applications, enterprise software, and data analytics.

Features of MySQL
- Open source
- High speed
- Secure

- Multi-user
- Cross-platform
- Supports transactions
- Large community support

MySQL Architecture

Applications
|
SQL Queries
|
MySQL Server
|
Database
|
Tables

SQL (Structured Query Language)

SQL is the standard language used to communicate with relational databases.

Basic SQL Commands

Create Database

```
CREATE DATABASE College;
```

Use Database

```
USE College;
```

Create Table

```
CREATE TABLE Students(
Student_ID INT PRIMARY KEY,
Name VARCHAR(50),
Department VARCHAR(30),
Marks INT
);
```

Insert Data

```
INSERT INTO Students
VALUES(101,'Rahul','CSE',90);
```

View Data

```
SELECT * FROM Students;
```

Output

Student_ID	Name	Department	Marks
101	Rahul	CSE	90

Update Data

```
UPDATE Students
SET Marks=95
WHERE Student_ID=101;
```

Delete Data

```
DELETE FROM Students
WHERE Student_ID=101;
```

Connecting MySQL with Python

Install library

```
pip install mysql-connector-python
```

Python Program

```
import mysql.connector

conn = mysql.connector.connect(
   host="localhost",
```

```
    user="root",
    password="password",
    database="College"
)

cursor = conn.cursor()

cursor.execute("SELECT * FROM Students")

for row in cursor:
    print(row)

conn.close()
```

Advantages of MySQL

- Free
- Fast
- Reliable
- Secure
- Easy integration with Python

4. Data Exploration

What is Data Exploration?

Data Exploration is the process of understanding the structure, quality, and characteristics of a dataset before analysis or modeling.
It helps answer questions such as:

- How many records are there?
- Which columns exist?
- Are there missing values?
- Are there duplicate records?
- What are the data types?
- What is the distribution of values?

Importance of Data Exploration

- Detects missing values
- Detects duplicates
- Identifies outliers
- Understands relationships
- Improves data quality

Data Exploration Using Pandas

Suppose

```
import pandas as pd

df = pd.read_csv("students.csv")
```

View First Records

df.head()
Displays the first 5 rows.

View Last Records

df.tail()
Displays the last 5 rows.

Data Shape

df.shape
Output
(500,6)

Meaning
- 500 rows
- 6 columns

Column Names
df.columns

Data Types
df.dtypes

Dataset Information
df.info()
Shows:
- Number of rows
- Number of columns
- Data types
- Non-null values
- Memory usage

Statistical Summary
df.describe()
Displays:
- Count
- Mean
- Standard deviation
- Minimum
- Quartiles (25%, 50%, 75%)
- Maximum

Missing Values
df.isnull().sum()

Duplicate Records
df.duplicated().sum()

Unique Values
df["Department"].unique()

Value Counts
df["Department"].value_counts()

5. Table Functions
Table functions are commonly used to inspect, summarize, filter, and transform data stored in tables.
Common SQL Table Functions
COUNT()
Counts the number of records.
SELECT COUNT(*) FROM Students;

SUM()
Calculates the total.
SELECT SUM(Marks)
FROM Students;

AVG()
Calculates the average.
SELECT AVG(Marks)
FROM Students;

MAX()
Finds the maximum value.
```
SELECT MAX(Marks)
FROM Students;
```

MIN()
Finds the minimum value.
```
SELECT MIN(Marks)
FROM Students;
```

DISTINCT
Displays unique values.
```
SELECT DISTINCT Department
FROM Students;
```

ORDER BY
Sorts records.
```
SELECT *
FROM Students
ORDER BY Marks DESC;
```

GROUP BY
Groups records based on a column.
```
SELECT Department,
AVG(Marks)
FROM Students
GROUP BY Department;
```

HAVING
Filters grouped results.
```
SELECT Department,
AVG(Marks)
FROM Students
GROUP BY Department
HAVING AVG(Marks)>80;
```

LIMIT
Displays a specified number of rows.
```
SELECT *
FROM Students
LIMIT 5;
```

6. Joining Datasets
What is Joining?
Joining combines data from two or more related tables using a common column, usually a **Primary Key** or **Foreign Key**.

Example Tables
Students

Student_ID	Name	Department
101	Rahul	CSE
102	Priya	AI
103	John	ECE

Marks

Student_ID	Subject	Marks
101	Python	90
102	Database	85
104	AI	92

Types of Joins

1. INNER JOIN

Returns only matching rows.

```
SELECT Students.Student_ID,
Students.Name,
Marks.Subject,
Marks.Marks
FROM Students
INNER JOIN Marks
ON Students.Student_ID=Marks.Student_ID;
```

Result

Student_ID	Name	Subject	Marks
101	Rahul	Python	90
102	Priya	Database	85

2. LEFT JOIN

Returns all rows from the left table and matching rows from the right table.

```
SELECT *
FROM Students
LEFT JOIN Marks
ON Students.Student_ID=Marks.Student_ID;
```

Result

Student_ID	Name	Subject	Marks
101	Rahul	Python	90
102	Priya	Database	85
103	John	NULL	NULL

3. RIGHT JOIN

Returns all rows from the right table and matching rows from the left table.

```
SELECT *
FROM Students
RIGHT JOIN Marks
ON Students.Student_ID=Marks.Student_ID;
```

Result

Student_ID	Name	Subject	Marks
101	Rahul	Python	90
102	Priya	Database	85
104	NULL	AI	92

4. FULL OUTER JOIN

Returns all rows from both tables, matching where possible. (Not directly supported in MySQL; it can be simulated using UNION of LEFT JOIN and RIGHT JOIN.)

Expected result:

Student_ID	Name	Subject	Marks
101	Rahul	Python	90
102	Priya	Database	85
103	John	NULL	NULL
104	NULL	AI	92

5. CROSS JOIN

Returns the Cartesian product of two tables.

```
SELECT *
FROM Students
CROSS JOIN Marks;
```

If the Students table has 3 rows and the Marks table has 3 rows, the result will contain **9 rows**.

Difference Between SQL Joins

Join Type	Returns
INNER JOIN	Only matching rows from both tables
LEFT JOIN	All rows from the left table and matching rows from the right table
RIGHT JOIN	All rows from the right table and matching rows from the left table
FULL OUTER JOIN	All rows from both tables (matching and non-matching)
CROSS JOIN	Every row of the first table combined with every row of the second table

Complete Data Wrangling Workflow Using Databases

Data Sources
(CSV, Excel, APIs, PDF, Databases)
↓
Import into MySQL Database
↓
Organize Data into Tables
↓
Explore Data (SELECT, COUNT, AVG,
GROUP BY, ORDER BY, etc.)
↓
Join Related Tables
↓
Clean Missing and Duplicate Data
↓
Export to Pandas DataFrame
↓
Perform EDA and Visualization
↓
Build Machine Learning Models

Key Examination Points

1. A **database** is an organized collection of related data managed by a **DBMS**.
2. A **Relational Database Management System (RDBMS)** stores data in tables connected through **Primary Keys** and **Foreign Keys**.
3. **MySQL** is a widely used open-source RDBMS that uses **SQL** to create, retrieve, update, and manage data.
4. **Data exploration** helps understand dataset structure, identify missing values, duplicates, data types, and statistical properties before analysis.
5. Common SQL aggregate functions include **COUNT()**, **SUM()**, **AVG()**, **MAX()**, and **MIN()**.
6. Clauses such as **GROUP BY**, **HAVING**, **ORDER BY**, and **DISTINCT** are used to summarize and organize data.

7. **INNER JOIN** returns only matching records, **LEFT JOIN** returns all left-table records, **RIGHT JOIN** returns all right-table records, **FULL OUTER JOIN** returns all records from both tables (not directly supported in MySQL), and **CROSS JOIN** returns the Cartesian product.
8. Python libraries such as **mysql-connector-python** and **Pandas** enable seamless integration between MySQL databases and data analysis workflows.

Unit 4 Exploratory Data Analysis **10**

Visualization with Matplotlib and Seaborn**,** Customizing Plots: Titles, Legends, Labels and Themes Distribution Plots: Histograms, Boxplots, KDE, Bar Charts, Count Plots, Pie Charts, Scatter Plots, Pair Plots, Heatmaps, Correlation and Covariance Analysis, Advanced Visuals: Violin Plots, Strip Plots, Swarm Plots, Multivariate Visualization and Subplots

---

Visualization with Matplotlib and Seaborn

What is Data Visualization?

Data visualization is the **process of representing data graphically** to identify patterns, trends, and relationships that are difficult to see in raw numbers.

- It transforms **raw data into visual stories**.
- Helps in **decision making**, **exploratory data analysis**, and **presentations**.

Why Visualization is Important

Purpose	Example
Identify trends	Line plots showing monthly sales
Compare categories	Bar charts for product performance
Analyse relationships	Scatter plots between variables
Explore distributions	Histograms, box plots
Display correlations	Heatmaps

Matplotlib — The Foundation Library

Matplotlib is a **low-level** plotting library that provides control over every aspect of a figure.

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

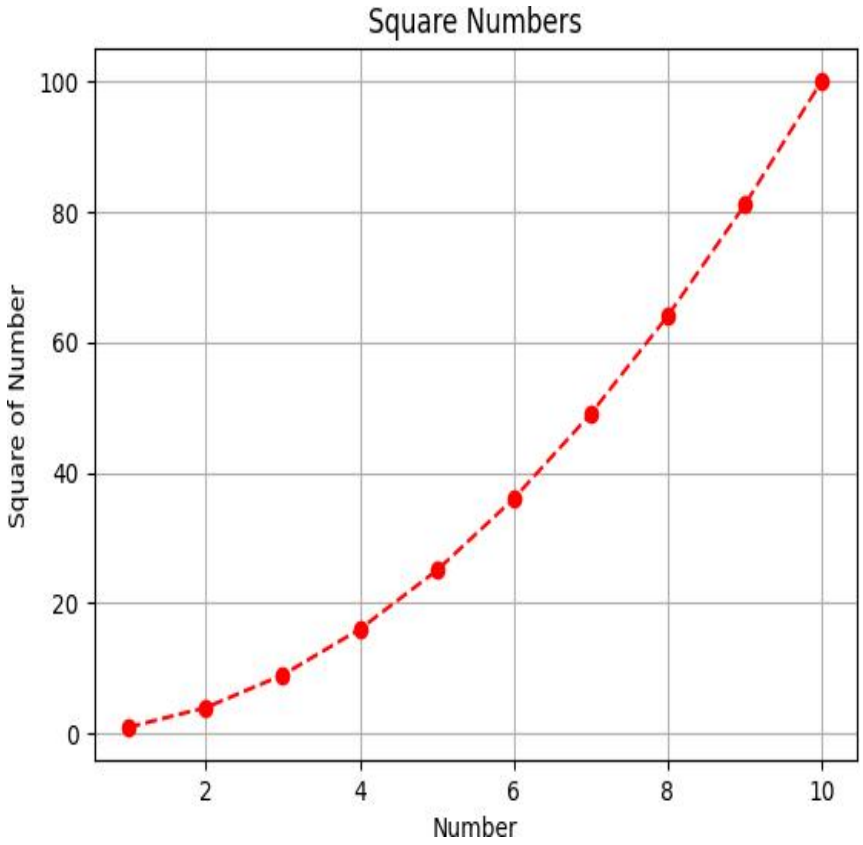
Basic Plot Types in Matplotlib

Type	Function	Description	Example
Line Plot	plt.plot()	For trends over time or continuous data	plt.plot(x, y)
Scatter Plot	plt.scatter()	For relationships between two variables	plt.scatter(x, y)
Bar Plot	plt.bar()	Comparing categorical values	plt.bar(categories, values)
Histogram	plt.hist()	Visualizing distributions	plt.hist(data, bins=10)
Pie Chart	plt.pie()	Showing proportions of categories	plt.pie(sizes, labels=labels)
Box Plot	plt.boxplot()	Spread, median, outliers	plt.boxplot(data)

Example: Line Plot

```
import matplotlib.pyplot as plt import numpy
as np
x = np.arange(1, 11) y = x **
2
plt.plot(x, y, color='red', marker='o', linestyle='--') plt.title("Square Numbers")
plt.xlabel("Number") plt.ylabel("Square of
Number") plt.grid(True)
plt.show()
```

Output:



Seaborn — Statistical Visualization Simplified

Seaborn is a high-level data visualization library built on top of Matplotlib.It is designed for creating attractive, informative, and statistical graphics easily.

Important Features

- Built-in themes and color palettes
- Integrated with Pandas DataFrames
- Simplifies statistical visualizations (e.g., regression lines, error bars)
- Includes built-in datasets (Iris, Titanic, Tips, etc.)
- Automatically handles categorical vs. numerical data

#Importing Seaborn

```
import seaborn as sns
import matplotlib.pyplot as plt
```

Basic Seaborn Plot Types

Type	Function	Description
Scatter Plot	sns.scatterplot()	Relationship between two numeric variables
Line Plot	sns.lineplot()	Trend visualization
Bar Plot	sns.barplot()	Mean/summary of data with error bars
Count	sns.countplot	Frequency of categorical values

Plot	()	
Box Plot	sns.boxplot()	Summary statistics (median, IQR, outliers)
Violin Plot	sns.violinplot()	Distribution + density together
Pair Plot	sns.pairplot()	Pairwise relationship between all variables
Heatmap	sns.heatmap()	Correlation matrix or data matrix visualization

Seaborn Plot Categories

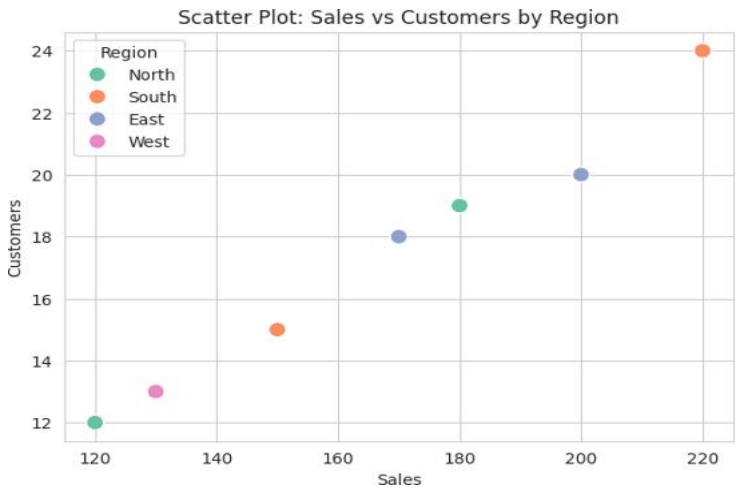
Seaborn plots can be grouped into **four major categories**:

Category	Purpose	Example Functions
Relational	Relationship between variables	`scatterplot()`, `lineplot()`
Categorical	Compare categories	`barplot()`, `boxplot()`, `violinplot()`, `stripplot()`, `swarmplot()`
Distributional	Data distributions	`histplot()`, `kdeplot()`, `rugplot()`
Multivariate/Matrix	Complex relationships	`heatmap()`, `pairplot()`, `clustermap()`

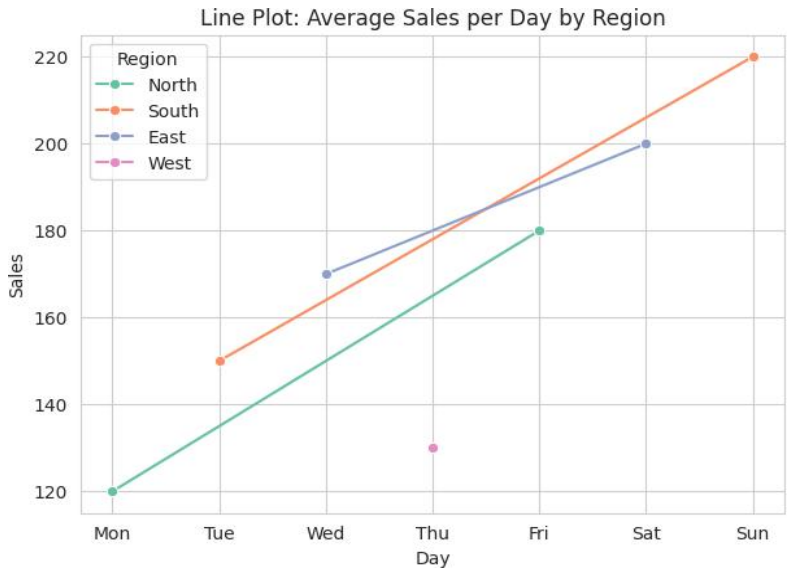
```
# ============================================================
# Seaborn Visualization: All Major Plots on One Pandas DataFrame #
============================================================
========

# Import Libraries import
pandas as pd import
seaborn as sns
import matplotlib.pyplot as plt

# Create Sample Pandas DataFrame data =
{
    'Day': ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'] * 10, 'Sales': [120, 150, 170,
    130, 180, 200, 220] * 10,
    'Customers': [12, 15, 18, 13, 19, 20, 24] * 10,
    'Discount': [5, 10, 7, 8, 12, 6, 9] * 10,
    'Region': ['North', 'South', 'East', 'West', 'North', 'East', 'South']
* 10
}

df = pd.DataFrame(data)
# Set Seaborn Style and Color Theme
sns.set_style("whitegrid")
sns.set_palette("Set2")

#
============================================================
======= # RELATIONAL PLOTS
# ============================================================

# Scatter Plot plt.figure(figsize=(7,5))
sns.scatterplot(x="Sales", y="Customers", hue="Region", data=df, s=100) plt.title("Scatter Plot:
```

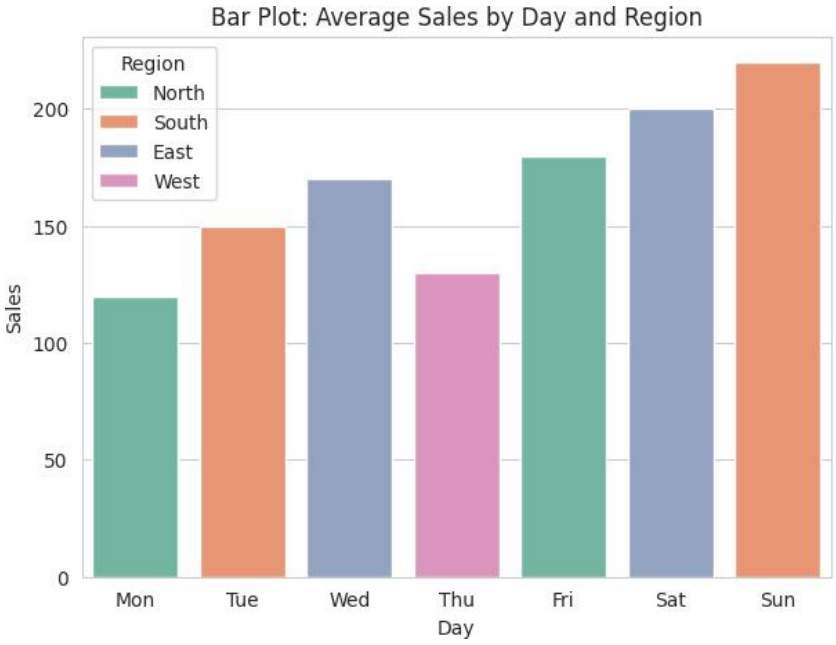
```
Sales vs Customers by Region")
plt.show()

# Line Plot plt.figure(figsize=(7,5))
sns.lineplot(x="Day", y="Sales", hue="Region", data=df, estimator='mean', marker='o')
plt.title("Line Plot: Average Sales per Day by Region") plt.show()

#
==========================================================
======= # CATEGORICAL PLOTS
# ============================================================

# Bar Plot plt.figure(figsize=(7,5))
sns.barplot(x="Day", y="Sales", hue="Region", data=df, estimator='mean', ci=None)
plt.title("Bar Plot: Average Sales by Day and Region") plt.show()

# Count Plot plt.figure(figsize=(7,5))
sns.countplot(x="Region",data=df)
plt.title("Count Plot: Frequency of Each Region") plt.show()

# Box Plot plt.figure(figsize=(7,5))
sns.boxplot(x="Region", y="Sales", data=df) plt.title("Box Plot: Sales
Distribution by Region") plt.show()

# Violin Plot plt.figure(figsize=(7,5))
sns.violinplot(x="Region", y="Customers", data=df, palette="pastel") plt.title("Violin
Plot: Customers Distribution by Region") plt.show()




# Swarm Plot plt.figure(figsize=(7,5))
sns.swarmplot(x="Region", y="Sales", data=df, color="black", alpha=0.6) plt.title("Swarm Plot:
Sales Spread by Region")
plt.show()

#
==========================================================
======= # DISTRIBUTION PLOTS
# ============================================================

# Histogram plt.figure(figsize=(7,5))
sns.histplot(df["Sales"], bins=10, kde=True, color='skyblue') plt.title("Histogram: Distribution of
Sales")
plt.show()

# KDE Plot plt.figure(figsize=(7,5))
sns.kdeplot(df["Customers"], shade=True, color='green') plt.title("KDE Plot:
Customer Density")
```

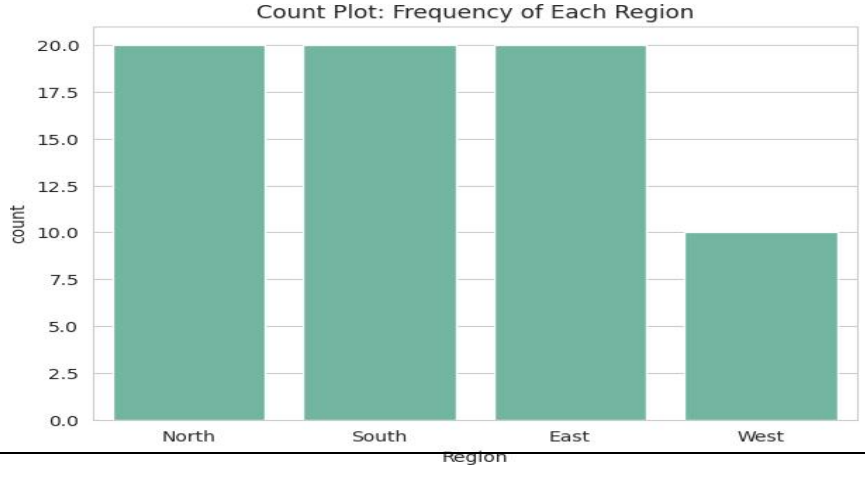
```
plt.show()

#
============================================================
======= # MATRIX / MULTIVARIATE PLOTS
# ============================================================

# Pair Plot
sns.pairplot(df, hue="Region")
plt.suptitle("Pair Plot: Relationships Between Numeric Variables", y=1.02) plt.show()

# Heatmap plt.figure(figsize=(7,5))
corr = df.corr(numeric_only=True)
sns.heatmap(corr, annot=True, cmap="coolwarm", linewidths=0.5) plt.title("Heatmap: Correlation
Matrix")
plt.show()

#
============================================================
======= # FACETGRID (MULTIPLE SUBPLOTS AUTOMATICALLY)
# ============================================================

g = sns.FacetGrid(df, col="Region") g.map_dataframe(sns.histplot, x="Sales",
bins=8, color="teal")
g.fig.suptitle("FacetGrid: Sales Distribution by Region", y=1.05) plt.show()
```

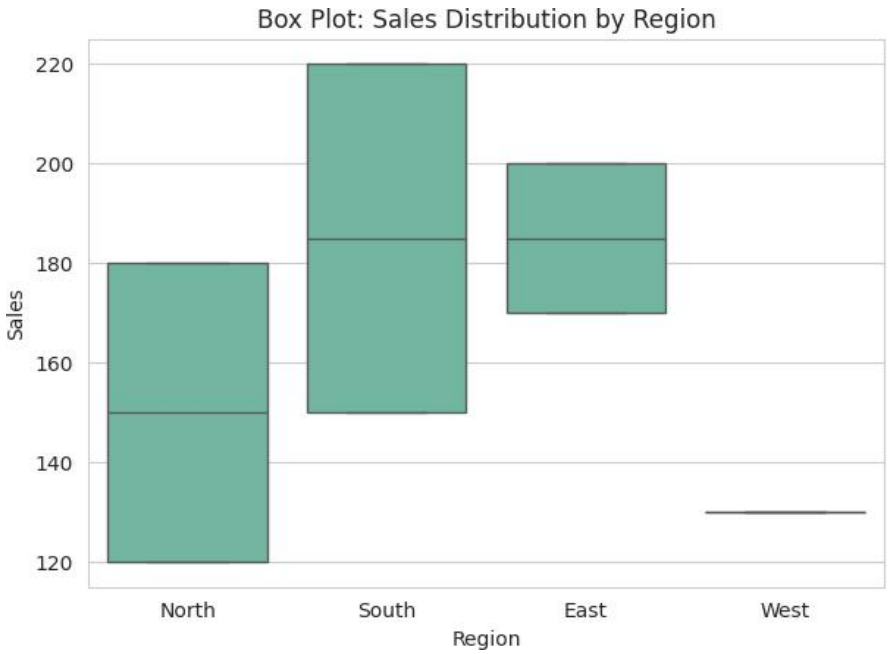


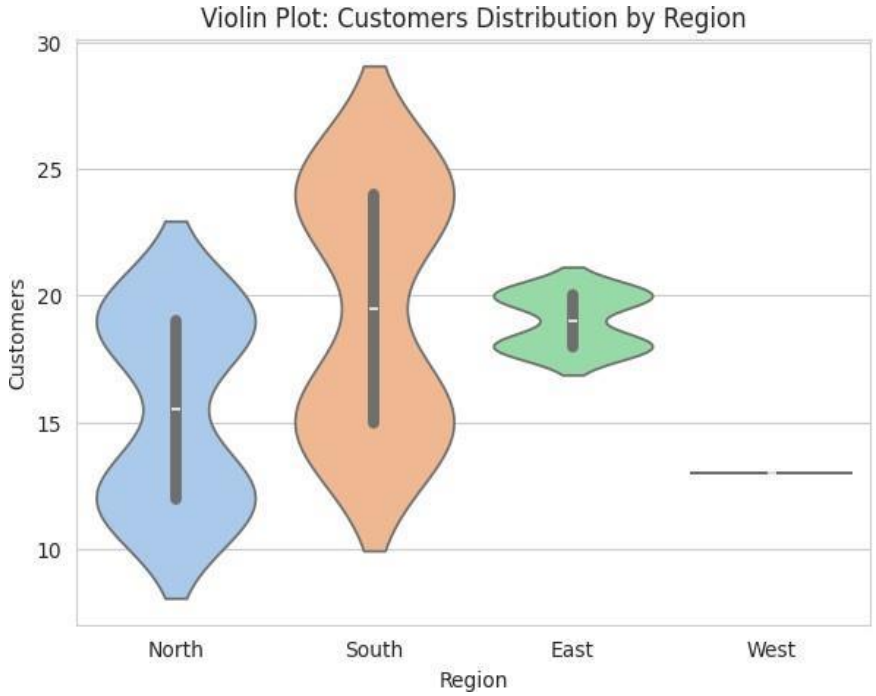
OHM SRI SAIRAM

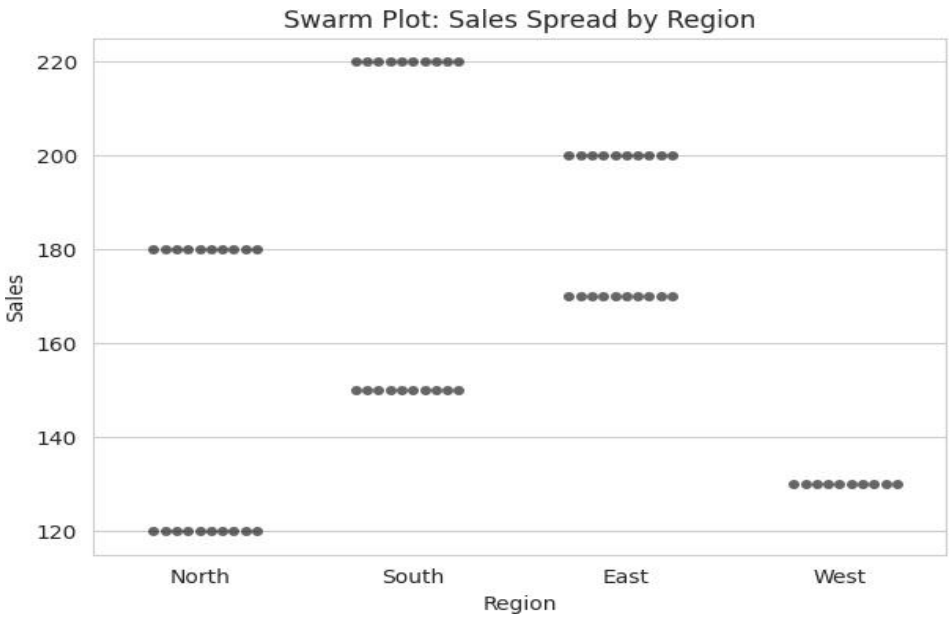


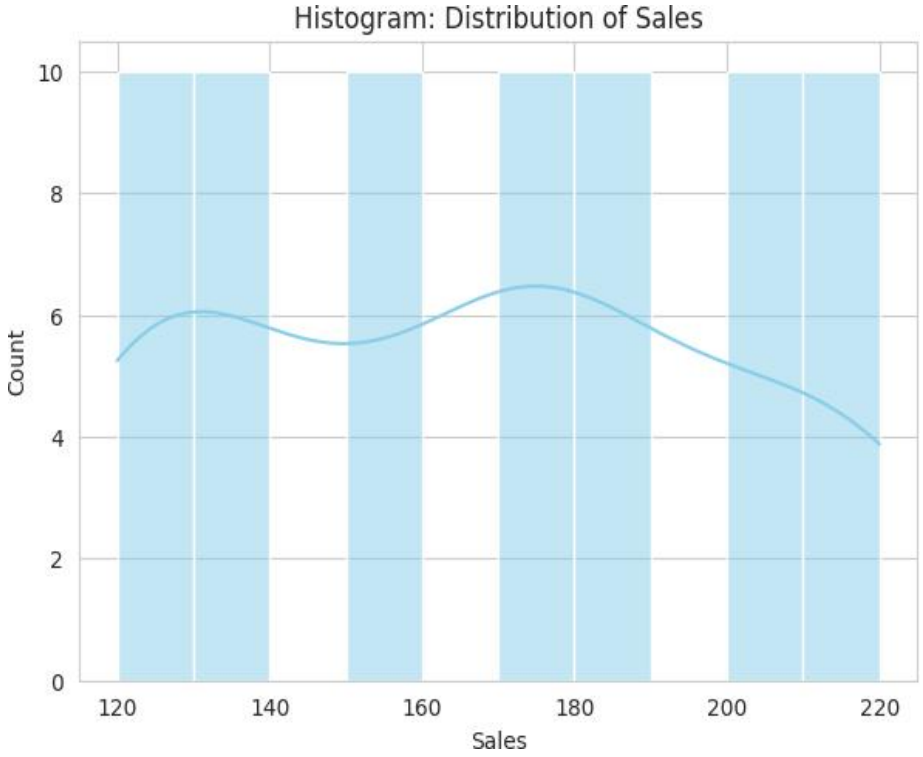


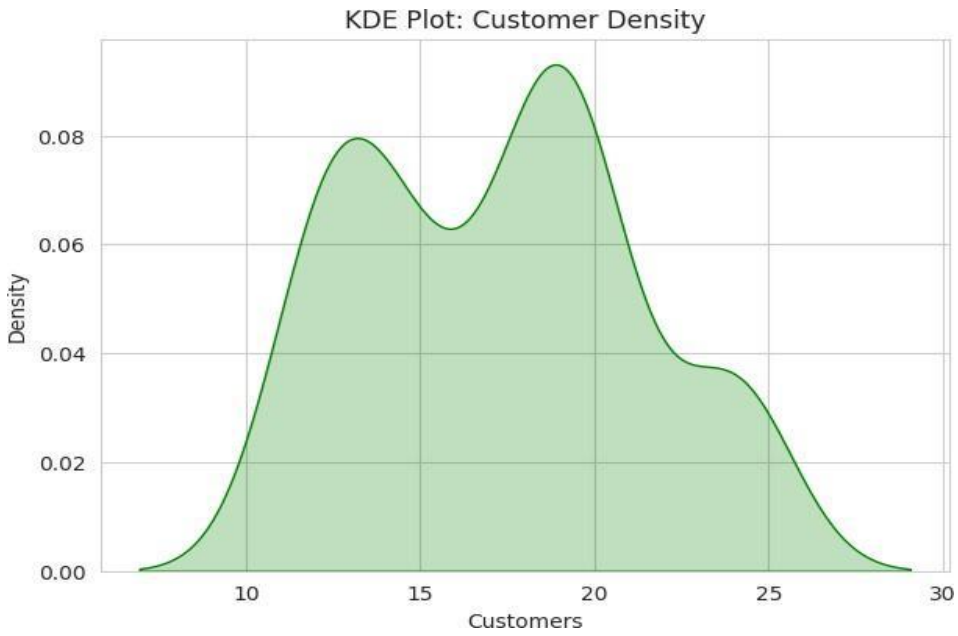


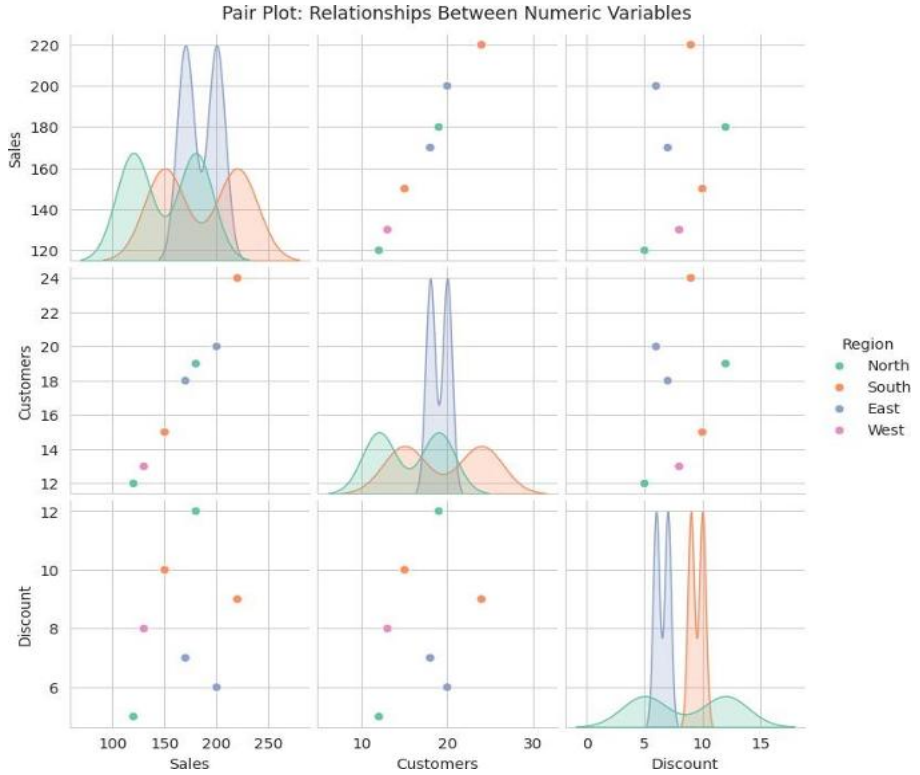

Box Plot: Sales Distribution by Region
Sales
220
200
180
160
140
120
North
South
East
West
Region

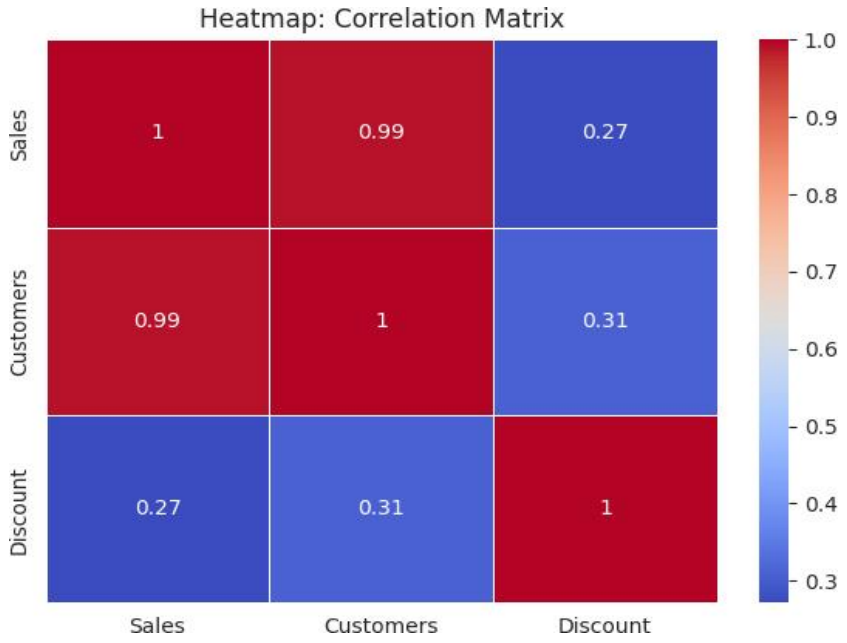

Violin Plot: Customers Distribution by Region
Customers
30
25
20
15
10
North
South
East
West
Region

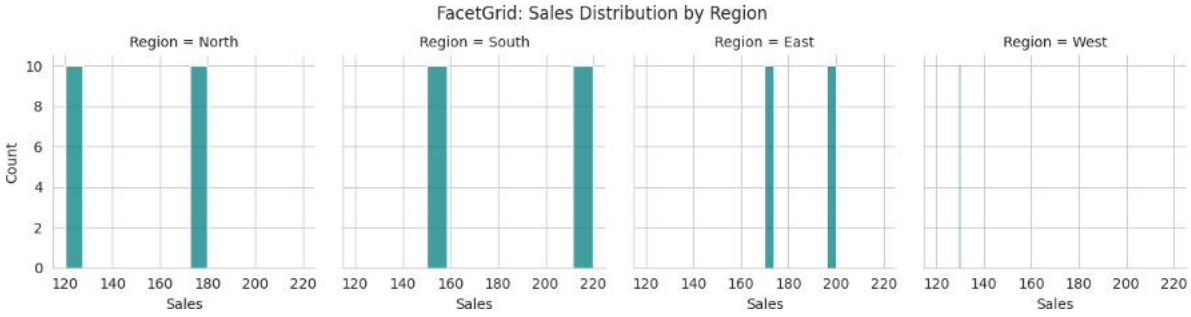

Swarm Plot: Sales Spread by Region
Sales
220
200
180
160
140
120
North
South
East
West
Region


Histogram: Distribution of Sales
Count
10
8
6
4
2
0
120
140
160
180
200
220
Sales


KDE Plot: Customer Density
Density
0.00
0.02
0.04
0.06
0.08
10
15
20
25
30
Customers


Pair Plot: Relationships Between Numeric Variables
Sales
Customers
Discount
Region
North
South
East
West
100
150
200
250
Sales
10
20
30
Customers
0
5
10
15
Discount





Customizing Plots: Titles, Legends, Labels, Themes

Visualization is not just about plotting data — it's about **communicating insights clearly**. Customization helps viewers **quickly understand the story behind the data**.

In **Matplotlib and Seaborn**, we will focus on **all customization options**:

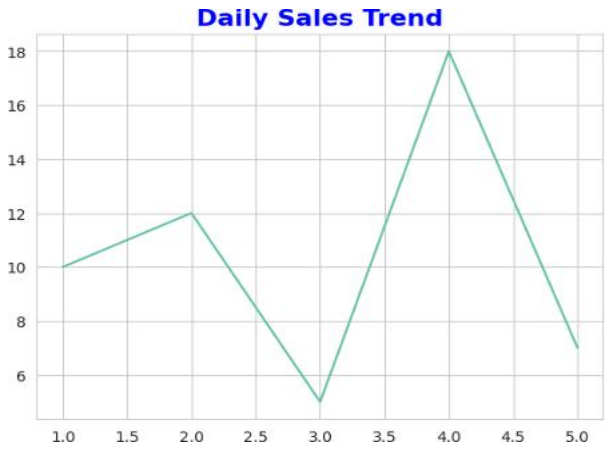
1. Titles
2. Axis Labels
3. Legends
4. Themes / Styles
5. Figure Size
6. Colors, Markers, and Line Styles
7. Font Styles & Sizes
8. Combining Multiple Customizations

1. Titles:

Titles are the **first element viewers notice**. A good title tells **what the plot is about**.

Matplotlib

```
import matplotlib.pyplot as plt x = [1, 2, 3,
4, 5]
y = [10, 12, 5, 18, 7]
plt.plot(x, y)
plt.title("Daily Sales Trend", fontsize=16, color='blue', fontweight='bold', loc='center')
plt.show()
```



Seaborn

Seaborn uses Matplotlib for rendering, so customization is the same:

```
import seaborn as sns
import pandas as pd
df = pd.DataFrame({'Day': ['Mon','Tue','Wed','Thu','Fri'], 'Sales':[10,12,5,18,7]})
sns.lineplot(x='Day', y='Sales', data=df)
plt.title("Weekly Sales Trend", fontsize=16, fontweight='bold', color='darkgreen')
plt.show()
```

2. Axis Labels

Axis labels describe what each axis represents (units and meaning).

#Matplotlib Example

```
plt.plot(x, y, marker='o', linestyle='--', color='purple') plt.xlabel("Day of Week",
fontsize=14)
plt.ylabel("Total Sales ($)", fontsize=14) plt.title("Sales Trend with
Labels", fontsize=16) plt.show()
```

#Seaborn Example

```
sns.lineplot(x='Day', y='Sales', data=df, marker='o') plt.xlabel("Day", fontsize=12)
plt.ylabel("Sales ($)", fontsize=12) plt.title("Weekly
Sales Trend", fontsize=14) plt.show()
```

3. Legends

Legends identify **different series or categories** in a plot.

#Matplotlib Example

```
y2 = [5, 9, 7, 12, 10]
plt.plot(x, y, label="Store A", marker='o')
plt.plot(x, y2, label="Store B", marker='s', color='red') plt.xlabel("Day")
plt.ylabel("Sales")
plt.title("Sales Comparison: Store A vs B")
plt.legend(loc='upper left', fontsize=10, shadow=True, frameon=True) plt.show()
```

#Seaborn Example

```
df2 = pd.DataFrame({
        'Day': ['Mon','Tue','Wed','Thu','Fri']*2,
        'Sales': [10,12,5,18,7, 5,9,7,12,10],
        'Store': ['A']*5 + ['B']*5
})
sns.lineplot(x='Day', y='Sales', hue='Store', data=df2, marker='o') plt.title("Sales Comparison by
Store")
plt.show()
```

Themes and Styles:

Themes (or styles) are pre-defined **visual templates** that change the **look and feel** of your plots. They control things like:

- Background color
- Grid visibility
- Font styles
- Color cycles
- Line thickness and markers

Using themes ensures **consistent and professional visuals** without manually styling every element.

Matplotlib Styles
Matplotlib comes with many built-in styles. You can **list all available styles** using:

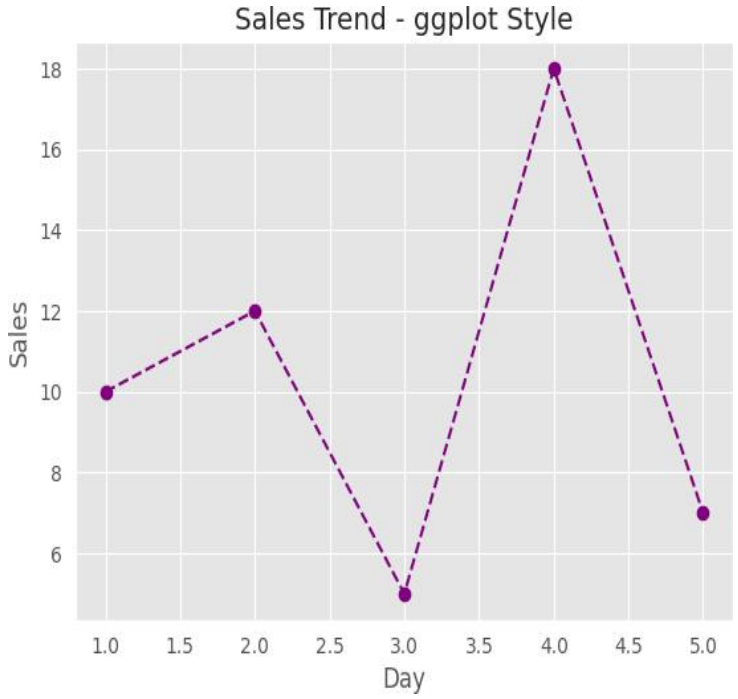
```
import matplotlib.pyplot as plt
print(plt.style.available)
```

Output:

['bmh', 'classic', 'dark_background', 'fast', 'fivethirtyeight', 'ggplot', 'grayscale', 'seaborn', 'seaborn-bright', 'seaborn-dark', ...]

Applying a Style

```
import matplotlib.pyplot as plt x = [1,
2, 3, 4, 5]
y = [10, 12, 5, 18, 7]
plt.style.use('ggplot')                    # Set ggplot style
plt.plot(x, y, marker='o', linestyle='--', color='purple') plt.title("Sales Trend - ggplot
Style")
plt.xlabel("Day")
plt.ylabel("Sales")
plt.show()
```



2. Seaborn Themes (Styles)
Seaborn comes with **high-level themes** that automatically set background, grid, and axes aesthetics.
Available Seaborn Styles

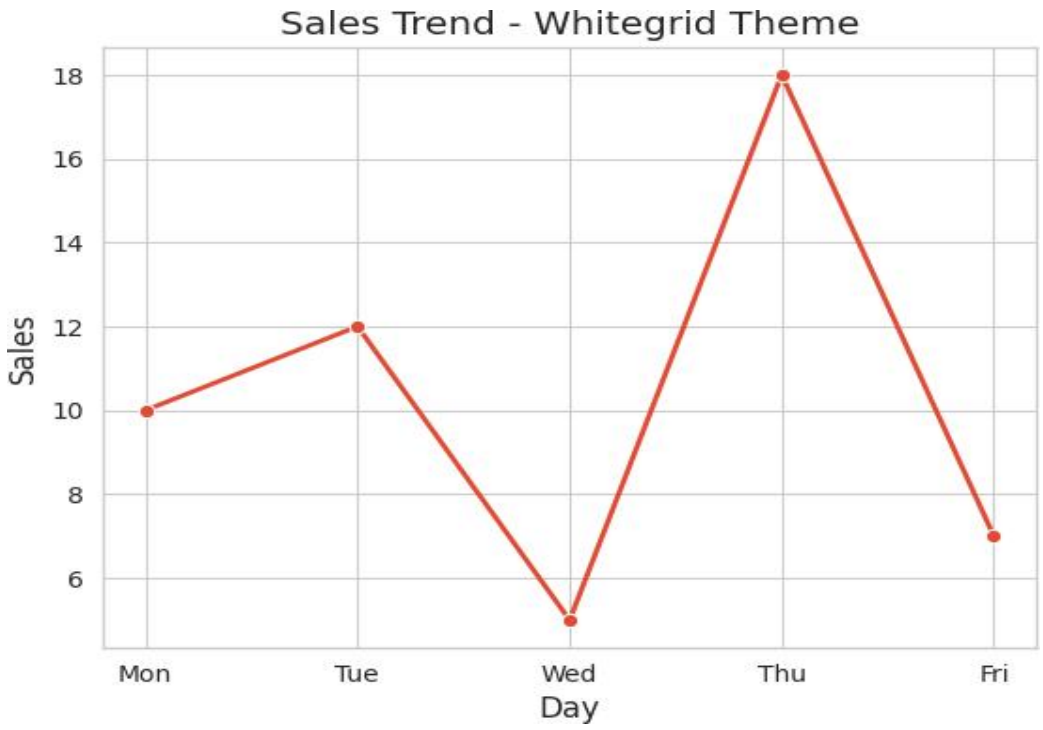
```
import seaborn as sns
print(sns.axes_style().keys())
# Returns keys: 'white', 'dark', 'whitegrid', 'darkgrid', 'ticks'
```

Syntax to Apply a Theme

```
import seaborn as sns
sns.set_style(style='whitegrid')  # Apply 'whitegrid' theme
```

#Example

```
import seaborn as sns
import matplotlib.pyplot as plt

import pandas as pd
# Sample data
df = pd.DataFrame({
    'Day':['Mon','Tue','Wed','Thu','Fri'], 'Sales': [10, 12,
    5, 18, 7]
})

# Set Seaborn theme
sns.set_style("whitegrid")

# Plot
sns.lineplot(x='Day', y='Sales', data=df, marker='o', linewidth=2) plt.title("Sales Trend - Whitegrid Theme")
plt.show()
```



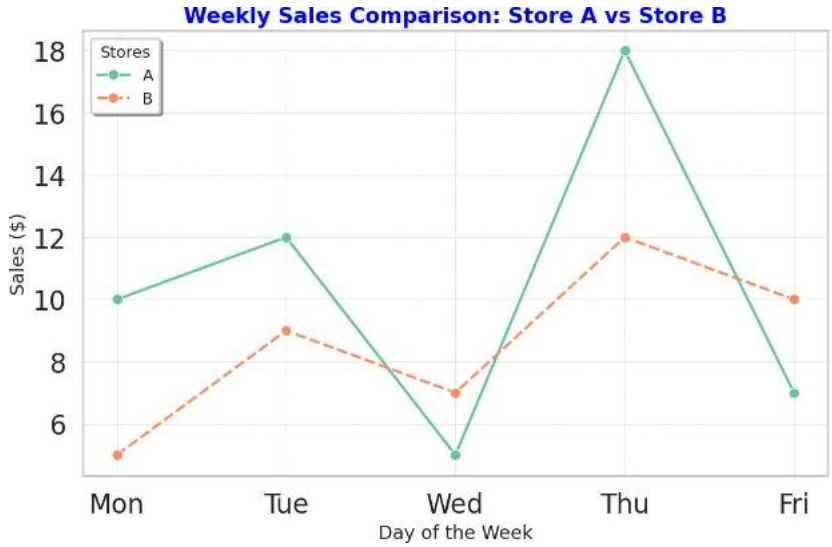
Sample Program for Customizing Plots: Titles, Legends, Labels, Themes using matplotlib

```
# ============================================================
# Customizing Matplotlib Plots: Titles, Legends, Labels, Themes #
============================================================
========

import matplotlib.pyplot as plt

# Sample Data
days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'] sales_A = [10, 12,
5, 18, 7]
sales_B = [5, 9, 7, 12, 10]

#Set Figure Size plt.figure(figsize=(10,6),
dpi=120)

#Apply Matplotlib Style (Theme)
plt.style.use('ggplot')  # Options: 'classic', 'seaborn', 'fivethirtyeight', 'bmh'

#Plot Multiple Series with Custom Markers, Lines, Colors plt.plot(days, sales_A,
label='Store A', marker='o', linestyle='--', color='purple', markersize=8, linewidth=2)
plt.plot(days, sales_B, label='Store B', marker='s', linestyle='-', color='green', markersize=8,
linewidth=2)

#Customize Title
plt.title("Weekly Sales Comparison: Store A vs Store B", fontsize=16, fontweight='bold',
color='blue')

#Customize Axis Labels
plt.xlabel("Day of the Week", fontsize=14) plt.ylabel("Sales ($)",
fontsize=14)

#Customize Legend
plt.legend(loc='upper left', fontsize=12, shadow=True, frameon=True, title="Stores")

#Customize Grid and Ticks
plt.grid(True, linestyle='--', linewidth=0.5, alpha=0.7)  # Minor customization of grid

# Show Plot plt.show()
```

Sample Program for Customizing Plots: Titles, Legends, Labels, Themes using matplotlib

```
# ============================================================
# Customizing Seaborn Plots: Titles, Legends, Labels, Themes
# ============================================================

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

#Sample Data (Same as Matplotlib Example) df =
pd.DataFrame({
    'Day': ['Mon','Tue','Wed','Thu','Fri']*2,
    'Sales': [10,12,5,18,7, 5,9,7,12,10],
    'Store': ['A']*5 + ['B']*5
})

#Set Figure Size plt.figure(figsize=(10,6),
dpi=120)

#Set Seaborn Theme / Style
sns.set_style("whitegrid")  # Options: white, dark, whitegrid, darkgrid, ticks
sns.set_context("talk", font_scale=1.2)  # Larger fonts for presentations sns.set_palette("Set2")  #
Color palette for lines

#Create Line Plot with Markers sns.lineplot(
    x='Day', y='Sales',
    hue='Store',
    data=df,
    marker='o',
    linewidth=2,

    style='Store',          # Optional: different line styles for each Store
    markersize=8
```

```
)

#Customize Title
plt.title("Weekly Sales Comparison: Store A vs Store B", fontsize=16,
                fontweight='bold', color='blue')

#Customize Axis Labels
plt.xlabel("Day of the Week", fontsize=14) plt.ylabel("Sales ($)",
fontsize=14)

#Customize Legend
plt.legend(title="Stores", loc='upper left', fontsize=12, title_fontsize=12,
frameon=True, shadow=True)

#Optional: Adjust Gridlines (Seaborn handles grid via theme) plt.grid(True,
linestyle='--', linewidth=0.5, alpha=0.7)

# Show Plot plt.show()
```



Advanced Visuals: Violin Plots, Strip Plots, Swarm Plots

1. Violin Plots

A violin plot combines aspects of a box plot and a kernel density plot.

It shows the distribution (shape, spread, and probability density) of the data across different categories.

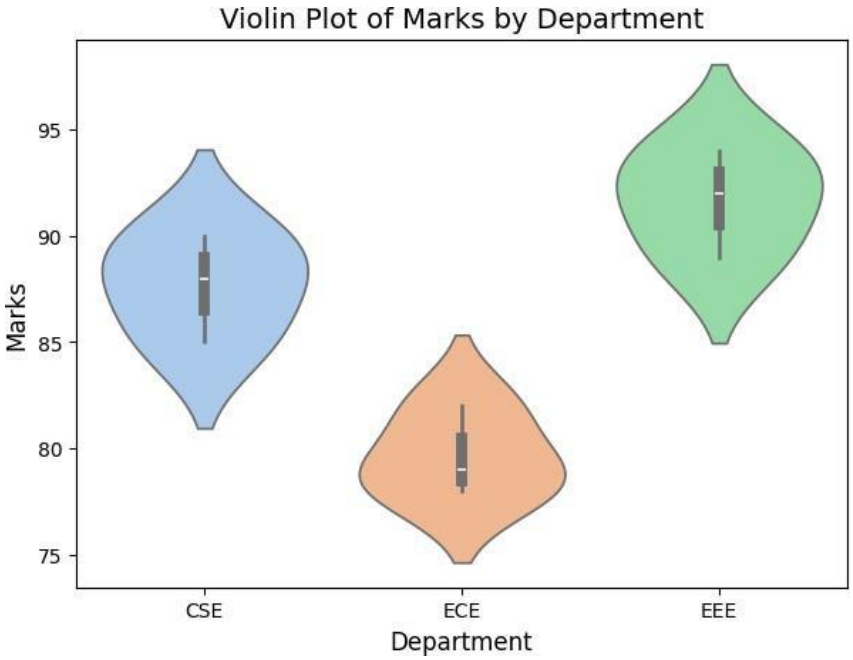
- The thick bar in the middle: similar to a box plot (shows interquartile range).
- The thin line (whisker): indicates the data range.
- The wider parts of the violin: show where the data is more concentrated (higher probability density).

Syntax

```
sns.violinplot(
    x=None,              # Categorical variable on x-axis y=None,      #
    Numeric variable to show distribution
    hue=None,            # Optional: split data by another category data=None,   #
    Pandas DataFrame containing the data order=None,  # Optional: order of
    categories on x-axis hue_order=None,# Optional: order of hue categories
    scale='area',  # How to scale violin widths ('area', 'count', 'width') inner='box',  # Show summary
    inside violin ('box', 'quartile', 'point', 'stick', None) split=False,      # Split violin in half if hue is used
    palette=None,  # Colors for categories or hue levels
)
```

Example Program

```
#import necessary libraries import
pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Create data using a dictionary data = {
    'Department': ['CSE', 'CSE', 'CSE', 'ECE', 'ECE', 'ECE', 'EEE', 'EEE', 'EEE'],
    'Marks': [85, 90, 88, 78, 82, 79, 92, 89, 94]
}

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

# Display the DataFrame
print("DataFrame:") print(df)

# Create a violin plot
plt.figure(figsize=(7,5))
sns.violinplot(x='Department', y='Marks', data=df, inner='box', palette='pastel')

# Add title and labels
plt.title('Violin Plot of Marks by Department', fontsize=14) plt.xlabel('Department', fontsize=12)
plt.ylabel('Marks', fontsize=12)

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



2. Strip Plots

A strip plot is used to display individual data points along a categorical axis. It's like a scatter plot for categorical data and is useful for:

- Visualizing distribution of values in each category
- Showing all individual observations (unlike boxplots or violin plots which summarize)
- Comparing subgroups with hue

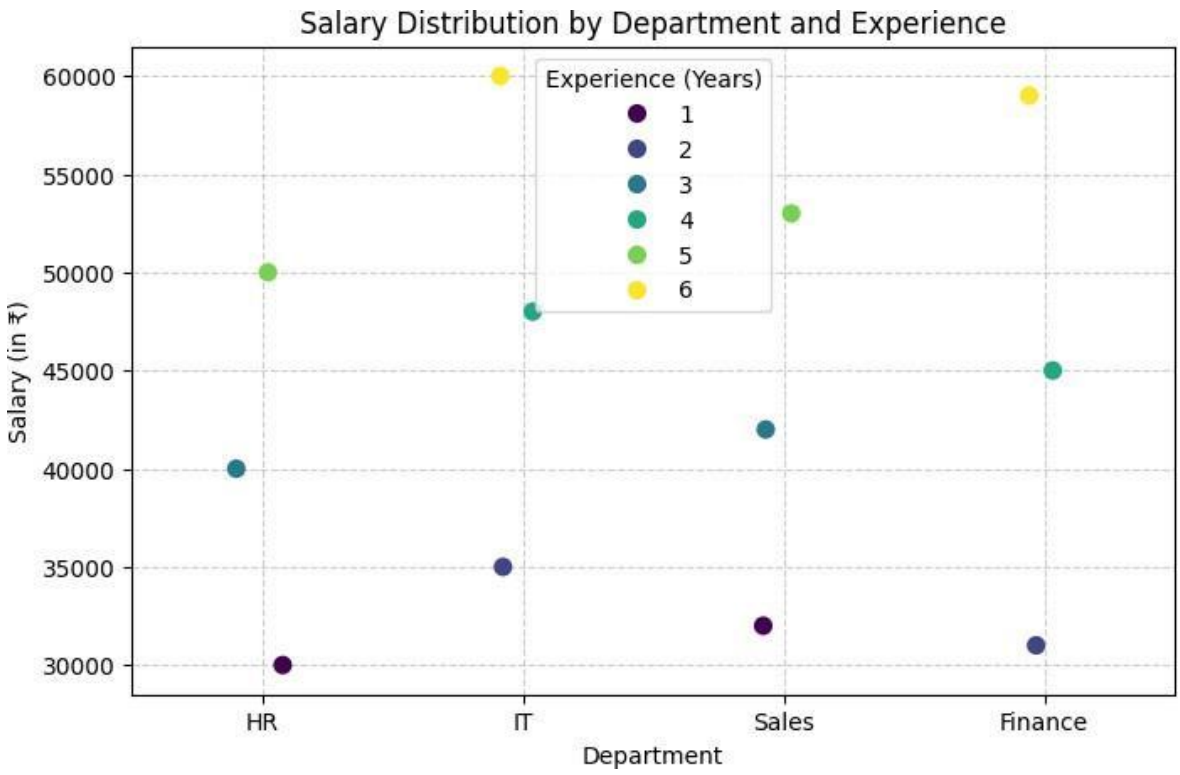
Syntax:

```
sns.stripplot(
    x=None,              # Categorical variable for x-axis y=None,
                         # Numeric variable for y-axis
    hue=None,            # Optional categorical variable to color data points data=None,#
    Pandas DataFrame containing the data order=None,  # Optional: order of
    categories on x-axis hue_order=None,  # Optional: order of hue levels
    jitter=True,         # Add random noise to points along categorical axis (True/False or float) size=5,  #
    Marker size
    palette=None,        # Color palette for hue or categories
)

# Import libraries import
pandas as pd import
seaborn as sns
import matplotlib.pyplot as plt

# ----------------------------------
# 1. Create dataset using a Python dictionary #
data = {----------------------------------
    'Department': ['HR', 'HR', 'HR', 'IT', 'IT', 'IT', 'Sales', 'Sales',
'Sales', 'Finance', 'Finance', 'Finance'],
    'Experience_Years': [1, 3, 5, 2, 4, 6, 1, 3, 5, 2, 4, 6],
```

```
    'Salary': [30000, 40000, 50000, 35000, 48000, 60000, 32000, 42000,
53000, 31000, 45000, 59000]
}

# Convert dictionary to pandas DataFrame df =
pd.DataFrame(data)

print("Employee Dataset:") print(df)

# ------------------------------------
# 2. Create a vertical strip plot #
plt.figure(figsize=(8,5)) ----------------------
sns.stripplot(x='Department', y='Salary', data=df, jitter=True, palette='Set2', size=8)
plt.title("Strip Plot of Salary by Department") plt.xlabel("Department")
plt.ylabel("Salary (in ₹)") plt.grid(True,
linestyle="--", alpha=0.6) plt.show()

# ------------------------------------
# 3. Horizontal strip plot #
plt.figure(figsize=(8,5)) ----------------------
sns.stripplot(y='Department', x='Salary', data=df, jitter=True, palette='cool', size=8)
plt.title("Horizontal Strip Plot of Salary by Department") plt.xlabel("Salary (in ₹)")
plt.ylabel("Department")
plt.grid(True, linestyle="--", alpha=0.6) plt.show()

# ------------------------------------
# 4. Strip plot with Experience hue #
plt.figure(figsize=(8,5)) ----------------------
sns.stripplot(x='Department', y='Salary', hue='Experience_Years', data=df, jitter=True,
palette='viridis', size=8)
plt.title("Salary Distribution by Department and Experience") plt.xlabel("Department")
plt.ylabel("Salary (in ₹)") plt.legend(title="Experience
(Years)") plt.grid(True, linestyle="--", alpha=0.6)
plt.show()
```

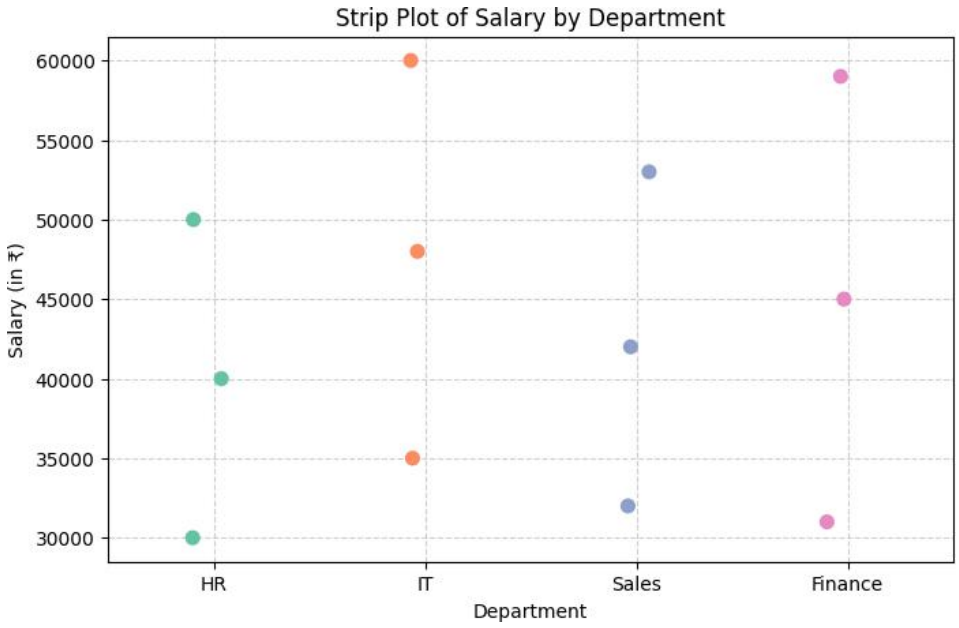
Output

Vertical Strip Plot

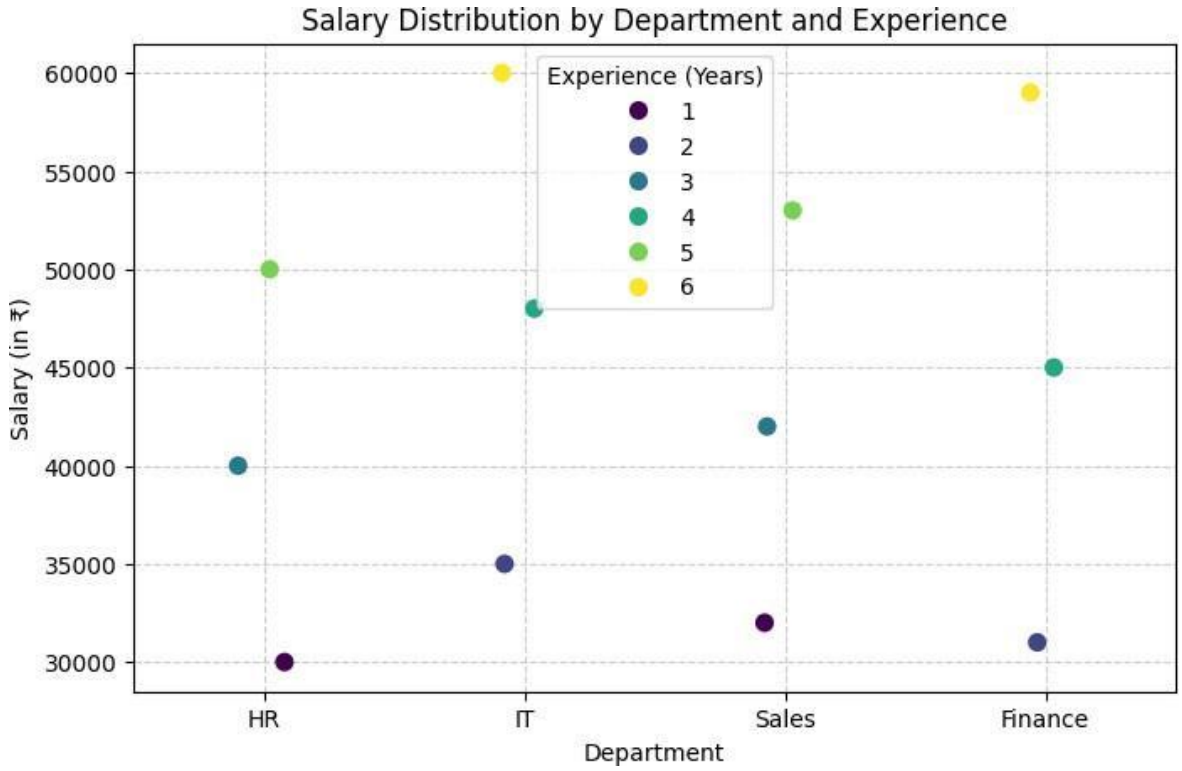


Output

Horizontal Strip Plot



Horizontal Strip Plot with Hue



3. Swarm Plots

A **swarm plot** is similar to a **strip plot** but **avoids overlapping points** by adjusting their positions along the categorical axis.

Use cases:

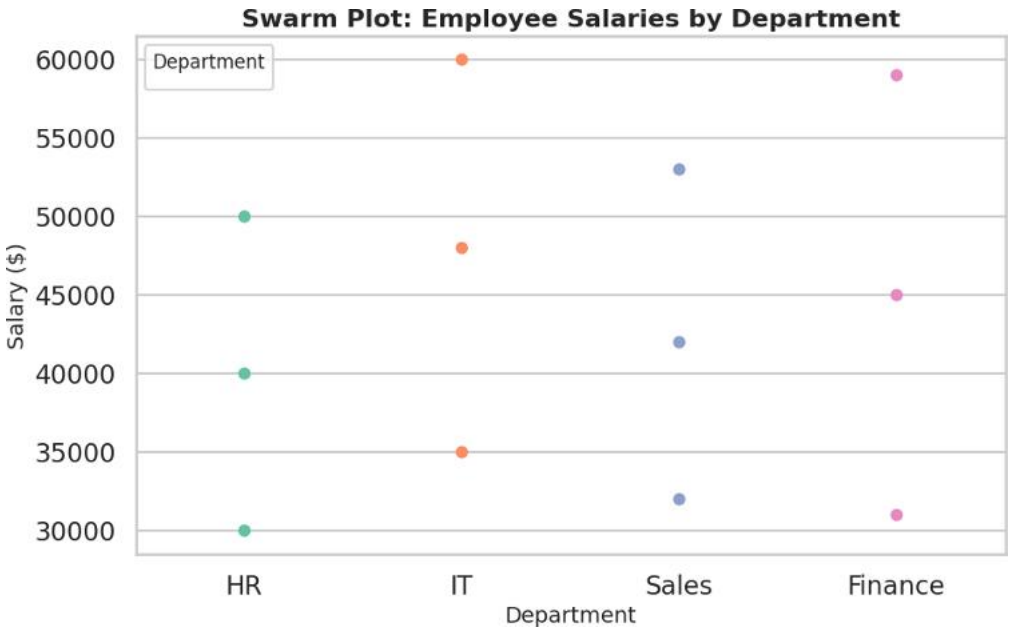
- Visualizing **individual observations** in each category
- Showing **distribution** without overlapping points
- Comparing **subgroups with hue**

Syntax:

```
sns.swarmplot(
    x=None,          # Categorical variable on x-axis
    y=None,          # Numeric variable on y-axis
    hue=None,        # Optional categorical variable to color points data=None,
                     # Pandas DataFrame containing the data order=None,
                     # Order of categories on x-axis hue_order=None,# Order
    of hue categories
    size=5,          # Size of points
    palette=None,  # Color palette for hue or categories marker='o',
                     # Shape of points
)
```

Example: Swarm Plot for Employee Salary Dataset

```
# Import libraries import
pandas as pd import
seaborn as sns
import matplotlib.pyplot as plt

# Dataset (already provided) data = {
    'Department': ['HR', 'HR', 'HR', 'IT', 'IT', 'IT', 'Sales', 'Sales',
'Sales', 'Finance', 'Finance', 'Finance'],
    'Experience_Years': [1, 3, 5, 2, 4, 6, 1, 3, 5, 2, 4, 6],
    'Salary': [30000, 40000, 50000, 35000, 48000, 60000, 32000, 42000,
53000, 31000, 45000, 59000]
}
df = pd.DataFrame(data)

# Set Seaborn theme for aesthetics
sns.set_style("whitegrid")
sns.set_context("talk")

# Create Swarm Plot
plt.figure(figsize=(10,6)) sns.swarmplot(
    x='Department',   # Categorical axis y='Salary',     #
    Numeric variable data=df,
    size=8,                          # Size of points palette='Set2',       #
    Colors for departments hue='Department', # Optional: color by
    department
    dodge=False                      # Keep points overlapping vertically for each
department
)

# Add title and axis labels
plt.title("Swarm Plot: Employee Salaries by Department", fontsize=16, fontweight='bold')
plt.xlabel("Department", fontsize=14) plt.ylabel("Salary
($)", fontsize=14)

#Optional: Legend customization plt.legend(title="Department",
loc='upper left', fontsize=12, title_fontsize=12)

# Show plot plt.show()
```



Multivariate Visualizations and Subplots

Multivariate visualization is used when we want to **analyze relationships between 3 or more variables** in a dataset.

Common techniques:

1. **Pair Plots**: Scatterplots for every pair of variables (with histograms on diagonal)
2. **Heatmaps** :Correlation matrices
3. **Joint Plots / Pair Grids** :Detailed analysis between two or more variables
4. **Subplots** : Multiple plots in one figure for comparison

Multivariate Visualization Program:

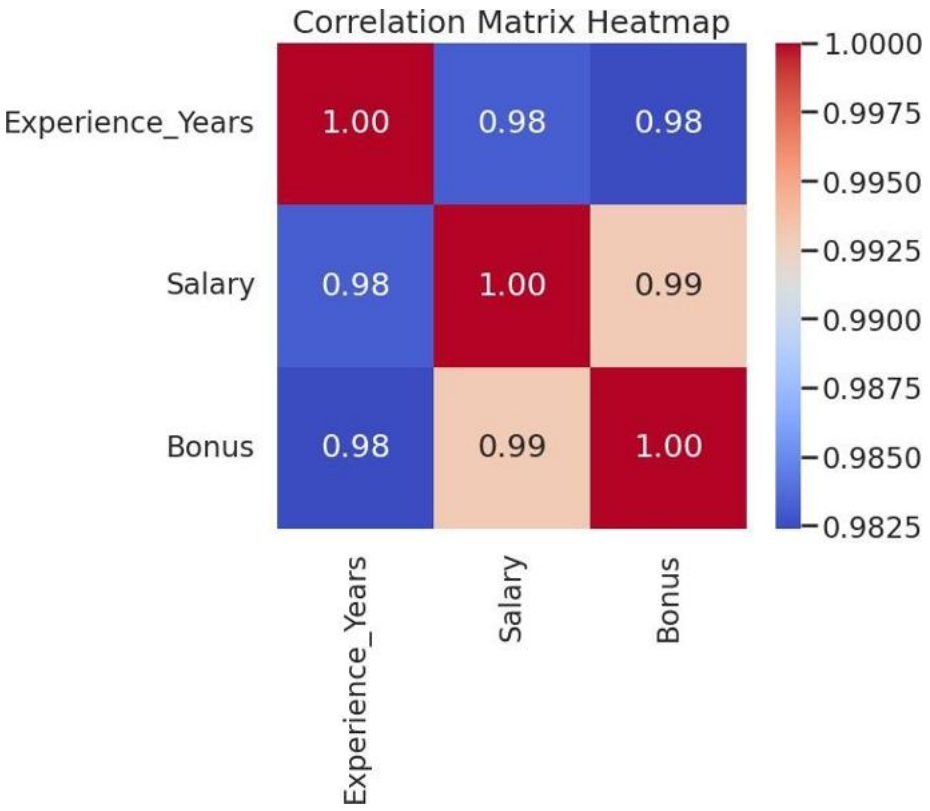
```
# ----------------------------
# Import Libraries #
# ----------------------------
import pandas as pd import
seaborn as sns
import matplotlib.pyplot as plt

# ----------------------------
# Create Dataset #
data = {----------------------
    'Department': ['HR','HR','HR','IT','IT','IT',
                   'Sales','Sales','Sales',
                   'Finance','Finance','Finance'], 'Experience_Years':
    [1,3,5,2,4,6,1,3,5,2,4,6],
    'Salary': [30000,40000,50000,35000,48000,60000,
               32000,42000,53000,31000,45000,59000],
    'Bonus': [2000, 3000, 4000, 2500, 3500, 4500, 2200,
              3200, 4200, 2100, 3300, 4700]
}
```

```
df = pd.DataFrame(data)
print("Employee Dataset:") print(df)

# ----------------------------
# Set Seaborn Style #
# ----------------------------
sns.set_style("whitegrid")
sns.set_context("talk")

# ----------------------------
# Pair Plot: Multivariate Relationships #
# ----------------------------
sns.pairplot(df, hue='Department', diag_kind='hist', palette='Set2') plt.suptitle("Pair Plot:
Salary, Bonus, and Experience by Department", y=1.02)
plt.show()

# ----------------------------
# Heatmap: Correlation Matrix #
# ----------------------------
corr = df[['Experience_Years', 'Salary', 'Bonus']].corr() plt.figure(figsize=(6,5))
sns.heatmap(corr, annot=True, cmap='coolwarm', fmt=".2f") plt.title("Correlation Matrix
Heatmap")
plt.show()

# ----------------------------
# Subplots: Multiple Scatter Plots #
# ----------------------------
fig, axes = plt.subplots(1,3, figsize=(18,5))

# Scatter plot: Experience vs Salary sns.scatterplot(x='Experience_Years', y='Salary',
hue='Department', data=df, ax=axes[0], palette='Set2', s=100)
axes[0].set_title("Experience vs Salary")

# Scatter plot: Experience vs Bonus
sns.scatterplot(x='Experience_Years', y='Bonus', hue='Department', data=df, ax=axes[1],
palette='Set2', s=100)
axes[1].set_title("Experience vs Bonus")

# Scatter plot: Salary vs Bonus
sns.scatterplot(x='Salary', y='Bonus', hue='Department', data=df, ax=axes[2], palette='Set2',
s=100)
axes[2].set_title("Salary vs Bonus")

plt.tight_layout() plt.show()
```

Output:

Employee Dataset:

	Department	Experience_Years	Salary	Bonus
0	HR	1	30000	2000
1	HR	3	40000	3000
2	HR	5	50000	4000
3	IT	2	35000	2500
4	IT	4	48000	3500
5	IT	6	60000	4500
6	Sales	1	32000	2200
7	Sales	3	42000	3200
8	Sales	5	53000	4200
9	Finance	2	31000	2100
10	Finance	4	45000	3300
11	Finance	6	59000	4700



Plotly and Interactive Visualizations

Plotly is a modern, open-source Python library for creating **interactive, browser-based visualizations**.
It's built on **D3.js** and **WebGL**, allowing smooth, dynamic, and high-quality graphics.

Using Plotly we can

- Zoom, pan, and hover to explore data
- Build dashboards and web apps
- Export charts as images or HTML

Plotly Features

Feature	Description
Interactivity	Zoom, hover tooltips, select data points
Wide Chart Support	Line, Bar, Scatter, 3D plots, Maps, Pie charts, Heatmaps, etc.
Web Integration	Works seamlessly in Jupyter, Dash apps, or web browsers
Beautiful Output	High-quality, responsive, and modern visuals
Customizable	Control over color, layout, themes, and animation

Plotly Syntax Styles

Plotly provides **two main interfaces**:

Interface	Description	Example
Plotly Express (px)	High-level, simple syntax	px.scatter(df, x, y)
Graph Objects (go)	Low-level, full control	go.Figure(data=[go.Scatter(...)])

1. Plotly Express supported plots

Category	Chart Type	Function	Description / Use Case
Basic Statistical Charts	Scatter Plot	px.scatter()	Relationship between two numeric variables
	Line Chart	px.line()	Trend over time or sequence
	Bar Chart	px.bar()	Compare categorical data
	Histogram	px.histogram()	Distribution of numeric values
	Box Plot	px.box()	Visualize spread and outliers
Pie & Polar Charts	Pie Chart	px.pie()	Proportion of whole
	Sunburst	px.sunburst()	Hierarchical categories
	Treemap	px.treemap()	Nested rectangles for hierarchy
	Polar Scatter	px.scatter_polar()	Circular data representation
3D Charts	3D Scatter	px.scatter_3d()	Three-variable relationship
	3D Line	px.line_3d()	3D trajectory or path
	3D Surface	px.surface()	Continuous surface visualization
	3D Mesh	px.scatter_3d()	Visualize surfaces or point clouds
	3D Density	px.density_contour()	Data density visualization
Map-Based Charts	Scatter Geo	px.scatter_geo()	Plot points on map (lat/long)
	Line Geo	px.line_geo()	Connect points geographically
	Choropleth	px.choropleth()	Color-coded map by value

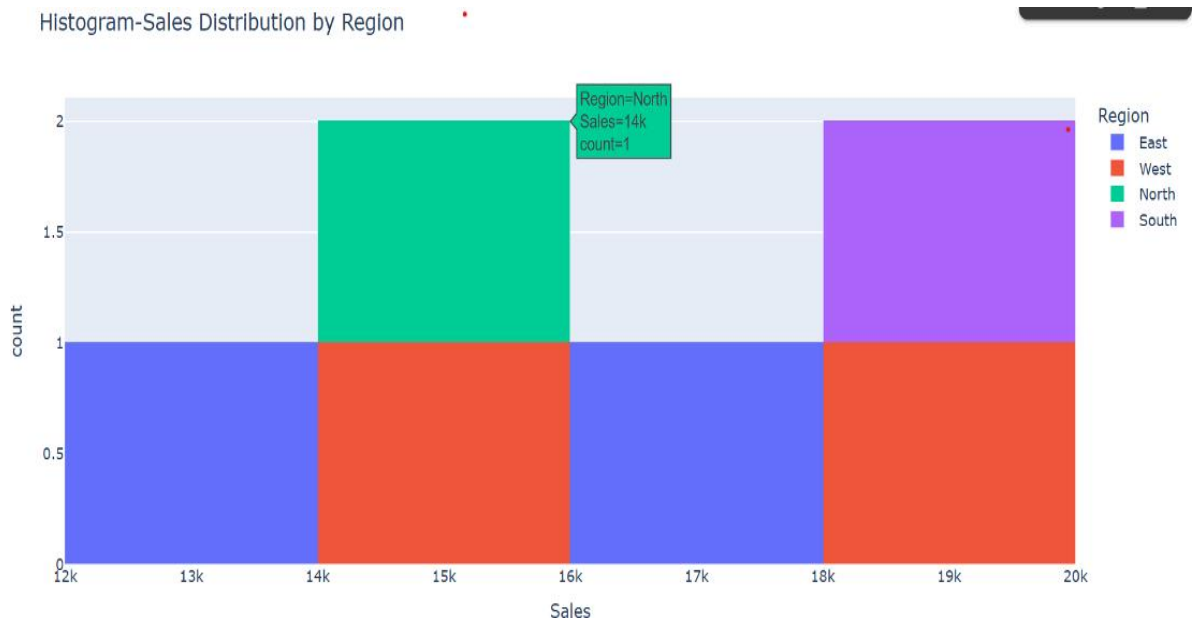
	Scatter Mapbox	px.scatter_mapbox()	Interactive bubble map
	Timeline	px.timeline()	Tasks/events over time

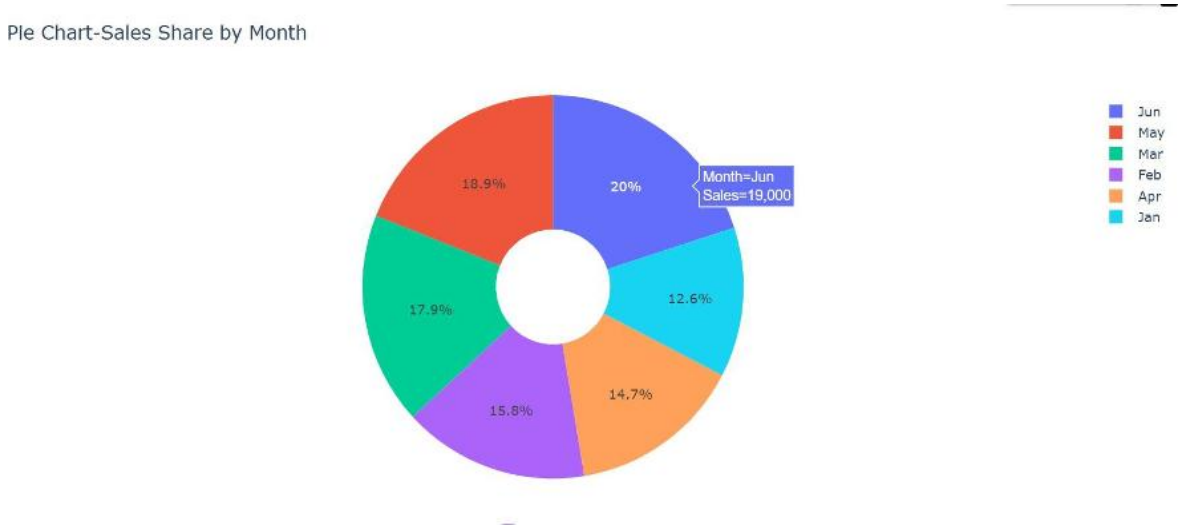
#Plotly express demo program on various charts

```
import pandas as pd
import plotly.express as px data = {
        'Month': ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'],

        'Sales': [12000, 15000, 17000, 14000, 18000, 19000],
        'Profit': [3000, 4000, 4500, 3200, 5000, 5200],
        'Region': ['East', 'West', 'East', 'North', 'South', 'West']
}

df = pd.DataFrame(data)
print("Sample DataFrame:") print(df)

# BAR CHART — Sales and Profit by Month fig_bar =
px.bar(
        df, x='Month',
        y=['Sales','Profit'], barmode='group',
        title='Bar Chart-Sales and Profit by Month'
)
fig_bar.show()

#  SCATTER PLOT — Sales vs Profit by Region
fig_scatter = px.scatter(
        df, x='Sales',
        y='Profit',
        color='Region',
        size='Sales',
        text='Month',
        title='Scatter Plot-Sales vs Profit Scatter Plot'
)
fig_scatter.update_traces(textposition='top center') fig_scatter.show()

#  HISTOGRAM — Distribution of Sales by Region fig_hist =
px.histogram(
        df, x='Sales',
        nbins=5,
        color='Region',
        title='Histogram-Sales Distribution by Region'
)
fig_hist.show()

# PIE CHART — Sales Share by Month
fig_pie = px.pie(
```

```
        df, names='Month',
        values='Sales',
        title='Pie Chart-Sales Share by Month', hole=0.3  #
        makes it a donut chart
)
fig_pie.show()
```

OUTPUT:





2. Plotly Graph Objects

Plotly Graph objcts allows creation of fully customizable, interactive charts with complete control over traces, markers, and layout elements like titles, axes, and grids. It supports multiple traces, subplots, 3D charts, and annotations, making it ideal for complex visualizations and dashboards. Common charts include scatter, line, bar, box, pie, histogram, 3D scatter/line/surface, and tables. Typical use cases include multi-trace dashboards, combined scatter and box plots, or annotated 3D visualizations.

Syntax

```
import plotly.graph_objects as go #
Create a figure
fig = go.Figure()
# Add a trace (line, scatter, bar, etc.) fig.add_trace(go.Scatter(
    x=[x_values],
    y=[y_values],
    mode='lines+markers',          # 'lines', 'markers', or 'lines+markers'
    name='Trace Name'
))
```

Category	Chart Type	Graph Object Function	Description
Basic Charts	Scatter / Line	go.Scatter()	Plots lines, markers, or both; used for trends and relationships.
Basic Charts	Bar	go.Bar()	Creates vertical or horizontal bar charts to compare values across categories.
Distribution	Histogram	go.Histogram()	Shows frequency distribution of numeric data; supports multiple traces and bins.
Distribution	Box	go.Box()	Visualizes spread, quartiles, outliers, and mean of data.
Pie / Donut	Pie	go.Pie()	Displays proportion of values in a categorical variable; hole parameter creates donut charts.
3D Charts	3D Scatter	go.Scatter3d()	Visualizes relationships between three numeric variables in 3D space.
3D Charts	3D Line	go.Scatter3d()	Plots lines in 3D for trajectories or time series.
3D Charts	3D Surface	go.Surface()	Creates 3D surface plots for continuous data.

Subplots / Layout	Subplots	make_subplots() + add_trace()	Combines multiple charts into one figure with rows and columns.
Table / Data Display	Table	go.Table()	Displays tabular data interactively with headers and cells.

#Plotly graph demo program on pie charts

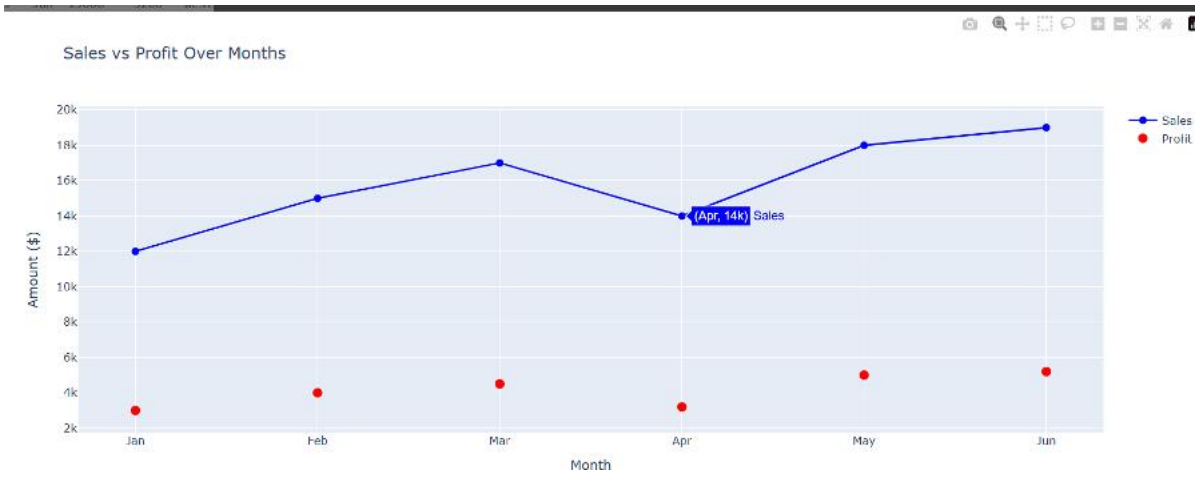
(Execute Program and paste outputs)

```
import pandas as pd
import plotly.graph_objects as go

# Sample DataFrame
data = {
    'Month': ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'], 'Sales': [12000,
    15000, 17000, 14000, 18000, 19000],
    'Profit': [3000, 4000, 4500, 3200, 5000, 5200],
    'Region': ['East', 'West', 'East', 'North', 'South', 'West']
}

df = pd.DataFrame(data)
print("Sample DataFrame:") print(df)

# Create figure fig =
go.Figure()

# Add Sales line + markers
fig.add_trace(go.Scatter(
    x=df['Month'],
    y=df['Sales'],
    mode='lines+markers',          # 'lines', 'markers', or 'lines+markers'
    name='Sales',
    line=dict(color='blue'),
    marker=dict(size=8)
))

# Add Profit scatter
fig.add_trace(go.Scatter(
    x=df['Month'],
    y=df['Profit'],
    mode='markers',
    name='Profit',
    marker=dict(color='red', size=10)
))
```

```
# Customize layout fig.update_layout(
        title='Sales vs Profit Over Months',
        xaxis_title='Month', yaxis_title='Amount ($)'
)

# Show figure fig.show()
```

OUTPUT:

Month	Sales	Profit	Region
Jan	12000	3000	East
Feb	15000	4000	West
Mar	17000	4500	East
Apr	14000	3200	North
May	18000	5000	South
Jun	19000	5200	West



#Plotly graph demo program on various charts (Execute and paste outputs)

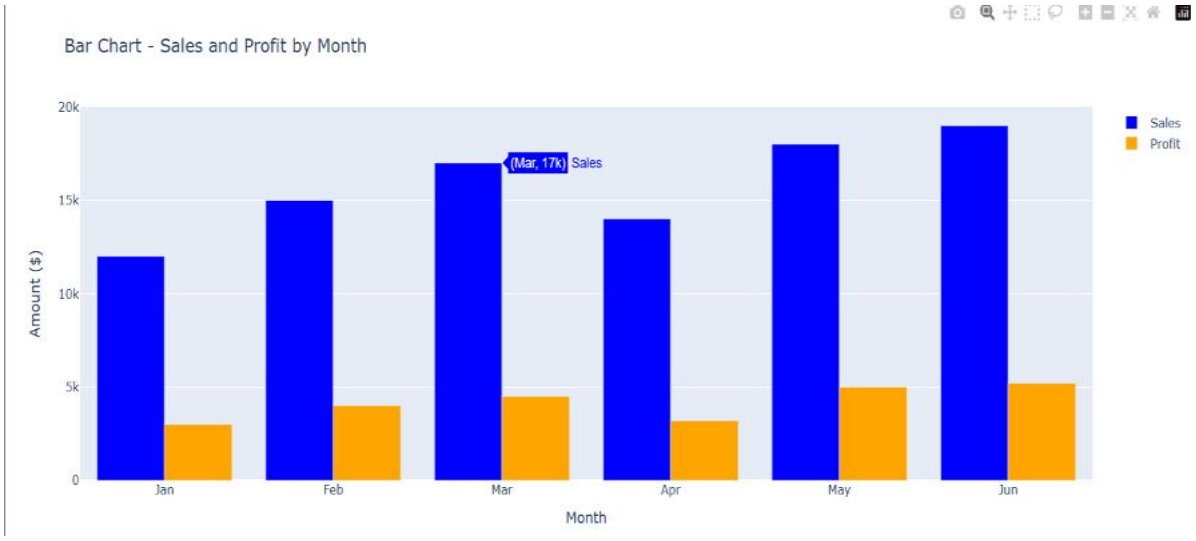
```
import pandas as pd
import plotly.graph_objects as go

# Sample DataFrame data
= {
        'Month': ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'], 'Sales': [12000, 15000, 17000,
        14000, 18000, 19000],
        'Profit': [3000, 4000, 4500, 3200, 5000, 5200],
        'Region': ['East', 'West', 'East', 'North', 'South', 'West']
}

df = pd.DataFrame(data) print("Sample
DataFrame:") print(df)
```

```
# BAR CHART – Sales and Profit by Month fig_bar =
go.Figure() fig_bar.add_trace(go.Bar(
        x=df['Month'],
        y=df['Sales'], name='Sales',
        marker_color='blue'
))
fig_bar.add_trace(go.Bar( x=df['Month'],
        y=df['Profit'], name='Profit',
        marker_color='orange'
))
fig_bar.update_layout(


        title='Bar Chart - Sales and Profit by Month', barmode='group',
        xaxis_title='Month',
        yaxis_title='Amount ($)'
)
fig_bar.show()

# SCATTER PLOT – Sales vs Profit by Region fig_scatter =
go.Figure() fig_scatter.add_trace(go.Scatter(
        x=df['Sales'],
        y=df['Profit'], mode='markers+text',
        text=df['Month'], textposition='top
        center',
        marker=dict(size=df['Sales']/1000, color=df['Sales'], colorscale='Viridis'),
        name='Sales vs Profit'
))
fig_scatter.update_layout(
        title='Scatter Plot - Sales vs Profit by Region', xaxis_title='Sales',
        yaxis_title='Profit'
)
fig_scatter.show()

# HISTOGRAM – Distribution of Sales by Region fig_hist =
go.Figure() fig_hist.add_trace(go.Histogram(
        x=df['Sales'], name='Sales',
        nbinsx=5,
        marker_color='blue',
        opacity=0.75
))
fig_hist.add_trace(go.Histogram(
        x=df['Profit'], name='Profit',
        nbinsx=5, marker_color='orange',
        opacity=0.75
))
fig_hist.update_layout(
```

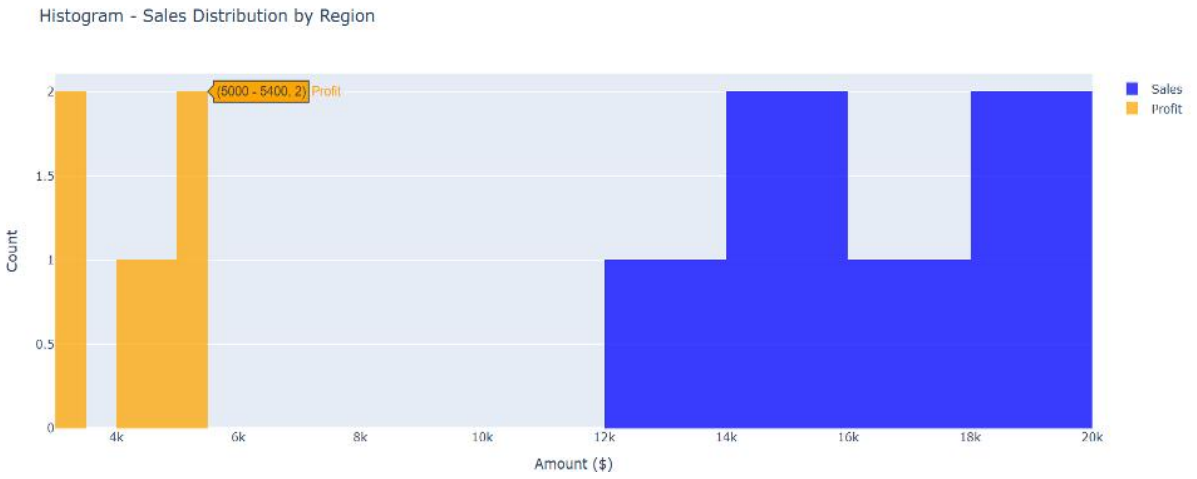
```
        title='Histogram - Sales Distribution by Region', barmode='overlay',
        xaxis_title='Amount ($)',
        yaxis_title='Count'



)
fig_hist.show()

# PIE CHART – Sales Share by Month fig_pie =
go.Figure(go.Pie(
        labels=df['Month'],
        values=df['Sales'], hole=0.3,  #
        Donut chart
        marker=dict(colors=['#636EFA', '#EF553B', '#00CC96', '#AB63FA', '#FFA15A', '#19D3F3'])
))
fig_pie.update_layout(
        title='Pie Chart - Sales Share by Month'
)
fig_pie.show()
```

OUTPUT:

Index	Month	Sales	Profit	Region
0	Jan	12000	3000	East
1	Feb	15000	4000	West
2	Mar	17000	4500	East
3	Apr	14000	3200	North
4	May	18000	5000	South
5	Jun	19000	5200	West



OHM SRI SAIRAM



Unit 5 EDA Case Studies and Web Scraping **11**

Step-by-step EDA on Sample Datasets (Titanic, Iris, Sales, etc.), Feature Engineering Techniques in EDA, EDA Report Generation using Python Notebooks, Preparing Data for Machine Learning Models, Web Scraping: What to Scrape and How, Analyzing and Parsing Web Pages with LXML and XPath, Advanced Web Scraping Using Selenium.

---

1. Step-by-Step Exploratory Data Analysis (EDA) on Sample Datasets

What is Exploratory Data Analysis (EDA)?

Exploratory Data Analysis (EDA) is the process of examining, cleaning, summarizing, and visualizing datasets to discover patterns, detect anomalies, test hypotheses, and prepare data for machine learning.

Definition:

EDA is a systematic approach to analyze datasets using statistical summaries and visualizations before applying machine learning algorithms.

Objectives of EDA

- Understand the structure of the dataset.
- Detect missing values.
- Identify duplicate records.
- Find outliers.
- Understand relationships among variables.
- Select useful features.
- Prepare clean data for machine learning.

EDA Workflow

Collect Dataset

Load Dataset

Understand Dataset

Clean Dataset

Handle Missing Values

Handle Outliers

Feature Engineering

Visualization

Generate Report

Prepare Data for ML

Sample Dataset 1: Titanic Dataset
Description
The Titanic dataset contains passenger information and whether they survived.

PassengerId	Name	Age	Sex	Fare	Survived
1	John	22	Male	7.25	0
2	Anna	38	Female	71.28	1

Step 1: Import Libraries
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Step 2: Load Dataset
```
df = pd.read_csv("titanic.csv")
```

Step 3: View Dataset
df.head()
Displays the first five rows.

Step 4: Dataset Information
df.info()
Shows:
- Number of rows
- Number of columns
- Data types
- Missing values

Step 5: Statistical Summary
df.describe()
Provides:
- Mean
- Median
- Standard deviation
- Minimum
- Maximum
- Quartiles

Step 6: Check Missing Values
df.isnull().sum()

Example Output:

Column	Missing
Age	177
Cabin	687

Step 7: Handle Missing Values

```
df["Age"].fillna(df["Age"].median(), inplace=True)
```

Step 8: Remove Duplicate Records

```
df.drop_duplicates(inplace=True)
```

Step 9: Detect Outliers

```
sns.boxplot(df["Fare"])
```

Step 10: Visualize Survival

```
sns.countplot(x="Survived", data=df)
```

Step 11: Relationship Between Age and Survival

```
sns.barplot(x="Sex", y="Survived", data=df)
```

Step 12: Correlation Matrix

```
sns.heatmap(df.corr(numeric_only=True), annot=True)
```

Step 13: Save Clean Dataset

```
df.to_csv("clean_titanic.csv", index=False)
```

Sample Dataset 2: Iris Dataset

Dataset Description

The Iris dataset contains measurements of flowers.

Sepal Length	Sepal Width	Petal Length	Petal Width	Species
5.1	3.5	1.4	0.2	Setosa

Visualizing Species

```
sns.pairplot(df, hue="Species")
```

Distribution Plot

```
sns.histplot(df["SepalLength"])
```

Correlation

```
sns.heatmap(df.corr(numeric_only=True), annot=True)
```

Sample Dataset 3: Sales Dataset

Dataset

Date	Product	Sales
1-Jan	Laptop	25000

Monthly Sales
df.groupby("Month")["Sales"].sum()

Sales Trend
plt.plot(df["Date"], df["Sales"])

Product-wise Sales
sns.barplot(x="Product", y="Sales", data=df)

2. Feature Engineering Techniques in EDA
What is Feature Engineering?
Feature Engineering is the process of creating, modifying, or selecting variables (features) from raw data to improve the performance of machine learning models.

Importance
- Improves prediction accuracy.
- Reduces noise.
- Simplifies models.
- Helps algorithms learn better patterns.

Common Feature Engineering Techniques
1. Handling Missing Values
Replace missing values with:
- Mean
- Median
- Mode

Example
df["Age"].fillna(df["Age"].mean(), inplace=True)

2. Encoding Categorical Variables
Machine learning models require numeric values.
Label Encoding
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

df["Gender"] = le.fit_transform(df["Gender"])
Example:
Male → 1
Female → 0

One-Hot Encoding
pd.get_dummies(df["Department"])
Example:

CSE	ECE	AI
1	0	0

3. Feature Scaling
Scaling brings features to a similar range.
Standardization

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

df[["Age"]] = scaler.fit_transform(df[["Age"]])
```

Normalization

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

df[["Salary"]] = scaler.fit_transform(df[["Salary"]])
```

4. Creating New Features
Example
Age
↓
Age Group

```
df["AgeGroup"] = df["Age"] // 10
```

5. Binning

```
pd.cut(df["Age"], bins=5)
```

6. Date Feature Extraction

```
df["Year"] = pd.to_datetime(df["Date"]).dt.year
```

7. Removing Unnecessary Features

```
df.drop(["PassengerId"], axis=1)
```

3. EDA Report Generation Using Python Notebooks
What is a Python Notebook?
A Python Notebook (Jupyter Notebook) is an interactive environment where code, text, tables, equations, and visualizations are combined in one document.

Advantages

- Interactive coding
- Visualization support
- Documentation
- Easy sharing
- Reproducible analysis

Structure of an EDA Notebook
Title

Objective

Import Libraries

Load Dataset

Dataset Overview

Data Cleaning

Missing Values

Visualization

Feature Engineering

Insights

Conclusion

Example Notebook Workflow
```
import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

df = pd.read_csv("sales.csv")

df.info()

df.describe()

sns.histplot(df["Sales"])

plt.show()
```

Automatic Report Generation
Libraries
- ydata-profiling (formerly pandas-profiling)
- Sweetviz
- Dataprep

Example
```
from ydata_profiling import ProfileReport
```

```
profile = ProfileReport(df)

profile.to_file("EDA_Report.html")
```

Generated Report Includes

- Missing values
- Correlation
- Outliers
- Distribution
- Data types
- Duplicate records

4. Preparing Data for Machine Learning Models

Why Data Preparation?

Machine learning algorithms require clean, structured, and numerical data.

Steps

Step 1

Remove duplicates

```
df.drop_duplicates(inplace=True)
```

Step 2

Handle missing values

```
df.fillna(df.mean(numeric_only=True), inplace=True)
```

Step 3

Encode categorical variables

```
pd.get_dummies(df)
```

Step 4

Scale numerical data

```
StandardScaler()
```

Step 5

Split Features and Target

```
X = df.drop("Target", axis=1)

y = df["Target"]
```

Step 6

Train-Test Split

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
   X, y,
   test_size=0.2,
   random_state=42
```

)

Final Workflow

Raw Data
  |
Cleaning
  |
Missing Values
  |
Encoding
  |
Scaling
  |
Feature Engineering
  |
Train-Test Split
  |
Machine Learning Model

5. Web Scraping: What to Scrape and How

What is Web Scraping?

Web scraping is the automated process of extracting information from websites using software instead of manually copying data.

Applications

- Product prices
- News articles
- Weather data
- Job listings
- Stock market information
- Sports statistics
- Hotel prices
- Research papers

What Can Be Scraped?

- Text
- Images
- Tables
- Links
- Prices
- Reviews
- Metadata

Basic Web Scraping Process

Website
  |
HTTP Request
  |

HTML Page
  |
Parse HTML
  |
Extract Data
  |
Save Data

Example Using Requests
```
import requests

response = requests.get("https://example.com")

print(response.text)
```

6. Analyzing and Parsing Web Pages with LXML and XPath
What is LXML?
LXML is a high-performance Python library for parsing HTML and XML documents.

Installation
```
pip install lxml
```

What is XPath?
XPath (XML Path Language) is used to locate and extract elements from HTML/XML documents.

Example HTML
```
<html>
<body>

<h1>College</h1>

<p>Department of AI</p>

</body>
</html>
```

Parse HTML
```
from lxml import html

tree = html.fromstring(page_content)
```

Extract Heading
```
tree.xpath("//h1/text()")
```
Output
College

Extract Paragraph
tree.xpath("//p/text()")

Extract Links
tree.xpath("//a/@href")

Advantages of XPath
- Fast
- Accurate
- Supports nested elements
- Easy navigation
- Flexible queries

7. Advanced Web Scraping Using Selenium
Why Selenium?
Some websites load content dynamically using JavaScript. Libraries like requests cannot access this dynamic content because they only fetch the initial HTML.
Selenium automates a real web browser, allowing it to interact with pages like a human user.

Features
- Opens websites
- Clicks buttons
- Fills forms
- Scrolls pages
- Waits for JavaScript
- Takes screenshots
- Downloads dynamic data

Installation
pip install selenium

Basic Example
```
from selenium import webdriver

driver = webdriver.Chrome()

driver.get("https://example.com")

print(driver.title)

driver.quit()
```

Locate an Element
```
from selenium.webdriver.common.by import By

title = driver.find_element(By.TAG_NAME, "h1")
```

```
print(title.text)
```

Click a Button

```
driver.find_element(By.ID, "login").click()
```

Enter Text

```
driver.find_element(By.NAME, "username").send_keys("admin")
```

Wait for Elements

```
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

WebDriverWait(driver, 10).until(
   EC.presence_of_element_located((By.ID, "result"))
)
```

Scroll Down

```
driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
```

Take Screenshot

```
driver.save_screenshot("page.png")
```

Comparison of Web Scraping Libraries

Feature	Requests	BeautifulSoup	LXML	Selenium
Download HTML	Yes	No (Parser)	No (Parser)	Yes
Parse HTML	No	Yes	Yes	Yes
JavaScript Support	No	No	No	Yes
Speed	Very Fast	Fast	Very Fast	Slow
Dynamic Websites	No	No	No	Yes
Best Use	Static pages	HTML parsing	HTML/XML parsing with XPath	Dynamic, interactive websites

Complete Workflow of EDA and Web Scraping