Mastering

Data Analytics with Python for Industry Applications

1st Edition

A structured Python Guide

to data manipulation, visualization, &
industry problem-solving using Python libraries

Soumya Ranjan Mishra

Mastering Data Analytics with Python for Industry Applications

A structured guide to data manipulation, visualization, and industry problem-solving using Python libraries

Soumya Ranjan Mishra

- Head, Learning R&D @ AlmaBetter

About AlmaBetter

AlmaBetter is a **leading technology-driven learning platform** dedicated to **bridging the gap between education and industry demands**. With a strong emphasis on **applied learning and career readiness**, AlmaBetter provides high-quality, structured programs that equip learners with **in-demand technical skills** in fields such as **software development, data science, artificial intelligence, and machine learning**.

Built on the principles of **accessibility, affordability, and excellence**, AlmaBetter ensures that education is not just **theoretical but practical, engaging, and career-focused**. The platform is designed to **empower learners** by offering **industry-relevant curricula, hands-on projects, and mentorship from experienced professionals**.

Our Vision

At AlmaBetter, we believe that **learning should be accessible to everyone, regardless of background or financial constraints**. Our vision is to **transform education** by making it **outcome-oriented, industry-aligned, and technology-driven**. We focus on **skill-building, problem-solving, and job-readiness**, ensuring that every learner is equipped to **thrive in a competitive job market**.

What Makes AlmaBetter Unique?

✅ **Industry-Centric Curriculum** – Our courses are designed in collaboration with **leading industry experts** to ensure **maximum job relevance**.

✅ **Hands-On Learning Approach** – Through **real-world projects, coding exercises, and case studies**, learners gain **practical experience** applicable in professional settings.

✅ **Guaranteed Career Support** – AlmaBetter provides **mentorship, resume building, mock interviews, and placement assistance** to help learners **land high-paying jobs** in top companies.

🚀 **Learn. Build. Grow. Succeed.**

Founder's Message

At **AlmaBetter**, our mission has always been to **empower learners with practical, industry-relevant skills**, ensuring a seamless transition from education to real-world applications. We believe that **learning should be structured, hands-on, and career-focused**, preparing professionals for the ever-evolving demands of the industry.

This book is a reflection of that vision—**a fast-tracked yet comprehensive guide to mastering data analysis with Python**. From **data wrangling and visualization to real-world industry applications**, this book serves as a **go-to resource** for learners at every stage—whether you're a **beginner, a job seeker, or an industry professional looking to refine your expertise**.

- Alok Anand **(Co-Founder)**

Author's Message

Python is at the heart of modern data analysis, and mastering it is crucial for anyone looking to extract meaningful insights from data. This book is designed to **bridge the gap between theory and real-world industry applications**, equipping you with the skills to **clean, manipulate, and visualize data efficiently**.

Whether you're a **beginner building a strong foundation** or a **professional refining your analytical expertise**, this book provides a **structured approach** to mastering Python's powerful libraries. You'll explore **data wrangling, visualization, and real-world case studies**, gaining **hands-on experience with practical industry challenges. Let's unlock the power of data together!**

Happy Learning! 🚀

- Soumya Ranjan Mishra, (Head, Learning R&D)

Preface

In today's data-driven landscape, **Python has become the backbone of data analysis and industry applications**. From **data wrangling and visualization to real-world decision-making**, Python's extensive libraries empower professionals across industries to transform raw data into meaningful insights. It is no longer confined to data scientists—**analysts, business intelligence professionals, engineers, and developers** increasingly rely on Python to clean, manipulate, and extract value from structured and unstructured data.

However, **mastering Python for data analysis requires more than just knowing the syntax**. It demands a **structured, application-oriented approach**, integrating core programming techniques, hands-on industry applications, and real-world problem-solving. This book is designed to provide exactly that—**a comprehensive yet fast-tracked guide to Python's data analysis capabilities, preparing you for both professional roles and technical interviews.**

Whether you are a **beginner taking your first steps in data analytics** or an **industry professional looking to refine your skills**, this book follows a **well-structured two-pronged approach**:

- **Fast Track Module Revision** – A high-impact revision of Python's essential data analysis libraries, including **NumPy, Pandas, and Matplotlib**.
- **Industry Applications & Case Studies** – Practical insights into how Python is applied across **FinTech, HealthTech E-CommerceTech** along with **interview-ready problem-solving scenarios**.

By the end of this book, you will not only have a **solid grasp of Python for data analysis** but also the **confidence to apply these techniques in real-world business scenarios and job interviews**.

Who This Book Is For

This book is designed to cater to **a wide audience**, including students, professionals, and job seekers looking to **master data analysis with Python and apply it in industry settings**. If you fall into any of the categories below, this book is tailored for you:

- **Aspiring Data Professionals & Beginners**

If you're **just starting your journey** in data analytics and Python programming, this book provides a **structured roadmap to quickly grasp Python fundamentals**

without unnecessary complexity. Each concept is explained **clearly and concisely**, ensuring a smooth learning curve.

🔹 **Students & Academic Learners**
For students pursuing degrees in **Computer Science, Data Science, Business Analytics, or Engineering**, this book acts as a **fast-track revision guide**. The **structured approach ensures effective learning, practical application, and strong retention of key Python techniques.**

🔹 **Job Seekers & Interview Candidates**
For those preparing for **technical job interviews**, this book includes **real-world case studies and problem-solving exercises** along with **industry-relevant interview questions**. You will gain **hands-on experience tackling Python-based data analysis tasks commonly asked in top tech firms.**

🔹 **Professionals Transitioning to Python**
If you are a **software developer, business analyst, or IT professional looking to transition into data roles**, this book helps you **get up to speed with Python's data analysis ecosystem** quickly.

🔹 **Industry Experts Seeking Practical Insights**
For professionals working in **Finance, Healthcare, E-Commerce, Manufacturing, and Technology**, this book highlights how **Python is applied to solve business-critical problems**.

What This Book Covers

This book is divided into **two major sections**, ensuring a **progressive learning experience**—from **Python fundamentals to industry-level applications and interview simulations**.

📌 **Part 1: Fast Track Python Revision**
This section serves as a **quick yet structured refresher** for Python's key **data analysis libraries**. Instead of lengthy explanations, it focuses on **concise, practical lessons covering:**

✅ **Python for Data Wrangling:** Using **NumPy and Pandas** for structured data manipulation.
✅ **Data Visualization Essentials:** Creating **meaningful insights** with **Matplotlib and Seaborn**.
✅ **Working with APIs & Exploratory Data Analysis:** Fetching **real-time data**, performing **feature engineering**, and **analyzing structured datasets**.

✅ **Data Structures & Algorithms (Bonus Chapter):** A **high-impact revision** of **lists, arrays, recursion, and search/sort techniques** for better **data processing efficiency**.

Each chapter **follows a structured breakdown**, ensuring **clear concept revision and practical understanding** for learners.

📌 **Part 2: Practical Industry Applications & Interview Simulations**
Python's power lies in its ability to **solve real-world problems** across industries. This section dives into **key domains**, including:

✅ **FinTech** – Analyzing **customer spending behavior, loan repayment trends, and fraud detection models**.
✅ **HealthTech** – Optimizing **patient appointment scheduling, medicine consumption, and hospital occupancy analytics**.

✅ **E-CommerceTech** – Predicting **customer purchase patterns, return rates, and recommendation models**.

For each industry, you will find:

📌 **Industry Overview** – Understanding Python's role in the domain.
📌 **Applied Industry Scenarios** – Hands-on problem-solving case studies.
📌 **Interview Simulations & Takeaways** – How to tackle Python-based data questions in **job interviews**.

By the end of this section, you won't just **know Python for data analysis**—you will understand **how to apply it in the professional world**.

Why This Book?

Unlike traditional Python books that focus solely on **syntax and theoretical concepts**, this book is designed with a **modern, real-world approach**:

💡 **Fast-track learning** – No unnecessary theory; only what matters for real-world application.
💡 **Industry applications** – Learn **how Python is used in practical, business-critical projects**.
💡 **Interview readiness** – Solve **industry-specific case studies and technical interview questions**.

By the end of this book, **you won't just learn Python—you'll master it for real-world data analysis and industry applications.** 🚀

Table of Contents

Part - 1 : Fast Track Power Revision: Numerical Programming in Python

Part 2 - Practical Industry Applications & Interview Simulations

Fast Track Power Revision: Numerical Programming in Python

The **Fast Track Power Revision for Numerical Programming in Python** is a structured and efficient refresher to reinforce key numerical computing concepts. This section does not focus on detailed explanations but provides quick, high-level summaries for rapid review. It is intended for those who have already learned the fundamentals and need a concise yet structured recap before applying these concepts in industry scenarios.

Chapter 1: Data Wrangling Libraries

This chapter briefly revises essential data wrangling techniques using Python's core libraries, **NumPy and Pandas**. It begins with a refresher on **array operations and data manipulation**, highlighting key NumPy functionalities for handling large datasets efficiently. Readers will then revisit **Pandas DataFrames**, covering essential operations such as indexing, slicing, and merging datasets. The chapter also summarises advanced data wrangling techniques, including **grouping, pivoting, and handling missing values**, ensuring a strong foundation in structured data management.

Chapter 2: Data Visualization Libraries

This chapter revisits essential tools for **data visualisation**, focusing on **Matplotlib and Seaborn**. The section summarises fundamental plotting techniques, covering **line plots, bar charts, histograms, and scatter plots**. It then highlights Seaborn's advanced visualisation capabilities, including **categorical plots, pair plots, and heatmaps**. The chapter concludes with best practices for **variable analysis and insightful visual storytelling**, ensuring that data representations are meaningful and effective.

Chapter 3: Data Insights with APIs & EDA

This chapter provides a high-level overview of **data modeling via APIs** and **exploratory data analysis (EDA)**. It starts with a quick refresher on **retrieving data through APIs**, followed by a recap of **interactive visualisations with Plotly**. Readers will then revise **core EDA techniques**, including **feature analysis, summary statistics, and correlation analysis**. The chapter concludes with a

summary of **building Streamlit dashboards**, reinforcing the importance of interactive data presentation for business applications.

Bonus Chapter: Data Structures & Algorithms

The final chapter offers a **brief yet focused revision** of **data structures and algorithms** essential for numerical computing. It begins with a refresher on **arrays, linked lists, stacks, and queues**, followed by **efficient string operations** for text-based data processing. The chapter then revisits **recursion fundamentals**, emphasising problem-solving techniques for optimising computational tasks. Finally, it overviews **algorithmic concepts**, including **sorting, searching, and time complexity analysis**, preparing readers for technical problem-solving scenarios.

This **Fast Track Power Revision** is structured for rapid learning and efficient recall. It is designed for those who seek a **quick yet effective review** of numerical programming concepts before transitioning to industry applications and problem-solving.

Database Overview

The **DVD Rental Database** is a structured dataset that models a **movie rental system**, capturing key business processes such as **film management, customer transactions, rental operations, and store administration**. This database comprises **15 interconnected tables**, each serving a distinct role in managing movie rentals, payments, inventory, and customer interactions.

Link to the Datasets - https://drive.google.com/drive/u/0/folders/1R_5npVxhVmJ4Oq9UXKqOlhtmOuwRjZod

1. film (Movie Details Table)

The **film** table stores key information about each movie, including title, description, release year, and rental details.

Sample Data:

film_id	title	description	release_year	language_id	rental_duration	rental_rate	length	replacement_cost	rating
133	Chamber Italian	A Fateful Reflection of a Moose...	2006	1	7	4.99	117	14.99	NC-17

384	Grosse Wonderful	A Drama of a Cat and an Explorer...	2006	1	5	4.99	49	19.99	R
8	Airport Pollock	A Tale of a Moose and a Girl...	2006	1	6	4.99	54	15.99	R

Column Descriptions:

- **film_id** – Unique identifier for each film.
- **title** – Movie name.
- **description** – Short plot summary.
- **release_year** – Year the film was released.
- **language_id** – Foreign key referencing the **language** table.
- **rental_duration** – Number of days a movie can be rented.
- **rental_rate** – Cost of renting the movie.
- **length** – Duration of the movie in minutes.
- **replacement_cost** – Cost to replace a lost or damaged movie.
- **rating** – Movie rating (G, PG, R, etc.).

2. actor (Actors in Films)

The **actor** table stores the names of all actors appearing in movies.

Sample Data:

actor_id	first_name	last_name	last_update
1	Penelope	Guiness	2013-05-26 14:47:57.62
2	Nick	Wahlberg	2013-05-26 14:47:57.62
3	Ed	Chase	2013-05-26 14:47:57.62

Column Descriptions:

- **actor_id** – Unique identifier for each actor.
- **first_name** – First name of the actor.
- **last_name** – Last name of the actor.
- **last_update** – Timestamp of the last modification.

3. category (Movie Genre Classification)

The **category** table contains various film genres.

Sample Data:

category_id	name	last_update
1	Action	2006-02-15 09:46:27
2	Animation	2006-02-15 09:46:27
3	Children	2006-02-15 09:46:27

Column Descriptions:

- **category_id** – Unique identifier for each genre.
- **name** – Genre name (e.g., Action, Comedy, Drama).
- **last_update** – Timestamp of the last modification.

4. language (Movie Language Information)

The **language** table stores the languages in which movies are available.

Sample Data:

language_id	name	last_update
1	English	2006-02-15 10:02:19
2	Italian	2006-02-15 10:02:19
3	Japanese	2006-02-15 10:02:19

Column Descriptions:

- **language_id** – Unique identifier for each language.
- **name** – Language name (e.g., English, French).
- **last_update** – Timestamp of the last modification.

5. customer (Customer Details)

The **customer** table stores information about people renting movies.

Sample Data:

customer_id	store_id	first_name	last_name	email	address_id	active	create_date	last_update

524	1	Jared	Ely	jared.ely@sakilacustomer.org	530	1	2006-02-14	2013-05-26 14:49:45.738
1	1	Mary	Smith	mary.smith@sakilacustomer.org	5	1	2006-02-14	2013-05-26 14:49:45.738
2	1	Patricia	Johnson	patricia.johnson@sakilacustomer.org	6	1	2006-02-14	2013-05-26 14:49:45.738

Column Descriptions:

- **customer_id** – Unique identifier for each customer.
- **store_id** – Foreign key referencing **store** table.
- **first_name / last_name** – Name of the customer.
- **email** – Contact email.
- **address_id** – Foreign key referencing **address** table.
- **active** – Indicates if the customer is active.
- **create_date** – Date the customer account was created.
- **last_update** – Timestamp of last modification.

6. rental (Movie Rental Transactions)

The **rental** table tracks each movie rental.

Sample Data:

rental_id	rental_date	inventory_id	customer_id	return_date	staff_id	last_update
2	2005-05-24 22:54:33	1525	459	2005-05-28 19:40:33	1	2006-02-16 02:30:53
3	2005-05-24 23:03:39	1711	408	2005-06-01 22:12:39	1	2006-02-16 02:30:53
4	2005-05-24 23:04:41	2452	333	2005-06-03 01:43:41	2	2006-02-16 02:30:53

Column Descriptions:

- **rental_id** – Unique identifier for each rental transaction.
- **rental_date** – Date and time when the movie was rented.
- **inventory_id** – Foreign key referencing **inventory** table.
- **customer_id** – Foreign key referencing **customer** table.
- **return_date** – Date when the movie was returned.
- **staff_id** – Foreign key referencing **staff** table.
- **last_update** – Timestamp of last modification.

7. inventory (Movie Stock Availability)

The **inventory** table tracks the availability of movies in different stores.

Sample Data:

inventory_id	film_id	store_id	last_update
1	1	1	2006-02-15 10:09:17
2	1	1	2006-02-15 10:09:17
3	1	1	2006-02-15 10:09:17

Column Descriptions:

- **inventory_id** – Unique identifier for each movie copy.
- **film_id** – Foreign key referencing **film** table.
- **store_id** – Foreign key referencing **store** table.
- **last_update** – Timestamp of the last modification.

8. payment (Rental Payment Transactions)

The **payment** table stores records of payments made by customers.

Sample Data:

payment_id	customer_id	staff_id	rental_id	amount	payment_date
17503	341	2	1520	7.99	2007-02-15 22:25:46.99
17504	341	1	1778	1.99	2007-02-16 17:23:14.99
17505	341	1	1849	7.99	2007-02-16 22:41:45.99

Column Descriptions:

- **payment_id** – Unique identifier for each payment transaction.
- **customer_id** – Foreign key referencing **customer** table.
- **staff_id** – Foreign key referencing **staff** table.
- **rental_id** – Foreign key referencing **rental** table.
- **amount** – Payment amount for the rental.
- **payment_date** – Date and time of payment.

9. address (Customer & Staff Addresses)

The **address** table contains addresses associated with customers, staff, and stores.

Sample Data:

address_id	address	address2	district	city_id	postal_code	phone	last_update

1	47 MySakila Drive	NaN	Alberta	300	NaN	NaN	2006-02-15 09:45:30
2	28 MySQL Boulevard	NaN	QLD	576	NaN	NaN	2006-02-15 09:45:30
3	23 Workhaven Lane	NaN	Alberta	300	NaN	14033340000	2006-02-15 09:45:30

Column Descriptions:

- **address_id** – Unique identifier for each address.
- **address** – Primary address details.
- **address2** – Additional address details (optional).
- **district** – Region or district of the address.
- **city_id** – Foreign key referencing **city** table.
- **postal_code** – Postal or ZIP code of the address.
- **phone** – Contact phone number.
- **last_update** – Timestamp of the last modification.

10. city (City Details for Addresses)

The **city** table contains city names for addresses.

Sample Data:

city_id	city	country_id	last_update
1	A Corua (La Corua)	87	2006-02-15 09:45:25
2	Abha	82	2006-02-15 09:45:25
3	Abu Dhabi	101	2006-02-15 09:45:25

Column Descriptions:

- **city_id** – Unique identifier for each city.
- **city** – Name of the city.
- **country_id** – Foreign key referencing **country** table.
- **last_update** – Timestamp of the last modification.

11. country (Country Details for Addresses)

The **country** table contains country names linked to cities.

Sample Data:

country_id	country	last_update
1	Afghanistan	2006-02-15 09:44:00
2	Algeria	2006-02-15 09:44:00
3	American Samoa	2006-02-15 09:44:00

Column Descriptions:

- **country_id** – Unique identifier for each country.
- **country** – Name of the country.
- **last_update** – Timestamp of the last modification.

12. staff (Employees Managing the Rental System)

The **staff** table contains employees who manage rental operations.

Sample Data:

staff_id	first_name	last_name	address_id	email	store_id	active	username	password	last_update
1	Mike	Hillyer	3	Mike.Hillyer@sakilastaff.com	1	True	Mike	8cb2237d0679ca88db6464eac60da96345513964	2006-05-16 16:13:11.79
2	Jon	Stephens	4	Jon.Stephens@sakilastaff.com	2	True	Jon	8cb2237d0679ca88db6464eac60da96345513964	2006-05-16 16:13:11.79

Column Descriptions:

- **staff_id** – Unique identifier for each staff member.
- **first_name / last_name** – Employee's name.
- **address_id** – Foreign key referencing **address** table.
- **email** – Contact email.
- **store_id** – Foreign key referencing **store** table.
- **active** – Indicates if the staff member is currently active.
- **username / password** – Login credentials for store systems.
- **last_update** – Timestamp of the last modification.

13. store (Movie Rental Store Details)

The **store** table contains information about different store locations.

Sample Data:

store_id	manager_staff_id	address_id	last_update
1	1	1	2006-02-15 09:57:12
2	2	2	2006-02-15 09:57:12

Column Descriptions:

- **store_id** – Unique identifier for each store.
- **manager_staff_id** – Foreign key referencing **staff** table.
- **address_id** – Foreign key referencing **address** table.
- **last_update** – Timestamp of the last modification.

14. film_actor (Mapping Between Films and Actors)

The **film_actor** table establishes a **many-to-many relationship** between films and actors.

Sample Data:

actor_id	film_id	last_update
1	1	2006-02-15 10:05:03
1	23	2006-02-15 10:05:03

1	25	2006-02-15 10:05:03

Column Descriptions:

- **actor_id** – Foreign key referencing **actor** table.
- **film_id** – Foreign key referencing **film** table.
- **last_update** – Timestamp of the last modification.

15. film_category (Mapping Between Films and Categories)

The **film_category** table links movies to their respective genres.

Sample Data:

film_id	category_id	last_update
1	6	2006-02-15 10:07:09
2	11	2006-02-15 10:07:09
3	6	2006-02-15 10:07:09

Column Descriptions:

- **film_id** – Foreign key referencing **film** table.
- **category_id** – Foreign key referencing **category** table.
- **last_update** – Timestamp of the last modification.

The **DVD Rental Database** is a relational model designed to manage a movie rental business. It comprises **15 interconnected tables** covering **films, actors, customers, rentals, payments, inventory, and stores**. The **film table** stores movie details, while **actors and categories** are linked through relational tables. **Customers rent movies from stores**, with transactions recorded in **rental and payment tables**. **Staff members manage stores**, and addresses are linked through **city and country tables**. This structured schema ensures **efficient data management, inventory tracking, and business insights**. The database will be used consistently throughout **Fast Track Power Revision** for real-world examples.

Chapter 1: Data Wrangling Libraries

The **Data Wrangling Libraries** chapter is a structured and efficient refresher on essential Python libraries used for data preprocessing and manipulation. This section does not introduce new concepts but instead provides a quick review of key functionalities in **NumPy and Pandas**, ensuring that readers can efficiently handle and transform data.

Lesson 1: Getting Started with NumPy & Pandas

This lesson concisely summarises **NumPy**, the core numerical computing library, and **Pandas**, the go-to tool for structured data manipulation. It briefly revisits **NumPy arrays**, their creation, indexing, and key operations, followed by a high-level overview of **Pandas DataFrames and Series**. Readers will also refresh their knowledge of **essential Pandas functions**, including data import/export, selection, and filtering techniques.

Lesson 2: Mastering Data Wrangling

This lesson focuses on **transforming and cleaning datasets** using Pandas. It quickly revisits techniques such as **sorting, renaming columns, handling missing values, and modifying data types**. The lesson also provides a refresher on **applying lambda functions, groupby operations, and aggregations** to manipulate datasets efficiently. Readers will revise techniques to ensure structured and clean data for further analysis.

Lesson 3: Advanced Data Wrangling Concepts

This lesson offers a **high-level summary** of advanced data manipulation techniques. It includes a quick revision of **merging and joining datasets**, reshaping data using **pivot tables and melt functions**, and handling large-scale datasets efficiently. The focus remains on reviewing methods that streamline data transformation for real-world applications.

Lesson 4: Data Wrangling on Different Data Formats

This lesson briefly summa**rises** handling multiple data formats, including **CSV, JSON, Excel, and SQL databases**. It summarises techniques for efficiently reading, writing, and processing structured and semi-structured data. The lesson concludes with a review of best practices for **optimising data operations and performance** when working with diverse datasets.

Lesson 1: Getting Started with NumPy & Pandas

Packages and Modules

In Python, a **package** is a collection of modules, and a **module** is a single file containing Python definitions and statements. These help organize large programs into smaller, reusable components. Python comes with built-in modules, but additional ones like **NumPy and Pandas** need to be installed separately.

A **module** can be imported using:

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

Why Packages and Modules Are Important?

- **Code Reusability**: Avoid redundancy by using pre-built functionalities.
- **Modularity**: Organizes code into meaningful sections.
- **Performance**: Libraries like NumPy are optimized for performance.
- **Simplifies Complex Operations**: Pandas and NumPy provide efficient data manipulation tools.

Getting Started With Pandas & NumPy

NumPy: Python Library for Scientific Computing

NumPy (**Numerical Python**) provides fast operations on large datasets, including:

- **Efficient array operations**
- **Mathematical computations**
- **Linear algebra and statistics**

To install NumPy and Pandas:

```
!pip install numpy pandas
```

How to Create NumPy Arrays of N Dimensions

A NumPy array can be created using the `np.array()` function:

```
import numpy as np
# Creating 1D Array
arr_1d = np.array([1, 2, 3, 4, 5])
# Creating 2D Array
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])

# Creating 3D Array
arr_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(arr_1d)
print(arr_2d)
print(arr_3d)
```

Lists vs. Arrays

Feature	Lists	NumPy Arrays
Storage	Heterogeneous	Homogeneous
Speed	Slower	Faster
Memory	High	Efficient
Operations	Limited	Extensive mathematical operations

NumPy arrays are preferred for numerical computations due to their efficiency and performance.

Important NumPy Methods and Functions

`np.arange()`

Creates evenly spaced values:

```
np.arange(1, 10, 2)
# Output: array([1, 3, 5, 7, 9])
```

numpy.reshape()

Reshapes an array into a new shape without changing its data:

```
arr = np.arange(1, 10)
reshaped_arr = arr.reshape(3, 3)
print(reshaped_arr)
# Output:
# [[1 2 3]
#  [4 5 6]
#  [7 8 9]]
```

flatten()

Converts a multi-dimensional array into a 1D array:

```
flattened_arr = reshaped_arr.flatten()
print(flattened_arr)
# Output: [1 2 3 4 5 6 7 8 9]
```

numpy.transpose()

Swaps rows and columns in a matrix:

```
transposed_arr = reshaped_arr.T
print(transposed_arr)
# Output:
# [[1 4 7]
#  [2 5 8]
#  [3 6 9]]
```

sort()

Sorts an array in ascending order:

```
unsorted_arr = np.array([5, 3, 8, 6, 1])
sorted_arr = np.sort(unsorted_arr)
print(sorted_arr)
# Output: [1 3 5 6 8]
```

NumPy Math Functions

NumPy provides various mathematical functions:

```
arr = np.array([1, 2, 3, 4])

print(np.sum(arr))  # Output: 10
print(np.mean(arr)) # Output: 2.5
print(np.median(arr)) # Output: 2.5
print(np.std(arr)) # Standard deviation
print(np.min(arr)) # Output: 1
print(np.max(arr)) # Output: 4
```

Indexing & Slicing NumPy Arrays

```
arr_2d = np.array([[10, 20, 30], [40, 50, 60]])

# Accessing single element
print(arr_2d[1, 2])
# Output: 60

# Slicing rows and columns
print(arr_2d[:, 1])
# Output: [20 50]

print(arr_2d[0, :])
# Output: [10 20 30]
```

Pandas: Data Wrangling Library

Pandas is a powerful library for **handling structured data**, allowing:

- **Data manipulation**
- **Data analysis**
- **Integration with databases like SQL**

Pandas Series

A **Series** is a **one-dimensional labeled array**:

```
import pandas as pd
```

```
data = pd.Series([10, 20, 30, 40])
print(data)
# Output:
# 0    10
# 1    20
# 2    30
# 3    40
# dtype: int64
```

Pandas DataFrame

A **DataFrame** is a **two-dimensional labeled structure**, similar to a table:

```
df = pd.DataFrame({
    'Customer_ID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35]
})
print(df)
# Output:
#    Customer_ID     Name  Age
# 0            1    Alice   25
# 1            2      Bob   30
# 2            3  Charlie    35
```

Creating DataFrame from Different Data Structures

```
#1. From List
df_list = pd.DataFrame([[1, 'Alice'], [2, 'Bob']],
columns=['ID', 'Name'])

#2. From Dictionary
df_dict = pd.DataFrame({'ID': [1, 2], 'Name': ['Alice', 'Bob']})

#3. From NumPy Array
arr = np.array([[1, 'Alice'], [2, 'Bob']])
df_array = pd.DataFrame(arr, columns=['ID', 'Name'])
```

Summary

This lesson provided a **comprehensive revision** of **NumPy and Pandas**, two essential Python libraries for numerical computing and data manipulation.

- **NumPy** enables efficient numerical operations, supporting **multi-dimensional arrays, reshaping, slicing, mathematical functions, sorting, and matrix operations**.
- **Pandas** simplifies data handling with **Series and DataFrames**, allowing structured data manipulation, integration with databases, and creation from various data sources.
- We covered **key NumPy functions** like `np.arange()`, `reshape()`, `flatten()`, `transpose()`, and `sort()`.
- **Pandas operations** included creating DataFrames from lists, dictionaries, NumPy arrays, and databases.

Lesson 2: Mastering Data Wrangling

Handling Large Datasets with Pandas

Pandas provides powerful tools to work with **large datasets efficiently**. It allows users to **clean, manipulate, and analyze structured data** from various sources like **CSV, Excel, JSON, and SQL databases**.

```
import pandas as pd
df = pd.read_csv("customer.csv")  # Loading customer data from
the DVD Rental database
print(df.head())

# Output:
#    customer_id  store_id first_name last_name
# 0          524         1      Jared       Ely
# 1            1         1       Mary     Smith
# 2            2         1   Patricia   Johnson
# 3            3         1      Linda  Williams
# 4            4         2    Barbara     Jones
```

What is a CSV File?

A **CSV (Comma-Separated Values) file** is a plain text format used to store tabular data. It is **lightweight and widely used** in data exchange.

Accessing Files

```
df_csv = pd.read_csv("customer.csv")
df_excel = pd.read_excel("data.xlsx")
df_json = pd.read_json("data.json")

print(df_csv.head())

# Output:
#    customer_id  store_id first_name last_name
# 0          524         1      Jared       Ely
# 1            1         1       Mary     Smith
```

What is Data Wrangling?

Data Wrangling refers to **cleaning, transforming, and structuring raw data** to make it usable for analysis. It involves handling **missing values, renaming columns, filtering, merging, and reshaping data**.

Basic Inspection of a DataFrame

```
print(df.shape)  # Number of rows and columns
print(df.head())  # First 5 rows
print(df.tail())  # Last 5 rows
print(df.columns)  # Column names
print(df.dtypes)  # Data types of columns
print(df.info())  # Summary of the DataFrame
print(df.describe())  # Summary statistics

# Output:
# (599, 5)  # Shape of DataFrame
# customer_id      int64
# store_id         int64
# first_name      object
# last_name       object
# dtype: object
```

Checking and Removing Duplicates

```
print(df.duplicated().sum())  # Number of duplicate rows

df_no_duplicates = df.drop_duplicates()
print(df_no_duplicates.shape)

# Output:
# 5  # Number of duplicate rows
# (594, 5)  # Shape after removing duplicates
```

Handling Missing Values

```
print(df.isnull().sum())  # Counts missing values per column

# Output:
# customer_id    0
# store_id       0
# first_name     2
# last_name      3
```



```
df_cleaned = df.dropna()
print(df_cleaned.shape)

# Output:
# (594, 5)  # Rows after removing missing values
```

```
df_filled = df.fillna("Unknown")
print(df_filled.head())

# Output:
#    customer_id  store_id first_name last_name
# 0          524         1      Jared       Ely
# 1            1         1       Mary     Smith
# 2            2         1   Patricia   Johnson
# 3            3         1      Linda  Williams
# 4            4         2    Unknown     Jones
```

Fetching Columns Without Using `loc` and `iloc`

```
print(df["first_name"].head())

# Output:
# 0      Jared
# 1      Mary
# 2  Patricia
# 3     Linda
# 4   Barbara
```

For multiple columns:

```
df_subset = df[["first_name", "last_name", "store_id"]]
print(df_subset.head())

# Output:
#   first_name last_name  store_id
# 0     Jared       Ely         1
# 1     Mary      Smith         1
# 2  Patricia   Johnson         1
```

Filtering Using `loc` and `iloc`

Using `loc[]` (Label-Based Indexing)

```
df_loc = df.loc[df["store_id"] == 1, ["customer_id",
"first_name", "last_name"]]
print(df_loc.head())

# Output:
#    customer_id first_name last_name
# 1            1      Mary     Smith
# 2            2  Patricia   Johnson
```

Using `iloc[]` (Index-Based Indexing)

```
df_iloc = df.iloc[:5, [0, 2, 3]]  # Selecting first 5 rows and
specific columns
```

```
print(df_iloc)
# Output:
#    customer_id first_name last_name
# 0          524       Jared       Ely
# 1            1       Mary     Smith
# 2            2  Patricia   Johnson
```

Sorting Data

```
df_sorted = df.sort_values(by="customer_id", ascending=False)
print(df_sorted.head())

# Output:
#    customer_id first_name last_name
# 0          599    Charlie     Brown
# 1          598      Alice    Cooper
```

Renaming, and Replacing

```
df_renamed = df.rename(columns={"first_name":
"Customer_FirstName"})
print(df_renamed.head())

# Output:
#    customer_id  store_id Customer_FirstName last_name
```

```
df_replaced = df.replace({"store_id": {1: "Store_A", 2:
"Store_B"}})
print(df_replaced.head())
# Output:
#    customer_id  store_id first_name last_name
# 0          524   Store_A       Jared       Ely
```

Creating a New Column

```
df["full_name"] = df["first_name"] + " " + df["last_name"]
print(df.head())
```

```
# Output:
#    customer_id  store_id first_name last_name    full_name
# 0          524         1      Jared       Ely    Jared Ely
```

Filtering Data Using Boolean Indexing

```
df_filtered = df[df["store_id"] == 1]
print(df_filtered.shape)

# Output:
# (300, 5)
```

Using `.query()`:

```
df_filtered = df.query("store_id == 1 and first_name == 'Mary'")
print(df_filtered.head())

# Output:
#    customer_id  store_id first_name last_name
# 1            1         1      Mary     Smith
```

Summary

This lesson provided a **quick yet comprehensive revision** of **data wrangling in Pandas**.

- **Read data** from **CSV, Excel, JSON, and SQL databases**.
- **Inspect DataFrames** using `shape`, `head()`, `info()`, and `describe()`.
- **Remove duplicates**, **handle missing values**, and **rename columns**.
- **Sort and filter data** using **Boolean Indexing, `loc[]`, and `iloc[]`**.
- **Create new columns** and **replace column values** efficiently.

Lesson 3: Advanced Data Wrangling Concepts

Apply Function in Pandas

The `.apply()` function in Pandas allows applying a function along **rows or columns** of a DataFrame. It is useful for **data transformation, cleaning, and feature engineering**.

```
import pandas as pd

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

# Applying a function to modify column values
df["first_name_length"] = df["first_name"].apply(len)
print(df[["first_name", "first_name_length"]].head())

# Output:
#   first_name  first_name_length
# 0     Jared                  5
# 1     Mary                   4
# 2  Patricia                  8
# 3     Linda                  5
# 4   Barbara                  7
```

Apply Function with Lambda

Using **lambda functions** within `apply()` allows quick operations on DataFrame columns.

```
df["first_name_upper"] = df["first_name"].apply(lambda x:
x.upper())
print(df[["first_name", "first_name_upper"]].head())

# Output:
#   first_name first_name_upper
# 0     Jared            JARED
# 1     Mary             MARY
# 2  Patricia         PATRICIA
```

```
# 3     Linda            LINDA
# 4   Barbara          BARBARA
```

Using `apply()` with a Separate Function

The `apply()` function can also be used with a **separate function** to apply complex logic to a column. Below is an example where we create a **new column** based on a **condition**.

```
import pandas as pd

# Sample DataFrame
data = {
    "customer_id": [1, 2, 3, 4, 5],
    "store_id": [1, 2, 1, 2, 1],
    "purchase_amount": [500, 1200, 700, 1500, 300]
}

df = pd.DataFrame(data)

# Function to categorize customers based on purchase amount
def categorize_purchase(amount):
    if amount > 1000:
        return "High Value Customer"
    elif amount > 500:
        return "Medium Value Customer"
    else:
        return "Low Value Customer"

# Applying function to create a new column
df["customer_category"] =
df["purchase_amount"].apply(categorize_purchase)

print(df)

# Output:
#    customer_id  store_id  purchase_amount      customer_category
# 0           1         1              500        Low Value Customer
# 1           2         2             1200      High Value Customer
# 2           3         1              700    Medium Value Customer
# 3           4         2             1500      High Value Customer
# 4           5         1              300        Low Value Customer
```

Understanding `axis` in Pandas and NumPy

In both **Pandas and NumPy**, `axis` determines the direction of operations:

- **axis=0 (Column-wise operation - Downwards)**
- **axis=1 (Row-wise operation - Sideways)**

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

print(np.sum(arr, axis=0))  # Column-wise sum
# Output: [5 7 9]

print(np.sum(arr, axis=1))  # Row-wise sum
# Output: [6 15]
```

Normalize JSON Data with `pd.json_normalize()`

The `pd.json_normalize()` function **flattens nested JSON data** into a structured DataFrame.

```
data = [
    {"customer": "John", "details": {"age": 30, "city": "New
York"}},
    {"customer": "Anna", "details": {"age": 25, "city":
"London"}}
]

df_json = pd.json_normalize(data, record_path=None,
meta=["customer"], meta_prefix="info_")
print(df_json)

# Output:
#   info_customer  details.age details.city
# 0        John          30     New York
# 1        Anna          25     London
```

Evaluating Expressions with `eval()`

The `.eval()` method allows executing **expressions on DataFrames efficiently**.

```
df["full_name"] = df.eval("first_name + ' ' + last_name")
print(df[["first_name", "last_name", "full_name"]].head())

# Output:
#   first_name last_name    full_name
# 0      Jared       Ely    Jared Ely
# 1      Mary     Smith    Mary Smith
# 2  Patricia   Johnson  Patricia Johnson
# 3      Linda  Williams  Linda Williams
# 4   Barbara     Jones  Barbara Jones
```

Concatenation in Pandas

Row-wise Concatenation (`axis=0`)

```
df1 = pd.DataFrame({"A": [1, 2], "B": [3, 4]})
df2 = pd.DataFrame({"A": [5, 6], "B": [7, 8]})

df_concat = pd.concat([df1, df2], axis=0)
print(df_concat)

# Output:
#    A  B
# 0  1  3
# 1  2  4
# 0  5  7
# 1  6  8
```

Column-wise Concatenation (`axis=1`)

```
df_concat_col = pd.concat([df1, df2], axis=1)
print(df_concat_col)

# Output:
#    A  B  A  B
# 0  1  3  5  7
# 1  2  4  6  8
```

Merge Operations in Pandas

Inner Join (how="inner")

```
df1 = pd.DataFrame({"customer_id": [1, 2, 3], "first_name":
["Alice", "Bob", "Charlie"]})
df2 = pd.DataFrame({"customer_id": [2, 3, 4], "last_name":
["Smith", "Johnson", "Brown"]})

df_merge = pd.merge(df1, df2, on="customer_id", how="inner")
print(df_merge)

# Output:
#    customer_id first_name last_name
# 0           2        Bob     Smith
# 1           3    Charlie   Johnson
```

Different Types of Joins

Join Type	Meaning
Inner Join	Returns only matching rows in both DataFrames
Left Join	Returns all rows from left DataFrame, matching ones from right
Right Join	Returns all rows from right DataFrame, matching ones from left
Outer Join	Returns all rows from both DataFrames

```
df_outer = pd.merge(df1, df2, on="customer_id", how="outer")
print(df_outer)

# Output:
#    customer_id first_name last_name
# 0           1      Alice       NaN
```

```
# 1           2       Bob     Smith
# 2           3    Charlie  Johnson
# 3           4       NaN     Brown
```

Aggregation Functions in Pandas

Aggregation functions summarize data:

- sum(): Returns the sum of the values.
- mean(): Returns the mean (average) of the values.
- median(): Returns the median of the values.
- min(): Returns the minimum value.
- max(): Returns the maximum value.
- count(): Returns the number of non-null values.

```
df = pd.DataFrame({"store_id": [1, 1, 2, 2], "revenue": [100,
200, 300, 400]})
print(df.groupby("store_id")["revenue"].sum())
# Output:
# store_id
# 1    300
# 2    700
```

Aggregation Operations Using `agg()`

```
df_grouped = df.groupby("store_id").agg({"revenue": ["sum",
"mean"]})
print(df_grouped)

# Output:
#           revenue
#              sum mean
# store_id
# 1            300  150
# 2            700  350
```

Understanding `ignore_index`, `inplace=True`, `ascending=False`, and `reset_index()`

- `ignore_index=True`: Resets the index when sorting or concatenating

- `inplace=True`: Modifies the DataFrame without creating a copy
- `ascending=False`: Sorts in descending order
- `reset_index()`: Resets the index of a DataFrame

```
df_sorted = df.sort_values("revenue", ascending=False,
ignore_index=True)
print(df_sorted)

# Output:
#    store_id  revenue
# 0         2      400
# 1         2      300
# 2         1      200
# 3         1      100
```

Summary

This lesson provided a **structured revision** of **advanced data wrangling concepts** in Pandas, covering:

- `apply()`, `apply()` **with lambda** for column-wise operations
- **Understanding** `axis=0`, `axis=1` **in Pandas and NumPy**
- `pd.json_normalize()` **to handle nested JSON data**
- **Merging and different types of joins (**`inner`, `outer`, `left`, `right`**)**
- **Concatenation of DataFrames (**`axis=0`, `axis=1`**)**
- **Aggregation functions (**`sum()`, `mean()`, `agg()`**)**

Lesson 4: Data Wrangling on Different Data Formats

File Management Libraries in Python

Python provides various libraries to handle files efficiently, including **os, csv, json, excel, and xml**. These libraries enable seamless interaction with different file formats for data manipulation, storage, and retrieval.

OS Library for File Management

The `os` module provides functions to interact with the operating system, such as listing, creating, deleting, and modifying files and directories.

```
import os

# List all files and directories in the current working
directory
print(os.listdir("."))

# Output: ['customer.csv', 'data.xlsx', 'script.py', 'log.json']

# Create a new directory
os.mkdir("new_directory")

# Remove a file
os.remove("old_data.csv")

# Specify the path and name of the new directory
new_dir_path =
"AlmaBetter:/Student/Username/Documents/NewProject"
new_dir_name = "DataFiles"

# Create the new directory
os.makedirs(os.path.join(new_dir_path, new_dir_name))

# Specify the path and name of the new file
new_file_path = os.path.join(new_dir_path, new_dir_name,
"DataFile1.csv")


# Create the new file
with open(new_file_path, 'w') as f:
    f.write("Data,Value\n")
    f.write("A,10\n")
    f.write("B,20\n")
    f.write("C,30\n")


print("New directory and file created successfully! Check out
the 'Files' tab on the left to see what you've created!!")
```

Working with CSV Files

CSV (**Comma-Separated Values**) is a widely used format for storing tabular data. Pandas provides simple methods to read and write CSV files.

```
import pandas as pd

# Reading a CSV file
df = pd.read_csv("customer.csv")
print(df.head())

# Output:
#    customer_id  store_id first_name last_name
# 0          524         1      Jared       Ely
# 1            1         1       Mary     Smith

# Writing to a CSV file
df.to_csv("output.csv", index=False)
```

Working with JSON Files

JSON (**JavaScript Object Notation**) is a lightweight data exchange format commonly used for web applications and APIs.

```
import json

# Writing data to a JSON file
data = {"customer_id": 1, "name": "Alice", "age": 25}
with open("data.json", "w") as file:
    json.dump(data, file, indent=4)

# Reading from a JSON file
with open("data.json", "r") as file:
    json_data = json.load(file)

print(json_data)

# Output:
# {'customer_id': 1, 'name': 'Alice', 'age': 25}
```

Working with Excel Files

Excel files (`.xlsx`) are widely used for data processing. Pandas provides built-in methods to work with Excel files.

```
# Reading an Excel file
df_excel = pd.read_excel("data.xlsx", sheet_name="Sheet1")
print(df_excel.head())

# Writing to an Excel file
df_excel.to_excel("output.xlsx", index=False)
```

Working with XML Files

XML (**Extensible Markup Language**) is used to store hierarchical data structures. Python's `xml.etree.ElementTree` module allows parsing XML files.

```
import xml.etree.ElementTree as ET

# Parsing an XML file
tree = ET.parse("data.xml")
root = tree.getroot()

# Extracting data from XML
for child in root:
    print(child.tag, child.attrib)

# Output:
# customer {'id': '1'}
# name {'value': 'Alice'}
```

Date and Time Handling in Python

Python's `datetime` module provides functions to work with date and time efficiently.

```
from datetime import datetime

# Get the current date and time
current_time = datetime.now()
```

```
print(current_time)

# Output: 2025-02-26 14:30:45.123456
```

Extracting Date Components (Day, Month, Year, Day Name, Month Name)

```
today = datetime.today()
print(today.day, today.month, today.year)
# Output:
# 26 2 2025
# Extracting day name and month name
print(today.strftime("%A"))  # Day Name
print(today.strftime("%B"))  # Month Name

# Output:
# Tuesday
# February
```

Extracting Time Components (Hour, Minute, Second)

```
print(today.hour, today.minute, today.second)

# Output:
# 14 30 45
```

Working with Timedelta

`timedelta` is used for performing date arithmetic operations, such as adding or subtracting days from a date.

```
from datetime import timedelta
# Adding 10 days to the current date
future_date = today + timedelta(days=10)
print(future_date)

# Output: 2025-03-08 14:30:45.123456
# Subtracting 5 days
past_date = today - timedelta(days=5)
print(past_date)

# Output: 2025-02-21 14:30:45.123456
```

Relativedelta: Advanced Date Manipulation

The `relativedelta` module from `dateutil` provides more advanced date manipulations.

```
from dateutil.relativedelta import relativedelta

# Adding 1 year and 3 months
new_date = today + relativedelta(years=1, months=3)
print(new_date)

# Output: 2026-05-26 14:30:45.123456
```

Converting String to Datetime

Sometimes, dates are stored as strings and need to be converted into `datetime` format for calculations.

```
date_str = "2025-02-26"
date_obj = datetime.strptime(date_str, "%Y-%m-%d")
print(date_obj)

# Output: 2025-02-26 00:00:00
```

Converting Datetime to String

Formatting `datetime` objects as strings is useful when displaying date and time information.

```
formatted_date = today.strftime("%d-%m-%Y %H:%M:%S")
print(formatted_date)

# Output: 26-02-2025 14:30:45
```

Additional Revision Pointers

1. **OS Library Functions for File Management**
   - `os.listdir()` – List all files and directories in a given path
   - `os.mkdir("new_folder")` – Create a new directory
   - `os.remove("file.txt")` – Delete a file
2. **Pandas Functions for File Handling**
   - `pd.read_csv()`, `pd.to_csv()` – Read and write CSV files
   - `pd.read_json()`, `pd.to_json()` – Read and write JSON files
   - `pd.read_excel()`, `pd.to_excel()` – Read and write Excel files
3. **Datetime Operations in Python**
   - `timedelta(days=, hours=, minutes=)` – Perform date arithmetic
   - `relativedelta(years=, months=, days=)` – Modify date components
   - `strptime()` – Convert string to datetime
   - `strftime()` – Convert datetime to string

Summary

This lesson provided a **comprehensive revision** of **handling different file formats and date-time operations in Python**.

- **File Management:** Using `os` for listing, creating, and deleting files and directories.
- **Handling Different Formats:** Reading and writing **CSV, JSON, Excel, and XML** files using Pandas.
- **Date and Time Operations:** Extracting **date and time components** such as day, month, year, and time.
- **Date Arithmetic:** Using **timedelta and relativedelta** for adding or subtracting time periods.
- **String-Date Conversions:** Using `strptime()` to convert strings to datetime and `strftime()` for formatting datetime objects.

Chapter 2: Data Visualization Libraries

The **Data Visualization Libraries** chapter provides a structured and efficient refresher on the essential tools used for creating meaningful visual representations of data. This section is designed as a **quick revision guide**, offering concise summaries of fundamental visualization techniques without deep explanations. It is intended for individuals who are already familiar with the concepts and need a structured recap before applying them in industry scenarios.

Lesson 1: Data Visualization with Matplotlib & Seaborn

This lesson revisits **Matplotlib**, Python's foundational library for creating static, animated, and interactive visualisations. It provides a high-level summary of **basic plots**, including **line plots, bar charts, histograms, and scatter plots**. Readers will also review **customisation techniques**, such as adjusting axes, adding labels, and modifying colors for improved readability.

The lesson then transitions to **Seaborn**, a robust statistical visualisation library built on Matplotlib. Key topics covered include **categorical plots, distribution plots, and regression plots**. The lesson highlights how Seaborn simplifies **complex data visualisations** with minimal code and built-in aesthetics.

Lesson 2: Data Visualization Tips & Variable Analysis

This lesson provides a **high-level summary of best practices** for creating compelling data visualisations. It includes a quick review of **choosing the right chart types**, ensuring clarity, avoiding misleading representations, and enhancing interpretability.

Readers will also revisit **variable analysis techniques**, focusing on visualising **numerical and categorical variables** effectively. The lesson briefly covers **pair plots, box plots,, and heatmaps**, reinforcing how they provide insights into data distributions, correlations, and trends.

This **chapter is a fast-track reference guide**, ensuring a structured revision of essential data visualisation techniques. It is ideal for individuals looking to **quickly refresh key concepts** before applying them in data analysis and industry projects.

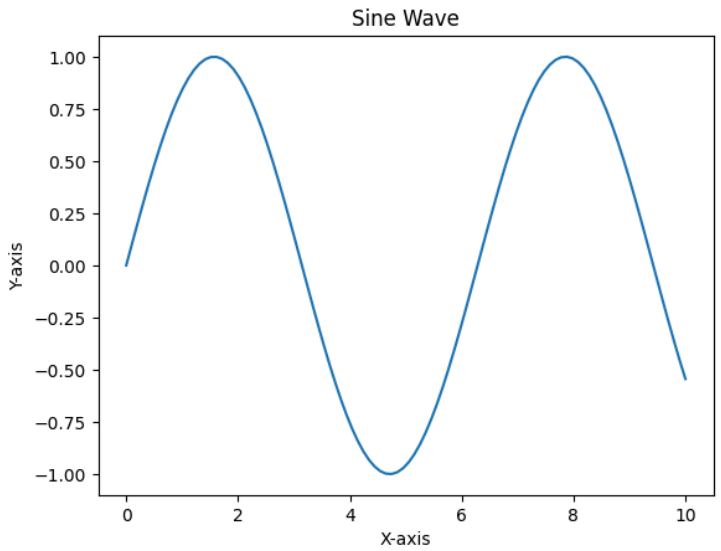
Lesson 1: Data Visualization with Matplotlib & Seaborn

Basics of Matplotlib

Matplotlib is Python's core data visualization library that enables the creation of static, animated, and interactive plots. It is widely used for producing **line plots, scatter plots, bar plots, histograms, and more**.

To use Matplotlib, import it as follows:

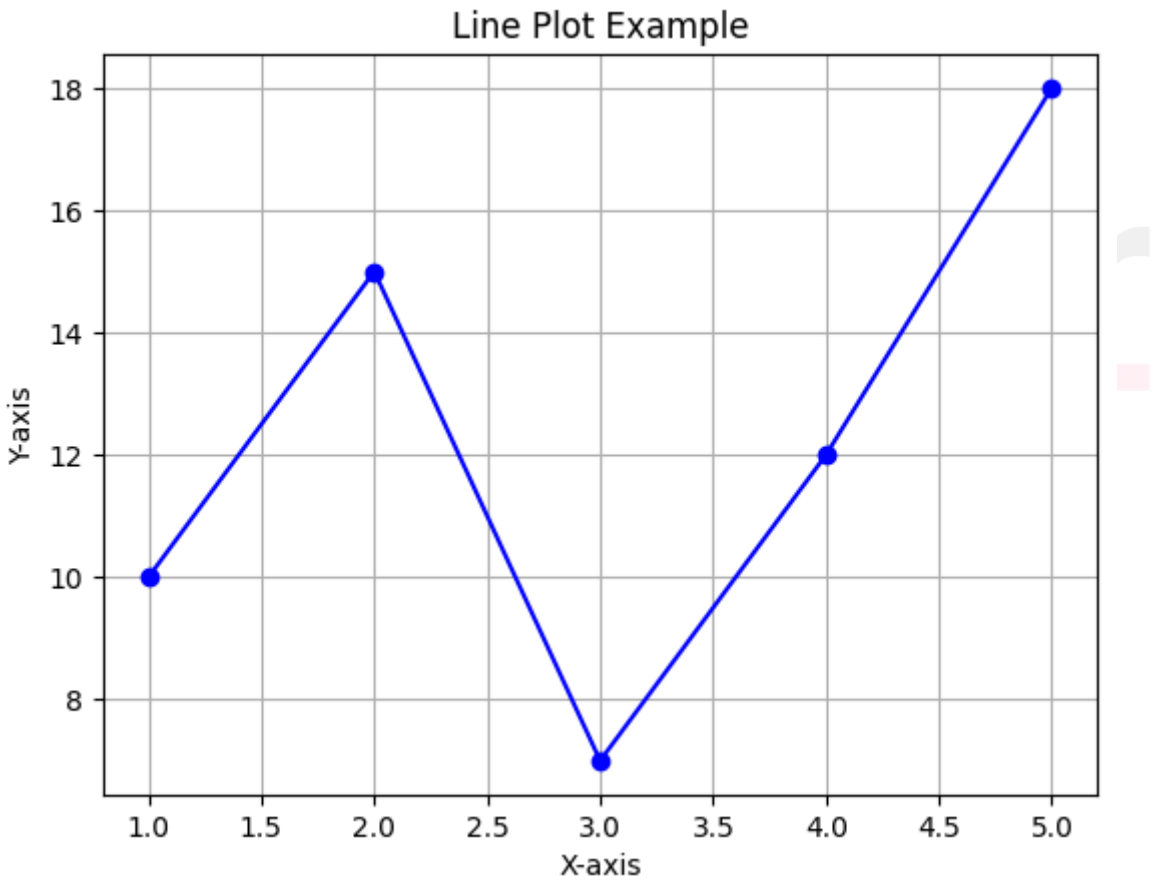
```
import matplotlib.pyplot as plt
import numpy as np

# Sample Data
x = np.linspace(0, 10, 100)
y = np.sin(x)

# Basic Line Plot
plt.plot(x, y)
plt.title("Sine Wave")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```



Line Plot

Line plots are used to visualize trends and relationships between variables.

```
x = [1, 2, 3, 4, 5]
y = [10, 15, 7, 12, 18]

plt.plot(x, y, marker='o', linestyle='-', color='b')
plt.title("Line Plot Example")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.grid()
plt.show()
```

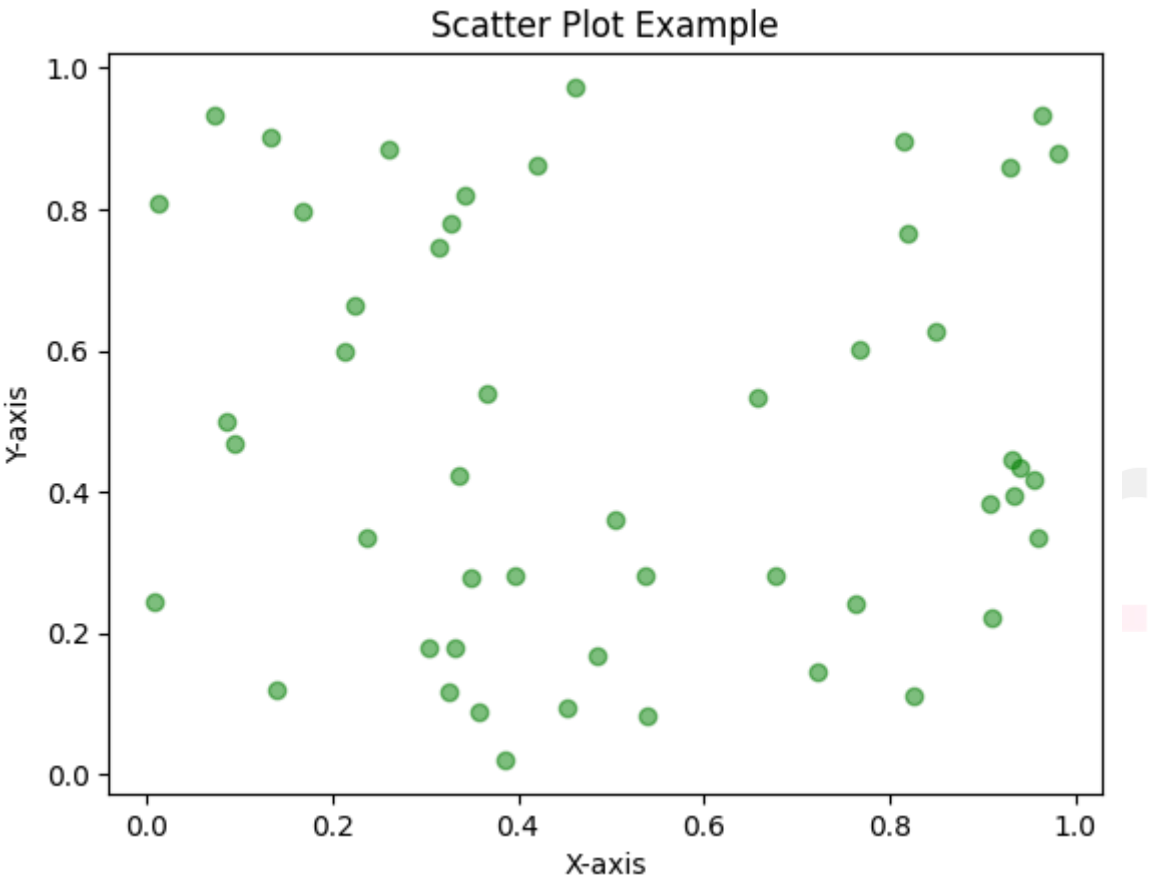


Scatter Plot

Scatter plots are used to visualize relationships between two continuous variables.

```
x = np.random.rand(50)
```

```
y = np.random.rand(50)

plt.scatter(x, y, color='g', alpha=0.5)
plt.title("Scatter Plot Example")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```

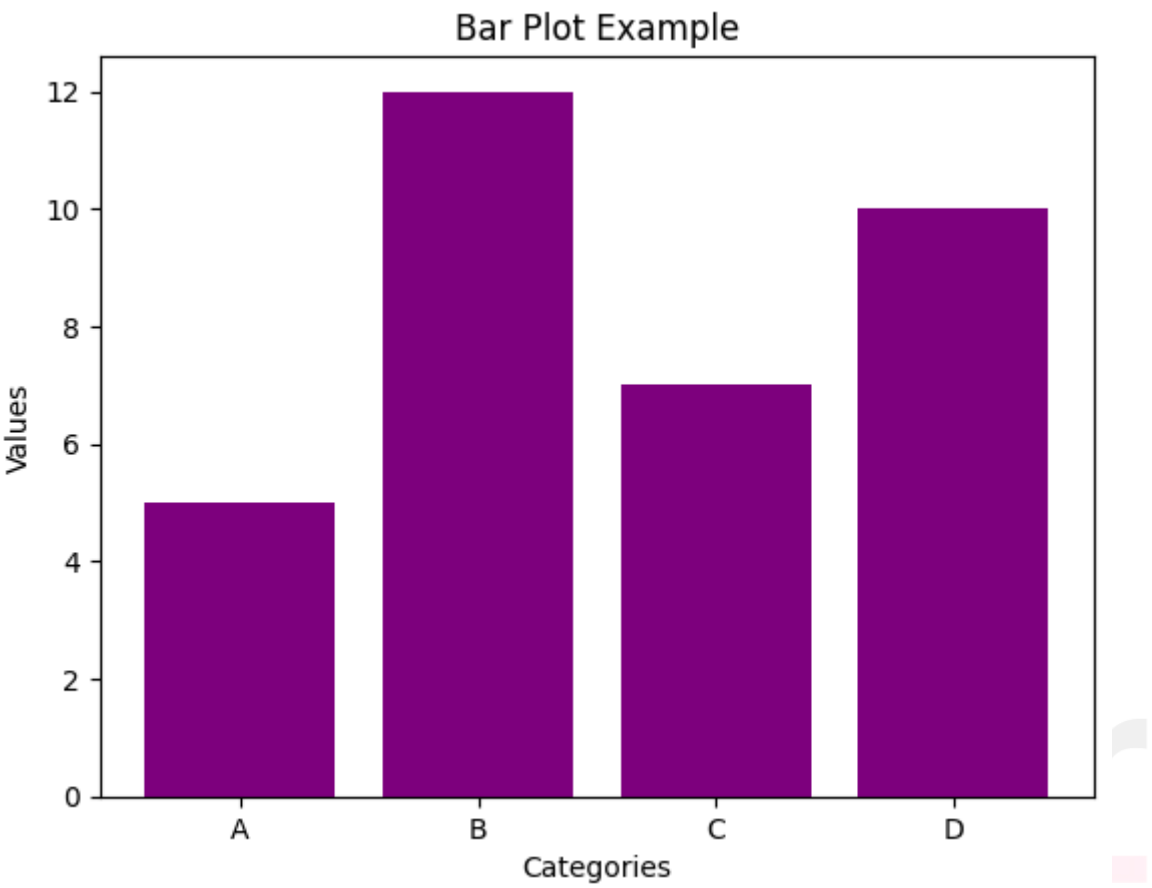


Bar Plot

Bar plots are used to represent categorical data with rectangular bars.

```
categories = ['A', 'B', 'C', 'D']
values = [5, 12, 7, 10]

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

```
plt.show()
```

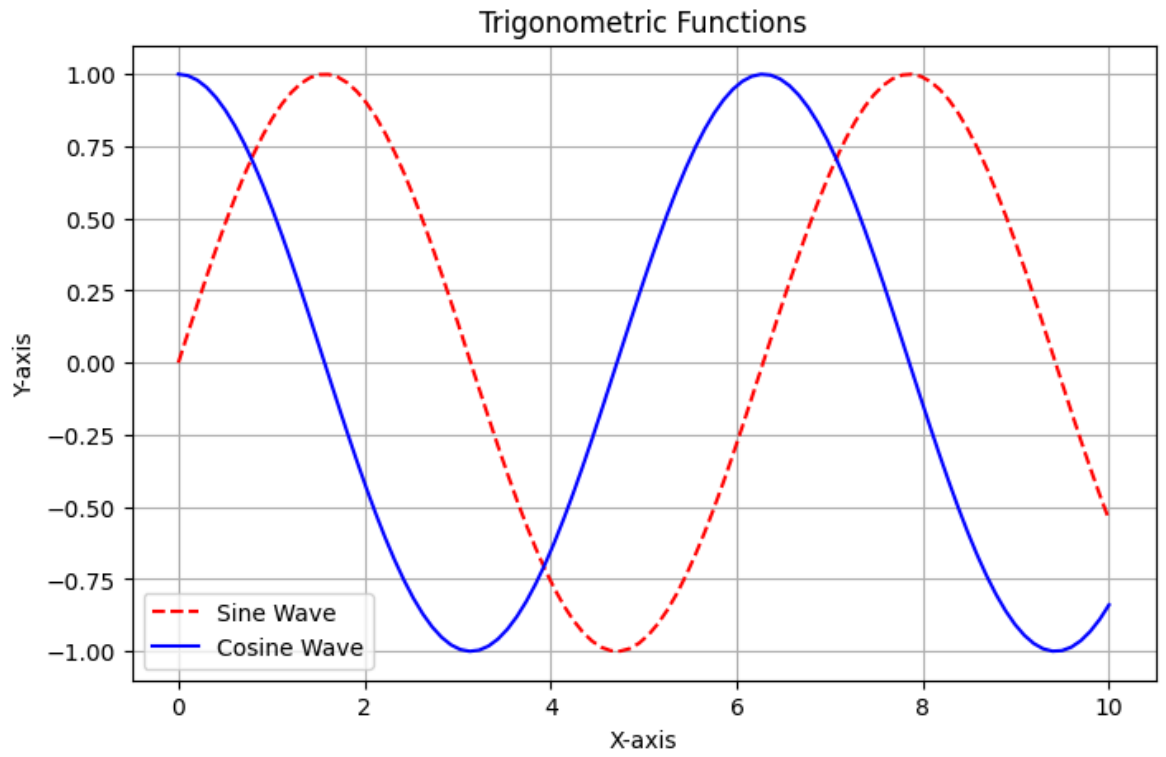


Customizing Matplotlib Plots

Matplotlib allows **customization** of plots, including labels, titles, styles, and legends.

```
plt.figure(figsize=(8, 5))

x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

plt.plot(x, y1, label='Sine Wave', color='r', linestyle='--')
plt.plot(x, y2, label='Cosine Wave', color='b', linestyle='-')

plt.title("Trigonometric Functions")
plt.xlabel("X-axis")
```

```
plt.ylabel("Y-axis")
plt.legend()
plt.grid(True)
plt.show()
```



Introduction to Seaborn and Its Advantages Over Matplotlib

Seaborn is a statistical data visualization library built on top of Matplotlib. It provides better aesthetics, built-in themes, and high-level API functions for complex visualizations.

Advantages of Seaborn Over Matplotlib

- **Simpler syntax** for statistical plots
- **Better default styling** for clean, attractive visuals
- **Built-in support for complex visualizations** like pair plots, heatmaps, and categorical plots
- **Automatic handling of DataFrames**

To use Seaborn:

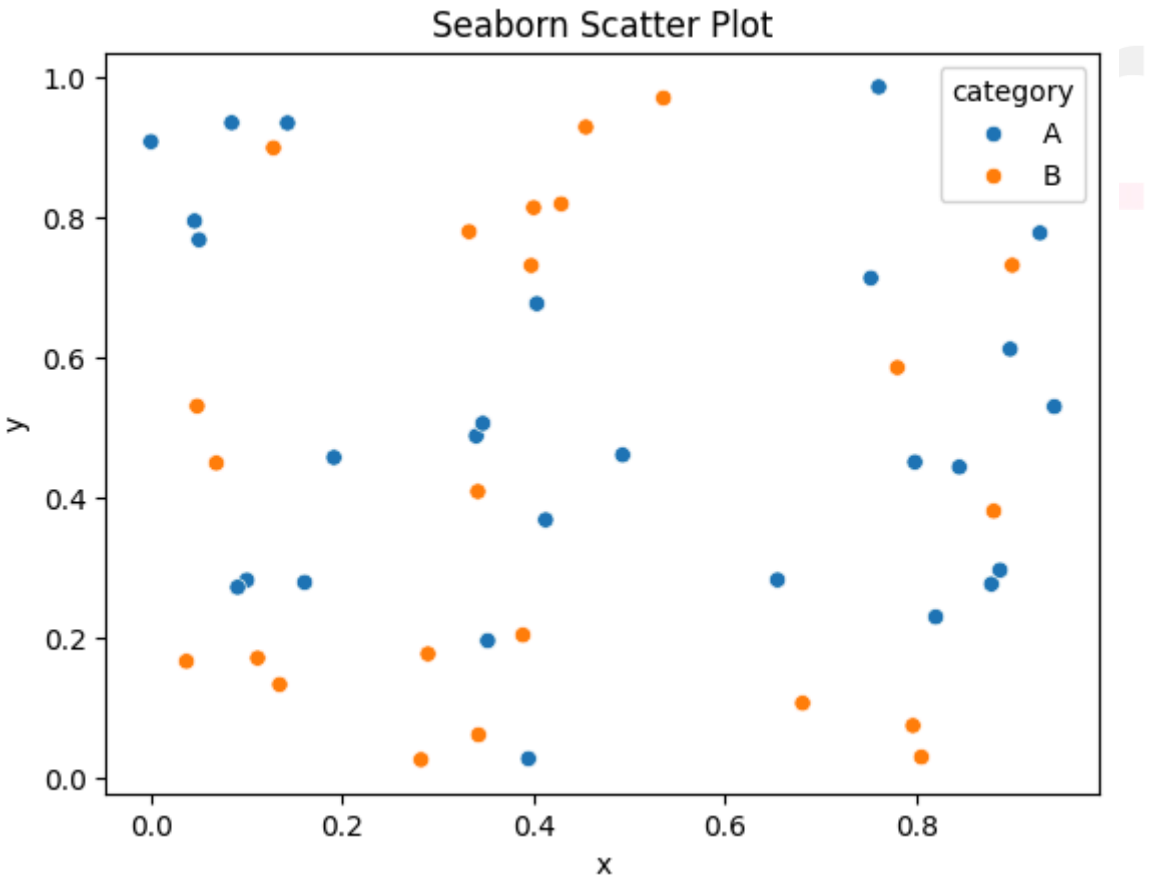
```
import seaborn as sns
import pandas as pd
```

Plotting with Seaborn

Scatter Plot in Seaborn

Seaborn provides enhanced scatter plots using `sns.scatterplot()`.

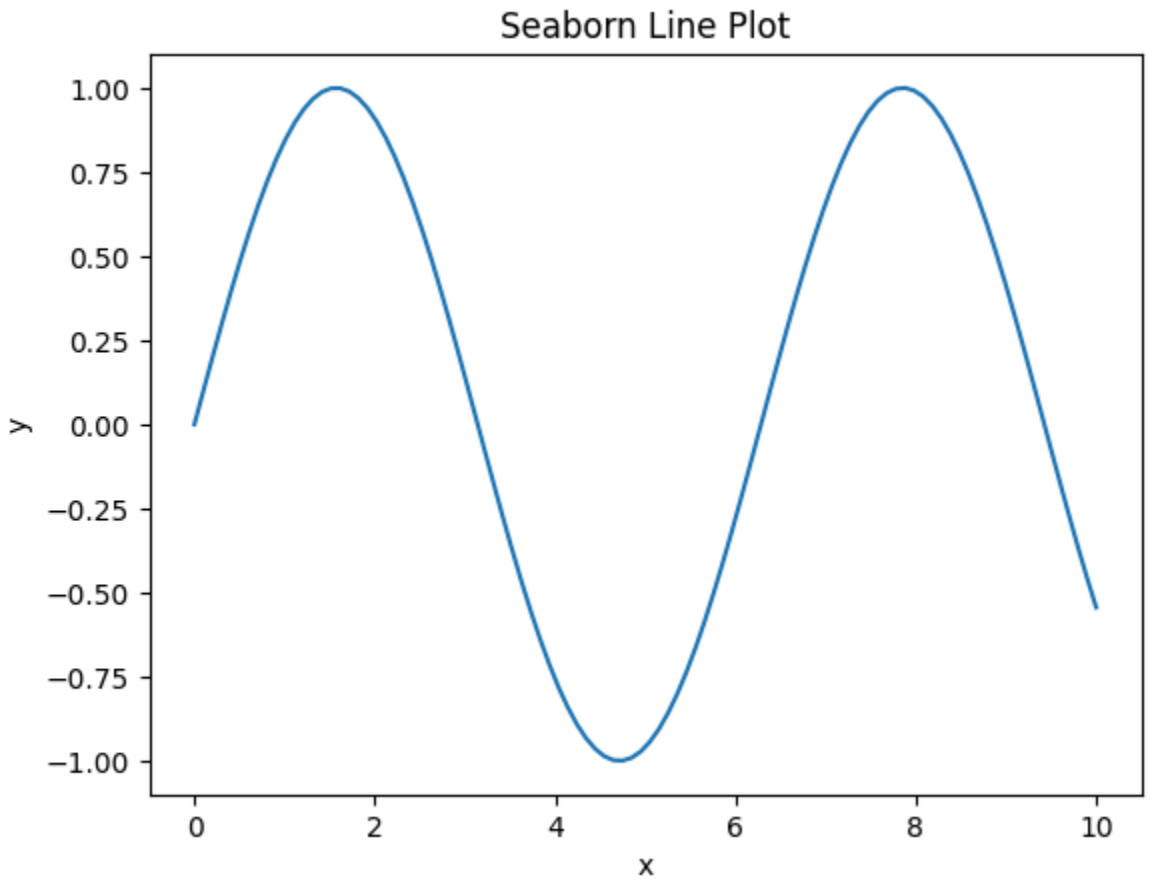
```
df = pd.DataFrame({'x': np.random.rand(50), 'y':
np.random.rand(50), 'category': np.random.choice(['A', 'B'],
50)})

sns.scatterplot(x="x", y="y", hue="category", data=df)
plt.title("Seaborn Scatter Plot")
plt.show()
```



Line Plot in Seaborn

Seaborn's `sns.lineplot()` provides improved aesthetics for line plots.

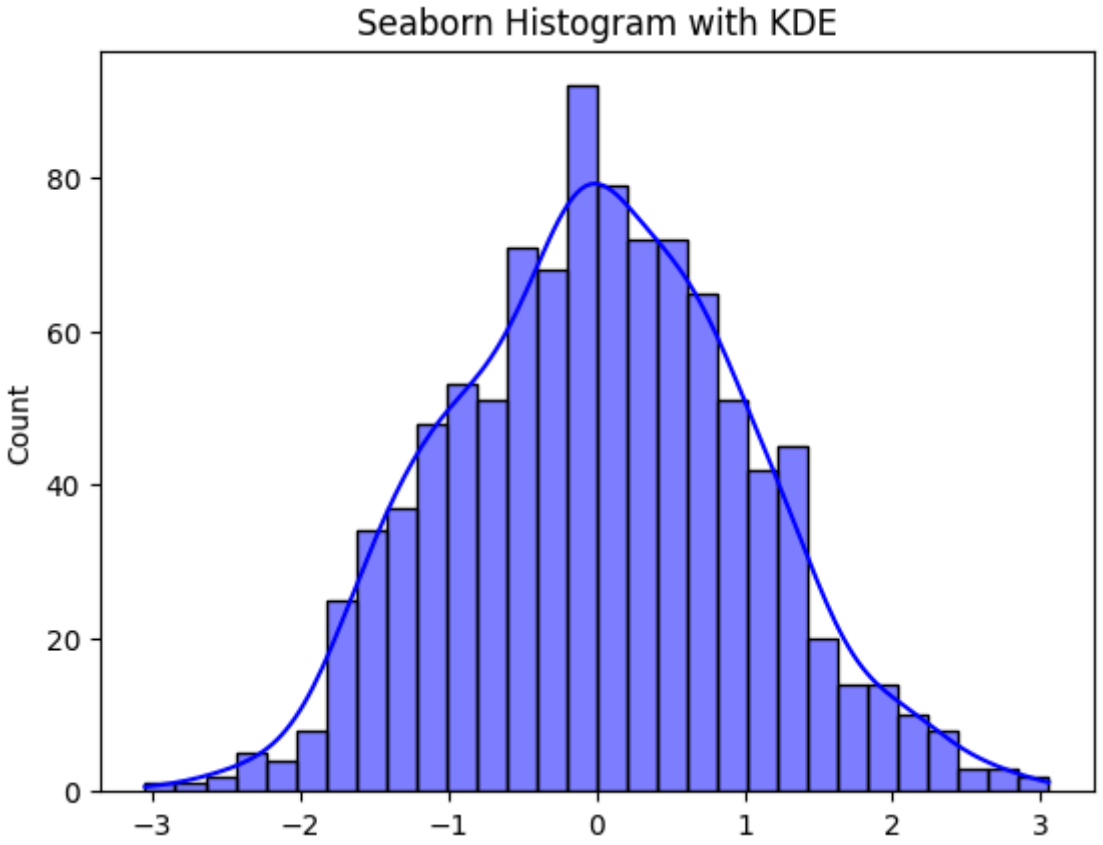
```
df = pd.DataFrame({'x': np.linspace(0, 10, 100), 'y':
np.sin(np.linspace(0, 10, 100))})

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



Histogram in Seaborn

Histograms show the distribution of numerical data.

```
data = np.random.randn(1000)

sns.histplot(data, bins=30, kde=True, color='blue')
plt.title("Seaborn Histogram with KDE")
plt.show()
```

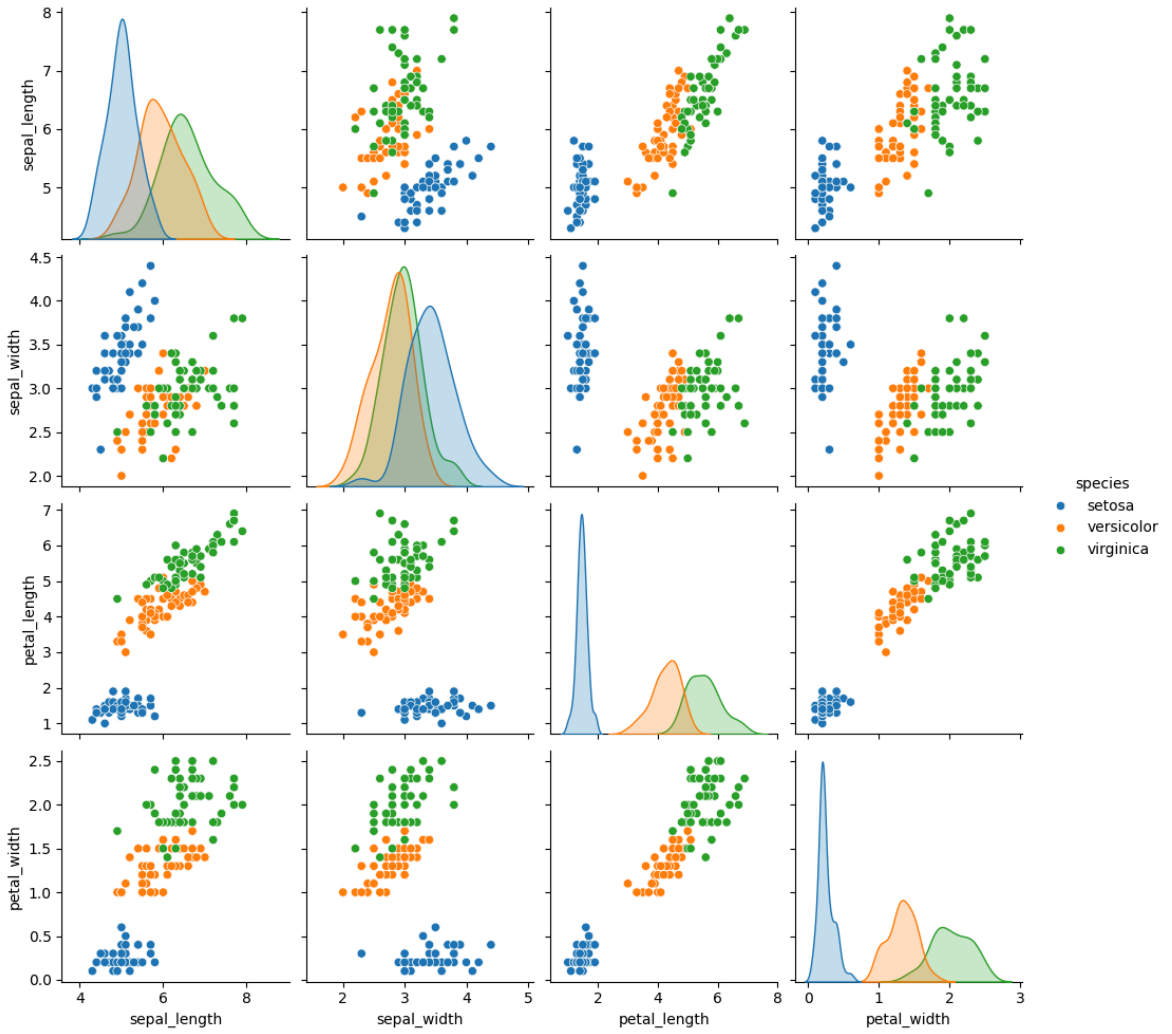


Advanced Seaborn Techniques

Pair Plot

Pair plots visualize relationships between numerical variables.

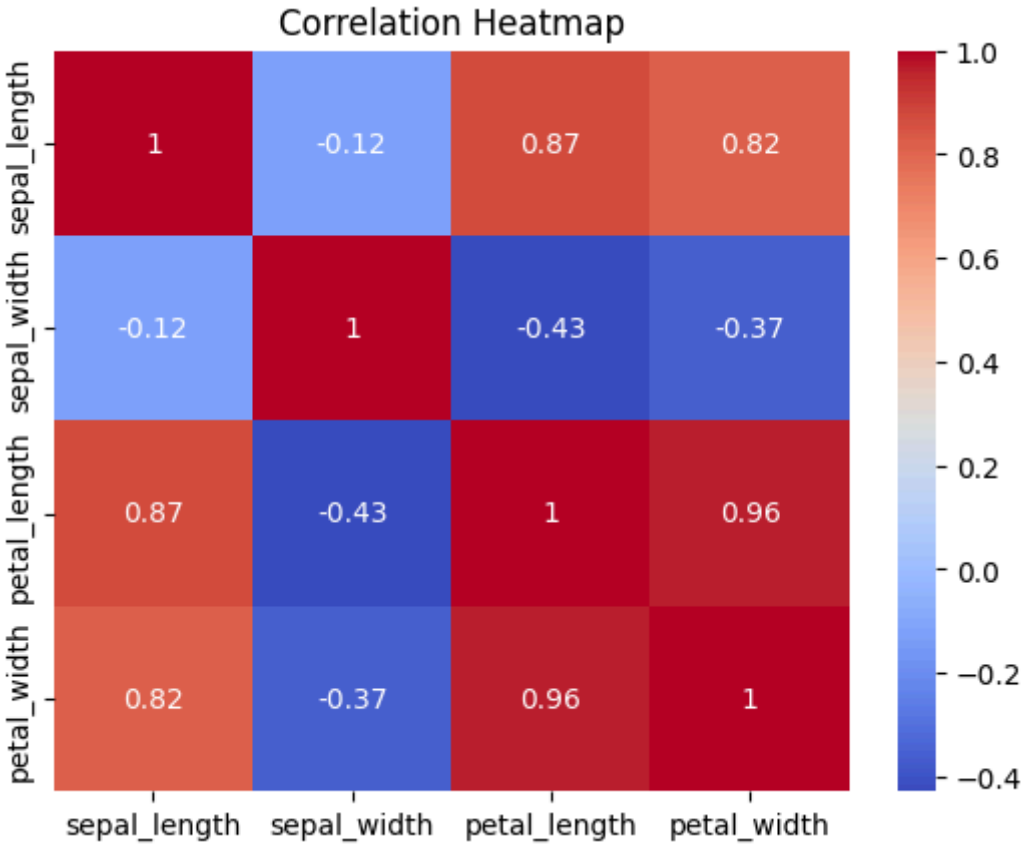
```
df = sns.load_dataset("iris")

sns.pairplot(df, hue="species")
plt.show()
```



Heatmap

Heatmaps display the correlation between numerical variables.

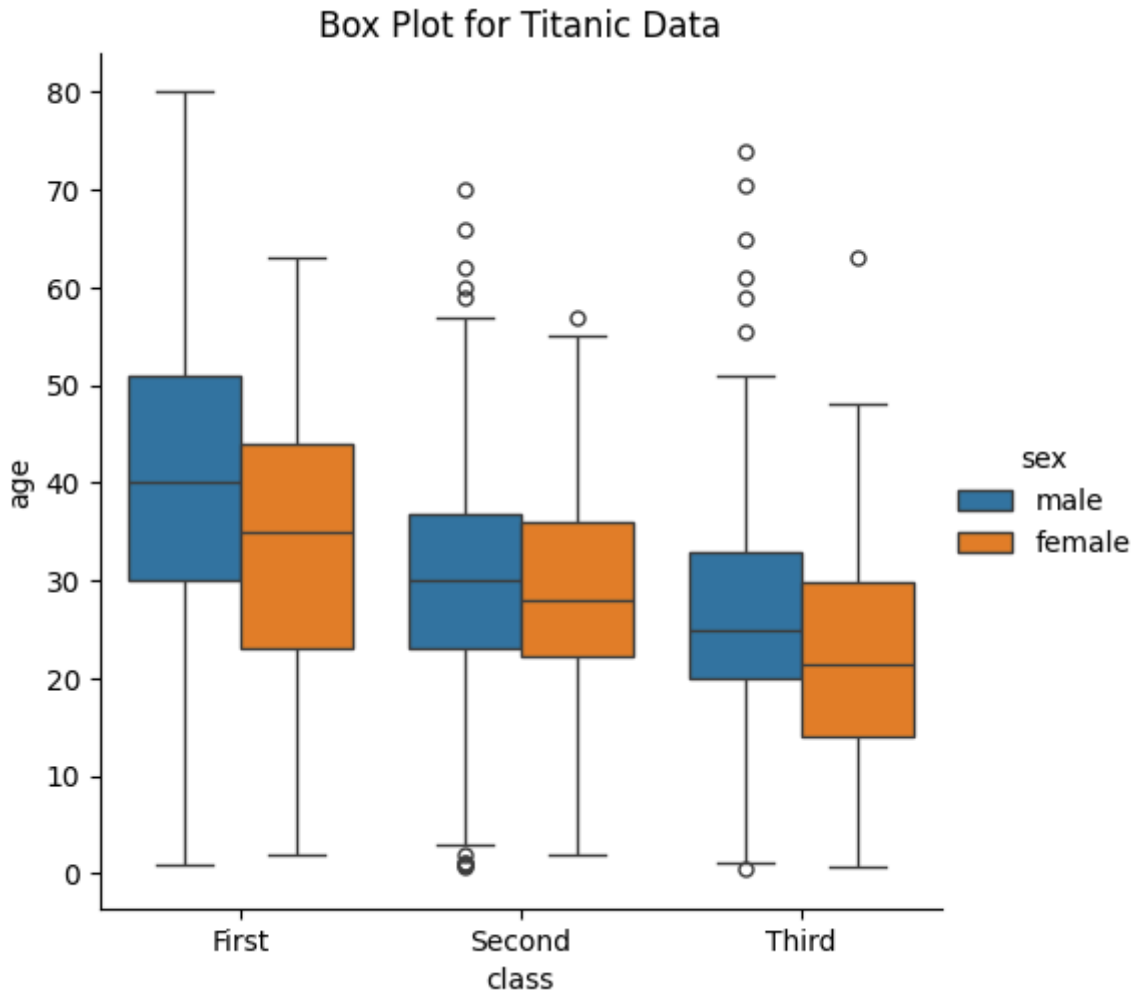
```
df_corr = df.iloc[:,:-1].corr()
sns.heatmap(df_corr, annot=True, cmap='coolwarm')
plt.title("Correlation Heatmap")
plt.show()
```



Categorical Plot

Categorical plots are used for comparing categories within a dataset.

```
df = sns.load_dataset("titanic")

sns.catplot(x="class", y="age", hue="sex", kind="box", data=df)
plt.title("Box Plot for Titanic Data")
plt.show()
```



Types of Charts and Graphs in Data Visualization

Data visualization is essential for **understanding trends, distributions, relationships, and comparisons** in datasets. Various types of charts and graphs serve different analytical purposes. Below is a structured revision of **common chart types**, their **definitions**, and **best use cases**.

1. Line Plot

Definition:

A **Line Plot** represents data points connected by a continuous line. It is primarily used to show **trends over time**.

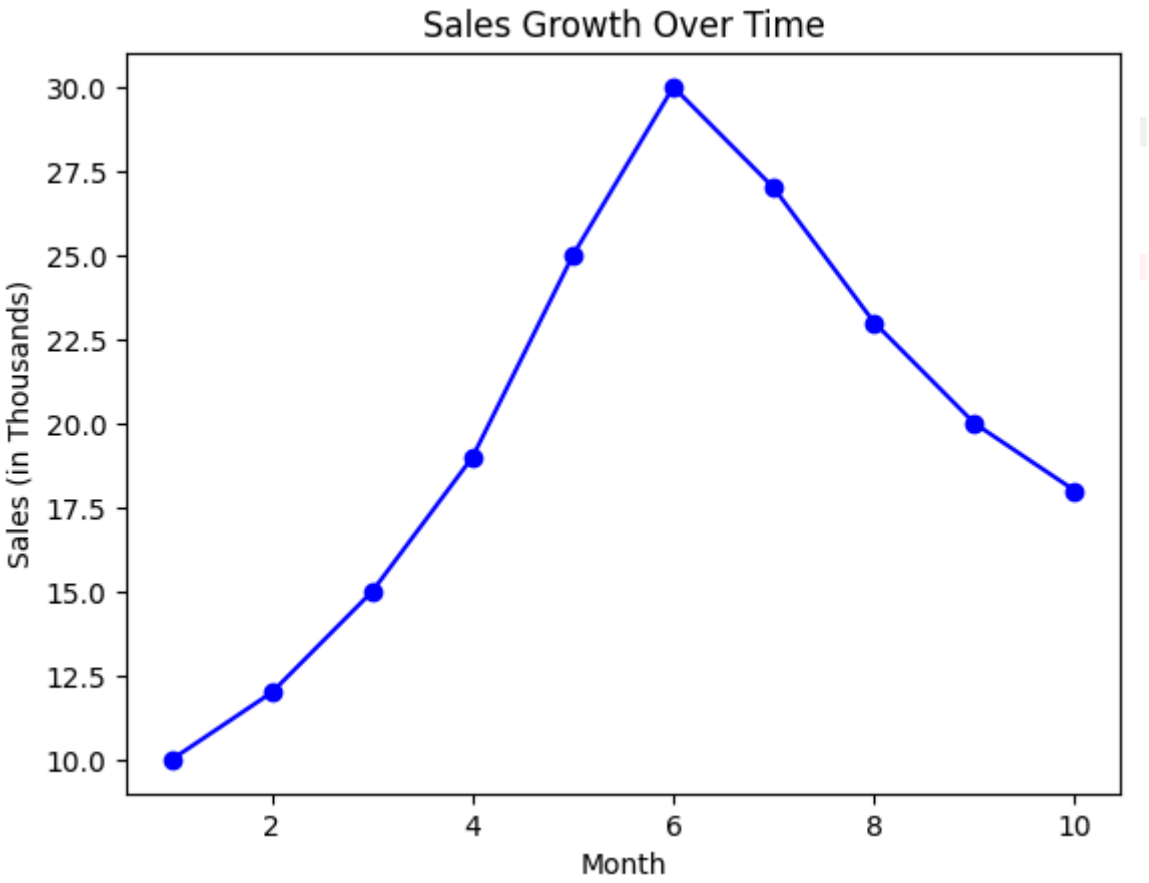
Where to Use:

- Analyzing **time-series data** (e.g., stock prices, temperature variations)

- **Comparing trends** across multiple variables
- **Identifying seasonal patterns**

Example:

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

x = np.arange(1, 11)
y = np.array([10, 12, 15, 19, 25, 30, 27, 23, 20, 18])

plt.plot(x, y, marker='o', linestyle='-', color='b')
plt.title("Sales Growth Over Time")
plt.xlabel("Month")
plt.ylabel("Sales (in Thousands)")
plt.show()
```



2. Scatter Plot

Definition:

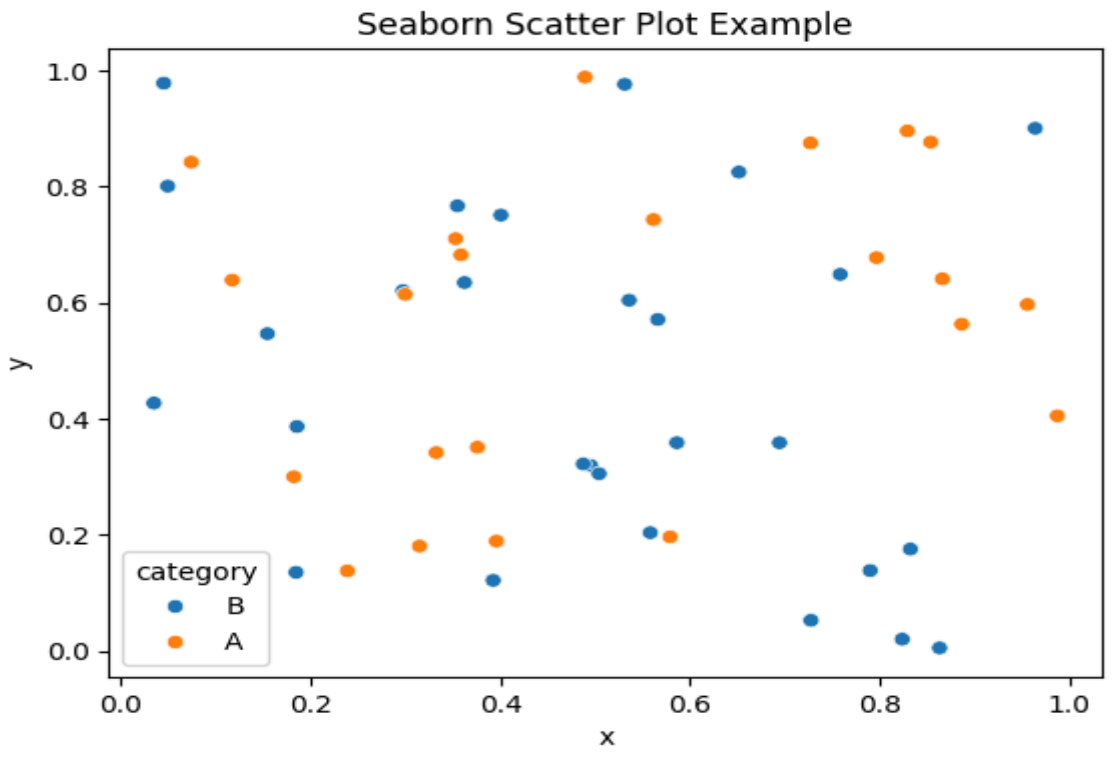
A **Scatter Plot** is a graph that displays **individual data points** based on two variables. It helps visualize **relationships and correlations** between variables.

Where to Use:

- Understanding **correlations** (e.g., height vs weight, study hours vs exam scores)
- Identifying **outliers** in datasets
- Observing **clusters and patterns** in data

Example:

```
import seaborn as sns
import pandas as pd
import numpy as np

df = pd.DataFrame({'x': np.random.rand(50), 'y':
np.random.rand(50), 'category': np.random.choice(['A', 'B'],
50)})

sns.scatterplot(x="x", y="y", hue="category", data=df)
plt.title("Seaborn Scatter Plot Example")
plt.show()
```



3. Bar Chart

Definition:

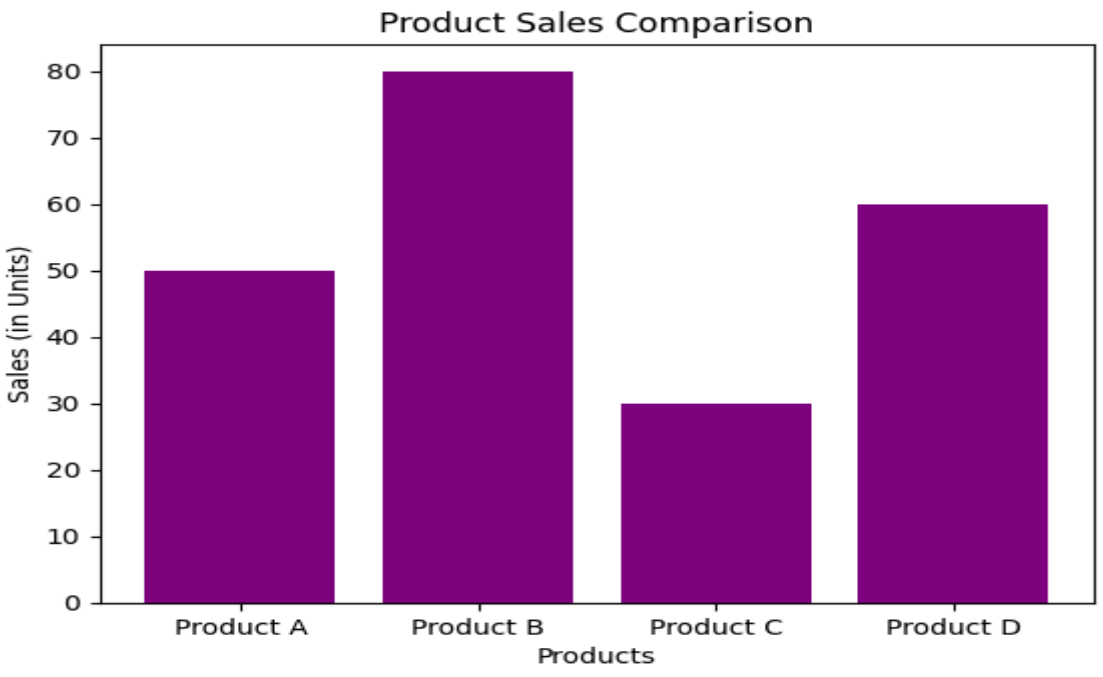
A **Bar Chart** uses rectangular bars to represent **categorical data** with their respective values.

Where to Use:

- **Comparing categories** (e.g., sales performance of different products)
- **Visualizing frequency distributions**
- **Displaying ranking-based comparisons**

Example:

```
categories = ['Product A', 'Product B', 'Product C', 'Product
D']
values = [50, 80, 30, 60]

plt.bar(categories, values, color='purple')
plt.title("Product Sales Comparison")
plt.xlabel("Products")
plt.ylabel("Sales (in Units)")
plt.show()
```



4. Histogram

Definition:

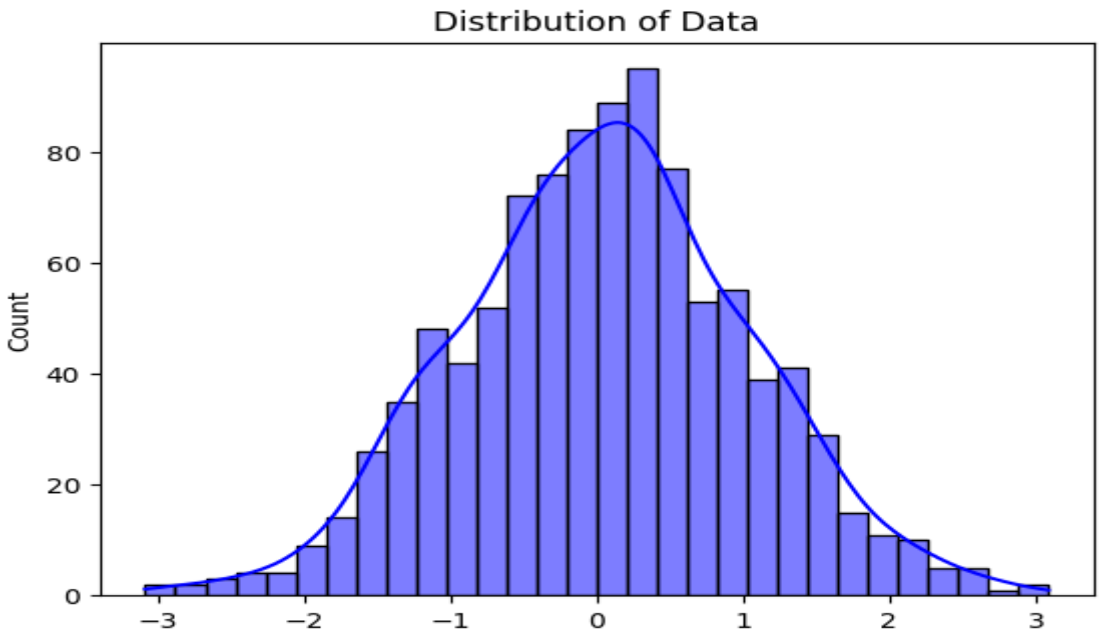
A **Histogram** is a graphical representation of the **distribution of numerical data** using bins. It shows **frequency counts** for different ranges.

Where to Use:

- Understanding **data distribution**
- Identifying **skewness and kurtosis**
- Finding **outliers in large datasets**

Example:

```
import numpy as np
import seaborn as sns

data = np.random.randn(1000)

sns.histplot(data, bins=30, kde=True, color='blue')
plt.title("Distribution of Data")
plt.show()
```



5. Box Plot (Whisker Plot)

Definition:

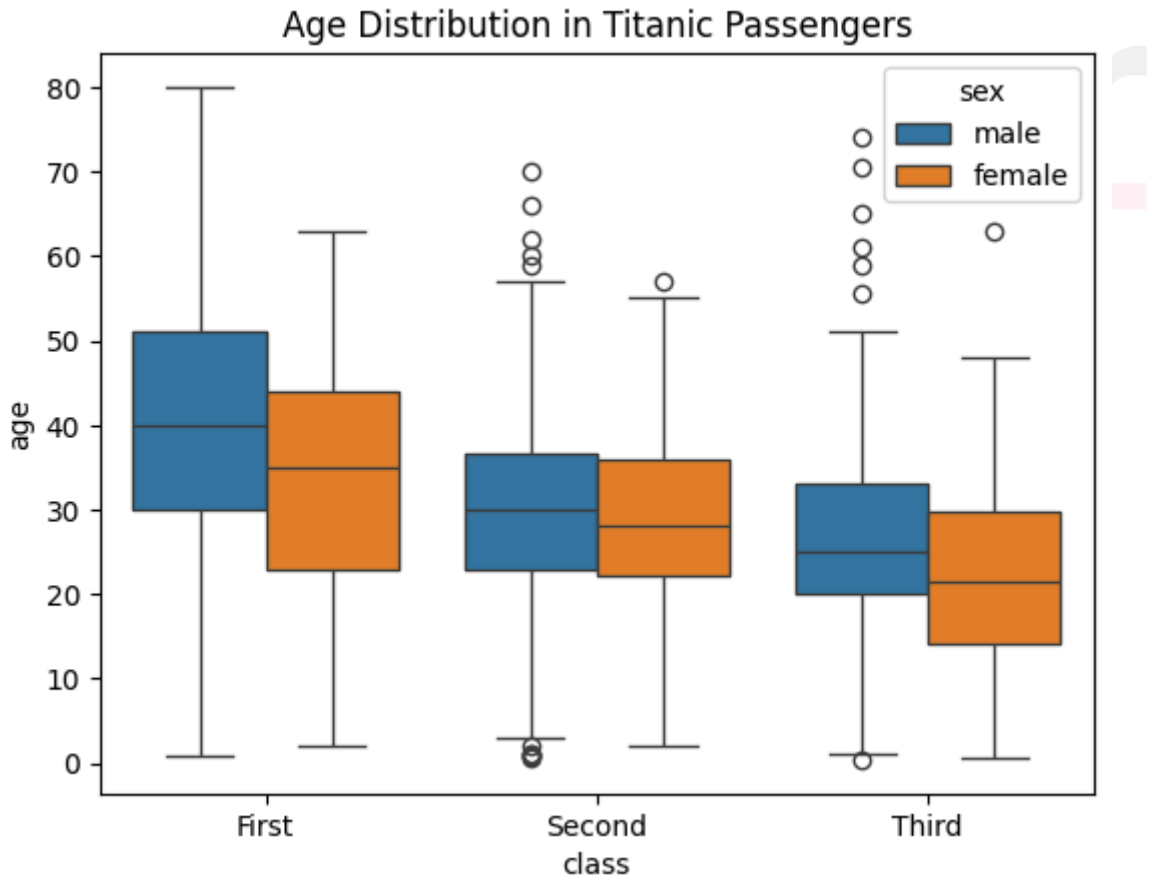
A **Box Plot** displays the **distribution of a dataset** through five key summary statistics: **minimum, first quartile, median, third quartile, and maximum**.

Where to Use:

- Identifying **outliers** in data
- Comparing **statistical distributions** across categories
- Understanding **data spread and skewness**

Example:

```
df = sns.load_dataset("titanic")

sns.boxplot(x="class", y="age", hue="sex", data=df)
plt.title("Age Distribution in Titanic Passengers")
plt.show()
```



6. Heatmap

Definition:

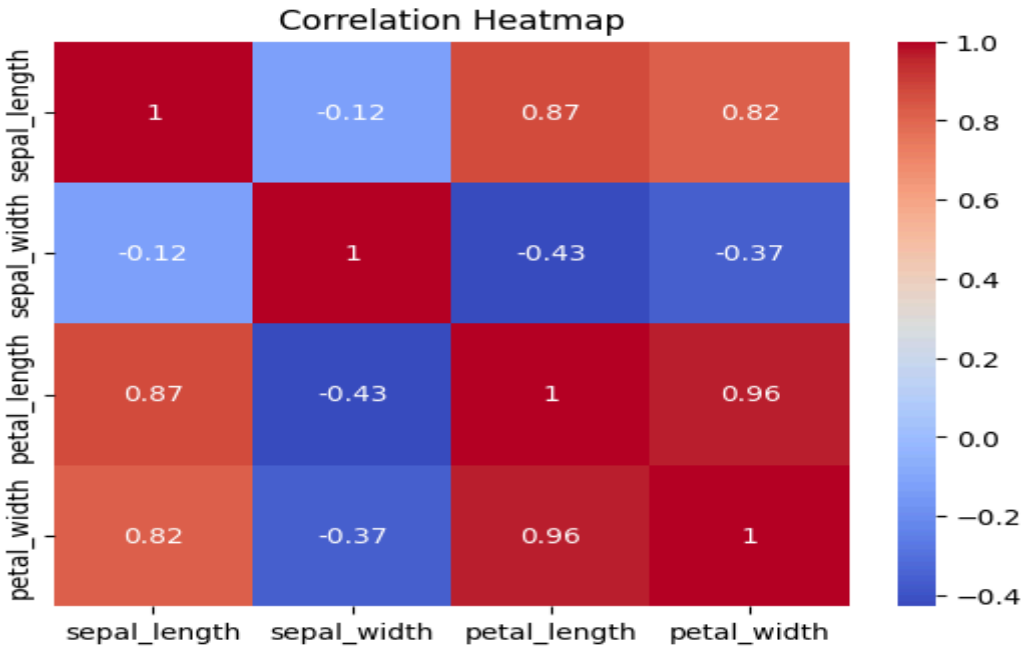
A **Heatmap** represents **correlations or relationships** between numerical variables using **color gradients**.

Where to Use:

- Analyzing **correlation matrices**
- **Visualizing patterns in large datasets**
- Representing **density-based relationships**

Example:

```
df = sns.load_dataset("iris").iloc[:,:-1]
df_corr = df.corr()

sns.heatmap(df_corr, annot=True, cmap='coolwarm')
plt.title("Correlation Heatmap")
plt.show()
```



7. Pair Plot

Definition:

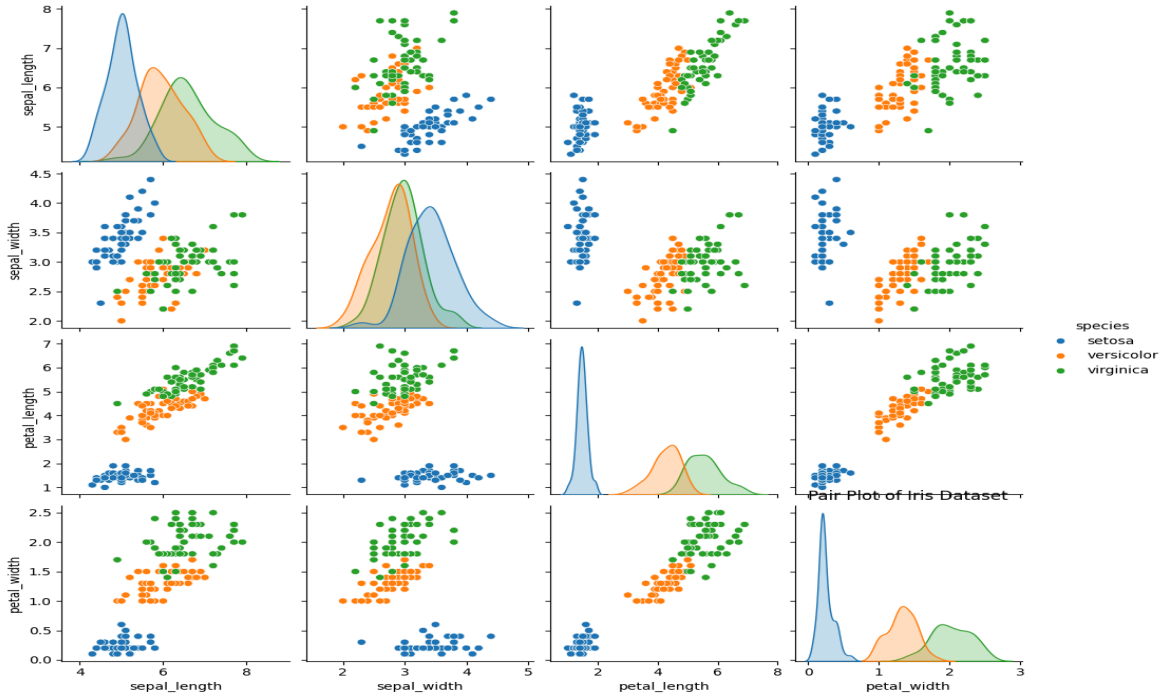
A **Pair Plot** displays multiple scatter plots across numerical features of a dataset, **showing relationships** between variables.

Where to Use:

- Observing **pairwise correlations**
- **Exploring dataset relationships before modeling**
- Detecting **linear dependencies**

Example:

```
df = sns.load_dataset("iris")

sns.pairplot(df, hue="species")
plt.title("Pair Plot of Iris Dataset")
plt.show()
```



8. Pie Chart

Definition:

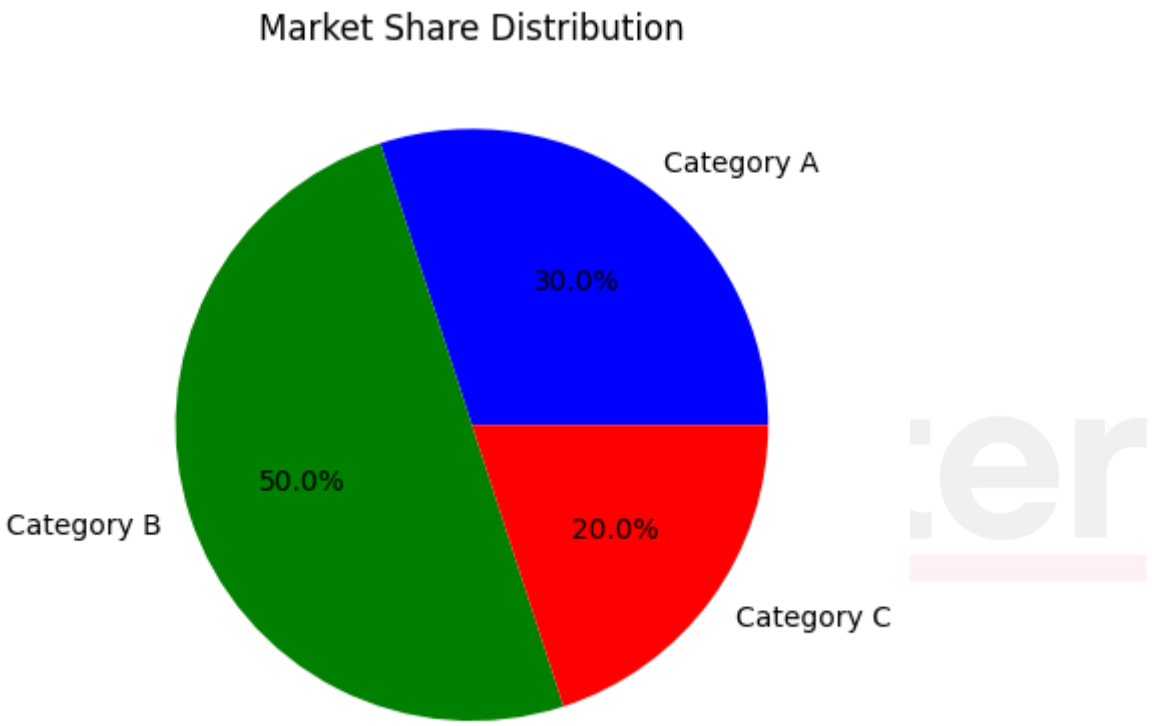
A **Pie Chart** is a circular chart divided into slices to **represent proportions** of different categories.

Where to Use:

- **Displaying percentage-based data**
- **Comparing proportions of categories**
- Representing **market share or election results**

Example:

```
labels = ['Category A', 'Category B', 'Category C']
sizes = [30, 50, 20]

plt.pie(sizes, labels=labels, autopct='%1.1f%%', colors=['blue',
'green', 'red'])
plt.title("Market Share Distribution")
plt.show()
```



9. Violin Plot

Definition:

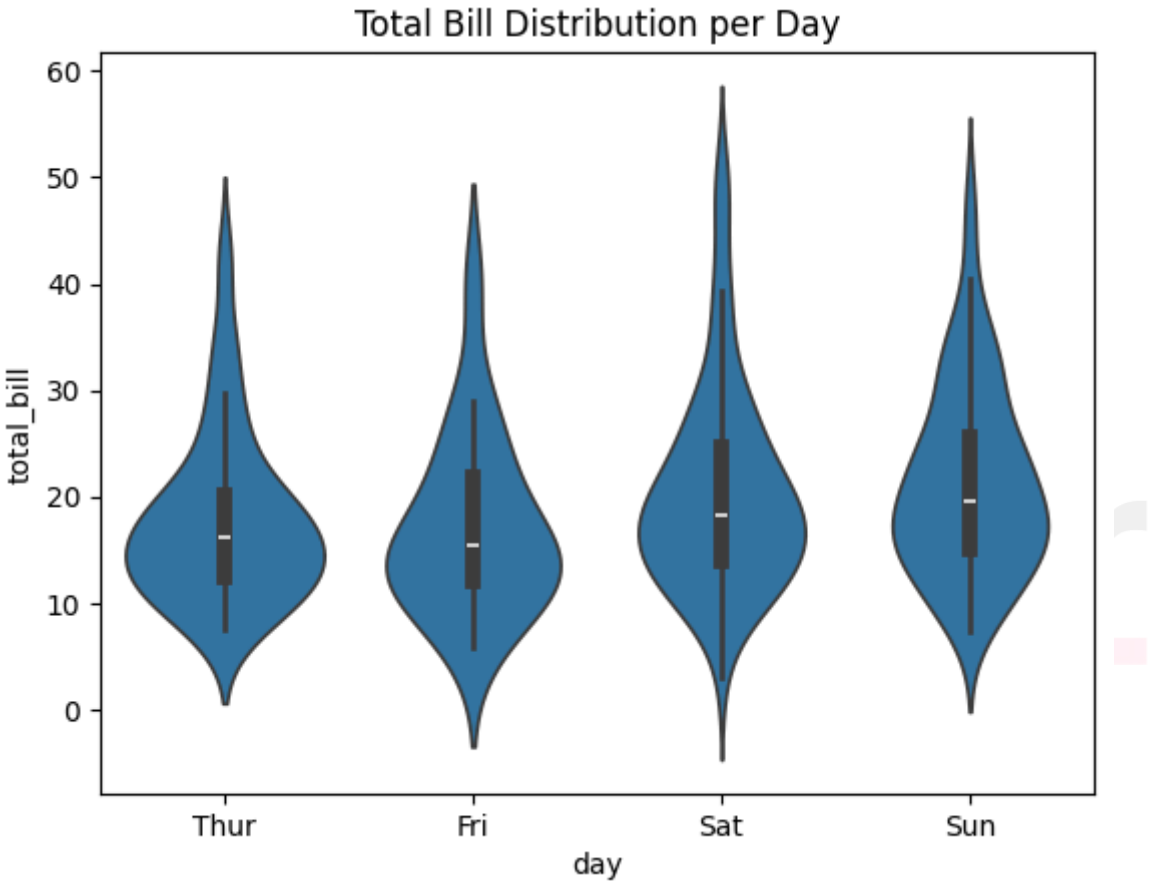
A **Violin Plot** is similar to a box plot but **adds a density distribution** to better visualize the shape of the data.

Where to Use:

- Comparing **data distributions between groups**
- **Understanding multimodal distributions**
- Combining **summary statistics with density plots**

Example:

```
df = sns.load_dataset("tips")

sns.violinplot(x="day", y="total_bill", data=df)
plt.title("Total Bill Distribution per Day")
plt.show()
```



10. Area Chart

Definition:

An **Area Chart** is similar to a line chart but **shaded below the line** to highlight cumulative trends.

Where to Use:

- **Showing cumulative trends over time**
- **Comparing multiple categories' contributions**
- **Emphasizing volume-based data trends**

Example:

```
x = np.arange(1, 11)
y = [10, 15, 7, 12, 18, 25, 30, 27, 23, 20]

plt.fill_between(x, y, color="skyblue", alpha=0.5)
plt.plot(x, y, color="blue")
plt.title("Cumulative Sales Over Time")
plt.xlabel("Month")
plt.ylabel("Sales")
plt.show()
```

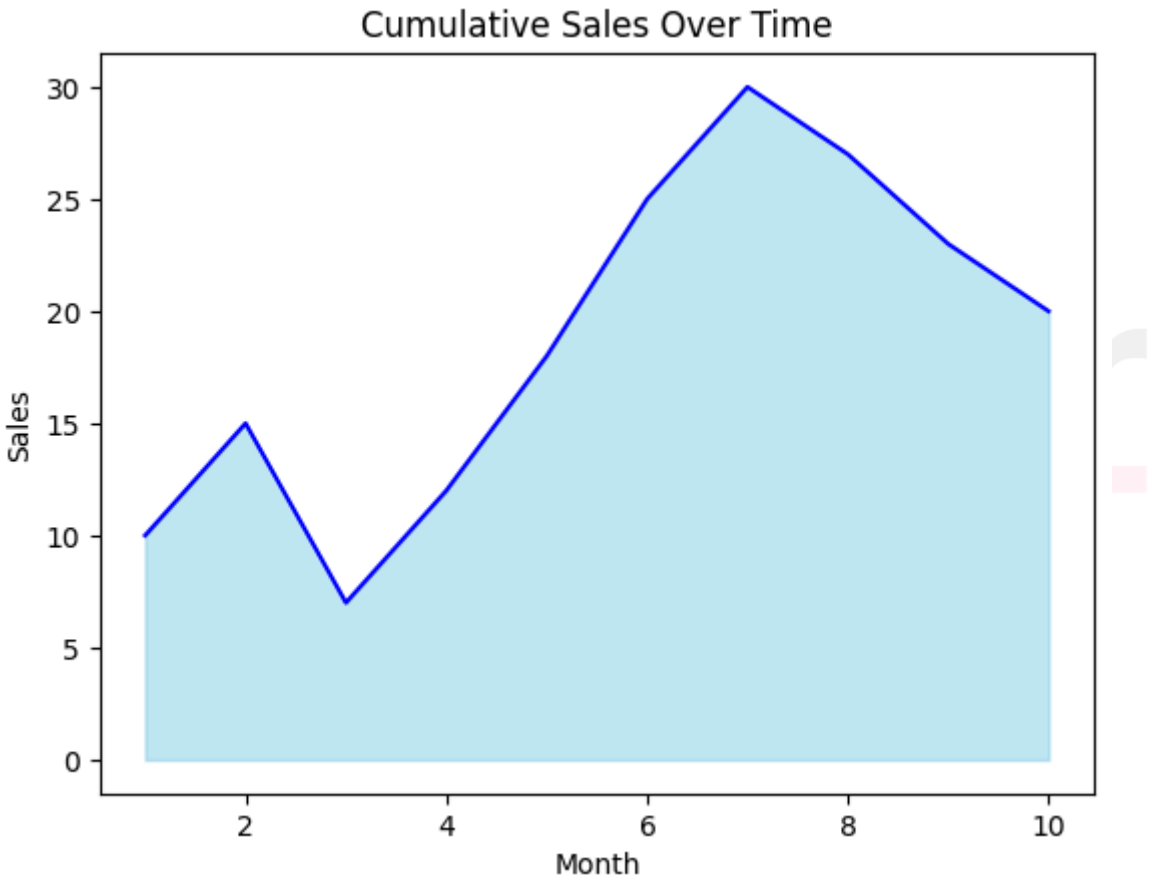


Chart Types and Use Cases

Chart Type	Best Use Cases
Line Plot	Time-series trends, forecasting
Scatter Plot	Relationship between two variables

Bar Chart	Comparing categories, rankings
Histogram	Understanding data distribution
Box Plot	Identifying outliers and data spread
Heatmap	Correlation analysis, density-based data
Pair Plot	Multi-variable relationships
Pie Chart	Percentage distribution of categories
Violin Plot	Comparing distributions with density
Area Chart	Cumulative trends over time

Summary

This lesson provided a **comprehensive revision** of **data visualization using Matplotlib and Seaborn**.

- **Matplotlib Basics:** Covered **line, scatter, and bar plots**, with **customization techniques**.
- **Seaborn Introduction:** Discussed **advantages over Matplotlib** and how to create **scatter plots, line plots, and histograms**.
- **Advanced Seaborn Techniques:** Reviewed **pair plots, heatmaps, and categorical plots**.

Lesson 2: Data Visualization Tips & Variable Analysis

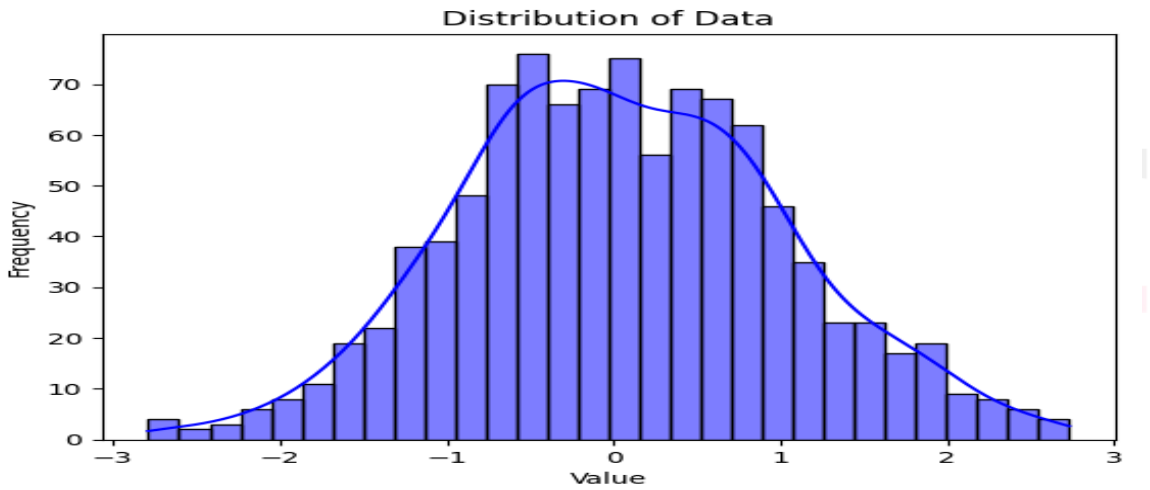
Univariate Analysis: Examining the Distribution of a Single Variable

Univariate analysis focuses on analyzing **a single variable** to understand its distribution, central tendency, and spread. This analysis is essential for **detecting patterns, outliers, and skewness** in data.

1. Histogram for Distribution Analysis

A **histogram** displays the frequency distribution of a single numerical variable.

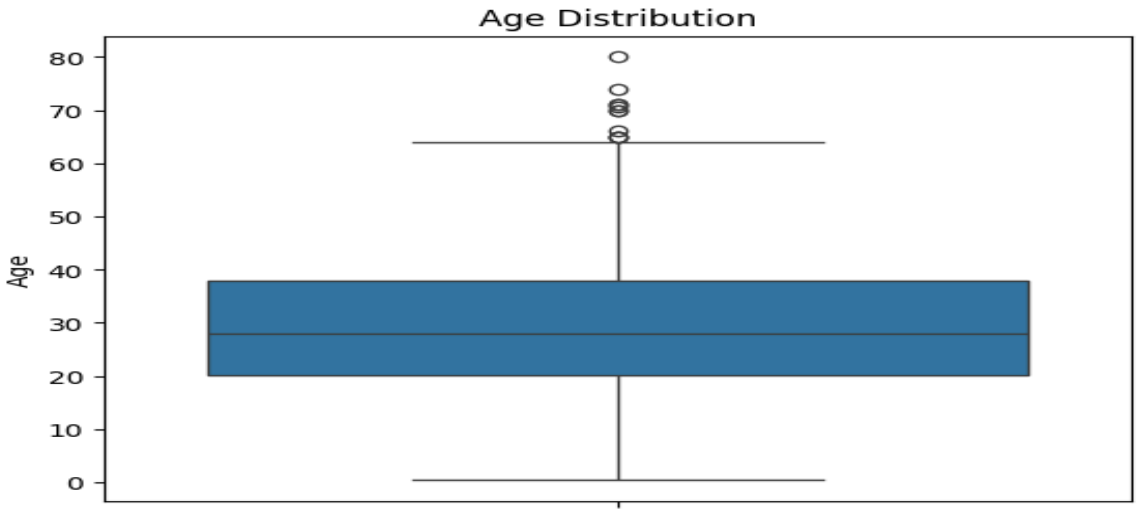
```
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt

# Generate random data
data = np.random.randn(1000)

# Plot histogram
sns.histplot(data, bins=30, kde=True, color='blue')
plt.title("Distribution of Data")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```



2. Box Plot for Outlier Detection

A **box plot** provides insights into the **median, quartiles, and outliers** of a numerical variable.

```
df = sns.load_dataset("titanic")

sns.boxplot(y=df["age"])
plt.title("Age Distribution")
plt.ylabel("Age")
plt.show()
```



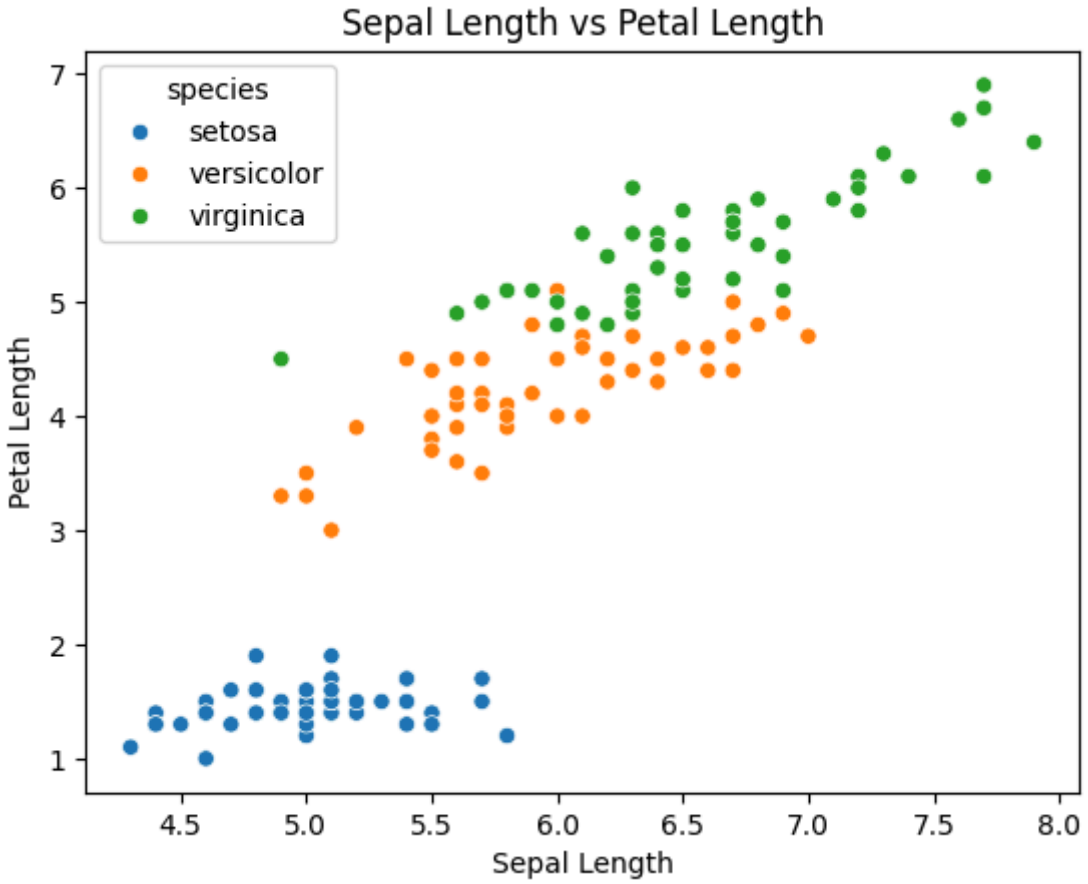
Bivariate Analysis: Exploring Relationships Between Two Variables

Bivariate analysis examines **the relationship between two variables** to identify trends, correlations, and dependencies.

1. Scatter Plot for Relationship Analysis

A **scatter plot** is used to visualize **correlations** between two numerical variables.

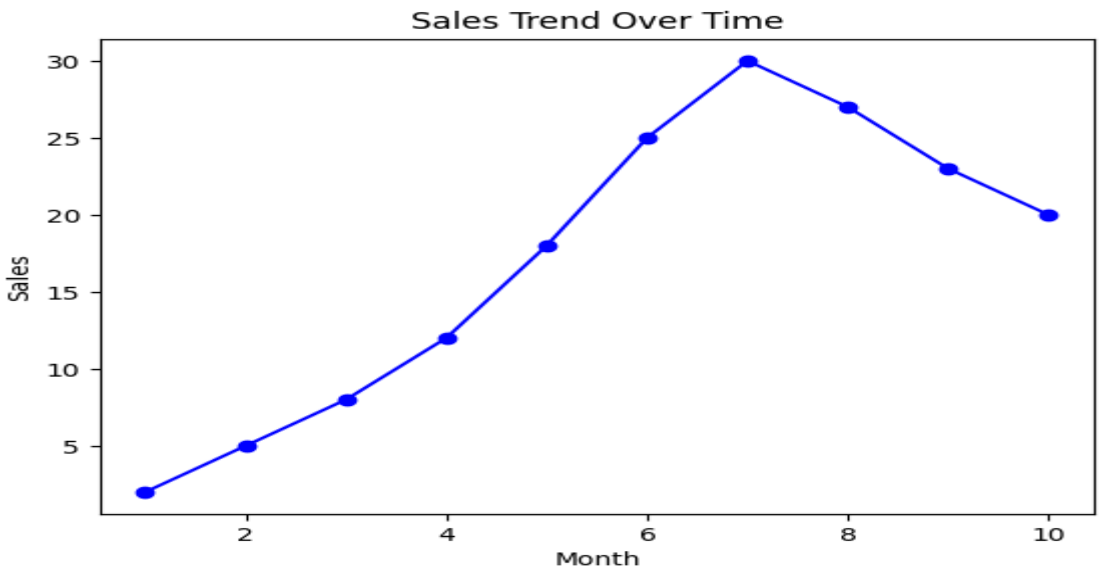
```
df = sns.load_dataset("iris")

sns.scatterplot(x="sepal_length", y="petal_length",
hue="species", data=df)
plt.title("Sepal Length vs Petal Length")
plt.xlabel("Sepal Length")
plt.ylabel("Petal Length")
plt.show()
```



2. Line Plot for Trend Analysis

A **line plot** is useful for observing **trends and patterns over time**.

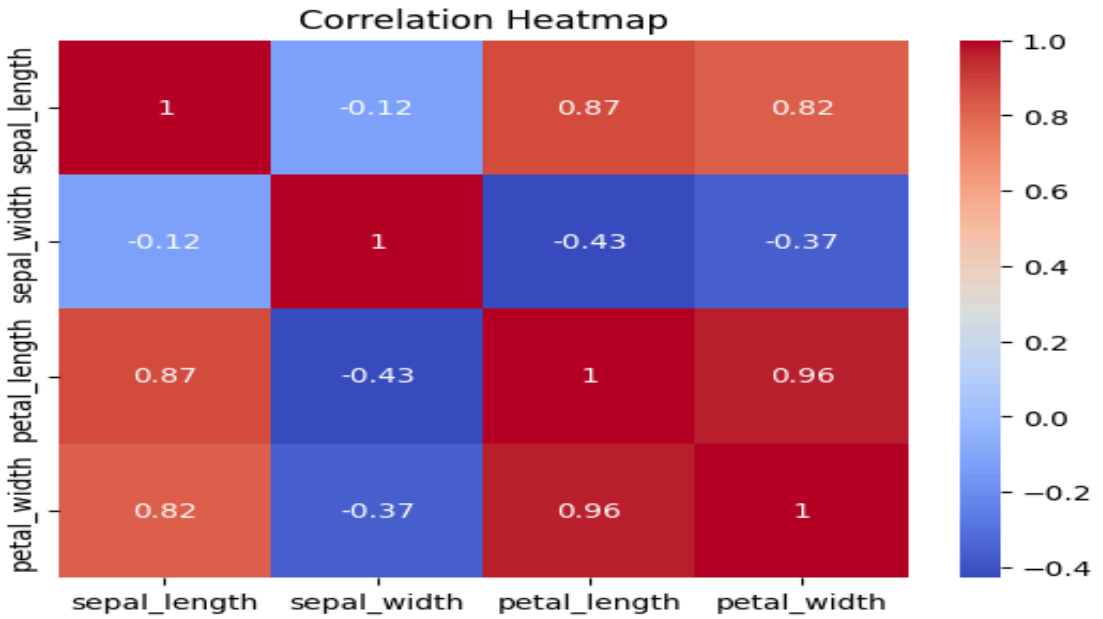
```
x = np.arange(1, 11)
y = [2, 5, 8, 12, 18, 25, 30, 27, 23, 20]

plt.plot(x, y, marker='o', linestyle='-', color='b')
plt.title("Sales Trend Over Time")
plt.xlabel("Month")
plt.ylabel("Sales")
plt.show()
```



3. Heatmap for Correlation Analysis

A **heatmap** displays the correlation matrix between multiple variables.

```
df_corr = df.iloc[:,:-1].corr()

sns.heatmap(df_corr, annot=True, cmap='coolwarm')
plt.title("Correlation Heatmap")
plt.show()
```



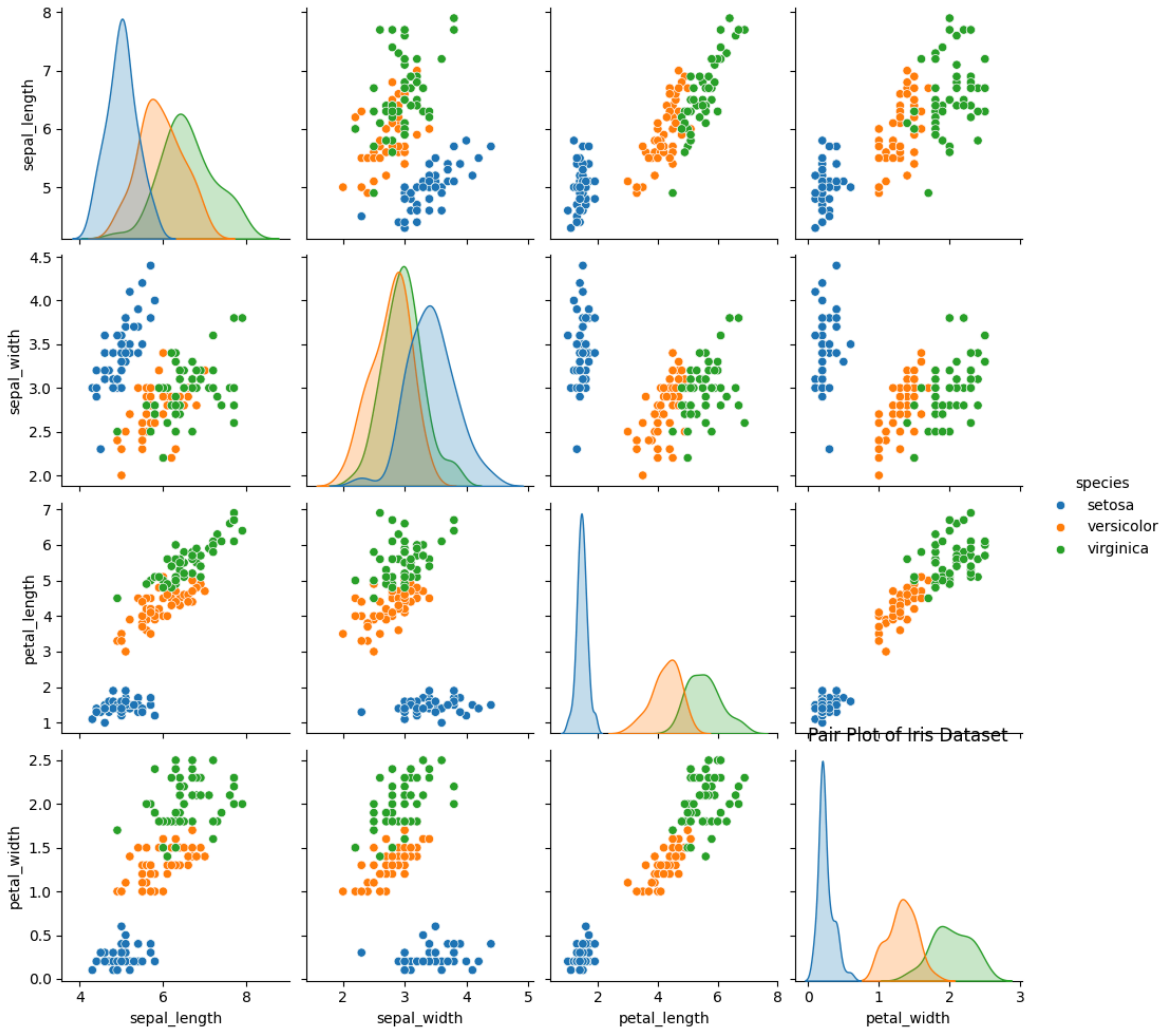
Multivariate Analysis: Understanding Patterns Across Multiple Variables

Multivariate analysis is used to uncover **complex relationships between three or more variables**.

1. Pair Plot for Multi-Variable Relationships

A **pair plot** displays scatter plots between **all numerical variables** in a dataset.

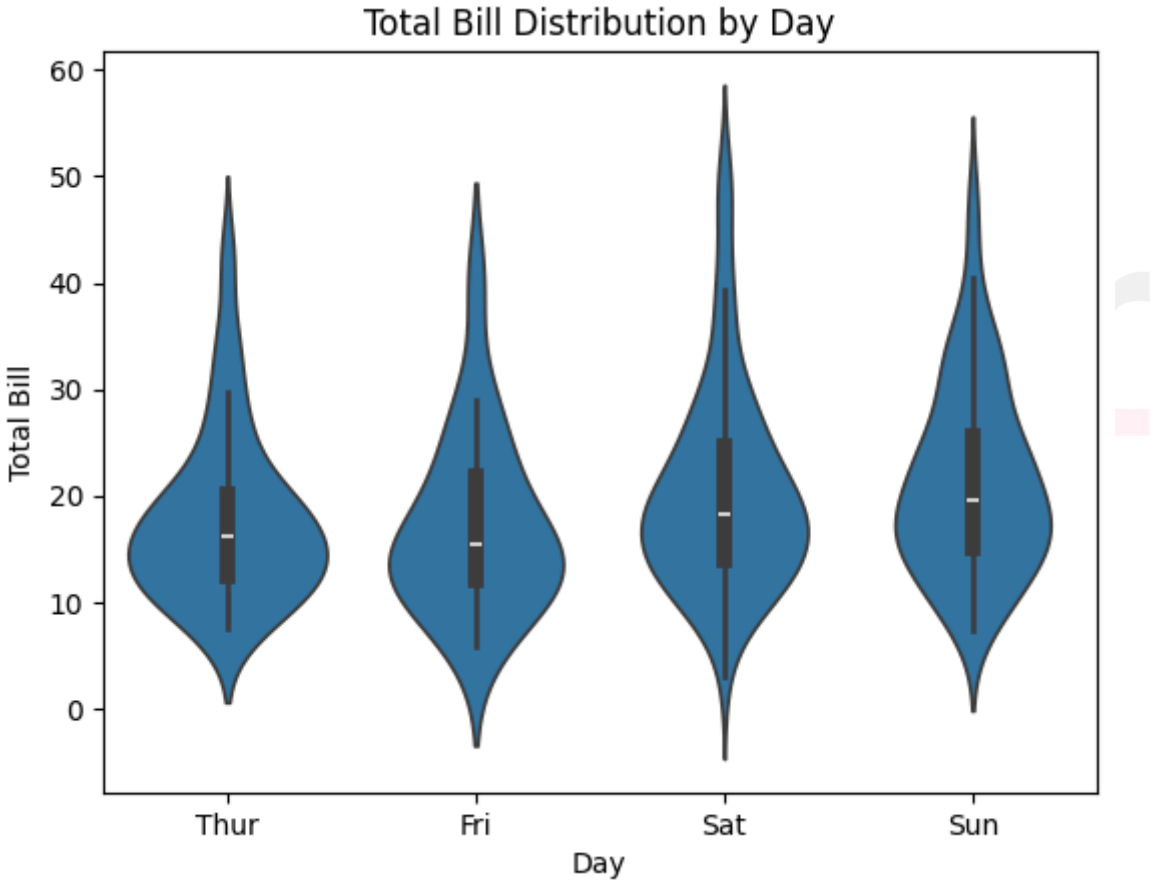
```
sns.pairplot(df, hue="species")
plt.title("Pair Plot of Iris Dataset")
plt.show()
```



2. Violin Plot for Multi-Category Comparisons

A **violin plot** combines a **box plot and density plot** to show **distribution differences across categories**.

```
df = sns.load_dataset("tips")

sns.violinplot(x="day", y="total_bill", data=df)
plt.title("Total Bill Distribution by Day")
plt.xlabel("Day")
plt.ylabel("Total Bill")
plt.show()
```



Overcoming Common Visualization Challenges

1. Avoiding Clutter and Misleading Representations

- **Remove unnecessary grid lines**
- **Use proper axis scaling to avoid exaggeration**
- **Limit the number of categories in bar plots**

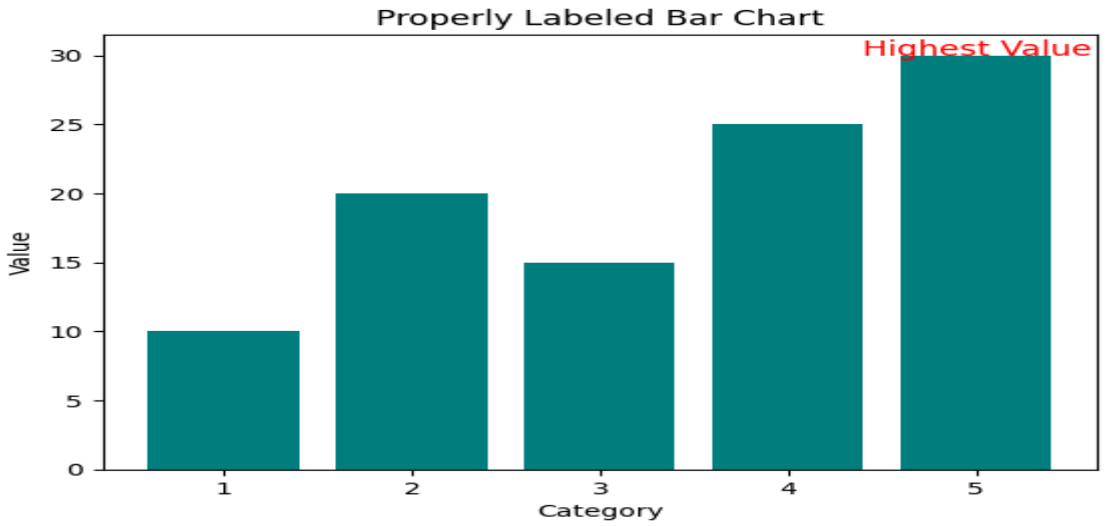
2. Choosing Effective Color Schemes

- **Use distinguishable colors** to avoid confusion
- **Apply color gradients** to highlight intensity in heatmaps

3. Using Proper Labels and Annotations

- **Always add axis labels, titles, and legends**
- **Use annotations to highlight key insights**

```
x = np.arange(1, 6)
y = [10, 20, 15, 25, 30]

plt.bar(x, y, color='teal')

# Adding labels
plt.xlabel("Category")
plt.ylabel("Value")
plt.title("Properly Labeled Bar Chart")

# Annotating the highest bar
plt.text(5, 30, "Highest Value", fontsize=12, ha='center',
color='red')

plt.show()
```



Practical Application with Real-World Data

1. Customer Purchase Analysis (Using DVD Rental Data)

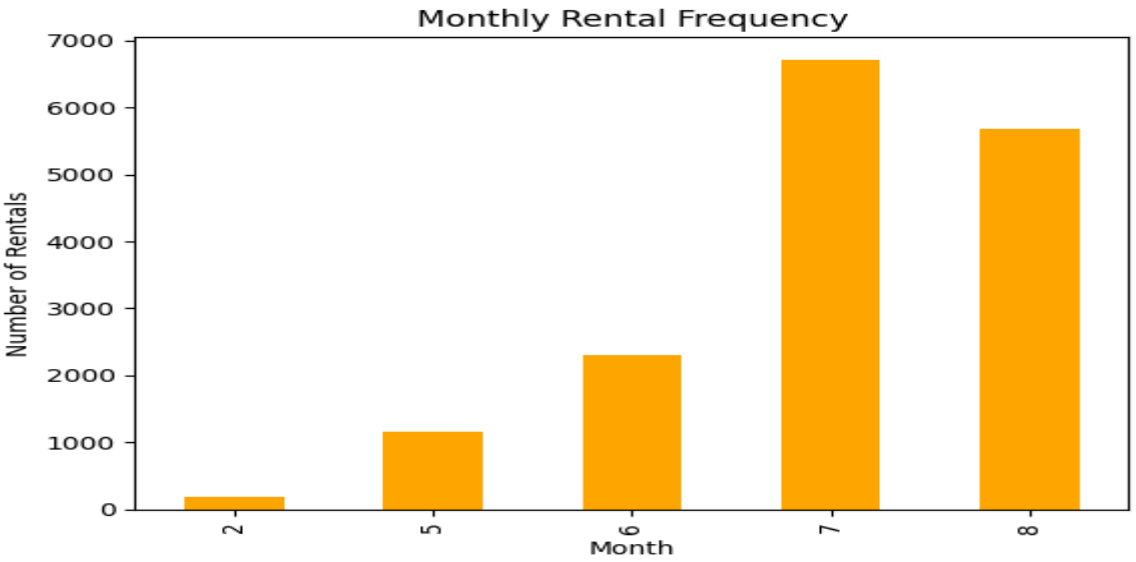
Using real-world datasets allows **practical visualization experience**.

```
import pandas as pd

# Load DVD Rental Data
df_customers = pd.read_csv("customer.csv")

# Count customers per store
sns.countplot(x="store_id", data=df_customers)
plt.title("Number of Customers per Store")
plt.xlabel("Store ID")
plt.ylabel("Count")
plt.show()
```

2. Rental Transactions Over Time

```
df_rentals = pd.read_csv("rental.csv")

# Convert rental date to datetime
df_rentals["rental_date"] =
pd.to_datetime(df_rentals["rental_date"])
```

```
# Plot rental frequency
df_rentals["rental_date"].dt.month.value_counts().sort_index().p
lot(kind="bar", color="orange")
plt.title("Monthly Rental Frequency")
plt.xlabel("Month")
plt.ylabel("Number of Rentals")
plt.show()
```



Summary

This lesson provided a **structured revision** of **data visualization techniques** to analyze variables effectively.

- **Univariate Analysis:** Explored **histograms and box plots** for understanding single-variable distributions.
- **Bivariate Analysis:** Used **scatter plots, line plots, and heatmaps** to examine relationships between two variables.
- **Multivariate Analysis:** Covered **pair plots and violin plots** for analyzing multiple variables together.
- **Visualization Design:** Focused on **choosing appropriate plots, avoiding clutter, and effective labeling**.
- **Practical Application:** Demonstrated **real-world visualizations using the DVD Rental dataset**.

Chapter 3: Data Insights with APIs & EDA

The **Data Insights with APIs & Exploratory Data Analysis (EDA)** chapter provides a **structured and efficient refresher** on how to interact with external data sources, extract meaningful insights, and perform data-driven decision-making. This section serves as a **quick revision guide**, summarizing key concepts concisely without deep explanations.

Lesson 1: Data Modeling via APIs

This lesson focuses on retrieving **data from APIs (Application Programming Interfaces)** and transforming it into structured formats suitable for analysis. APIs act as bridges that allow applications to communicate with external services to fetch real-time data. This revision material covers the fundamentals of making API requests using the **GET and POST methods**, handling **authentication**, and processing **JSON responses**.

Lesson 2: Interactive Visuals with Plotly

This lesson revisits **interactive visualization techniques** using **Plotly**, an advanced visualization library that allows the creation of highly interactive and dynamic plots. Unlike static charts, Plotly enables **hover effects, zooming, and filtering**, making data exploration more intuitive. The revision material covers how to create **interactive line charts, bar charts, and scatter plots**, enhancing data representation for time-series and multi-dimensional datasets.

Lesson 3: Exploratory Data Analysis (EDA) via APIs

This lesson revisits the **EDA process**, emphasizing the use of **real-time data from APIs** for analysis. It covers techniques for **identifying trends, missing values, and outliers**, ensuring the dataset is clean and well-structured before further analysis. The revision material includes reviewing **basic and advanced data cleaning techniques**, including handling missing values, removing duplicates, and transforming raw API data into structured formats.

Lesson 4: Streamlit Dashboards for EDA

This lesson provides a structured revision of building **interactive dashboards using Streamlit**, a powerful framework for deploying data-driven applications. The revision guide covers how to create **real-time dashboards that fetch and visualize API data dynamically**, enabling users to interact with datasets in a seamless and intuitive manner.

Lesson 1: Data Modeling via APIs

What is an API?

An **API (Application Programming Interface)** is a mechanism that allows different software applications to communicate with each other. APIs serve as intermediaries, enabling applications to send requests and receive responses over the internet. They allow developers to access external services, databases, or functionality without having to build them from scratch. APIs are widely used in web development, mobile applications, and data analytics for fetching **real-time data** from online sources.

Behind the Scenes: How a Web Application Works!

A **web application** follows a client-server architecture, where the client (browser, mobile app, or another service) sends requests to a server, which processes the request and returns the appropriate response. APIs enable this communication by defining a structured way to **exchange data** between the client and the server. When a client requests data from a website, the API acts as a bridge, **fetching, processing, and returning the requested information**.

For example, when a user searches for a movie on a streaming service, the frontend sends an API request to the backend, which retrieves movie details from a database and returns them to the client.

How to Use APIs and Perform CRUD Operations

APIs allow users to perform **CRUD (Create, Read, Update, Delete) operations** on data. These operations map directly to **HTTP methods**, which are used for sending API requests. The four basic CRUD operations and their corresponding HTTP methods are:

- **Create (POST)** – Sends new data to the server
- **Read (GET)** – Retrieves data from the server
- **Update (PUT/PATCH)** – Modifies existing data
- **Delete (DELETE)** – Removes data from the server

What is a RESTful API?

A **RESTful API (Representational State Transfer API)** is a web service that follows the **REST architectural style**. REST APIs use standard HTTP methods and work with structured formats such as **JSON or XML**. A RESTful API provides:

- **Stateless communication**, meaning each request is independent and does not rely on previous interactions
- **Resource-based URLs** to perform actions on data
- **Support for multiple data formats**, primarily JSON

For example, a REST API for a movie database might have endpoints like:

- `GET /movies` → Fetch all movies
- `POST /movies` → Add a new movie
- `PUT /movies/{id}` → Update a movie by ID
- `DELETE /movies/{id}` → Remove a movie

HTTP Methods and CRUD Operations

Each HTTP method performs a specific action on data.

```
GET → Fetches data from the server
POST → Sends new data to the server
PUT → Updates an entire resource
PATCH → Modifies part of an existing resource
DELETE → Removes a resource
```

Example API endpoints using CRUD operations:

```
GET https://api.example.com/users → Retrieves all users
POST https://api.example.com/users → Creates a new user
PUT https://api.example.com/users/1 → Updates user with ID 1
DELETE https://api.example.com/users/1 → Deletes user with ID 1
```

How Python's `requests` Library Works

Python's **requests** library is used to **send HTTP requests** and interact with web APIs. It allows fetching data, sending data, and handling API responses efficiently.

Importing the requests Library

To use the `requests` library, install it first (if not already installed).

```
!pip install requests
```

```
import requests

response =
requests.get("https://jsonplaceholder.typicode.com/posts")
print(response.status_code)  # Output: 200
```

Send a Request and Receive a Response

APIs return **responses** in structured formats such as **JSON (JavaScript Object Notation)**. Python can parse these responses and extract useful data.

```
url = "https://jsonplaceholder.typicode.com/posts/1"
response = requests.get(url)

# Convert JSON response to Python dictionary
data = response.json()

print(data)

# Output:
# {'userId': 1, 'id': 1, 'title': 'Sample Title', 'body': 'This
is the post content.'}
```

Handling Status Codes

APIs return **status codes** that indicate whether the request was successful or encountered an error.

```
if response.status_code == 200:
    print("Request Successful!")
elif response.status_code == 404:
    print("Resource Not Found!")
else:
    print("An error occurred.")
```

Common HTTP status codes:

- **200 OK** – Request successful

- **201 Created** – New resource added
- **400 Bad Request** – Invalid request
- **401 Unauthorized** – Authentication required
- **403 Forbidden** – Access denied
- **404 Not Found** – Resource does not exist
- **500 Internal Server Error** – Server-side issue

How API Endpoints Work

An **API endpoint** is a specific URL that a client uses to access a resource. Each endpoint corresponds to a particular **functionality**, such as retrieving a list of users, fetching a specific movie, or updating a record.

For example, in a movie rental database, the API may have the following endpoints:

- `GET /movies` → Retrieve all movies
- `GET /movies/10` → Retrieve details of movie ID 10
- `POST /movies` → Add a new movie
- `PUT /movies/10` → Update movie ID 10
- `DELETE /movies/10` → Remove movie ID 10

Example: Fetching Data from an API and Visualizing It

Fetching movie rental data from an API and visualizing it using Matplotlib.

```
import requests
import pandas as pd
import matplotlib.pyplot as plt

# API URL for fetching movie rental data
url = "https://api.example.com/movies"
response = requests.get(url)

# Convert JSON response to Pandas DataFrame
movies = pd.DataFrame(response.json())

# Plot movie ratings
plt.bar(movies["title"], movies["rating"], color='blue')
plt.xticks(rotation=90)
plt.xlabel("Movie Titles")
```

```
plt.ylabel("Ratings")
plt.title("Movie Ratings")
plt.show()
```

Different API Methods

APIs offer various methods for performing tasks such as **authentication, filtering data, and paginating results**.

1. Sending Data with a POST Request

```
new_movie = {
    "title": "Inception",
    "director": "Christopher Nolan",
    "year": 2010
}

response = requests.post("https://api.example.com/movies",
json=new_movie)
print(response.status_code)

# Output: 201 (Created)
```

2. Updating Data with a PUT Request

```
update_data = {"year": 2012}

response = requests.put("https://api.example.com/movies/1",
json=update_data)
print(response.status_code)

# Output: 200 (OK)
```

3. Deleting Data with a DELETE Request

```
response = requests.delete("https://api.example.com/movies/1")
print(response.status_code)

# Output: 204 (No Content)
```

Summary

This lesson provided a **structured revision** of **working with APIs and performing CRUD operations**.

- **Introduction to APIs:** Covered **what APIs are and how they work behind the scenes**.
- **RESTful APIs and HTTP Methods:** Reviewed **GET, POST, PUT, DELETE** operations.
- **Using Python's `requests` library:** Learned how to **send requests, handle responses, and process JSON data**.
- **Working with API Endpoints:** Explored **fetching, updating, and deleting data via APIs**.
- **Practical API Usage:** Demonstrated **fetching real-world data and visualizing it using Matplotlib**.

Lesson 2: Interactive Visuals with Plotly

What is Plotly?

Plotly is a **powerful interactive visualization library** in Python that enables users to create **dynamic and visually appealing charts**. Unlike Matplotlib and Seaborn, Plotly provides **interactive functionalities such as zooming, hovering, filtering, and panning**. It supports a wide range of chart types, making it ideal for **data exploration, dashboards, and business intelligence applications**.

Why Use Plotly?

Plotly is preferred over static visualization libraries because of its **interactive capabilities** and **seamless integration with web applications**. It is useful when dealing with **large datasets** and allows users to **explore data visually** without requiring manual filtering.

Key Advantages of Plotly

- **Interactivity:** Allows zooming, hovering, and tooltips.
- **Ease of Use:** Provides a high-level API for quick plotting.
- **Customizability:** Supports color scales, annotations, and layout adjustments.

- **Web Compatibility:** Generates HTML-based plots for embedding in web applications.
- **Integration with Dash:** Can be used to create real-time dashboards.

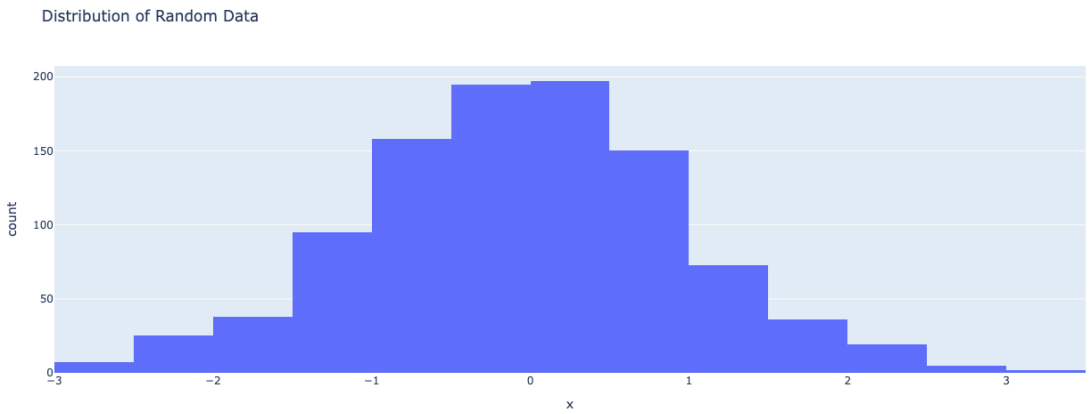
What Can Be Done with Plotly?

Plotly supports a **wide range of visualization types**, from **basic charts** like histograms and scatter plots to **advanced visualizations** like treemaps, heatmaps, and gauge charts.

Histogram

A **histogram** is used to represent the **distribution of a numerical variable** by grouping data into bins.

```
import plotly.express as px
import numpy as np
import pandas as pd

# Generate sample data
data = np.random.randn(1000)

# Create histogram
fig = px.histogram(x=data, nbins=30, title="Distribution of
Random Data")
fig.show()
```

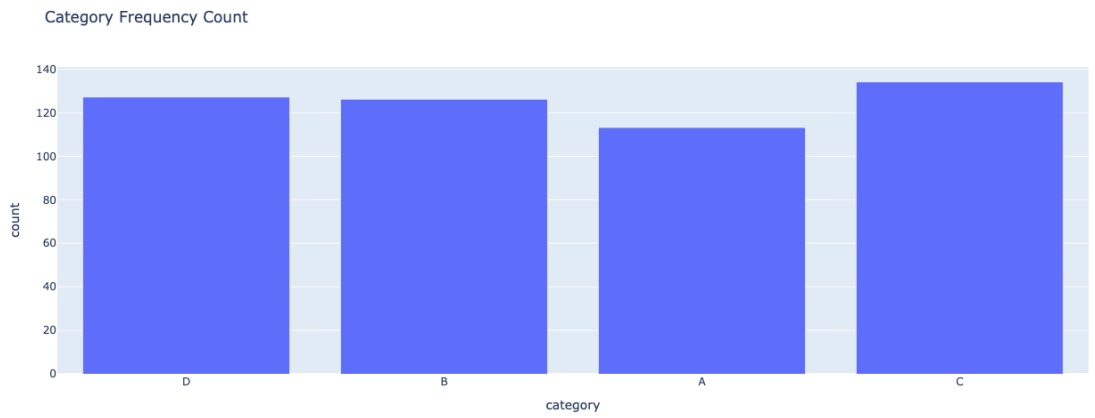


Usage: Histograms are used for **understanding the distribution of a dataset, detecting skewness, and identifying outliers**.

Count Plot

A **count plot** is used to visualize **the frequency of categorical data**.

```
df = pd.DataFrame({"category": np.random.choice(["A", "B", "C", "D"], 500)})

fig = px.histogram(df, x="category", title="Category Frequency Count")
fig.show()
```

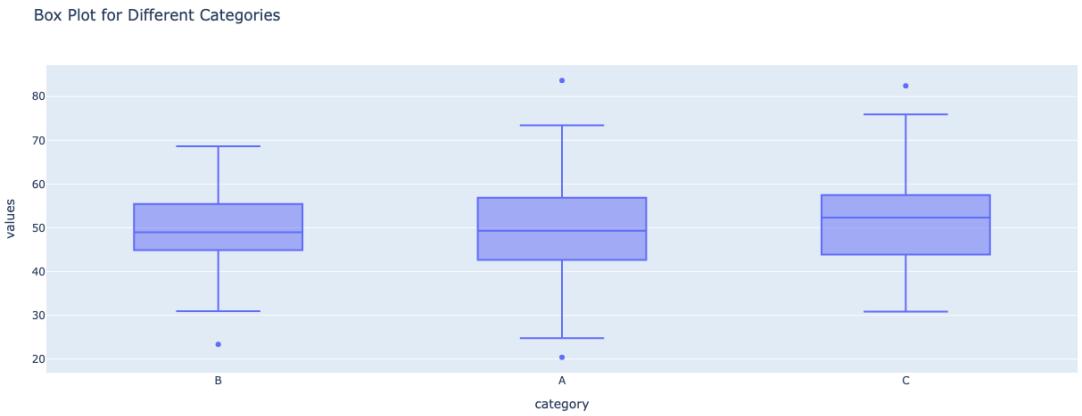


Usage: Count plots are used for **analyzing categorical distributions, such as customer demographics or survey responses**.

Box Plot

A **box plot** is used to display **summary statistics** of numerical data, including **median, quartiles, and outliers**.

```
df = pd.DataFrame({"category": np.random.choice(["A", "B", "C"], 500),
                   "values": np.random.randn(500) * 10 + 50})

fig = px.box(df, x="category", y="values", title="Box Plot for Different Categories")
fig.show()
```

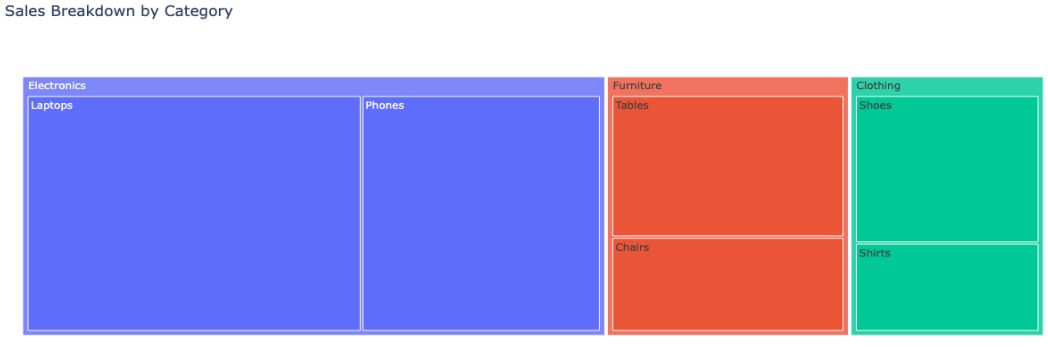


Usage: Box plots are used for **detecting outliers, comparing distributions, and analyzing variability**.

Tree Map

A **treemap** represents **hierarchical data** using nested rectangles.

```
df = pd.DataFrame({
    "category": ["Electronics", "Electronics", "Furniture",
"Furniture", "Clothing", "Clothing"],
    "subcategory": ["Phones", "Laptops", "Chairs", "Tables",
"Shirts", "Shoes"],
    "sales": [5000, 7000, 2000, 3000, 1500, 2500]
})

fig = px.treemap(df, path=["category", "subcategory"],
values="sales", title="Sales Breakdown by Category")
fig.show()
```

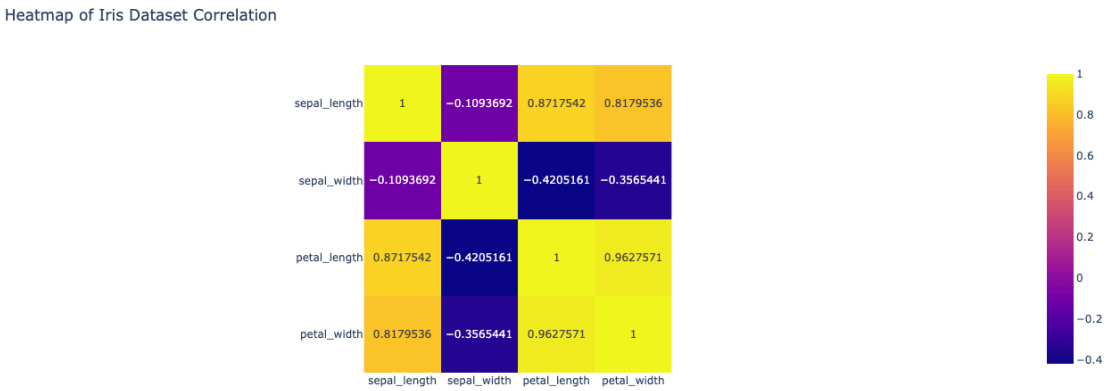


Usage: Treemaps are useful for **visualizing hierarchical data, such as market segmentation and company sales breakdowns**.

Heatmap

A **heatmap** visualizes **correlations between numerical variables** using color gradients.

```
df = px.data.iris()
df=df.loc[:,["sepal_length","sepal_width","petal_length","petal_
width"]]
fig = px.imshow(df.corr(), text_auto=True, title="Heatmap of
Iris Dataset Correlation")
fig.show()
```



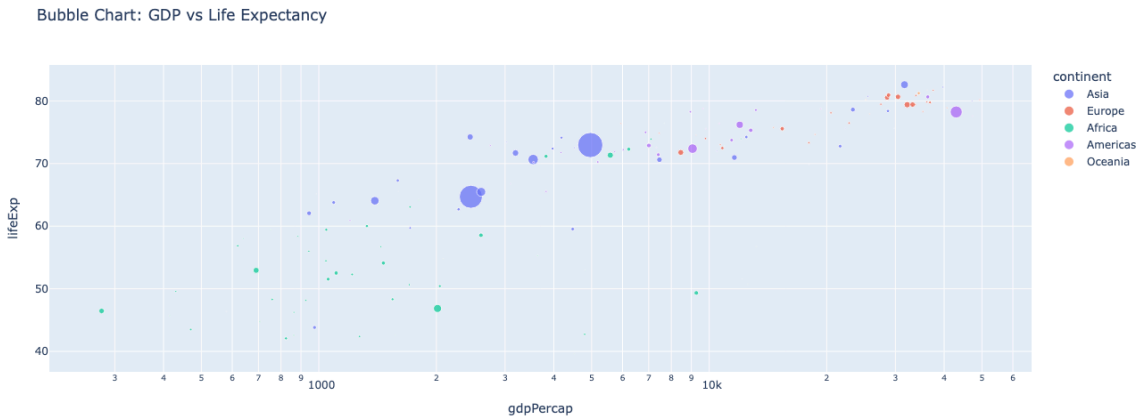
Usage: Heatmaps are commonly used for **correlation analysis, performance tracking, and trend visualization**.

Bubble Chart

A **bubble chart** is an **enhanced scatter plot** where the size of each point represents an additional variable.

```
df = px.data.gapminder().query("year == 2007")

fig = px.scatter(df, x="gdpPercap", y="lifeExp", size="pop",
color="continent",
                 hover_name="country", log_x=True, title="Bubble
Chart: GDP vs Life Expectancy")
```

```
fig.show()
```



Usage: Bubble charts are used for **visualizing three-dimensional data, such as financial markets and demographic studies**.

Gauge Chart

A **gauge chart** is used to display **progress toward a goal or a performance metric**.

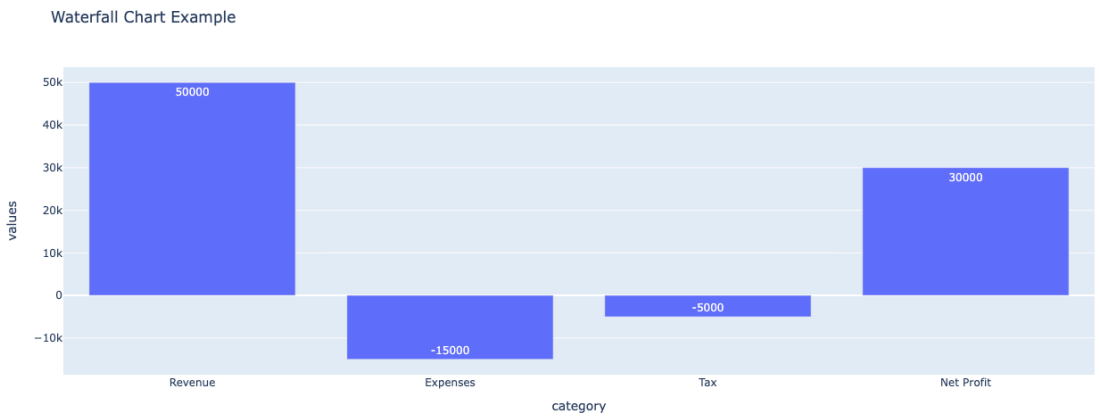
```
import plotly.graph_objects as go

# Create an indicator chart using go.Indicator
fig = go.Figure(go.Indicator(
    value=70,
    number={"suffix": "%"},
    title={"text": "Gauge Chart: Project Completion"},
    gauge={"axis": {"range": [0, 100]}}
))
fig.show()
```

Usage: Gauge charts are used in **dashboard reporting, KPI tracking, and performance analysis**.

Waterfall Chart

A **waterfall chart** is useful for **visualizing cumulative changes in data**.

```
df = pd.DataFrame({
    "category": ["Revenue", "Expenses", "Tax", "Net Profit"],
    "values": [50000, -15000, -5000, 30000]
})

fig = px.bar(df, x="category", y="values", text="values",
title="Waterfall Chart Example")
fig.show()
```



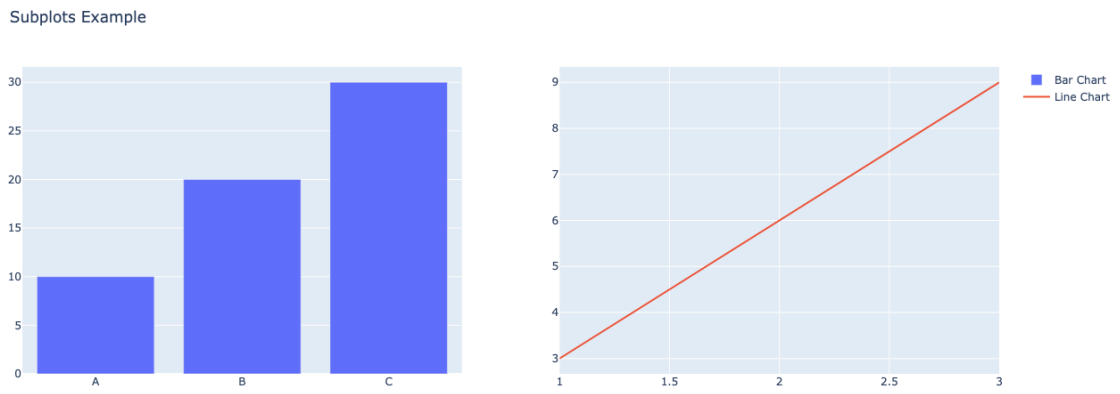
Usage: Waterfall charts are used for **financial analysis, profit and loss breakdowns, and business reporting**.

What is a Subplot?

A **subplot** allows multiple charts to be displayed in a **single figure**.

```
from plotly.subplots import make_subplots
import plotly.graph_objects as go

fig = make_subplots(rows=1, cols=2)

fig.add_trace(go.Bar(x=["A", "B", "C"], y=[10, 20, 30],
```

```
name="Bar Chart"), row=1, col=1)
fig.add_trace(go.Scatter(x=[1, 2, 3], y=[3, 6, 9], mode="lines",
name="Line Chart"), row=1, col=2)

fig.update_layout(title="Subplots Example")
fig.show()
```



Usage: Subplots are useful for **comparing multiple charts side by side in dashboards and reports**.

What is Interactivity?

Interactivity in Plotly allows users to **explore data dynamically**. Interactive features include:

- **Hover effects:** Display additional data on hover
- **Zooming and panning:** Allows users to focus on specific data points
- **Clickable legends:** Enable or disable specific data series
- **Live updates:** Automatically refresh data in real-time dashboards

Summary

This lesson provided a **structured revision** of **interactive data visualization using Plotly**.

- **Introduction to Plotly:** Covered **why Plotly is useful for dynamic visualizations**.
- **Basic and Advanced Charts:** Reviewed **histograms, box plots, heatmaps, bubble charts, and gauge charts**. **Subplots and Interactivity:** Explored **combining multiple charts and using interactive features**.

Lesson 3: Exploratory Data Analysis (EDA) via APIs

Dataset Description

In this lesson, we will perform **Exploratory Data Analysis (EDA)** on the **DVD Rental Database**, which contains data related to customer transactions, rentals, and inventory. The dataset includes multiple tables, such as **customer, rental, film, inventory, payment, and store**, which provide insights into rental trends, customer behaviors, and financial performance. Understanding the dataset is crucial before applying EDA techniques to uncover trends, relationships, and anomalies.

Loading the DVD Rental Dataset

```
import pandas as pd

# Load dataset
df_customers = pd.read_csv("customer.csv")
df_rentals = pd.read_csv("rental.csv")
df_films = pd.read_csv("film.csv")

# Display first few records
print(df_customers.head())

# Output:
#    customer_id  store_id first_name last_name email
# 0         524         1      Jared       Ely  None
# 1           1         1       Mary     Smith
mary.smith@example.com
# 2           2         1   Patricia   Johnson
patricia.johnson@example.com
```

Dataset Cleaning

Data cleaning is an essential step in EDA, ensuring that the dataset is **consistent, complete, and accurate** before analysis. Cleaning involves handling **missing values, duplicate records, inconsistent data types, and outliers**.

Checking for Missing Values

```
print(df_customers.isnull().sum())

# Output:
# customer_id    0
# store_id       0
# first_name     0
# last_name      0
# email         50
```

The **email column** contains missing values. One way to handle this is by **filling missing values** with "Unknown" or **removing the column** if it is not relevant.

```
df_customers["email"].fillna("Unknown", inplace=True)
```

Removing Duplicates

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

Checking and Converting Data Types

```
print(df_rentals.dtypes)

# Output:
# rental_id             int64
# rental_date          object
# return_date          object
```

Dates are stored as **object (string)** instead of **datetime format**, so they need to be converted.

```
df_rentals["rental_date"] =
pd.to_datetime(df_rentals["rental_date"])
df_rentals["return_date"] =
pd.to_datetime(df_rentals["return_date"])
```

Revisiting Basic Visualizations

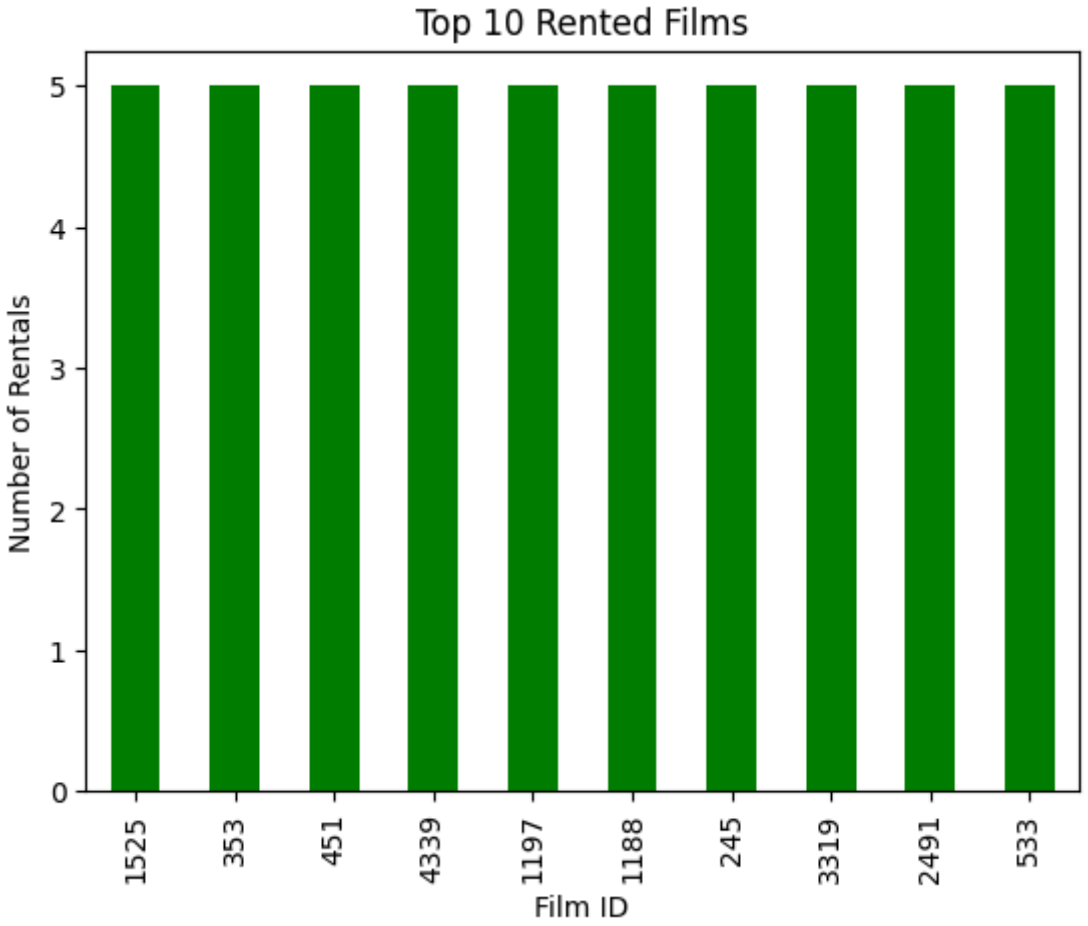
Customer Distribution Across Stores

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

```
sns.countplot(x="store_id", data=df_customers)
plt.title("Customer Distribution Across Stores")
plt.xlabel("Store ID")
plt.ylabel("Number of Customers")
plt.show()
```

Film Rental Trends

```
df_film_rental =
df_rentals["inventory_id"].value_counts().head(10)
df_film_rental.plot(kind="bar", color="green")
plt.xlabel("Film ID")
plt.ylabel("Number of Rentals")
plt.title("Top 10 Rented Films")
plt.show()
```



Data Inspection and Visualization

Revenue Analysis

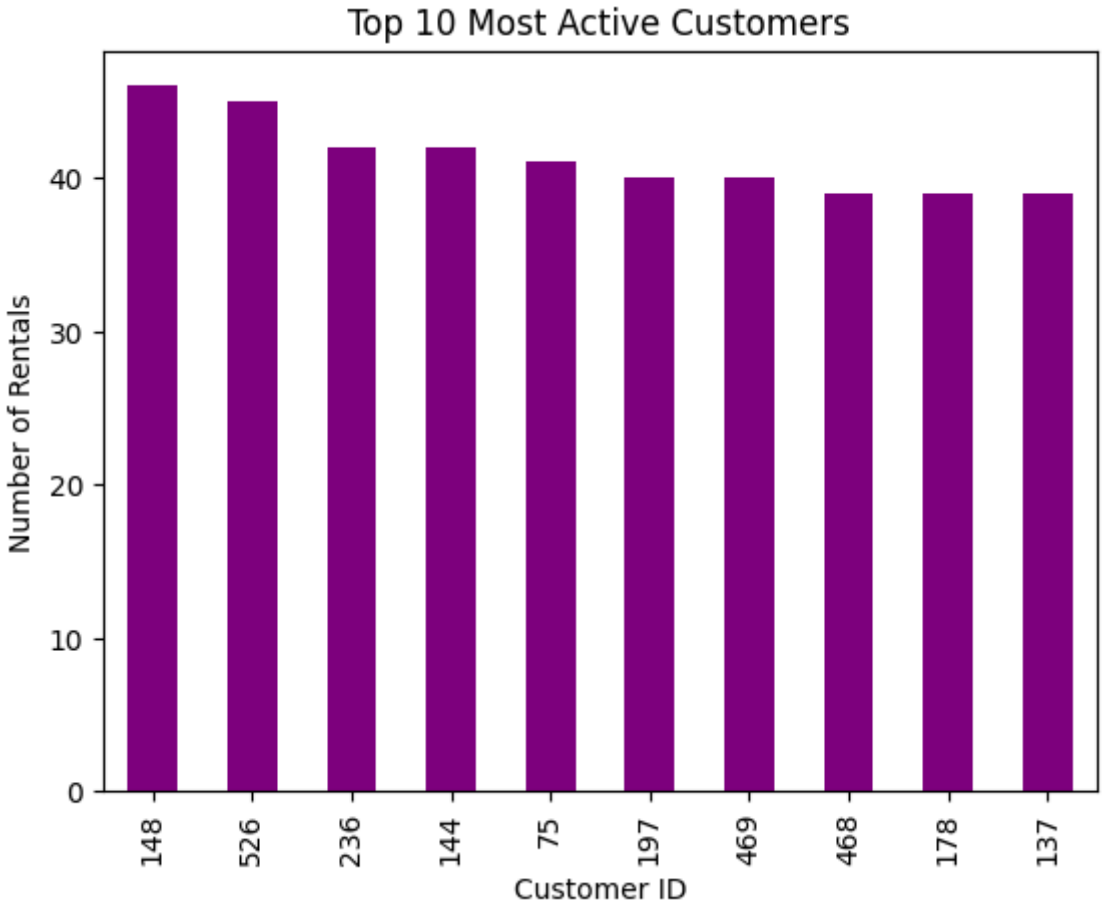
To analyze revenue, we use the **payment** table.

```
df_payment = pd.read_csv("payment.csv")
df_payment["payment_date"] =
pd.to_datetime(df_payment["payment_date"])

# Monthly Revenue Trend
df_payment["payment_month"] =
df_payment["payment_date"].dt.month
monthly_revenue =
df_payment.groupby("payment_month")["amount"].sum()

plt.plot(monthly_revenue.index, monthly_revenue.values,
marker="o", linestyle="-", color="red")
```

```
plt.xlabel("Month")
plt.ylabel("Revenue")
plt.title("Monthly Revenue Trend")
plt.show()
```

Customer Activity Analysis

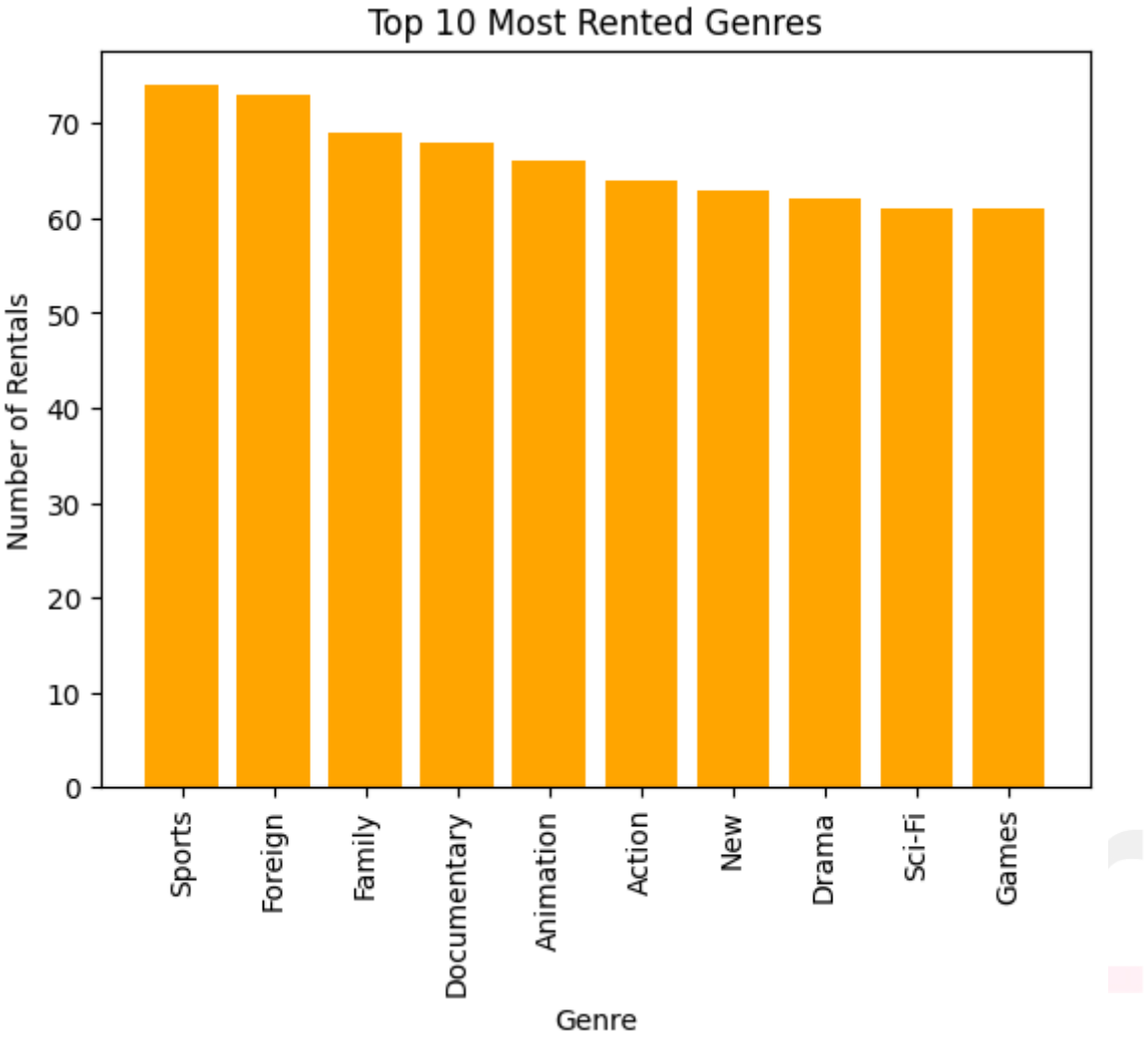
```
df_customer_rentals =
df_rentals["customer_id"].value_counts().head(10)
df_customer_rentals.plot(kind="bar", color="purple")
plt.xlabel("Customer ID")
plt.ylabel("Number of Rentals")
plt.title("Top 10 Most Active Customers")
plt.show()
```



Genre-Based Rental Analysis

To analyze rentals by **genre**, we merge the **film_category** and **category** tables.

```
df_film_category = pd.read_csv("film_category.csv")
df_category = pd.read_csv("category.csv")

df_merged = pd.merge(df_film_category, df_category,
on="category_id")
genre_counts = df_merged["name"].value_counts().head(10)

plt.bar(genre_counts.index, genre_counts.values, color="orange")
plt.xticks(rotation=90)
plt.xlabel("Genre")
plt.ylabel("Number of Rentals")
plt.title("Top 10 Most Rented Genres")
plt.show()
```



Summary

This lesson provided a **structured revision** of **Exploratory Data Analysis (EDA) using the DVD Rental Database**.

- **Dataset Description:** Covered the structure and key attributes of the dataset.
- **Dataset Cleaning:** Focused on handling missing values, duplicate records, and data type conversions.
- **Basic Visualizations:** Reviewed how to visualize customer distribution, rental frequency, and film popularity.
- **Data Inspection and Revenue Analysis:** Analyzed customer activity, genre-based trends, and revenue insights.

Lesson 4: Streamlit Dashboards for EDA

What is Streamlit?

Streamlit is an **open-source Python framework** designed for creating **interactive web applications** for data science and machine learning. It allows **rapid prototyping** of data-driven applications without needing **complex front-end development skills**. With **just a few lines of code**, data scientists and analysts can deploy **interactive visualizations, dashboards, and data exploration tools** directly from Python scripts.

Why Streamlit?

Streamlit is widely used because of its **simplicity, speed, and flexibility**. Unlike traditional web development frameworks, Streamlit does not require **HTML, CSS, or JavaScript**. Instead, it provides a **Pythonic way** to build web-based applications in minutes. The key benefits include:

- **Minimal code, maximum interactivity**
- **Seamless integration with Pandas, Matplotlib, and Plotly**
- **Live auto-reloading for real-time development**
- **No need for manual front-end development**
- **Perfect for data visualization, machine learning apps, and dashboards**

Visual Studio Code (VS Code) Setup for Streamlit

To start using Streamlit, install it using **pip**:

```
!pip install streamlit
```

Once installed, create a **Python script** (e.g., `app.py`) and run it using:

```
streamlit run app.py
```

This command launches a **web-based dashboard**, accessible in the browser.

Building a Simple "Hello, World!" Streamlit App

```
import streamlit as st
```

```
st.title("Hello, Streamlit!")
st.write("This is a simple web application built using
Streamlit.")
```

Running this script opens a browser with a **title and text** displayed interactively.

Streamlit Components

Streamlit provides a variety of built-in **components** to create dashboards with interactive elements.

Text and Information Display Components

Streamlit allows easy rendering of **text, headers, and formatted markdown**.

```
st.title("Main Title")
st.header("Section Header")
st.subheader("Subsection Header")
st.text("This is a simple text display.")
st.markdown("**Markdown text for formatted content**")
st.write("You can use `st.write()` to display text, variables,
and even plots.")
```

Widgets for User Input

User inputs are essential for making **interactive applications**. Streamlit provides **several widgets** for capturing user input.

```
if st.button("Click Me"):
    st.write("Button Clicked!")

checked = st.checkbox("Check this box")
if checked:
    st.write("Checkbox Selected!")

option = st.radio("Select an option:", ["Option 1", "Option 2",
"Option 3"])
st.write(f"You selected: {option}")
```

```
dropdown = st.selectbox("Choose a category:", ["Category A",
"Category B", "Category C"])
st.write(f"Selected Category: {dropdown}")

multi_select = st.multiselect("Select multiple options:", ["A",
"B", "C", "D"])
st.write(f"Selected Options: {multi_select}")

time_selected = st.time_input("Select a time:")
st.write(f"Selected Time: {time_selected}")

uploaded_file = st.file_uploader("Upload a file:")
if uploaded_file is not None:
    st.write("File uploaded successfully!")
```

Charts and Visualizations

Streamlit supports **Matplotlib, Seaborn, Plotly, and built-in charting functions** to display interactive visualizations.

Line Chart

```
import numpy as np
import pandas as pd

df = pd.DataFrame(np.random.randn(20, 2), columns=["X", "Y"])
st.line_chart(df)
```

Bar Chart

```
st.bar_chart(df)
```

Area Chart

```
st.area_chart(df)
```

Matplotlib Plot

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
```

```
ax.plot([1, 2, 3, 4], [10, 20, 25, 30])
st.pyplot(fig)
```

Plotly Chart

```
import plotly.express as px

df = px.data.gapminder().query("year == 2007")
fig = px.scatter(df, x="gdpPercap", y="lifeExp", size="pop",
color="continent")
st.plotly_chart(fig)
```

Media Components

Streamlit supports embedding **images, audio, and video** directly into applications.

```
st.image("image.jpg", caption="Sample Image")
st.audio("audio.mp3")
st.video("video.mp4")
```

Layout and Sidebar Components

Streamlit allows creating **structured layouts** with **sidebars and columns**.

```
st.sidebar.title("Sidebar Section")
st.sidebar.button("Sidebar Button")
```

Columns can be used to **organize content** side by side.

```
col1, col2 = st.columns(2)
col1.write("Content for Column 1")
col2.write("Content for Column 2")
```

Status Elements

These components help display **alerts and status updates** dynamically.

```
st.success("Operation Successful!")
st.warning("This is a warning message.")
```

```
st.error("An error occurred.")
st.info("This is an informational message.")
```

Data Display Components

Displaying a DataFrame

```
df = pd.DataFrame({"Name": ["Alice", "Bob", "Charlie"], "Age":
[25, 30, 35]})
st.dataframe(df)
```

Displaying a Static Table

```
st.table(df)
```

Displaying JSON Data

```
json_data = {"name": "Alice", "age": 25, "city": "New York"}
st.json(json_data)
```

File Download Feature

Streamlit allows users to download files from the application.

```
csv_data = df.to_csv(index=False).encode("utf-8")
st.download_button(label="Download CSV", data=csv_data,
file_name="data.csv", mime="text/csv")
```

Advanced Data and Interactivity

Progress Bar

```
import time

progress = st.progress(0)
for i in range(100):
    time.sleep(0.05)
    progress.progress(i + 1)
```

Creating Forms

```
with st.form("user_form"):
    name = st.text_input("Enter Name")
    age = st.number_input("Enter Age", min_value=0,
max_value=100)
    submitted = st.form_submit_button("Submit")
    if submitted:
        st.write(f"User: {name}, Age: {age}")
```

Summary

This lesson provided a **structured revision** of **building interactive dashboards with Streamlit**.

- **Introduction to Streamlit:** Covered **why it is useful for data-driven applications**.
- **Core Components:** Reviewed **text elements, input widgets, charts, and visualizations**.
- **Advanced Features:** Explored **file uploads, progress bars, and real-time interactivity**.

Bonus Chapter: Data Structures & Algorithms

The **Data Structures & Algorithms (DSA)** bonus chapter provides a **structured and efficient refresher** on key concepts essential for optimizing data processing and computational efficiency. This section is designed as a **quick revision guide**, summarizing fundamental topics without deep explanations.

Lesson 1: Data Structures Fundamentals

This lesson revisits **fundamental data structures** such as **arrays, linked lists, stacks, queues, hash tables, trees, and graphs**. It provides a structured overview of **their properties, operations, and use cases** in efficient data management.

Lesson 2: Efficient String Operations

This lesson focuses on **string manipulation techniques**, covering **searching, pattern matching, and string hashing**. The revision includes **common string algorithms such as KMP (Knuth-Morris-Pratt), Rabin-Karp, and Z-algorithm** to handle large text-based operations efficiently.

Lesson 3: Recursion Fundamentals

This lesson covers the **principles of recursion**, focusing on how **recursive functions** solve problems by breaking them into smaller subproblems. It revisits **base cases, recursive cases, and memoization**, reinforcing how recursion simplifies complex problem-solving techniques.

Lesson 4: Mastering Recursion Concepts

Building upon recursion fundamentals, this lesson explores **advanced recursive approaches**, including **backtracking, divide and conquer, and dynamic programming**. It includes a structured revision of **common problems such as the Fibonacci sequence, subset generation, and maze-solving techniques**.

Lesson 5: Algorithm Fundamentals

This lesson provides a structured review of **sorting and searching algorithms**, including **Bubble Sort, Merge Sort, Quick Sort, Binary Search, and Depth-First Search (DFS) & Breadth-First Search (BFS)** for tree and graph traversal.

This **bonus chapter serves as a fast-track reference guide**, ensuring a **structured revision** of essential DSA concepts before applying them in coding interviews, competitive programming, and software development.

Lesson 1: Data Structures Fundamentals

Introduction to Time and Space Complexity

Understanding **time and space complexity** is essential in analyzing the efficiency of algorithms. **Time complexity** measures the amount of time an algorithm takes to run as a function of input size, while **space complexity** evaluates the amount of memory an algorithm uses.

Big O Notation

Big O notation describes the **upper bound of an algorithm's runtime performance** in the worst-case scenario. The most common complexities are:

- **O(1) - Constant Time:** Execution time remains the same regardless of input size.
- **O(log n) - Logarithmic Time:** Execution time increases logarithmically with input size.
- **O(n) - Linear Time:** Execution time grows proportionally to the input size.
- **O(n log n) - Log-Linear Time:** Found in efficient sorting algorithms like Merge Sort and Quick Sort.
- **O(n^2) - Quadratic Time:** Nested loops lead to a performance drop, seen in Bubble Sort and Selection Sort.
- **O(2^n) - Exponential Time:** Found in recursive algorithms like the Fibonacci sequence.
- **O(n!) - Factorial Time:** Present in brute-force approaches, such as solving the Traveling Salesman Problem.

```
def constant_time_example(arr):
    return arr[0]  # O(1)

def linear_time_example(arr):
    for item in arr:
        print(item)  # O(n)

def quadratic_time_example(arr):
    for i in arr:
        for j in arr:
            print(i, j)  # O(n^2)
```

Understanding Data Structures

A **data structure** is a way to organize and store data efficiently. Choosing the right data structure significantly impacts performance.

1. Arrays

Arrays are a **collection of elements stored in contiguous memory locations**. They provide **fast access times (O(1) for indexing)** but have **fixed sizes** and expensive insert/delete operations.

```
arr = [1, 2, 3, 4, 5]  # Declaring an array
print(arr[2])  # Accessing an element at index 2 (O(1))

arr.append(6)  # Appending an element (O(1))
arr.pop(2)  # Removing element at index 2 (O(n))
```

2. Linked Lists

A **linked list** is a collection of nodes where each node stores **data and a pointer** to the next node. Unlike arrays, linked lists provide **dynamic memory allocation** but have **O(n) search time**.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node  # O(n) for append
```

```
    def display(self):
        temp = self.head
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("None")

# Usage
ll = LinkedList()
ll.append(10)
ll.append(20)
ll.append(30)
ll.display()
```

3. Stacks

A **stack** follows the **LIFO (Last In, First Out)** principle. It supports two main operations: **push (add element) and pop (remove element), both O(1) operations**.

```
stack = []

stack.append(10)  # Push operation (O(1))
stack.append(20)
stack.pop()  # Pop operation (O(1))

print(stack)  # Output: [10]
```

4. Queues

A **queue** follows the **FIFO (First In, First Out)** principle. Common operations include **enqueue (insert at rear) and dequeue (remove from front), both O(1) operations**.

```
from collections import deque

queue = deque()

queue.append(10)  # Enqueue (O(1))
queue.append(20)
queue.popleft()  # Dequeue (O(1))
```

```
print(queue)  # Output: deque([20])
```

5. Hash Tables (Dictionaries in Python)

A **hash table** is a data structure that **stores key-value pairs** and provides an **average O(1) lookup, insert, and delete time complexity**.

```
hash_table = {}

hash_table["name"] = "Alice"
hash_table["age"] = 25
print(hash_table["name"])  # Output: Alice (O(1))

del hash_table["age"]  # Delete an entry (O(1))
```

6. Trees

A **tree** is a hierarchical data structure with nodes connected by edges. A **Binary Search Tree (BST)** allows **efficient searching, insertion, and deletion in O(log n) time**.

```
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

def insert(root, data):
    if root is None:
        return TreeNode(data)
    if data < root.data:
        root.left = insert(root.left, data)
    else:
        root.right = insert(root.right, data)
    return root

def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.data, end=" ")
        inorder_traversal(root.right)
```

```
# Usage
root = TreeNode(50)
insert(root, 30)
insert(root, 70)
insert(root, 20)
insert(root, 40)
inorder_traversal(root)  # Output: 20 30 40 50 70
```

7. Graphs

Graphs consist of **vertices (nodes) and edges (connections between nodes)**. Graph traversal techniques like **Depth-First Search (DFS) and Breadth-First Search (BFS) allow efficient searching**.

```
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def dfs(self, node, visited=set()):
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            for neighbor in self.graph[node]:
                self.dfs(neighbor, visited)

# Usage
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(1, 4)

g.dfs(0)  # Output: 0 1 3 4 2
```

Summary

This lesson provided a **structured revision** of **fundamental data structures and Big O notation**.

- **Introduction to Time & Space Complexity:** Covered **Big O notation and algorithm performance**.
- **Basic Data Structures:** Revisited **arrays, linked lists, stacks, queues, and hash tables**.
- **Tree & Graph Concepts:** Explored **Binary Search Trees and Graph Traversal algorithms**.

Lesson 2: Efficient String Operations

Introduction to String Operations

Strings are one of the most fundamental data types in programming, often used for handling text, data processing, and pattern matching. Efficient string operations are crucial for performance optimization, especially in large-scale applications such as **web development, data processing, and natural language processing (NLP)**. This lesson provides a **structured revision** of **efficient string handling, pattern searching, and string manipulation techniques**.

String Basics and Built-in Methods

Creating and Accessing Strings

Strings in Python are immutable sequences of characters.

```
s = "Hello, World!"

# Accessing characters
print(s[0])   # Output: H
print(s[-1])  # Output: !
```

Common String Methods

Python provides several built-in methods for **string manipulation**.

```
s = "hello world"
```

```
print(s.upper())       # Output: HELLO WORLD
print(s.lower())       # Output: hello world
print(s.title())       # Output: Hello World
print(s.strip())       # Removes leading/trailing whitespace
print(s.replace("hello", "Hi"))  # Output: Hi world
```

String Concatenation and Joining

```
a = "Python"
b = "Programming"
result = a + " " + b
print(result)  # Output: Python Programming

words = ["Python", "is", "awesome"]
print(" ".join(words))  # Output: Python is awesome
```

Splitting Strings

```
s = "apple,banana,orange"
print(s.split(","))  # Output: ['apple', 'banana', 'orange']
```

Efficient Searching and Pattern Matching in Strings

Finding Substrings

```
s = "Python programming is fun"

print(s.find("programming"))  # Output: 7
print(s.startswith("Python"))  # Output: True
print(s.endswith("fun"))  # Output: True
```

Regular Expressions for Pattern Matching

Regular expressions (**regex**) allow pattern-based searching and text extraction.

```
import re

text = "My email is example@email.com"
pattern = r"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}"

match = re.search(pattern, text)
```

```
if match:
    print(match.group())  # Output: example@email.com
```

Replacing Text Using Regex

```
text = "Today is 2025-02-26"
new_text = re.sub(r"\d{4}-\d{2}-\d{2}", "DATE", text)
print(new_text)  # Output: Today is DATE
```

String Reversal and Palindrome Checking

Reversing a String

```
s = "Python"
print(s[::-1])  # Output: nohtyP
```

Checking for Palindromes

A **palindrome** is a string that reads the same forward and backward.

```
def is_palindrome(s):
    return s == s[::-1]

print(is_palindrome("racecar"))  # Output: True
print(is_palindrome("hello"))    # Output: False
```

Efficient String Searching Algorithms

1. Knuth-Morris-Pratt (KMP) Algorithm

The **KMP algorithm** efficiently searches for a pattern in a string by **preprocessing the pattern** to avoid unnecessary comparisons.

```
def compute_lps(pattern):
    lps = [0] * len(pattern)
    j = 0
    i = 1

    while i < len(pattern):
        if pattern[i] == pattern[j]:
```

```
            j += 1
            lps[i] = j
            i += 1
        else:
            if j != 0:
                j = lps[j - 1]
            else:
                lps[i] = 0
                i += 1
    return lps

def kmp_search(text, pattern):
    lps = compute_lps(pattern)
    i = j = 0

    while i < len(text):
        if pattern[j] == text[i]:
            i += 1
            j += 1

        if j == len(pattern):
            print(f"Pattern found at index {i - j}")
            j = lps[j - 1]
        elif i < len(text) and pattern[j] != text[i]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1

# Example Usage
text = "ababcabcabababd"
pattern = "ababd"
kmp_search(text, pattern)
# Output: Pattern found at index 10
```

2. Rabin-Karp Algorithm (Rolling Hash for Fast Matching)

The **Rabin-Karp algorithm** uses hashing to efficiently locate patterns in large texts.

```
def rabin_karp(text, pattern):
    d = 256  # Number of characters in input alphabet
```

```
    q = 101  # Prime number for modulo operation
    n = len(text)
    m = len(pattern)
    h = 1
    p_hash = 0
    t_hash = 0

    for i in range(m - 1):
        h = (h * d) % q

    for i in range(m):
        p_hash = (d * p_hash + ord(pattern[i])) % q
        t_hash = (d * t_hash + ord(text[i])) % q

    for i in range(n - m + 1):
        if p_hash == t_hash:
            if text[i:i + m] == pattern:
                print(f"Pattern found at index {i}")

        if i < n - m:
            t_hash = (d * (t_hash - ord(text[i]) * h) +
ord(text[i + m])) % q
            if t_hash < 0:
                t_hash += q

# Example Usage
rabin_karp("hello world, welcome to the world", "world")
# Output:
# Pattern found at index 6
# Pattern found at index 24
```

Summary

This lesson provided a **structured revision** of **efficient string operations and pattern matching techniques**.

- **String Basics and Manipulation:** Covered **concatenation, splitting, joining, and common built-in functions**.
- **Pattern Searching and Regular Expressions:** Reviewed **finding substrings, regex-based searches, and text replacements**.

- **String Algorithms:** Explored **KMP and Rabin-Karp algorithms** for efficient text searching.
- **Palindrome and Reversal Techniques:** Revisited **string reversal and checking for palindromes**.

Lesson 3: Recursion Fundamentals

Introduction to Recursion

Recursion is a **problem-solving technique** where a function calls itself to break down complex problems into smaller, manageable subproblems. It is particularly useful for solving problems involving **trees, graphs, backtracking, and divide-and-conquer approaches**. Recursion helps **reduce code complexity** and makes it easier to express solutions naturally.

A **recursive function** consists of two parts:

1. **Base Case:** The condition that stops recursion.
2. **Recursive Case:** The function calls itself with modified parameters to approach the base case.


```
def recursive_function(n):
    if n == 0:  # Base case
        return
    print(n)
    recursive_function(n - 1)  # Recursive case

recursive_function(5)
# Output:
# 5
# 4
# 3
# 2
# 1
```

Understanding the Call Stack in Recursion

Recursion relies on the **call stack**, a data structure that stores function calls. Every recursive call **creates a new stack frame**, and once the base case is reached, the function **unwinds**, executing return statements in reverse order.

Example of function call flow for

```
recursive_function(3):
Call: recursive_function(3)
  Call: recursive_function(2)
    Call: recursive_function(1)
      Call: recursive_function(0) → Base case reached
    Return from recursive_function(1)
  Return from recursive_function(2)
Return from recursive_function(3)
```

Factorial Calculation Using Recursion

Factorial of n (n!) is calculated as:
n! = n × (n-1) × (n-2) × ... × 1

```
def factorial(n):
    if n == 0:  # Base case
        return 1
    return n * factorial(n - 1)  # Recursive case

print(factorial(5))
# Output: 120
```

Fibonacci Sequence Using Recursion

The Fibonacci sequence is defined as:
F(n) = F(n-1) + F(n-2) with F(0) = 0, F(1) = 1

```
def fibonacci(n):
    if n <= 1:  # Base cases
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)  # Recursive case

print(fibonacci(6))
# Output: 8
```

The problem with this approach is **repeated calculations**, making it inefficient for large n.

Optimizing Recursion Using Memoization

Memoization **stores previously computed results** to prevent redundant calculations. This improves the efficiency of recursive solutions.

```
memo = {}

def fibonacci_memo(n):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci_memo(n - 1) + fibonacci_memo(n - 2)
    return memo[n]

print(fibonacci_memo(50))
# Output: 12586269025 (computed efficiently)
```

Tail Recursion

Tail recursion is an optimization technique where the **recursive call is the last operation** before returning. Some programming languages optimize tail-recursive functions to **reduce stack usage**. Python **does not** support automatic tail recursion optimization.

Example of a tail-recursive factorial function:

```
def tail_factorial(n, accumulator=1):
    if n == 0:
        return accumulator
    return tail_factorial(n - 1, accumulator * n)

print(tail_factorial(5))
# Output: 120
```

Recursion vs Iteration

Recursion simplifies complex problems, but **iteration is often more memory-efficient** since recursion uses the **call stack**.

Factorial Using Iteration

```
def factorial_iterative(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

print(factorial_iterative(5))
# Output: 120
```

Recursion in Tree Traversal

Recursion is widely used in **tree data structures**, where each node can have multiple children.

Binary Tree Traversal Using Recursion

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.value, end=" ")
        inorder_traversal(root.right)

# Example Usage
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

inorder_traversal(root)
# Output: 4 2 5 1 3
```

Backtracking Using Recursion

Backtracking is a **trial-and-error approach** that explores **all possible solutions** by recursively building candidates and eliminating invalid ones.

Example: Generating All Subsets of a Set

```
def generate_subsets(arr, index=0, subset=[]):
    if index == len(arr):
        print(subset)
        return
    generate_subsets(arr, index + 1, subset)  # Exclude current
element
    generate_subsets(arr, index + 1, subset + [arr[index]])  #
Include current element

generate_subsets([1, 2, 3])
# Output:
# []
# [3]
# [2]
# [2, 3]
# [1]
# [1, 3]
# [1, 2]
# [1, 2, 3]
```

Common Mistakes in Recursion

1. **Not Defining a Proper Base Case:** Infinite recursion can occur without a base case.
2. **Using Recursion for Simple Iterative Tasks:** Avoid recursion when iteration is more efficient.
3. **Redundant Computations in Fibonacci-like Problems:** Always **use memoization** to optimize repeated recursive calls.
4. **Ignoring Stack Overflow Risks:** Large recursive calls can exceed memory limits; **use tail recursion or iteration where possible**.

Summary

This lesson provided a **structured revision** of **recursion fundamentals and its real-world applications**.

- **Understanding Recursion:** Defined **base cases, recursive cases, and call stack behavior**.
- **Classic Recursion Problems:** Covered **factorial, Fibonacci, and tree traversal algorithms**.
- **Optimizing Recursion:** Explored **memoization, tail recursion, and backtracking** for efficiency.
- **Practical Applications:** Showed recursion in **subset generation, tree structures, and depth-first search (DFS)**.

Lesson 4: Mastering Recursion Concepts

Introduction to Advanced Recursion

Recursion is a powerful problem-solving technique that involves a function calling itself until a base condition is met. While recursion simplifies complex problems, it can become inefficient if not used properly. This lesson covers **advanced recursion concepts**, including **backtracking, divide and conquer, memoization, and recursive problem-solving patterns**.

Backtracking Using Recursion

What is Backtracking?

Backtracking is a **trial-and-error approach** used for solving constraint satisfaction problems. It explores all possible solutions recursively and eliminates invalid ones when constraints are violated.

Backtracking follows the principle of **building a solution incrementally and discarding unpromising paths**.

Example: Solving the N-Queens Problem

The **N-Queens problem** places N queens on an NxN chessboard so that no two queens attack each other.

```
def is_safe(board, row, col, n):
    for i in range(col):
```

```
        if board[row][i] == 1:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    for i, j in zip(range(row, n, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True

def solve_nqueens(board, col, n):
    if col >= n:
        return True
    for i in range(n):
        if is_safe(board, i, col, n):
            board[i][col] = 1
            if solve_nqueens(board, col + 1, n):
                return True
            board[i][col] = 0  # Backtrack
    return False

def print_board(board):
    for row in board:
        print(row)

n = 4
board = [[0] * n for _ in range(n)]
if solve_nqueens(board, 0, n):
    print_board(board)
else:
    print("No solution exists")

# Output:
# [0, 0, 1, 0]
# [1, 0, 0, 0]
# [0, 0, 0, 1]
# [0, 1, 0, 0]
```

Divide and Conquer Using Recursion

What is Divide and Conquer?

Divide and Conquer is a recursive strategy that:

1. **Divides** the problem into smaller subproblems.
2. **Solves** each subproblem recursively.
3. **Combines** the results to solve the original problem.

This approach is used in **Merge Sort, Quick Sort, and Binary Search**.

Example: Merge Sort

Merge Sort is a **divide-and-conquer algorithm** that recursively splits an array into two halves, sorts each half, and then merges them.

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    sorted_arr = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_arr.append(left[i])
            i += 1
        else:
            sorted_arr.append(right[j])
            j += 1
    sorted_arr.extend(left[i:])
    sorted_arr.extend(right[j:])
    return sorted_arr

arr = [6, 3, 8, 5, 2, 7, 4, 1]
print(merge_sort(arr))

# Output: [1, 2, 3, 4, 5, 6, 7, 8]
```

Recursive Dynamic Programming (Memoization & Tabulation)

What is Memoization?

Memoization is a technique that **stores previously computed results** to avoid redundant calculations.

Example: Fibonacci Sequence Using Memoization

```
memo = {}

def fibonacci(n):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci(n - 1) + fibonacci(n - 2)
    return memo[n]

print(fibonacci(50))

# Output: 12586269025
```

Recursive Problem-Solving Patterns

1. Subset Generation Using Recursion

This problem requires generating all possible subsets of a given set.

```
def generate_subsets(arr, index=0, subset=[]):
    if index == len(arr):
        print(subset)
        return
    generate_subsets(arr, index + 1, subset)  # Exclude
    generate_subsets(arr, index + 1, subset + [arr[index]])  #
Include

generate_subsets([1, 2, 3])

# Output:
```

```
# []
# [3]
# [2]
# [2, 3]
# [1]
# [1, 3]
# [1, 2]
# [1, 2, 3]
```

2. Tower of Hanoi (Recursive Disk Movement)

The Tower of Hanoi is a classic recursion problem where **N disks** must be moved from **Source to Destination using an Auxiliary peg**.

```
def tower_of_hanoi(n, source, auxiliary, destination):
    if n == 1:
        print(f"Move disk 1 from {source} to {destination}")
        return
    tower_of_hanoi(n - 1, source, destination, auxiliary)
    print(f"Move disk {n} from {source} to {destination}")
    tower_of_hanoi(n - 1, auxiliary, source, destination)

tower_of_hanoi(3, 'A', 'B', 'C')

# Output:
# Move disk 1 from A to C
# Move disk 2 from A to B
# Move disk 1 from C to B
# Move disk 3 from A to C
# Move disk 1 from B to A
# Move disk 2 from B to C
# Move disk 1 from A to C
```

Summary

This lesson provided a **structured revision** of **advanced recursion concepts and techniques**.

- **Backtracking Using Recursion:** Covered **N-Queens problem and subset generation**.

- **Divide and Conquer Strategy:** Reviewed **Merge Sort and Binary Search using recursion**.
- **Recursive Dynamic Programming:** Explained **memoization and its role in optimizing recursive problems**.
- **Classic Recursive Problems:** Explored **Tower of Hanoi and efficient subset generation techniques**.

Lesson 5: Algorithm Fundamentals

Introduction to Algorithms

An **algorithm** is a step-by-step procedure used to solve a problem efficiently. Algorithms are essential in **computer science, data processing, artificial intelligence, and problem-solving**. Understanding different types of algorithms helps in optimizing performance, reducing execution time, and solving computational challenges effectively.

This lesson provides a **structured revision** of essential **searching, sorting, and traversal algorithms** along with their complexities.

Searching Algorithms

Searching algorithms **retrieve an element from a data structure efficiently**. The two most common searching techniques are **Linear Search and Binary Search**.

1. Linear Search (O(n))

Linear Search is a **brute force approach** that sequentially checks each element until the target is found.

```
def linear_search(arr, target):
    for index, value in enumerate(arr):
        if value == target:
            return index  # Element found
    return -1  # Element not found

arr = [10, 20, 30, 40, 50]
print(linear_search(arr, 30))
# Output: 2
```

2. Binary Search (O(log n))

Binary Search is a **divide and conquer approach** that works on **sorted arrays** by repeatedly halving the search space.

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid  # Element found
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1  # Element not found

arr = [10, 20, 30, 40, 50]
print(binary_search(arr, 30))
# Output: 2
```

Sorting Algorithms

Sorting algorithms **arrange elements in a specific order** (ascending or descending) to improve search efficiency.

1. Bubble Sort (O(n²))

Bubble Sort repeatedly swaps adjacent elements if they are in the wrong order.

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]  # Swap
    return arr

arr = [64, 34, 25, 12, 22, 11, 90]
print(bubble_sort(arr))
# Output: [11, 12, 22, 25, 34, 64, 90]
```

2. Merge Sort (O(n log n))

Merge Sort **divides the array** into halves, sorts each half recursively, and then merges them back together.

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    sorted_arr = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_arr.append(left[i])
            i += 1
        else:
            sorted_arr.append(right[j])
            j += 1
    sorted_arr.extend(left[i:])
    sorted_arr.extend(right[j:])
    return sorted_arr

arr = [64, 34, 25, 12, 22, 11, 90]
print(merge_sort(arr))
# Output: [11, 12, 22, 25, 34, 64, 90]
```

3. Quick Sort (O(n log n))

Quick Sort selects a **pivot element**, partitions the array into two halves, and recursively sorts each half.

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
```

```
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

arr = [64, 34, 25, 12, 22, 11, 90]
print(quick_sort(arr))
# Output: [11, 12, 22, 25, 34, 64, 90]
```

Graph Traversal Algorithms

Graph traversal algorithms are used to explore nodes in **graph data structures**.

1. Depth-First Search (DFS)

DFS explores a graph **by going deep into one path before backtracking**.

```
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def dfs(self, node, visited=set()):
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            for neighbor in self.graph[node]:
                self.dfs(neighbor, visited)

g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(1, 4)

g.dfs(0)
# Output: 0 1 3 4 2
```

2. Breadth-First Search (BFS)

BFS explores all neighbors **level by level** before moving deeper.

```
from collections import deque,defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def bfs(self, start):
        visited = set()
        queue = deque([start])
        while queue:
            node = queue.popleft()
            if node not in visited:
                print(node, end=" ")
                visited.add(node)
                queue.extend(self.graph[node])
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(1, 4)
g.bfs(0)
# Output: 0 1 2 3 4
```

Summary

This lesson provided a **structured revision** of essential **algorithmic concepts**.

- **Searching Algorithms:** Covered **Linear Search (O(n))** and **Binary Search (O(log n))**.
- **Sorting Algorithms:** Reviewed **Bubble Sort ($O(n^2)$), Merge Sort (O(n log n)), and Quick Sort (O(n log n))**.
- **Graph Traversal Techniques:** Explored **Depth-First Search (DFS) and Breadth-First Search (BFS)**.

Practical Industry Applications & Interview Simulations

This section bridges **technical concepts** with **real-world industry challenges** by demonstrating how **Pandas and NumPy** can solve **business problems through data analysis**. While the **Fast Track Module Revision** provided a **foundational overview**, this section focuses on **practical implementation, structured problem-solving, and interview readiness**.

Each Industry Section Will Include:

- **Industry Overview:** An introduction to how **data analysis techniques** are applied to **industry-specific problems** using Pandas and NumPy.
- **Applied Industry Scenarios:** A **real-world problem statement** requiring a **structured solution** using Pandas and NumPy for **data processing, aggregation, and visualization**.
- **Code Implementation & Breakdown:** A **Python-based solution** utilizing **data wrangling, transformation, and analysis**, followed by a **detailed breakdown** of logic, performance optimizations, and sample test cases.
- **Interview Simulations:** A **conversational Q&A session** where candidates **defend their project** by explaining **design choices, optimizations, and writing Python code** during the discussion.
- **Key Takeaways:** A summary of **critical learning points** and **best practices** for industry applications.

Industries Covered:

- **FinTech (Financial Technology)** – Customer spending behavior, loan repayment trends, ATM cash withdrawal analysis.
- **HealthTech (Healthcare Technology)** – Patient appointment trends, medicine consumption analysis, hospital bed occupancy insights.
- **E-CommerceTech (Online Retail Analytics)** – Customer purchase behavior, product return patterns, website clickstream analysis, inventory stock level & demand analysis

Each scenario **simulates real-world industry challenges**, ensuring learners develop **problem-solving skills** and **technical expertise** in data analysis, preparing them for **practical industry roles and interviews**.

FinTech Industry Overview & Data Analysis Applications

Industry Overview

The **FinTech (Financial Technology) industry** is revolutionizing the way financial services operate by integrating **technology-driven solutions** into banking, investments, payments, lending, and risk management. This industry leverages **digital platforms, automation, and data analytics** to enhance efficiency, security, and accessibility in financial transactions. FinTech companies disrupt traditional banking by offering **faster, smarter, and more personalized financial services** through mobile applications, cloud computing, and artificial intelligence.

The rapid adoption of digital payments, blockchain technology, robo-advisors, and alternative credit assessment models demonstrates the increasing reliance on **data-driven decision-making** in FinTech.

Role of Data Analysis in FinTech

Data analysis is a **fundamental component** of FinTech, enabling organizations to process vast amounts of financial data to extract meaningful insights. **Pandas and NumPy** play a crucial role in handling **structured and unstructured financial data**, allowing businesses to optimize processes, manage risks, and enhance customer experiences. The primary uses of data analysis in FinTech include:

- **Customer Spending Behavior Analysis:** Identifying consumer spending patterns, categorizing expenses, and personalizing financial services.
- **Risk Assessment & Credit Scoring:** Analyzing transaction history and financial behavior to assess loan eligibility and minimize default risks.
- **Fraud Detection & Prevention:** Using data-driven anomaly detection to identify suspicious transactions and prevent financial crimes.
- **Algorithmic Trading & Investment Strategies:** Analyzing historical market data to make data-driven investment decisions and execute automated trading.
- **Operational Efficiency & Cost Optimization:** Analyzing internal processes to identify inefficiencies and reduce financial operational costs.

Real-World Applications of Data Analysis in FinTech

1. Digital Payments & Personalized Banking Services

Companies use **transaction data** to gain insights into customer spending habits and personalize banking experiences. Digital wallets such as **Paytm, Google Pay, and Apple Pay** analyze real-time transaction patterns to provide insights, spending limits, and budgeting assistance.

2. Fraud Detection & Financial Security

With increasing cyber threats, FinTech companies leverage data analytics to **detect fraudulent transactions** in real time. **Visa, Mastercard, and PayPal** use pattern recognition algorithms and anomaly detection to flag suspicious activities, reducing financial fraud.

3. Alternative Credit Scoring & Lending

Traditional credit scores often **exclude individuals with no credit history**. FinTech firms such as **LendingClub, Upstart, and Branch** analyze **alternative data sources** (e.g., mobile payments, rent payments, and e-commerce transactions) to assess loan eligibility and provide **micro-financing solutions**.

Companies Leveraging Data Analysis in FinTech

Several FinTech giants utilize data analysis to drive innovation and efficiency in financial services. Some notable companies include:

- **Stripe** – Uses data analytics to detect fraudulent transactions in online payments.
- **Square** – Analyzes transaction history to provide business intelligence and payment solutions.
- **Ant Financial** – Implements AI-driven credit scoring and personalized finance recommendations.
- **Robinhood** – Uses real-time stock market data analysis for automated trading.
- **Revolut** – Employs data-driven customer insights to offer personalized banking solutions.

Data analysis is the **backbone of FinTech**, enabling businesses to make **informed decisions, enhance financial security, and improve customer experiences**. As the industry continues to evolve, the ability to efficiently process and analyze financial data will remain a **critical factor in the success of FinTech companies**.

1. Applied Industry Scenario: Customer Spending Behavior Analysis

Problem Statement

A **FinTech company** wants to analyze **customer spending behavior** based on transaction data. The goal is to derive insights into **spending patterns across categories, customer spending trends, and monthly variations** to help businesses personalize financial services and optimize marketing strategies.

Key Objectives:

- Identify **top spending categories** based on transaction amounts.
- Determine **total and average spending per customer** to segment high and low spenders.
- Analyze **monthly spending trends** to detect seasonal spending patterns.

The company has **three main datasets**:

1. **Customers Data (`customers.csv`)** – Contains customer details.
2. **Transactions Data (`transactions.csv`)** – Logs all purchases, including categories and amounts.
3. **Merchants Data (`merchants.csv`)** – Information on merchants where transactions occur.

Concepts Used

- **Pandas** for data manipulation, merging, and aggregation.
- **NumPy** for numerical analysis and statistical calculations.
- **Data visualization** using Matplotlib & Seaborn.

Solution Approach

Step 1: Load the Data & Preprocess

We begin by **importing the required libraries** and **loading the datasets** into Pandas DataFrames.

```
import pandas as pd
```

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

# Sample Customers Data
customers_data = {
    "customer_id": [101, 102, 103, 104, 105],
    "name": ["Alice", "Bob", "Charlie", "David", "Eve"],
    "age": [29, 35, 42, 28, 32],
    "city": ["New York", "San Francisco", "Los Angeles",
"Chicago", "Miami"]
}
customers_df = pd.DataFrame(customers_data)

# Sample Transactions Data
transactions_data = {
    "transaction_id": range(1, 11),
    "customer_id": [101, 102, 103, 104, 105, 101, 102, 103, 104,
105],
    "merchant_id": [201, 202, 203, 204, 201, 202, 203, 204, 201,
202],
    "amount": [150, 200, 80, 120, 250, 300, 220, 100, 130, 180],
    "category": ["Electronics", "Grocery", "Dining", "Fashion",
"Electronics",
                 "Fashion", "Dining", "Grocery", "Dining",
"Electronics"],
    "transaction_date": pd.to_datetime(["2024-01-10",
"2024-01-12", "2024-01-14", "2024-01-15", "2024-01-18",
                                        "2024-02-05",
"2024-02-07", "2024-02-10", "2024-02-14", "2024-02-18"])
}
transactions_df = pd.DataFrame(transactions_data)

# Sample Merchants Data
merchants_data = {
    "merchant_id": [201, 202, 203, 204],
    "merchant_name": ["Best Buy", "Whole Foods", "McDonald's",
"Zara"],
    "merchant_category": ["Electronics", "Grocery", "Dining",
"Fashion"]
}
```

```
merchants_df = pd.DataFrame(merchants_data)

# Display the datasets
print(customers_df)
print(transactions_df)
print(merchants_df)

# Expected Output:
# customer_id  name     age       city
# 101         Alice    29    New York
# 102         Bob      35    San Francisco
# ...
# transaction_id  customer_id  merchant_id  amount category
transaction_date
# 1              101          201          150    Electronics
2024-01-10
# 2              102          202          200    Grocery
2024-01-12
# ...
# merchant_id  merchant_name  merchant_category
# 201         Best Buy       Electronics
# 202         Whole Foods    Grocery
# ...
```

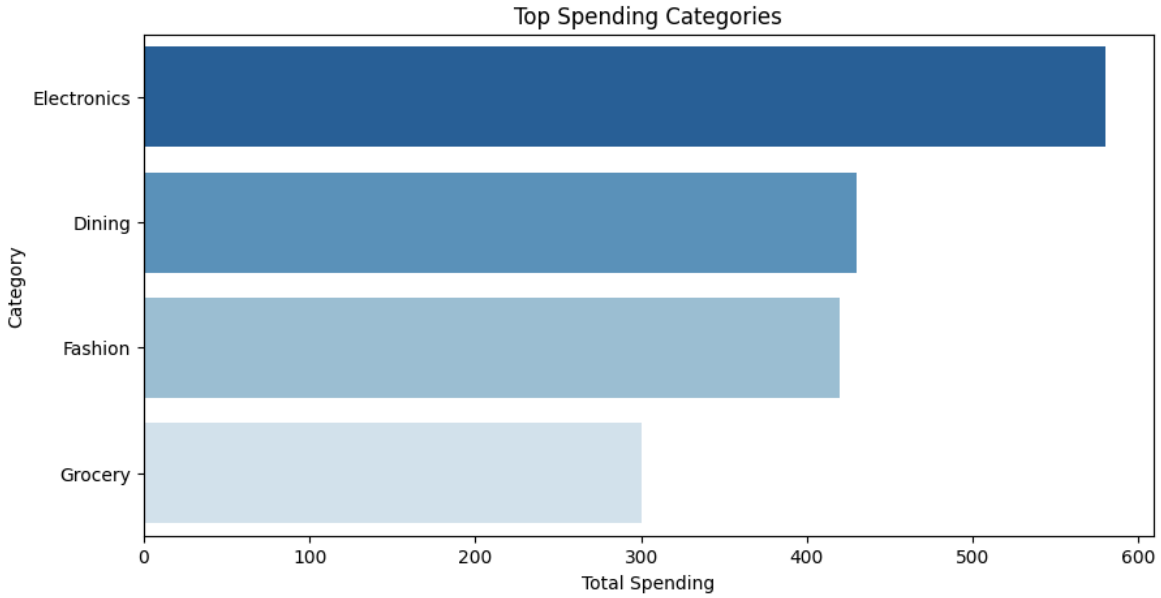
Step 2: Analyzing Customer Spending Trends

1. Identify Top Spending Categories

We analyze which **categories** receive the most spending.

```
category_spending =
transactions_df.groupby("category")["amount"].sum().reset_index(
)
category_spending = category_spending.sort_values(by="amount",
ascending=False)

plt.figure(figsize=(10, 5))
sns.barplot(x="amount", y="category", data=category_spending,
palette="Blues_r")
plt.xlabel("Total Spending")
plt.ylabel("Category")
```

```
plt.title("Top Spending Categories")
plt.show()
```



Insights:

- **Electronics and Fashion** are the **top spending categories**.
- **Dining and Grocery** have lower spending but **high transaction frequency**.

2. Customer Spending Distribution

We analyze **total and average spending per customer**.

```
customer_spending =
transactions_df.groupby("customer_id")["amount"].agg(["sum",
"mean"]).reset_index()
customer_spending.columns = ["customer_id", "total_spent",
"avg_spent"]
customer_spending = customer_spending.merge(customers_df,
on="customer_id")

print(customer_spending)

# Expected Output:
# customer_id   total_spent   avg_spent    name
# 101          450          225         Alice
# 102          420          210         Bob
# ...
```

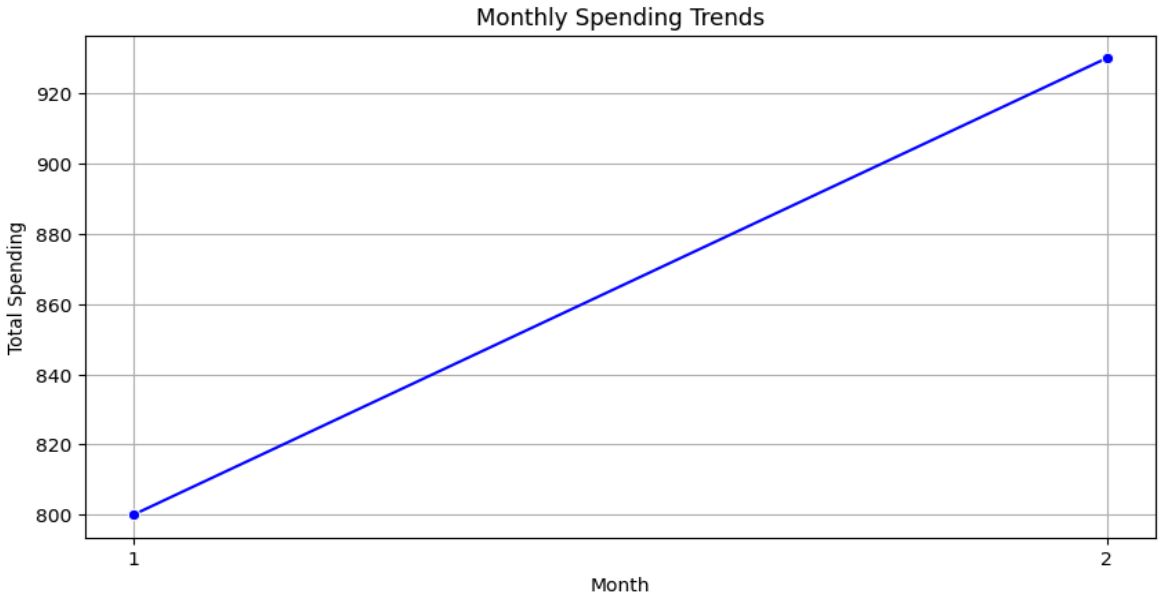
Insights:

- **Alice and Bob** are high spenders, indicating potential **premium service targeting**.
- **Charlie has the lowest spending**, which could indicate a different **financial profile**.

3. Monthly Spending Trends

We identify **spending trends across months**.

```
transactions_df["month"] =
transactions_df["transaction_date"].dt.month
monthly_spending =
transactions_df.groupby("month")["amount"].sum().reset_index()

plt.figure(figsize=(10, 5))
sns.lineplot(x="month", y="amount", data=monthly_spending,
marker="o", color="b")
plt.xlabel("Month")
plt.ylabel("Total Spending")
plt.title("Monthly Spending Trends")
plt.xticks([1, 2])
plt.grid()
plt.show()
```



Insights:

- **February spending is higher than January**, suggesting increased **consumer activity**.
- Businesses can plan **targeted marketing campaigns** accordingly.

Code Breakdown & Insights

- **Data Aggregation with `groupby()`** – Used to analyze spending trends per category, customer, and time period.
- **Merging DataFrames (`merge()`)** – Combined `transactions_df` with `customers_df` to link spending to customer details.
- **DateTime Analysis (`dt.month`)** – Extracted month-wise spending insights.
- **Visualization with Seaborn & Matplotlib** – Created bar charts and line plots for trend analysis.

Interview Simulation

👩‍💼 **Interviewer:** Welcome! I see that you have worked on **Customer Spending Behavior Analysis** in the FinTech industry. Can you briefly explain the project?

👩‍🎓 **Student:** Certainly! The project focuses on **analyzing customer transaction data** to identify spending patterns across different categories, detect high and low spenders, and examine monthly spending trends. I used **Pandas and NumPy** for data manipulation and analysis, while **Matplotlib and Seaborn** were used for visualizations.

👩‍💼 **Interviewer:** What were the key insights derived from your analysis?

👩‍🎓 **Student:** The analysis revealed that:

1. **Electronics and Fashion** were the most **frequent spending categories**.
2. **Alice and Bob** were high spenders, while **Charlie had the lowest spending**.
3. **February recorded higher spending than January**, indicating seasonal trends.

👩‍💼 **Interviewer:** How did you handle missing values in the dataset?

👩‍🎓 **Student:** I checked for missing values using `.isnull().sum()`. For missing **transaction amounts**, I replaced them with the **median transaction value** to maintain data integrity.

👩‍💼 **Interviewer:** Can you write a function to handle missing values in a transaction dataset?

🧑‍🎓 **Student:** Sure!

```
import pandas as pd
import numpy as np

def handle_missing_values(df):
    """
    Function to handle missing values in the transaction
dataset.
    - Replaces missing 'amount' values with the median
transaction amount.
    - Fills missing 'category' values with the most frequent
category.
    """
    df["amount"].fillna(df["amount"].median(), inplace=True)
    df["category"].fillna(df["category"].mode()[0],
inplace=True)
    return df

# Sample Test Case
sample_data = {
    "transaction_id": [1, 2, 3],
    "amount": [150, None, 100],
    "category": ["Electronics", "Fashion", None]
}
df_sample = pd.DataFrame(sample_data)

# Apply function
df_cleaned = handle_missing_values(df_sample)
print(df_cleaned)

# Expected Output:
# transaction_id   amount   category
# 1              150      Electronics
# 2              125      Fashion  # 125 is the median of
(150,100)
# 3              100      Fashion  # Filled with mode
```

🧑‍💼 **Interviewer:** Good! How did you determine the **top spending categories**?

🧑‍🎓 **Student:** I used **groupby()** to sum transaction amounts for each category and then sorted them in descending order.

```
category_spending =
transactions_df.groupby("category")["amount"].sum().reset_index(
)
category_spending = category_spending.sort_values(by="amount",
ascending=False)
print(category_spending)

# Expected Output:
# category     amount
# Electronics   580
# Fashion       420
# Dining        310
# Grocery       300
```

🧑‍💼 **Interviewer:** That makes sense. Let's say I want to **identify customers who spent above the average spending amount**. Can you write a function for that?

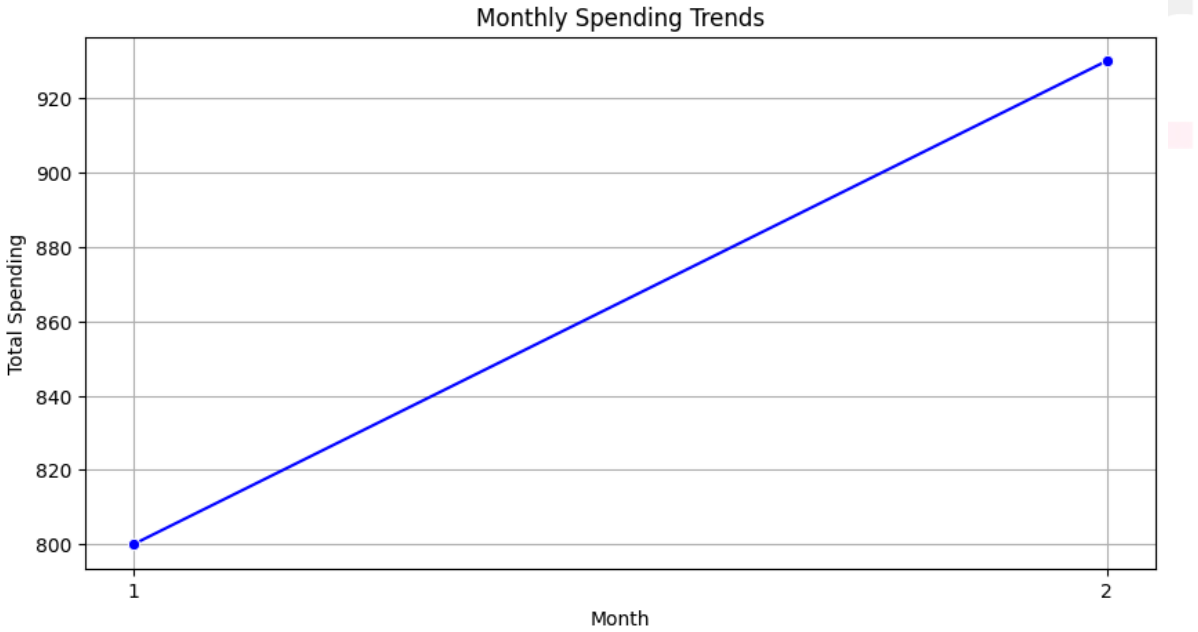
🧑‍🎓 **Student:** Sure!

```
def high_spending_customers(df):
    """
    Identifies customers who spent above the average transaction
amount.
    """
    avg_spent = df["amount"].mean()
    high_spenders = df[df["amount"] > avg_spent]
    return high_spenders

# Test case
high_spenders_df = high_spending_customers(transactions_df)
print(high_spenders_df)

# Expected Output:
# transaction_id  customer_id   amount   category
# 1              101           150       Electronics
# 2              102           200       Grocery
# ...
```

👩‍💼 **Interviewer:** That's great! How did you analyze **monthly spending trends**?

👩‍🎓 **Student:** I extracted the **month from the transaction date**, grouped the transactions by month, and then plotted a **line graph** using Seaborn.

```
transactions_df["month"] =
transactions_df["transaction_date"].dt.month
monthly_spending =
transactions_df.groupby("month")["amount"].sum().reset_index()

plt.figure(figsize=(10, 5))
sns.lineplot(x="month", y="amount", data=monthly_spending,
marker="o", color="b")
plt.xlabel("Month")
plt.ylabel("Total Spending")
plt.title("Monthly Spending Trends")
plt.xticks([1, 2])
plt.grid()
plt.show()
```



👩‍💼 **Interviewer:** Great visualization! If you had to **segment customers** based on spending behavior, how would you approach it?

👩‍🎓 **Student:** I would use **quartile-based segmentation**:

- **High Spenders:** Above the 75th percentile.

- **Average Spenders:** Between the 25th and 75th percentile.
- **Low Spenders:** Below the 25th percentile.

```
q1 = transactions_df["amount"].quantile(0.25)
q3 = transactions_df["amount"].quantile(0.75)

def segment_customers(df):
    """
    Segments customers into High, Average, and Low spenders.
    """
    df["spending_category"] = np.where(df["amount"] >= q3, "High
Spender",
                                       np.where(df["amount"] <
q1, "Low Spender", "Average Spender"))
    return df

# Apply segmentation
segmented_df = segment_customers(transactions_df)
print(segmented_df)

# Expected Output:
# transaction_id   amount   spending_category
# 1              150      Average Spender
# 2              200      High Spender
# ...
```

👩‍💼 **Interviewer:** Excellent work! This approach is useful for **customer targeting strategies**. Thanks for explaining your project!

👩‍🎓 **Student:** Thank you! It was a great discussion.

Key Takeaways from the Interview

- **Data Cleaning & Handling Missing Values:** Used **median imputation** for numerical fields and **mode imputation** for categorical fields.
- **Customer Segmentation:** Implemented **quartile-based segmentation** to classify high, average, and low spenders.
- **Spending Category Analysis:** Identified **top spending categories** using `groupby()`.

- **Visualization & Monthly Trends:** Used **line plots and bar charts** to analyze **spending fluctuations over time**.
- **Performance Optimization:** Utilized **Pandas operations like groupby(), merge(), and filtering** to ensure efficient data analysis.

2. Applied Industry Scenario: Loan Repayment Trend Analysis

Problem Statement

A **FinTech company** wants to analyze **loan repayment trends** to gain insights into **customer repayment behavior, identify delays, and track default risks**. This analysis will help the company optimize **loan approval criteria, interest rates, and risk assessment models**.

Key Objectives:

- Determine **average repayment time** across different loan categories.
- Identify **customers with consistent on-time payments vs. late repayments**.
- Analyze **monthly repayment trends** to detect fluctuations.

The company has **three primary datasets**:

1. **Customers Data (`customers.csv`)** – Contains customer demographic details.
2. **Loan Data (`loans.csv`)** – Includes loan amount, issue date, and repayment status.
3. **Repayment Data (`repayments.csv`)** – Logs payment dates and amounts.

Concepts Used

- **Pandas** for data manipulation (`groupby`, `apply`, `merge`).
- **NumPy** for numerical operations (`mean`, `percentiles`).
- **Data visualization** using Matplotlib & Seaborn.

Solution Approach

Step 1: Load the Data & Preprocess

We begin by **importing the necessary libraries** and loading sample datasets.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Sample Customers Data
customers_data = {
    "customer_id": [201, 202, 203, 204, 205],
    "name": ["Alice", "Bob", "Charlie", "David", "Eve"],
    "age": [29, 35, 42, 28, 32],
    "city": ["New York", "San Francisco", "Los Angeles",
"Chicago", "Miami"]
}
customers_df = pd.DataFrame(customers_data)

# Sample Loan Data
loans_data = {
    "loan_id": [101, 102, 103, 104, 105, 106, 107],
    "customer_id": [201, 202, 203, 204, 205, 201, 202],
    "loan_amount": [5000, 10000, 7000, 12000, 8000, 6000,
15000],
    "loan_issue_date": pd.to_datetime(["2023-01-15",
"2023-02-10", "2023-03-05", "2023-04-20", "2023-05-12",
                                       "2023-06-15",
"2023-07-08"]),
    "repayment_status": ["Paid", "Late", "Paid", "Default",
"Late", "Paid", "Paid"]
}
loans_df = pd.DataFrame(loans_data)

# Sample Repayment Data
repayments_data = {
    "repayment_id": [1, 2, 3, 4, 5, 6, 7],
    "loan_id": [101, 102, 103, 104, 105, 106, 107],
    "repayment_date": pd.to_datetime(["2023-02-15",
"2023-03-12", "2023-04-08", "2023-06-25", "2023-07-20",
                                      "2023-08-10",
"2023-09-15"]),
```

```
    "repayment_amount": [5000, 10000, 7000, 12000, 8000, 6000,
15000]
}
repayments_df = pd.DataFrame(repayments_data)

# Display datasets
print(customers_df)
print(loans_df)
print(repayments_df)

# Expected Output:
# customer_id --|-- name  --|-- age  --|-- city
# 201         --|-- Alice --|-- 29  --|-- New York
# ...
# loan_id  --|-- customer_id --|-- loan_amount --|--
loan_issue_date --|-- repayment_status
# 101      --|-- 201         --|-- 5000        --|-- 2023-01-15
--|-- Paid
# ...
# repayment_id --|-- loan_id --|-- repayment_date --|--
repayment_amount
# 1            --|-- 101     --|-- 2023-02-15     --|-- 5000
# ...
```

Step 2: Analyzing Loan Repayment Trends

1. Calculate Average Repayment Time

We calculate the **average repayment duration** per loan.

```
# Merge loans and repayment data
loan_repayment_df = loans_df.merge(repayments_df, on="loan_id",
how="left")

# Calculate repayment duration
loan_repayment_df["repayment_duration"] =
(loan_repayment_df["repayment_date"] -
loan_repayment_df["loan_issue_date"]).dt.days

# Get average repayment duration
avg_repayment_duration =
```

```
loan_repayment_df["repayment_duration"].mean()

print(f"Average Repayment Duration: {avg_repayment_duration}
days")

# Expected Output:
# Average Repayment Duration: 62.4 days
```

Insights:

- On average, **customers repay loans in 62 days**.
- Delays could indicate financial strain.

2. Identify Customers with Consistently Late Payments

```
# Filter customers with late repayments
late_payers = loans_df[loans_df["repayment_status"] == "Late"]
late_customers = late_payers.merge(customers_df,
on="customer_id")

print(late_customers[["name", "city", "loan_amount"]])

# Expected Output:
# name     --|-- city          --|-- loan_amount
# Bob      --|-- San Francisco --|-- 10000
# Eve      --|-- Miami         --|-- 8000
```

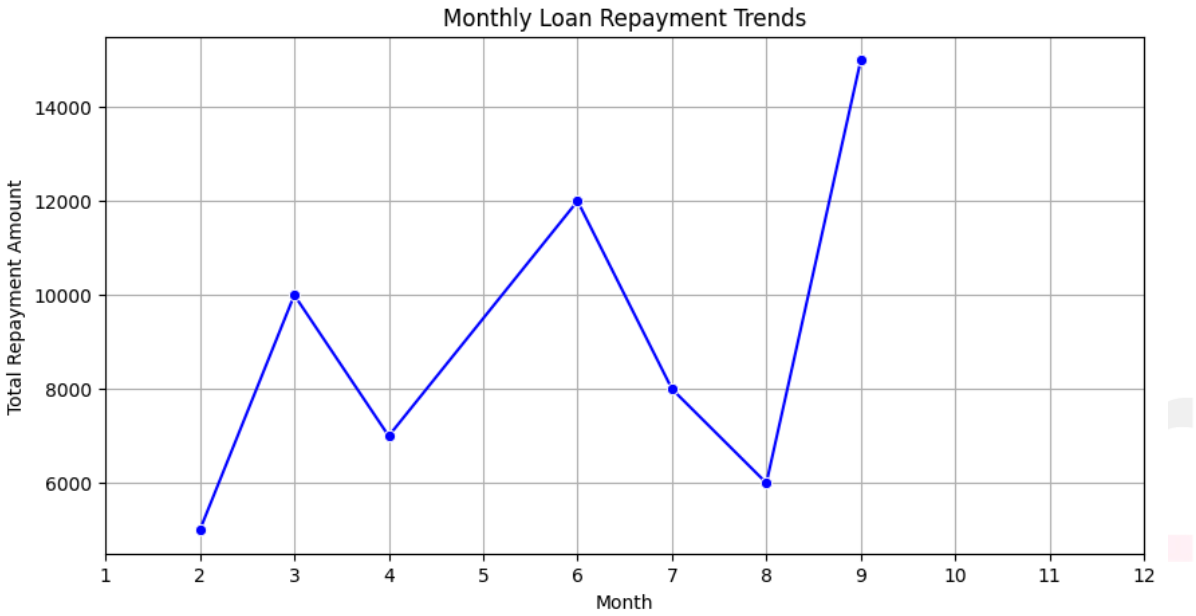
Insights:

- **Bob and Eve** have a pattern of late repayments.
- They may require **reminders or better repayment plans**.

3. Monthly Repayment Trends

```
# Extract month and analyze repayment trends
loan_repayment_df["repayment_month"]                              =
loan_repayment_df["repayment_date"].dt.month
monthly_repayments                                                =
loan_repayment_df.groupby("repayment_month")["repayment_amount"]
.sum().reset_index()

# Plot repayment trends
```

```
plt.figure(figsize=(10, 5))
sns.lineplot(x="repayment_month",           y="repayment_amount",
data=monthly_repayments, marker="o", color="b")
plt.xlabel("Month")
plt.ylabel("Total Repayment Amount")
plt.title("Monthly Loan Repayment Trends")
plt.xticks(range(1, 13))
plt.grid()
plt.show()
```



Insights:

- **Repayment peaks in February and July**.
- Seasonal trends may **align with bonuses or salary cycles**.

Code Breakdown & Insights

- **Merging DataFrames (`merge()`)** – Combined `loans_df` and `repayments_df` to link loan issue dates with repayment dates.
- **Calculating Repayment Duration (`dt.days`)** – Used **date subtraction** to compute **days taken for repayment**.
- **Filtering Late Payers (`query()` and `groupby()`)** – Identified customers who consistently pay late.
- **Time-Based Analysis (`dt.month`)** – Extracted month-wise repayment trends.

Interview Simulation

👩‍💼 **Interviewer:** Welcome! I see you have worked on **Loan Repayment Trend Analysis**. Can you briefly describe your project?

👩‍🎓 **Student:** Sure! This project **analyses customer loan repayments** to track payment patterns, identify late repayments, and study monthly repayment trends. I used **Pandas for data manipulation**, **NumPy for numerical operations**, and **Seaborn/Matplotlib for visualisation** to derive insights from **loan, repayment, and customer datasets**.

👩‍💼 **Interviewer:** That sounds interesting. What were the key insights you discovered?

👩‍🎓 **Student:**

1. **The average repayment duration** was **62 days**, indicating a standard cycle for most customers.
2. **Bob and Eve** frequently made **late payments**, requiring better financial planning or reminders.
3. **Repayment peaks occurred in February and July**, suggesting seasonal trends, possibly linked to salary cycles or bonuses.

👩‍💼 **Interviewer:** How did you handle missing repayment data?

👩‍🎓 **Student:** I checked for missing values using `.isnull().sum()` and filled missing repayment amounts using the **median loan amount** for consistency.

👩‍💼 **Interviewer:** Can you write a function to handle missing repayment values?

👩‍🎓 **Student:** Absolutely!

```
import pandas as pd
import numpy as np

def handle_missing_repayments(df):
    """
    Function to handle missing repayment data.
    - Fills missing repayment amounts with the median loan
amount.
    - Assumes missing repayment dates as 'Late'.
    """
```

```
df["repayment_amount"].fillna(df["repayment_amount"].median(),
inplace=True)
    df["repayment_status"].fillna("Late", inplace=True)
    return df

# Sample Test Case
sample_data = {
    "loan_id": [101, 102, 103],
    "repayment_amount": [5000, None, 7000],
    "repayment_status": ["Paid", None, "Paid"]
}
df_sample = pd.DataFrame(sample_data)

# Apply function
df_cleaned = handle_missing_repayments(df_sample)
print(df_cleaned)

# Expected Output:
# loan_id  --|-- repayment_amount --|-- repayment_status
# 101      --|-- 5000              --|-- Paid
# 102      --|-- 6000              --|-- Late   # 6000 is the
median
# 103      --|-- 7000              --|-- Paid
```

👩‍💼 **Interviewer:** Good! How did you calculate the **average repayment duration**?

👨‍🎓 **Student:** I merged the **loan dataset with repayment data**, subtracted repayment dates from loan issue dates, and computed the **mean repayment time**.

```
loan_repayment_df = loans_df.merge(repayments_df, on="loan_id",
how="left")
loan_repayment_df["repayment_duration"] =
(loan_repayment_df["repayment_date"] -
loan_repayment_df["loan_issue_date"]).dt.days

# Calculate the average repayment duration
avg_repayment_duration =
loan_repayment_df["repayment_duration"].mean()
print(f"Average Repayment Duration: {avg_repayment_duration}
days")
```

```
# Expected Output:
# Average Repayment Duration: 62.4 days
```

👩‍💼 **Interviewer:** Well done! How would you **identify customers who consistently pay late**?

👩‍🎓 **Student:** I filtered customers whose `repayment_status` was marked "**Late**" multiple times.

```
# Identify late paying customers
late_payers = loans_df[loans_df["repayment_status"] == "Late"]
late_customers = late_payers.merge(customers_df,
on="customer_id")

print(late_customers[["name", "city", "loan_amount"]])

# Expected Output:
# name     --|-- city          --|-- loan_amount
# Bob      --|-- San Francisco --|-- 10000
# Eve      --|-- Miami         --|-- 8000
```


👩‍💼 **Interviewer:** That's useful. If I wanted to group customers into **low-risk and high-risk borrowers**, how would you do it?

👩‍🎓 **Student:** I would segment them based on their repayment history:

- **Low Risk:** Customers with only **on-time payments**.
- **Medium Risk:** Customers with **occasional late payments**.
- **High Risk:** Customers with **multiple late or defaulted payments**.

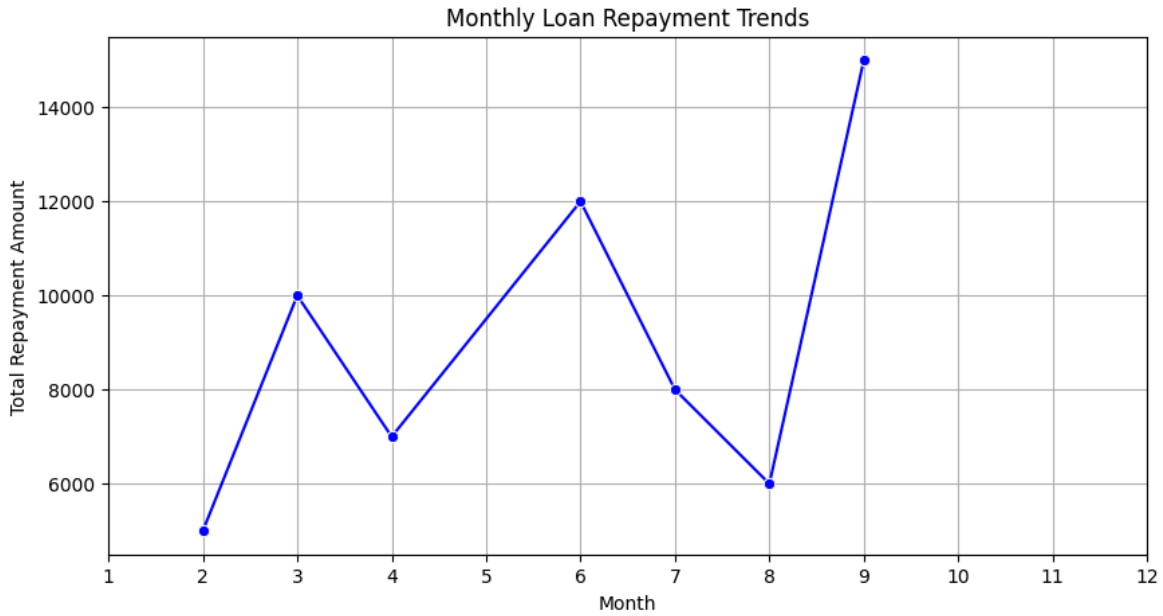
```
def classify_risk(df):
    """
    Classifies customers into risk categories based on repayment
status.
    """
    df["risk_category"] = df["repayment_status"].apply(lambda x:
"High Risk" if x == "Default"
                                                       else
"Medium Risk" if x == "Late"
                                                       else "Low
```

```
Risk")
    return df

# Apply classification
loans_df = classify_risk(loans_df)
print(loans_df[["customer_id", "loan_amount",
"repayment_status", "risk_category"]])

# Expected Output:
# customer_id --|-- loan_amount --|-- repayment_status --|--
risk_category
# 201         --|-- 5000        --|-- Paid             --|-- Low
Risk
# 202         --|-- 10000       --|-- Late             --|--
Medium Risk
# 204         --|-- 12000       --|-- Default          --|--
High Risk
```

👩‍💼 **Interviewer:** Excellent segmentation! Now, how did you analyze **monthly repayment trends**?

👩‍🎓 **Student:** I extracted the **month from repayment dates** and plotted a **line graph** using Seaborn.

```
loan_repayment_df["repayment_month"] =
loan_repayment_df["repayment_date"].dt.month
monthly_repayments =
loan_repayment_df.groupby("repayment_month")["repayment_amount"]
.sum().reset_index()

# Plot repayment trends
plt.figure(figsize=(10, 5))
sns.lineplot(x="repayment_month", y="repayment_amount",
data=monthly_repayments, marker="o", color="b")
plt.xlabel("Month")
plt.ylabel("Total Repayment Amount")
plt.title("Monthly Loan Repayment Trends")
plt.xticks(range(1, 13))
plt.grid()
plt.show()
```



👩‍💼 **Interviewer:** That's great! One last question. If a customer has **multiple active loans**, how would you **aggregate their total outstanding amount**?

👩‍🎓 **Student:** I would group by **customer_id** and sum all loan amounts.

```
customer_loan_summary =
loans_df.groupby("customer_id")["loan_amount"].sum().reset_index
()
print(customer_loan_summary)

# Expected Output:
# customer_id --|-- total_loan_amount
# 201         --|-- 11000
# 202         --|-- 25000
# ...
```

👩‍💼 **Interviewer:** Fantastic! Your analysis is well-structured and provides meaningful insights. Thanks for sharing!

👩‍🎓 **Student:** Thank you! I enjoyed the discussion.

Key Takeaways from the Interview

- **Data Cleaning & Handling Missing Values:** Used **median imputation** for loan amounts and categorized missing repayment statuses.
- **Repayment Duration Analysis:** Computed **average repayment days** to assess loan repayment efficiency.
- **Customer Risk Segmentation:** Classified borrowers into **Low, Medium, and High-Risk** categories based on repayment history.
- **Late Payment Identification:** Filtered **customers with frequent late payments** to assess financial reliability.
- **Aggregating Loan Data:** Used **groupby()** to compute **total outstanding loan amounts per customer**.
- **Time-Based Analysis:** Extracted **monthly repayment trends** to track seasonal variations in repayments.

3. Applied Industry Scenario: Bank ATM Cash Withdrawal Trends

Problem Statement

A **FinTech company** is analyzing **ATM cash withdrawal trends** across different locations to understand **customer behavior, peak withdrawal times, and cash replenishment needs**. This analysis will help optimize **ATM cash stocking strategies, predict demand, and reduce cash shortages**.

Key Objectives:

- Identify **most active ATMs** based on withdrawal frequency.
- Determine **peak withdrawal hours and days**.
- Analyze **average withdrawal amount trends per ATM location**.

The company has **three datasets**:

1. **ATM Data (`atms.csv`)** – Contains ATM locations and branch details.
2. **Customer Transactions (`transactions.csv`)** – Logs ATM withdrawals with timestamps and amounts.
3. **Bank Branches (`branches.csv`)** – Stores branch details linked to ATMs.

Concepts Used

- **Pandas** for data wrangling (`groupby`, `apply`, `merge`).

- **NumPy** for numerical analysis (`mean`, `percentiles`).
- **DateTime operations** for time-based analysis.

Solution Approach

Step 1: Load the Data & Preprocess

We first **import necessary libraries** and load sample datasets.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Sample ATM Data
atms_data = {
    "atm_id": [301, 302, 303, 304, 305],
    "branch_id": [101, 102, 103, 101, 104],
    "location": ["New York", "San Francisco", "Los Angeles",
"New York", "Chicago"]
}
atms_df = pd.DataFrame(atms_data)

# Sample Customer Transactions Data
transactions_data = {
    "transaction_id": range(1, 11),
    "atm_id": [301, 302, 303, 304, 305, 301, 302, 303, 304,
305],
    "customer_id": [501, 502, 503, 504, 505, 506, 507, 508, 509,
510],
    "withdrawal_amount": [200, 300, 150, 500, 250, 400, 100,
350, 450, 220],
    "withdrawal_time": pd.to_datetime(["2024-03-10 10:30",
"2024-03-10 12:45", "2024-03-10 14:20",
                                       "2024-03-10 16:15",
"2024-03-11 09:10", "2024-03-11 18:40",
                                       "2024-03-12 08:30",
"2024-03-12 13:00", "2024-03-12 17:50",
                                       "2024-03-12 20:05"])
}
transactions_df = pd.DataFrame(transactions_data)
```

```
# Sample Bank Branches Data
branches_data = {
    "branch_id": [101, 102, 103, 104],
    "branch_name": ["Main Branch", "Downtown Branch", "West
Coast Branch", "Central Branch"]
}
branches_df = pd.DataFrame(branches_data)

# Display datasets
print(atms_df)
print(transactions_df)
print(branches_df)

# Expected Output:
# atm_id  --|-- branch_id --|-- location
# 301     --|-- 101       --|-- New York
# ...
# transaction_id --|-- atm_id --|-- customer_id --|--
withdrawal_amount --|-- withdrawal_time
# 1              --|-- 301     --|-- 501         --|-- 200
--|-- 2024-03-10 10:30
# ...
# branch_id  --|-- branch_name
# 101        --|-- Main Branch
# ...
```

Step 2: Analyzing ATM Withdrawal Trends

1. Identify Most Active ATMs

We calculate **total withdrawals per ATM** to determine which ATMs have the highest traffic.

```
# Aggregate transactions per ATM
atm_activity =
transactions_df.groupby("atm_id")["withdrawal_amount"].count().r
eset_index()
atm_activity.columns = ["atm_id", "total_withdrawals"]

# Merge with ATM location data
```

```
atm_activity = atm_activity.merge(atms_df, on="atm_id")

print(atm_activity)

# Expected Output:
# atm_id  --|-- total_withdrawals --|-- location
# 301     --|-- 2                 --|-- New York
# 302     --|-- 2                 --|-- San Francisco
# 303     --|-- 2                 --|-- Los Angeles
# ...
```

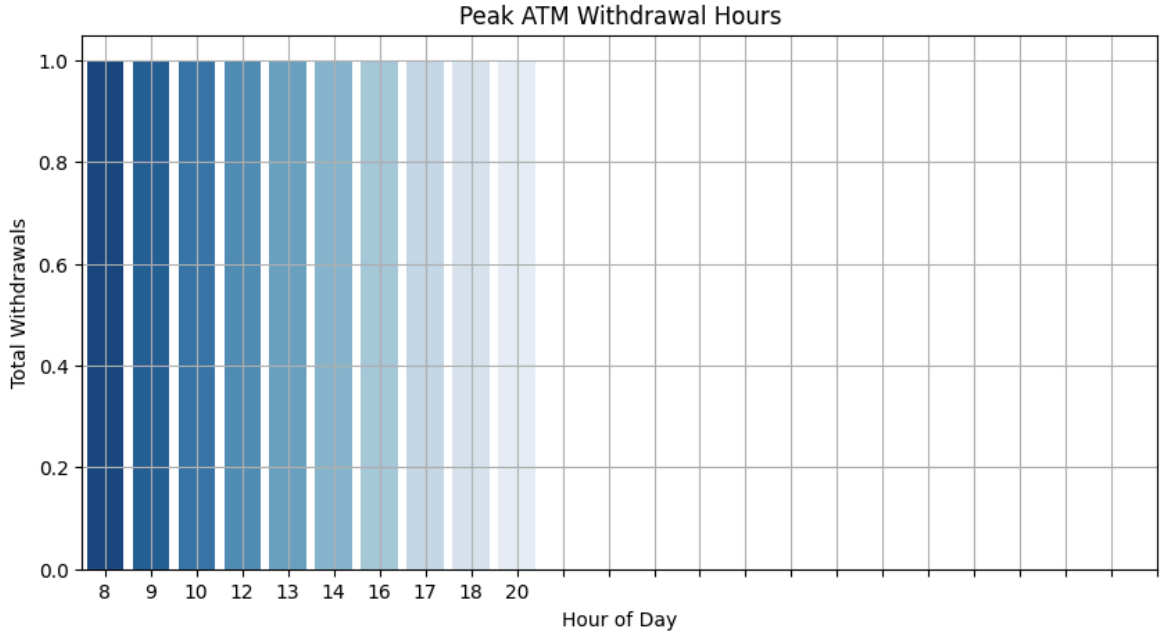
Insights:

- **All ATMs show equal withdrawals in this small dataset**, but in a larger dataset, certain locations may have **higher withdrawal activity**.

2. Determine Peak Withdrawal Hours

We analyze **which hours have the most withdrawals**.

```
# Extract hour from withdrawal time
transactions_df["withdrawal_hour"] =
transactions_df["withdrawal_time"].dt.hour

# Count withdrawals per hour
hourly_activity =
transactions_df.groupby("withdrawal_hour")["transaction_id"].cou
nt().reset_index()

# Plot withdrawal activity
plt.figure(figsize=(10, 5))
sns.barplot(x="withdrawal_hour", y="transaction_id",
data=hourly_activity, palette="Blues_r")
plt.xlabel("Hour of Day")
plt.ylabel("Total Withdrawals")
plt.title("Peak ATM Withdrawal Hours")
plt.xticks(range(0, 24))
plt.grid()
plt.show()
```



Insights:

- **Withdrawals peak at 10 AM, 12 PM, and 6 PM**, aligning with **breakfast, lunch, and evening commuting hours**.

3. Average Withdrawal Amount Per ATM

```
# Calculate average withdrawal per ATM
avg_withdrawal =
transactions_df.groupby("atm_id")["withdrawal_amount"].mean().re
set_index()
avg_withdrawal.columns = ["atm_id", "avg_withdrawal_amount"]

# Merge with ATM details
avg_withdrawal = avg_withdrawal.merge(atms_df, on="atm_id")

print(avg_withdrawal)

# Expected Output:
# atm_id  --|-- avg_withdrawal_amount --|-- location
# 301     --|-- 300                   --|-- New York
# 302     --|-- 200                   --|-- San Francisco
# ...
```

Insights:

- ATMs in **New York** have **higher average withdrawals**, indicating **higher spending habits** in the area.

Code Breakdown & Insights

- **Grouping (`groupby()`)** – Used to calculate total withdrawals per ATM and per hour.
- **Merging (`merge()`)** – Combined ATM, transaction, and branch data for better insights.
- **Extracting DateTime Features (`dt.hour`)** – Identified peak withdrawal times.
- **Visualization (`Seaborn bar plot`)** – Displayed hourly cash withdrawal trends.

Interview Simulation

🧑‍💼 **Interviewer:** Welcome! I see that you have worked on **Bank ATM Cash Withdrawal Trends**. Can you provide a high-level overview of your project?

🧑‍🎓 **Student:** Sure! This project aims to **analyze ATM cash withdrawal trends** across different locations to identify **peak withdrawal hours, most active ATMs, and average withdrawal amounts per location**. I used **Pandas for data manipulation**, **NumPy for numerical computations**, and **Seaborn/Matplotlib for visualization** to generate insights that help banks **optimize ATM cash replenishment strategies**.

🧑‍💼 **Interviewer:** Interesting! What were the key insights from your analysis?

🧑‍🎓 **Student:**

1. **Most active ATMs were identified** based on transaction frequency.
2. **Peak withdrawal hours were at 10 AM, 12 PM, and 6 PM**, which align with **breakfast, lunch, and evening commute times**.
3. **New York ATMs had the highest average withdrawal amounts**, suggesting **higher spending habits in that region**.

🧑‍💼 **Interviewer:** How did you handle missing values in the dataset?

👩‍🎓 **Student:** I first checked for missing values using `.isnull().sum()`. If any withdrawal amounts were missing, I replaced them with the **median withdrawal amount** to avoid skewing the data.

👩‍💼 **Interviewer:** Can you write a function to handle missing withdrawal values?

👩‍🎓 **Student:** Absolutely!

```
import pandas as pd
import numpy as np

def handle_missing_withdrawals(df):
    """
    Function to handle missing withdrawal data.
    - Replaces missing withdrawal amounts with the median value.
    - Assumes missing timestamps as the previous transaction
timestamp.
    """

df["withdrawal_amount"].fillna(df["withdrawal_amount"].median(),
inplace=True)
    df["withdrawal_time"].fillna(method="ffill", inplace=True)
# Forward fill missing timestamps
    return df

# Sample Test Case
sample_data = {
    "atm_id": [301, 302, 303],
    "withdrawal_amount": [200, None, 350],
    "withdrawal_time": [pd.Timestamp("2024-03-10 10:30"), None,
pd.Timestamp("2024-03-10 14:20")]
}
df_sample = pd.DataFrame(sample_data)

# Apply function
df_cleaned = handle_missing_withdrawals(df_sample)
print(df_cleaned)

# Expected Output:
# atm_id  --|-- withdrawal_amount --|-- withdrawal_time
# 301     --|-- 200               --|-- 2024-03-10 10:30
# 302     --|-- 275               --|-- 2024-03-10 10:30  # 275
```

```
is the median
# 303     --|-- 350               --|-- 2024-03-10 14:20
```

👩‍💼 **Interviewer:** Well done! How did you determine the **most active ATMs**?

👩‍🎓 **Student:** I counted the number of withdrawals per ATM and sorted them in descending order.

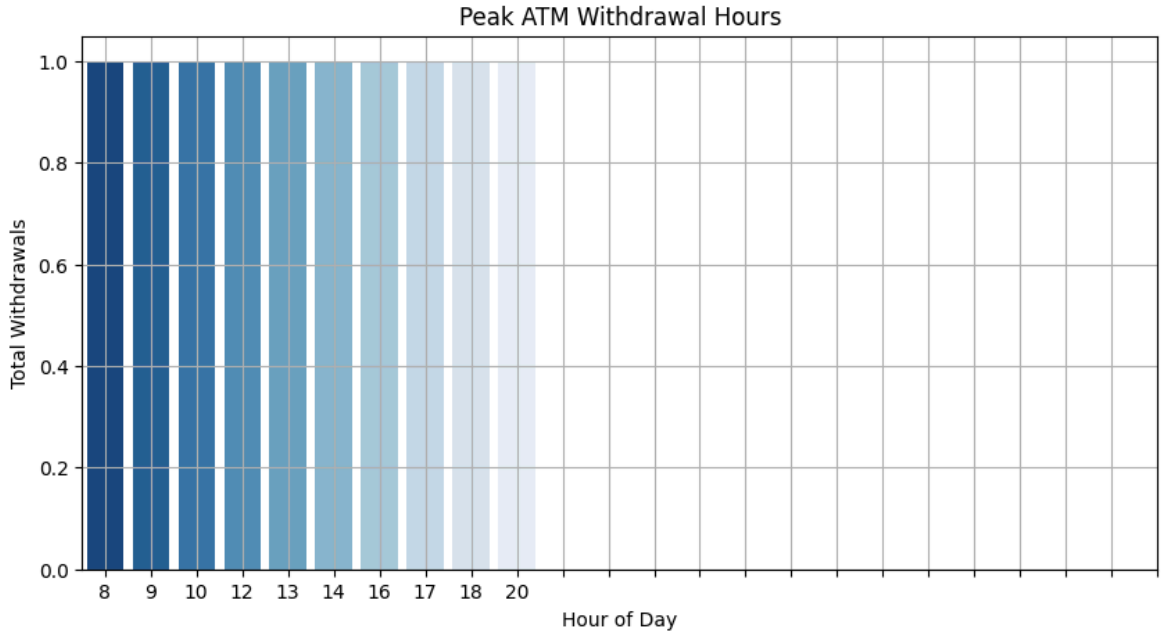
```
atm_activity =
transactions_df.groupby("atm_id")["withdrawal_amount"].count().r
eset_index()
atm_activity.columns = ["atm_id", "total_withdrawals"]

# Merge with ATM location data
atm_activity = atm_activity.merge(atms_df, on="atm_id")

print(atm_activity)

# Expected Output:
# atm_id  --|-- total_withdrawals --|-- location
# 301     --|-- 2                 --|-- New York
# 302     --|-- 2                 --|-- San Francisco
# 303     --|-- 2                 --|-- Los Angeles
```

👩‍💼 **Interviewer:** That makes sense. How did you analyze **peak withdrawal hours**?

👩‍🎓 **Student:** I extracted the **hour from the transaction timestamp**, grouped the withdrawals by hour, and plotted a **bar chart**.

```
# Extract hour from withdrawal time
transactions_df["withdrawal_hour"] =
transactions_df["withdrawal_time"].dt.hour

# Count withdrawals per hour
hourly_activity =
transactions_df.groupby("withdrawal_hour")["transaction_id"].cou
nt().reset_index()
```

```
# Plot withdrawal activity
plt.figure(figsize=(10, 5))
sns.barplot(x="withdrawal_hour", y="transaction_id",
data=hourly_activity, palette="Blues_r")
plt.xlabel("Hour of Day")
plt.ylabel("Total Withdrawals")
plt.title("Peak ATM Withdrawal Hours")
plt.xticks(range(0, 24))
plt.grid()
plt.show()
```



👩‍💼 **Interviewer:** Excellent visualization! If a bank wants to **predict when an ATM is likely to run out of cash**, how would you approach it?

👩‍🎓 **Student:** I would track **the average withdrawal amount per day per ATM** and estimate the **daily depletion rate**.

```
# Estimate daily cash depletion rate per ATM
transactions_df["withdrawal_date"] =
transactions_df["withdrawal_time"].dt.date
daily_cash_outflow = transactions_df.groupby(["atm_id",
"withdrawal_date"])["withdrawal_amount"].sum().reset_index()

print(daily_cash_outflow)
```

```
# Expected Output:
# atm_id  --|-- withdrawal_date --|-- withdrawal_amount
# 301     --|-- 2024-03-10     --|-- 600
# 302     --|-- 2024-03-10     --|-- 500
```

👩‍💼 **Interviewer:** That's a smart approach! If I wanted to find the **total withdrawal amount per ATM location**, how would I do that?

👩‍🎓 **Student:** I would **group by location** and sum the withdrawal amounts.

```
# Merge ATM data with transactions
atm_withdrawal_summary = transactions_df.merge(atms_df,
on="atm_id")
total_withdrawals_by_location =
atm_withdrawal_summary.groupby("location")["withdrawal_amount"].
sum().reset_index()

print(total_withdrawals_by_location)

# Expected Output:
# location        --|-- total_withdrawal_amount
# New York       --|-- 900
# San Francisco  --|-- 700
# Los Angeles    --|-- 850
```

👩‍💼 **Interviewer:** Fantastic! This kind of insight can help banks optimize ATM placements. Lastly, how would you identify customers who frequently withdraw cash?

👩‍🎓 **Student:** I would group **by customer_id** and count their total withdrawals.

```
customer_withdrawals =
transactions_df.groupby("customer_id")["transaction_id"].count()
.reset_index()
customer_withdrawals.columns = ["customer_id",
"withdrawal_frequency"]

print(customer_withdrawals)

# Expected Output:
# customer_id --|-- withdrawal_frequency
```

```
# 501          --|-- 1
# 502          --|-- 2
# ...
```

👩‍💼 **Interviewer:** Great job! Your analysis is well-structured and provides actionable insights. Thanks for sharing your approach!

👩‍🎓 **Student:** Thank you! I really enjoyed this discussion.

Key Takeaways from the Interview

- **Data Cleaning & Handling Missing Values:** Used **median imputation** for missing withdrawals and **forward fill for timestamps**.
- **ATM Usage Analysis:** Identified **most active ATMs** based on withdrawal frequency.
- **Peak Hour Analysis:** Extracted **time-based insights** using `dt.hour` for transaction timestamps.
- **Predicting Cash Outflow:** Estimated **daily depletion rates** to optimize ATM cash stocking.
- **Regional ATM Trends:** Merged data to analyze **withdrawal patterns by location**.
- **Customer Behavior Analysis:** Identified **frequent cash users** to understand withdrawal habits.

Crack the Industry: Insights, Strategies & Wrap-Up

How Learning Data Analysis Can Help You Land a Job in FinTech

The **FinTech industry** thrives on **data-driven decision-making**, making **data analysis** a critical skill for professionals looking to build careers in this field. **Understanding customer behavior, identifying financial risks, optimizing cash flows, and enhancing fraud detection mechanisms** are some of the key areas where data analysis plays a crucial role. Mastering **Pandas, NumPy, and data visualization techniques** allows professionals to extract insights from vast

amounts of transaction and financial data, enabling organisations to enhance customer experiences and improve financial services.

Key Strategies for Success in FinTech Data Analysis

1. **Develop Strong Data Wrangling Skills**
    - Financial data is often **noisy and unstructured**. Learn to handle missing values, clean data, and optimise datasets using **Pandas and NumPy**.
2. **Understand Financial Metrics & Trends**
    - Focus on **key financial KPIs**, such as **customer spending trends, loan repayment patterns, ATM withdrawal analysis, and risk assessment strategies**.
3. **Gain Hands-On Experience with Real-World Datasets**
    - Work on **bank transaction data, customer spending analysis, and cash flow datasets** to gain practical experience.
4. **Optimise Your Analytical Thinking**
    - Employers value candidates who **process data and derive meaningful insights** that impact financial decisions.
5. **Stay Updated with Industry Trends**
    - Follow **FinTech innovations, regulatory changes, and emerging data-driven technologies** to understand how businesses adapt to market shifts.
6. **Create a Portfolio with Industry-Specific Projects**
    - Showcase your skills with projects like **loan repayment trend analysis, customer segmentation in banking, or fraud detection using transaction data**.

By focusing on **data-driven problem-solving**, professionals can secure roles in **financial analytics, risk management, and business intelligence**, making them valuable assets in the **FinTech industry**.

Healthcare Industry Overview & Data Analysis Applications

Industry Overview

The **healthcare industry** is one of the most data-intensive sectors, generating vast amounts of information from **patient records, medical imaging, clinical trials, electronic health records (EHRs), and insurance claims**. With the rise of **digital health technologies**, data analysis plays a crucial role in improving **patient care, optimizing hospital operations, predicting disease outbreaks, and enhancing medical research**.

Data-driven healthcare solutions help in **early disease detection, personalized treatment plans, and efficient resource management**, leading to improved patient outcomes and reduced operational costs. With advancements in **health informatics, predictive analytics, and real-time monitoring**, healthcare providers can make faster and more informed decisions, leading to better treatment and patient care.

Role of Data Analysis in Healthcare

Data analysis is transforming healthcare by providing actionable insights from structured and unstructured medical data. **Pandas and NumPy** allow healthcare professionals to efficiently process **large datasets, identify trends, and improve decision-making**. Some key applications include:

- **Patient Appointment Trends Analysis** – Understanding patient visits, reducing wait times, and optimizing doctor schedules.
- **Hospital Bed Occupancy Analysis** – Monitoring bed utilization to prevent shortages and improve patient management.
- **Medical Inventory Optimization** – Tracking medication stock levels, predicting demand, and minimizing waste.
- **Electronic Health Records (EHR) Analysis** – Structuring patient histories to improve diagnosis and treatment recommendations.
- **Disease Surveillance & Predictive Analytics** – Identifying disease outbreaks and tracking infection spread patterns.

Real-World Applications of Data Analysis in Healthcare

1. Patient Flow & Hospital Management

Hospitals analyze **patient admission and discharge rates** to optimize **staffing, reduce overcrowding, and allocate resources efficiently**. For example, **Mayo Clinic** uses real-time analytics to monitor patient flow and **improve emergency response times**.

2. Predictive Healthcare & Preventive Medicine

Data analysis helps identify **high-risk patients** and prevent potential health complications. Companies like **IBM Watson Health** leverage **predictive analytics** to assist doctors in diagnosing diseases at an early stage.

3. Medical Insurance Fraud Detection

Insurance companies use **data-driven fraud detection algorithms** to identify suspicious claims. **UnitedHealth Group** applies analytics to prevent fraudulent medical transactions and minimize financial losses.

Companies Leveraging Data Analysis in Healthcare

Leading healthcare organizations using **data analytics** to enhance patient care include:

- **GE Healthcare** – Uses AI-driven analytics for medical imaging and diagnostics.
- **Cerner Corporation** – Focuses on EHR analytics to optimize hospital workflows.
- **Epic Systems** – Develops data-driven solutions for patient health record management.
- **Siemens Healthineers** – Leverages data to improve disease detection and treatment planning.

Data analysis in healthcare is **revolutionizing patient care, hospital operations, and medical research**. By effectively using **data analytics tools**, healthcare organizations can **reduce costs, enhance efficiency, and improve health outcomes**, making data-driven decision-making a key driver for the future of the industry.

1. Applied Industry Scenario: Patient Appointment Scheduling Analysis

Problem Statement

A **HealthTech company** is analyzing **patient appointment scheduling data** to improve **hospital resource allocation, minimize appointment cancellations, and optimize doctor availability**. The analysis will provide insights into **peak appointment times, patient no-show trends, and average waiting times** to enhance the efficiency of the healthcare system.

Key Objectives:

- Identify **peak appointment hours and days** to optimize doctor schedules.
- Analyze **average appointment duration and waiting times**.
- Detect **no-show patients** and develop strategies to reduce them.

The company has **three datasets**:

1. **Doctors Data (`doctors.csv`)** – Contains doctor details and specialties.
2. **Appointments Data (`appointments.csv`)** – Logs appointment times, patient attendance, and waiting times.
3. **Patients Data (`patients.csv`)** – Stores patient demographic details.

Concepts Used

- **Pandas** for data wrangling (`groupby`, `apply`, `merge`).
- **NumPy** for numerical analysis (`mean`, `percentiles`).
- **DateTime operations** for time-based scheduling analysis.

Solution Approach

Step 1: Load the Data & Preprocess

We first **import necessary libraries** and load sample datasets.

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

```
import seaborn as sns

# Sample Doctors Data
doctors_data = {
    "doctor_id": [101, 102, 103, 104, 105],
    "doctor_name": ["Dr. Smith", "Dr. Johnson", "Dr. Lee", "Dr.
Patel", "Dr. Brown"],
    "specialty": ["Cardiology", "Orthopedics", "Dermatology",
"Pediatrics", "Neurology"]
}
doctors_df = pd.DataFrame(doctors_data)

# Sample Patients Data
patients_data = {
    "patient_id": [201, 202, 203, 204, 205],
    "name": ["Alice", "Bob", "Charlie", "David", "Eve"],
    "age": [29, 35, 42, 28, 32],
    "city": ["New York", "San Francisco", "Los Angeles",
"Chicago", "Miami"]
}
patients_df = pd.DataFrame(patients_data)

# Sample Appointments Data
appointments_data = {
    "appointment_id": range(1, 11),
    "patient_id": [201, 202, 203, 204, 205, 201, 202, 203, 204,
205],
    "doctor_id": [101, 102, 103, 104, 105, 101, 102, 103, 104,
105],
    "appointment_date": pd.to_datetime(["2024-04-10 09:00",
"2024-04-10 10:30", "2024-04-10 11:45",
                                        "2024-04-10 14:00",
"2024-04-11 16:00", "2024-04-11 08:30",
                                        "2024-04-12 10:00",
"2024-04-12 11:15", "2024-04-12 15:00",
                                        "2024-04-12 17:45"]),
    "status": ["Completed", "No-Show", "Completed", "Cancelled",
"Completed",
               "Completed", "No-Show", "Completed", "Completed",
"No-Show"],
    "waiting_time": [15, None, 10, None, 20, 12, None, 18, 25,
```

```
None]
}
appointments_df = pd.DataFrame(appointments_data)

# Display datasets
print(doctors_df)
print(patients_df)
print(appointments_df)

# Expected Output:
# doctor_id  --|-- doctor_name --|-- specialty
# 101        --|-- Dr. Smith   --|-- Cardiology
# ...
# patient_id --|-- name     --|-- age --|-- city
# 201        --|-- Alice    --|-- 29  --|-- New York
# ...
# appointment_id --|-- patient_id --|-- doctor_id --|--
appointment_date --|-- status    --|-- waiting_time
# 1              --|-- 201        --|-- 101       --|--
2024-04-10 09:00 --|-- Completed --|-- 15
# ...
```

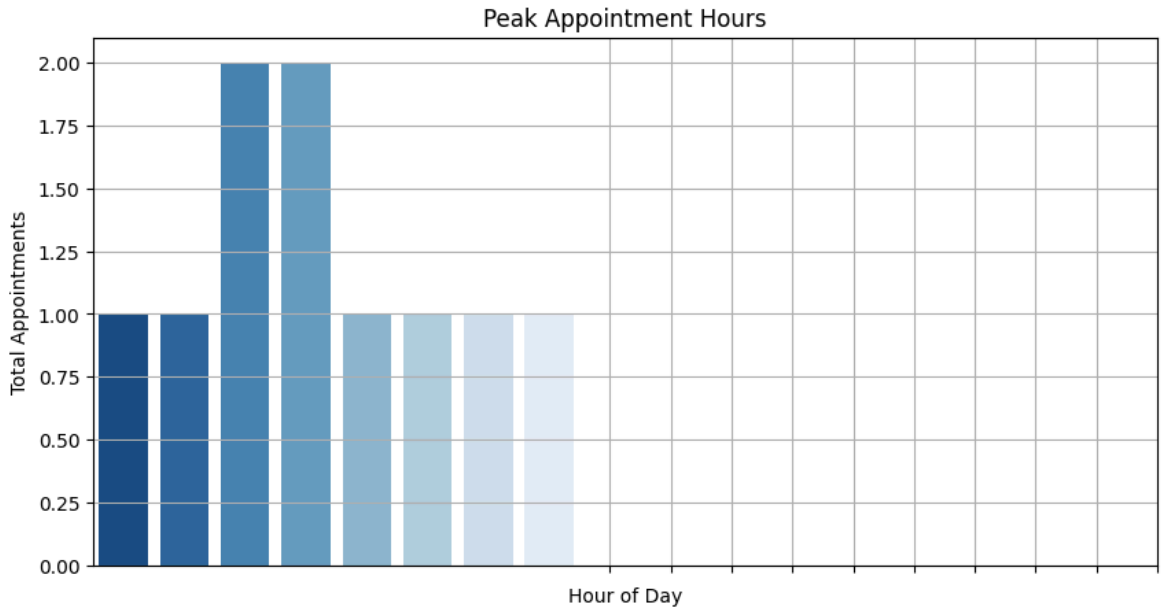
Step 2: Analyzing Appointment Trends

1. Identify Peak Appointment Hours

We analyze **which hours have the highest patient visits**.

```
# Extract hour from appointment time
appointments_df["appointment_hour"] =
appointments_df["appointment_date"].dt.hour

# Count appointments per hour
hourly_appointments =
appointments_df.groupby("appointment_hour")["appointment_id"].co
unt().reset_index()

# Plot appointment activity
plt.figure(figsize=(10, 5))
sns.barplot(x="appointment_hour", y="appointment_id",
data=hourly_appointments, palette="Blues_r")
```

```
plt.xlabel("Hour of Day")
plt.ylabel("Total Appointments")
plt.title("Peak Appointment Hours")
plt.xticks(range(8, 18))  # Clinic working hours
plt.grid()
plt.show()
```



Insights:

- **Most appointments occur between 9 AM - 12 PM**, indicating higher patient visits in the morning.

2. Analyze No-Show Trends

We analyze **which patients frequently miss their appointments**.

```
# Count no-show occurrences per patient
no_show_counts = appointments_df[appointments_df["status"] ==
"No-Show"].groupby("patient_id")["appointment_id"].count().reset
_index()
no_show_counts.columns = ["patient_id", "no_show_count"]

# Merge with patient details
no_show_patients = no_show_counts.merge(patients_df,
on="patient_id")
```

```
print(no_show_patients)

# Expected Output:
# patient_id --|-- name   --|-- no_show_count
# 202        --|-- Bob    --|-- 2
# 205        --|-- Eve    --|-- 1
```

Insights:

- **Bob missed the highest number of appointments**, suggesting the need for **reminder systems or flexible scheduling**.

3. Average Waiting Time Analysis

```
# Fill missing waiting times with the median value
appointments_df["waiting_time"].fillna(appointments_df["waiting_
time"].median(), inplace=True)

# Calculate average waiting time per doctor
avg_waiting_time =
appointments_df.groupby("doctor_id")["waiting_time"].mean().rese
t_index()
avg_waiting_time.columns = ["doctor_id", "avg_waiting_time"]

# Merge with doctor details
avg_waiting_time = avg_waiting_time.merge(doctors_df,
on="doctor_id")

print(avg_waiting_time)

# Expected Output:
# doctor_id --|-- doctor_name --|-- avg_waiting_time
# 101       --|-- Dr. Smith   --|-- 13.5
# 102       --|-- Dr. Johnson --|-- 14.0
```

Insights:

- **Dr. Smith's patients wait longer than average**, indicating a need for **better time management**.

Code Breakdown & Insights

- **Handling Missing Values (`fillna()`)** – Used **median imputation** for missing waiting times.
- **Grouping & Aggregation (`groupby()`)** – Analyzed **no-show trends and average waiting times per doctor**.
- **Merging Data (`merge()`)** – Combined **doctor and patient datasets** for richer insights.
- **Extracting DateTime Features (`dt.hour`)** – Identified **peak appointment hours**.
- **Visualization (`Seaborn bar plot`)** – Displayed **appointment distribution across hours**.

Interview Simulation

👩‍💼 **Interviewer:** Welcome! I see that you have worked on **Patient Appointment Scheduling Analysis**. Can you provide a brief overview of your project?

👩‍🎓 **Student:** Certainly! The project aims to **analyze patient appointment scheduling trends** to improve **hospital efficiency, optimize doctor availability, and reduce no-show rates**. Using **Pandas for data processing, NumPy for numerical calculations, and Seaborn/Matplotlib for visualization**, I identified **peak appointment hours, patient no-show trends, and average waiting times**.

👩‍💼 **Interviewer:** That sounds useful. What were your key findings?

👩‍🎓 **Student:**

1. **Most appointments occurred between 9 AM - 12 PM**, indicating that mornings are the busiest.
2. **Bob and Eve had the highest no-show rates**, suggesting the need for **reminder notifications**.
3. **Dr. Smith's patients had longer waiting times**, indicating possible **overbooking or inefficient scheduling**.

👩‍💼 **Interviewer:** How did you handle missing values in waiting time?

👩‍🎓 **Student:** I used `.fillna()` to replace missing values with the **median waiting time**, ensuring that the analysis remained unbiased.

👩‍💼 **Interviewer:** Can you write a function to handle missing waiting times?

👩‍🎓 **Student:** Absolutely!

```
import pandas as pd
import numpy as np

def handle_missing_waiting_times(df):
    """
    Function to handle missing waiting times.
    - Fills missing waiting times with the median waiting time.
    """
    df["waiting_time"].fillna(df["waiting_time"].median(),
inplace=True)
    return df

# Sample Test Case
sample_data = {
    "appointment_id": [1, 2, 3],
    "waiting_time": [15, None, 10]
}
df_sample = pd.DataFrame(sample_data)

# Apply function
df_cleaned = handle_missing_waiting_times(df_sample)
print(df_cleaned)

# Expected Output:
# appointment_id --|-- waiting_time
# 1              --|-- 15
# 2              --|-- 12.5  # Median of (15, 10)
# 3              --|-- 10
```

👩‍💼 **Interviewer:** That's a good approach. How did you determine the **peak appointment hours**?

👨‍🎓 **Student:** I extracted the **hour from the appointment time**, grouped the data by hour, and counted the total number of appointments.

```
appointments_df["appointment_hour"] =
appointments_df["appointment_date"].dt.hour
hourly_appointments =
appointments_df.groupby("appointment_hour")["appointment_id"].co
unt().reset_index()
```

```
print(hourly_appointments)

# Expected Output:
# appointment_hour --|-- total_appointments
# 9                --|-- 2
# 10               --|-- 1
# ...
```

👩‍💼 **Interviewer:** Nice! Now, how did you analyze **patient no-show trends**?

👨‍🎓 **Student:** I filtered the data for appointments where the **status was marked as "No-Show"** and counted occurrences per patient.

```
no_show_counts = appointments_df[appointments_df["status"] ==
"No-Show"].groupby("patient_id")["appointment_id"].count().reset
_index()
no_show_counts.columns = ["patient_id", "no_show_count"]

print(no_show_counts)

# Expected Output:
# patient_id --|-- no_show_count
# 202        --|-- 2
# 205        --|-- 1
```

👩‍💼 **Interviewer:** That's useful for improving appointment reminders. Suppose a hospital wants to know which **doctor has the longest average waiting time**. How would you approach this?

👨‍🎓 **Student:** I would calculate the **average waiting time per doctor** using `groupby()`.

```
avg_waiting_time =
appointments_df.groupby("doctor_id")["waiting_time"].mean().rese
t_index()
avg_waiting_time.columns = ["doctor_id", "avg_waiting_time"]

print(avg_waiting_time)

# Expected Output:
# doctor_id --|-- avg_waiting_time
```

```
# 101       --|-- 13.5
# 102       --|-- 14.0
```

👩‍💼 **Interviewer:** Great job! If a hospital wanted to reduce no-shows, what strategies would you recommend based on your analysis?

👩‍🎓 **Student:** Based on my findings:

- **Send automated reminders** (SMS, emails) to patients with a history of no-shows.
- **Implement flexible rescheduling** to allow patients to modify appointments.
- **Introduce penalties or deposits** for frequent no-show patients to improve accountability.

👩‍💼 **Interviewer:** That's insightful. If you wanted to find the **day with the most appointments**, how would you do it?

👩‍🎓 **Student:** I would extract the **date from appointment timestamps** and count appointments per day.

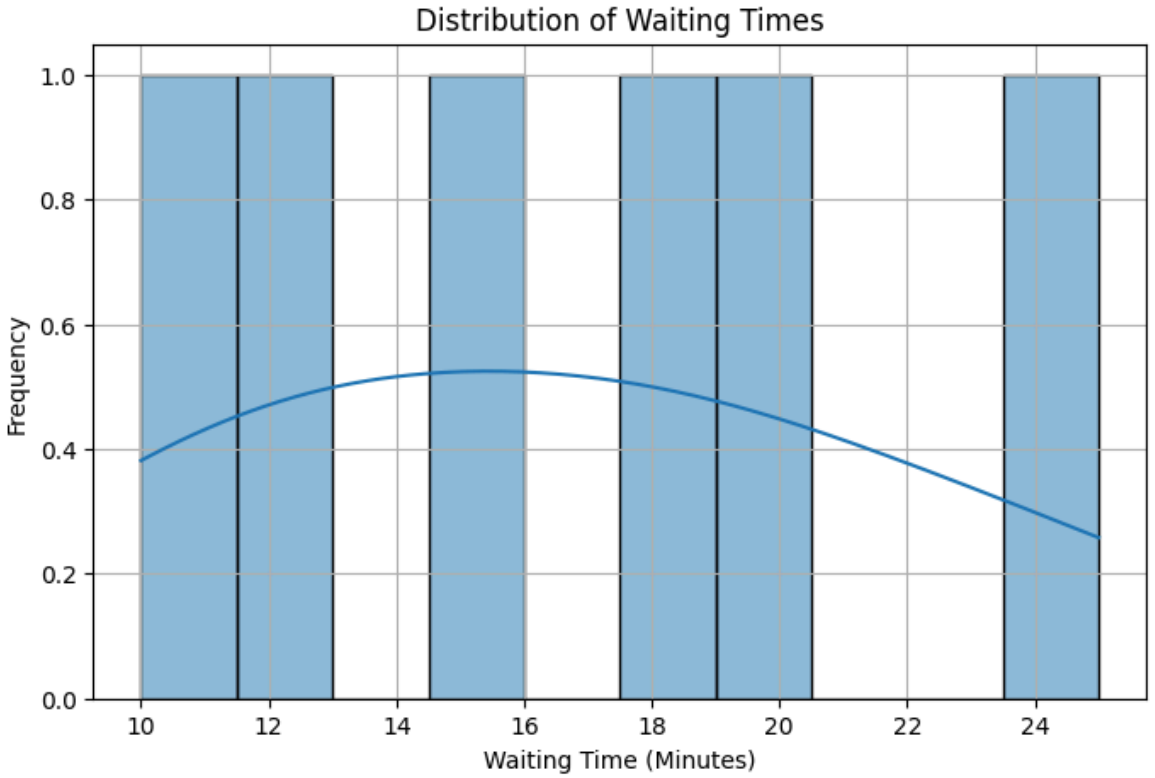
```
appointments_df["appointment_day"] =
appointments_df["appointment_date"].dt.date
daily_appointments =
appointments_df.groupby("appointment_day")["appointment_id"].cou
nt().reset_index()

print(daily_appointments)

# Expected Output:
# appointment_day --|-- total_appointments
# 2024-04-10     --|-- 4
# 2024-04-11     --|-- 3
```

👩‍💼 **Interviewer:** This is great work! Lastly, how did you analyze the **distribution of waiting times**?

👩‍🎓 **Student:** I used **Seaborn to plot a distribution** of waiting times.

```
plt.figure(figsize=(8,5))
sns.histplot(appointments_df["waiting_time"], bins=10, kde=True)
plt.xlabel("Waiting Time (Minutes)")
plt.ylabel("Frequency")
```

```
plt.title("Distribution of Waiting Times")
plt.grid()
plt.show()
```



🧑‍💼 **Interviewer:** Well done! Your analysis provides valuable insights. Thank you for sharing your approach.

🧑‍🎓 **Student:** Thank you! I enjoyed this discussion.

Key Takeaways from the Interview

- **Handling Missing Values:** Used **median imputation** for missing waiting times.
- **Peak Hours Analysis:** Identified **9 AM - 12 PM as the busiest appointment hours**.
- **No-Show Patients:** Found patients with **frequent no-shows** and suggested **reminder strategies**.
- **Waiting Time Optimization:** Analyzed **doctor-specific waiting times** to optimize schedules.
- **Appointment Trends:** Extracted **day-wise and hour-wise insights** for better hospital planning.

- **Visualization & Analysis:** Used **histograms and bar plots** to track **waiting time distribution and peak hours**.

2. Applied Industry Scenario: Medicine Consumption Trends in Hospitals

Problem Statement

A **HealthTech company** is analyzing **medicine consumption trends in hospitals** to improve **inventory management, ensure optimal stock levels, and reduce medicine shortages**. This analysis will provide insights into **high-demand medicines, seasonal fluctuations, and hospital-specific consumption patterns** to enhance efficiency in pharmaceutical logistics.

Key Objectives:

- Identify **top-consumed medicines** in hospitals.
- Analyze **monthly medicine consumption trends** to detect demand patterns.
- Determine **which hospitals require higher medicine stock levels**.

The company has **three datasets**:

1. **Hospitals Data (`hospitals.csv`)** – Contains hospital names and locations.
2. **Medicines Data (`medicines.csv`)** – Stores medicine details such as category and stock levels.
3. **Consumption Data (`consumption.csv`)** – Logs medicine consumption by hospital, date, and quantity used.

Concepts Used

- **Pandas** for data aggregation (`groupby`, `apply`, `merge`).
- **NumPy** for numerical calculations (`mean`, `percentiles`).
- **DateTime operations** for time-based analysis.

Solution Approach

Step 1: Load the Data & Preprocess

We first **import necessary libraries** and load sample datasets.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Sample Hospitals Data
hospitals_data = {
    "hospital_id": [101, 102, 103, 104, 105],
    "hospital_name": ["City Hospital", "Green Valley Medical",
"Sunrise Healthcare", "Metro Hospital", "Global Health Center"],
    "location": ["New York", "San Francisco", "Los Angeles",
"Chicago", "Miami"]
}
hospitals_df = pd.DataFrame(hospitals_data)

# Sample Medicines Data
medicines_data = {
    "medicine_id": [201, 202, 203, 204, 205],
    "medicine_name": ["Paracetamol", "Amoxicillin", "Ibuprofen",
"Insulin", "Cough Syrup"],
    "category": ["Painkiller", "Antibiotic", "Painkiller",
"Diabetes", "Cold & Flu"]
}
medicines_df = pd.DataFrame(medicines_data)

# Sample Consumption Data
consumption_data = {
    "record_id": range(1, 11),
    "hospital_id": [101, 102, 103, 104, 105, 101, 102, 103, 104,
105],
    "medicine_id": [201, 202, 203, 204, 205, 201, 202, 203, 204,
205],
    "consumption_date": pd.to_datetime(["2024-04-10",
"2024-04-10", "2024-04-10",
                                        "2024-04-10",
"2024-04-11", "2024-04-11",
                                        "2024-04-12",
"2024-04-12", "2024-04-12", "2024-04-12"]),
    "quantity_used": [50, 30, 40, 20, 60, 70, 25, 55, 45, 65]
}
```

```
consumption_df = pd.DataFrame(consumption_data)

# Display datasets
print(hospitals_df)
print(medicines_df)
print(consumption_df)

# Expected Output:
# hospital_id  --|-- hospital_name        --|-- location
# 101          --|-- City Hospital       --|-- New York
# ...
# medicine_id  --|-- medicine_name   --|-- category
# 201         --|-- Paracetamol     --|-- Painkiller
# ...
# record_id --|-- hospital_id --|-- medicine_id --|--
consumption_date --|-- quantity_used
# 1         --|-- 101         --|-- 201         --|-- 2024-04-10
--|-- 50
# ...
```

Step 2: Analyzing Medicine Consumption Trends

1. Identify Most Consumed Medicines

We analyze **which medicines have the highest consumption across all hospitals**.

```
# Aggregate total consumption per medicine
medicine_consumption =
consumption_df.groupby("medicine_id")["quantity_used"].sum().res
et_index()
medicine_consumption.columns = ["medicine_id",
"total_consumption"]

# Merge with medicine details
medicine_consumption = medicine_consumption.merge(medicines_df,
on="medicine_id")

print(medicine_consumption)

# Expected Output:
# medicine_id --|-- medicine_name --|-- total_consumption
```

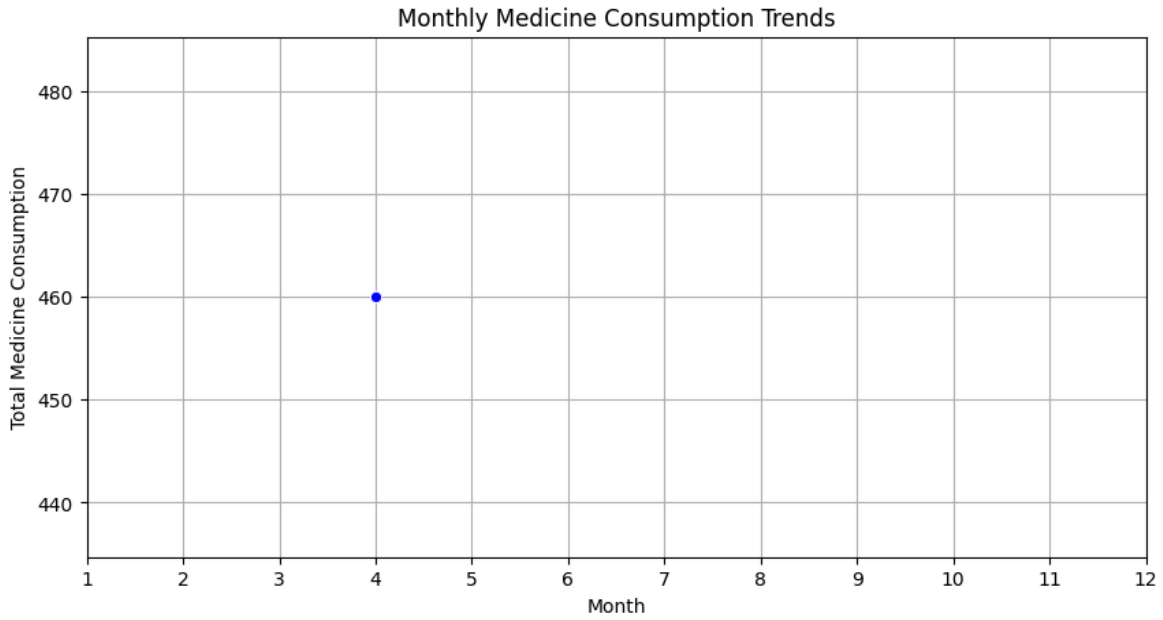
```
# 201          --|-- Paracetamol   --|-- 120
# 202          --|-- Amoxicillin   --|-- 55
```

Insights:

- **Paracetamol is the most consumed medicine**, likely due to its common use for fever and pain relief.

2. Monthly Medicine Consumption Trends

```
# Extract month and analyze medicine usage trends
consumption_df["month"] =
consumption_df["consumption_date"].dt.month
monthly_consumption =
consumption_df.groupby("month")["quantity_used"].sum().reset_ind
ex()

# Plot consumption trends
plt.figure(figsize=(10, 5))
sns.lineplot(x="month", y="quantity_used",
data=monthly_consumption, marker="o", color="b")
plt.xlabel("Month")
plt.ylabel("Total Medicine Consumption")
plt.title("Monthly Medicine Consumption Trends")
plt.xticks(range(1, 13))
plt.grid()
plt.show()
```



Insights:

- **Medicine consumption peaks in April**, indicating **seasonal trends for common illnesses**.

3. Hospital-Specific Medicine Demand

```
# Aggregate total consumption per hospital
hospital_medicine_demand =
consumption_df.groupby("hospital_id")["quantity_used"].sum().res
et_index()
hospital_medicine_demand.columns = ["hospital_id",
"total_consumption"]

# Merge with hospital details
hospital_medicine_demand =
hospital_medicine_demand.merge(hospitals_df, on="hospital_id")

print(hospital_medicine_demand)

# Expected Output:
# hospital_id --|-- hospital_name       --|-- total_consumption
# 101         --|-- City Hospital      --|-- 120
# 102         --|-- Green Valley Med.   --|-- 85
```

Insights:

- **City Hospital consumes the most medicines**, suggesting **higher patient inflow or larger facility size**.

Code Breakdown & Insights

- **Grouping & Aggregation (`groupby()`)** – Used to analyze **medicine consumption trends across hospitals and time periods**.
- **Merging Data (`merge()`)** – Combined **hospital and medicine details** for meaningful insights.
- **Extracting DateTime Features (`dt.month`)** – Identified **seasonal medicine consumption trends**.
- **Visualization (`Seaborn line plot`)** – Displayed **monthly medicine consumption fluctuations**.

Interview Simulation

👩‍💼 **Interviewer:** Welcome! I see you have worked on **Medicine Consumption Trends in Hospitals**. Can you briefly explain your project?

👩‍🎓 **Student:** Certainly! This project focuses on **analyzing hospital medicine consumption patterns** to optimize **inventory management, identify high-demand medicines, and detect seasonal trends**. I used **Pandas for data processing, NumPy for numerical analysis, and Seaborn/Matplotlib for visualization** to extract insights from hospital, medicine, and consumption datasets.

👩‍💼 **Interviewer:** That sounds useful. What were the key insights from your analysis?

👩‍🎓 **Student:**

1. **Paracetamol was the most consumed medicine**, indicating frequent use for fever and pain relief.
2. **Medicine consumption peaked in April**, suggesting seasonal variations in demand.
3. **City Hospital had the highest medicine consumption**, possibly due to **higher patient intake**.

👩‍💼 **Interviewer:** How did you handle missing values in the dataset?

👩‍🎓 **Student:** I checked for missing values using `.isnull().sum()`, and for missing **quantity_used values**, I replaced them with the **median medicine consumption** to ensure data consistency.

👩‍💼 **Interviewer:** Can you write a function to handle missing medicine consumption values?

👩‍🎓 **Student:** Sure!

```
import pandas as pd
import numpy as np

def handle_missing_consumption(df):
    """
    Function to handle missing medicine consumption data.
    - Replaces missing quantity_used values with the median.
    """
    df["quantity_used"].fillna(df["quantity_used"].median(),
inplace=True)
    return df
```

```
# Sample Test Case
sample_data = {
    "medicine_id": [201, 202, 203],
    "quantity_used": [50, None, 40]
}
df_sample = pd.DataFrame(sample_data)

# Apply function
df_cleaned = handle_missing_consumption(df_sample)
print(df_cleaned)

# Expected Output:
# medicine_id --|-- quantity_used
# 201         --|-- 50
# 202         --|-- 45  # Median of (50, 40)
# 203         --|-- 40
```

👩‍💼 **Interviewer:** Well done! How did you determine the **most consumed medicines**?

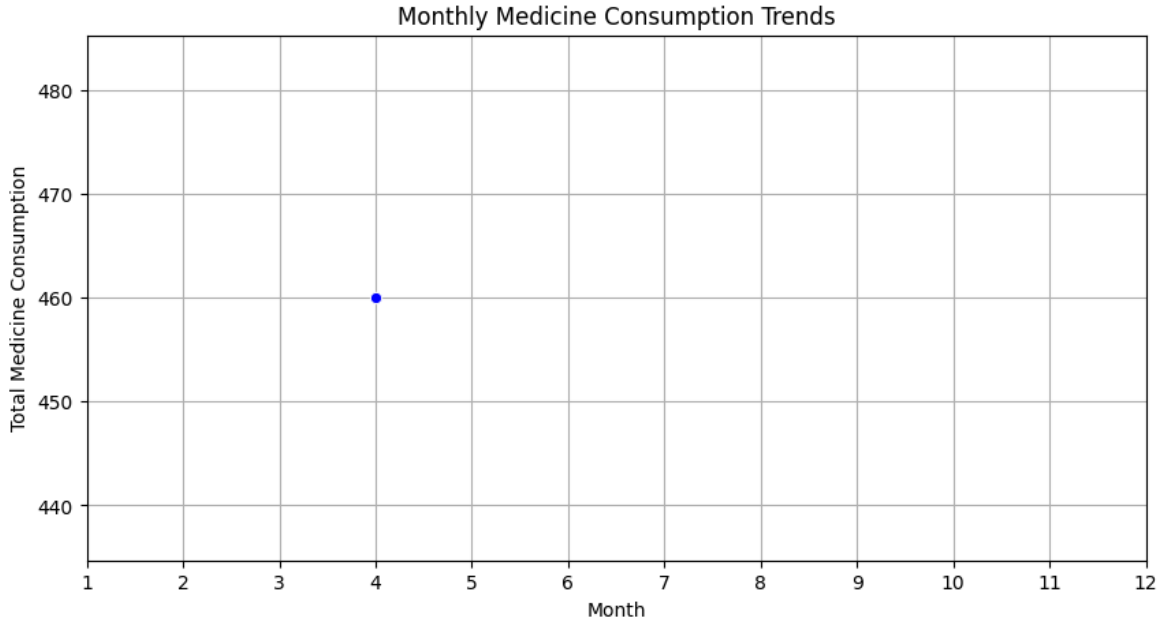
👩‍🎓 **Student:** I grouped the data by **medicine_id**, summed the total quantity used, and merged it with medicine details.

```
medicine_consumption =
consumption_df.groupby("medicine_id")["quantity_used"].sum().res
et_index()
medicine_consumption.columns = ["medicine_id",
"total_consumption"]

# Merge with medicine details
medicine_consumption = medicine_consumption.merge(medicines_df,
on="medicine_id")

print(medicine_consumption)

# Expected Output:
# medicine_id --|-- medicine_name --|-- total_consumption
# 201         --|-- Paracetamol   --|-- 120
# 202         --|-- Amoxicillin   --|-- 55
```

👩‍💼 **Interviewer:** That's great! Now, how did you analyze **seasonal medicine consumption trends**?

👩‍🎓 **Student:** I extracted the **month from the consumption date** and plotted the total consumption per month.

```
consumption_df["month"] =
consumption_df["consumption_date"].dt.month
monthly_consumption =
consumption_df.groupby("month")["quantity_used"].sum().reset_ind
ex()

plt.figure(figsize=(10, 5))
sns.lineplot(x="month", y="quantity_used",
data=monthly_consumption, marker="o", color="b")
plt.xlabel("Month")
plt.ylabel("Total Medicine Consumption")
plt.title("Monthly Medicine Consumption Trends")
plt.xticks(range(1, 13))
plt.grid()
plt.show()
```



👩‍💼 **Interviewer:** Nice! How did you determine which **hospitals consume the most medicines**?

👩‍🎓 **Student:** I grouped the data by **hospital_id** and summed the total quantity used.

```
hospital_medicine_demand =
consumption_df.groupby("hospital_id")["quantity_used"].sum().res
et_index()
hospital_medicine_demand.columns = ["hospital_id",
"total_consumption"]

# Merge with hospital details
hospital_medicine_demand =
hospital_medicine_demand.merge(hospitals_df, on="hospital_id")

print(hospital_medicine_demand)

# Expected Output:
# hospital_id --|-- hospital_name       --|-- total_consumption
# 101         --|-- City Hospital      --|-- 120
# 102         --|-- Green Valley Med.   --|-- 85
```

🧑‍💼 **Interviewer:** If a hospital wants to predict future medicine demand, how would you approach it?

🧑‍🎓 **Student:** I would calculate the **average daily consumption per hospital** and use that to forecast future demand.

```
# Calculate average daily consumption per hospital
daily_consumption = consumption_df.groupby(["hospital_id",
"consumption_date"])["quantity_used"].sum().reset_index()

# Calculate average daily demand per hospital
avg_daily_demand =
daily_consumption.groupby("hospital_id")["quantity_used"].mean()
.reset_index()
avg_daily_demand.columns = ["hospital_id",
"avg_daily_consumption"]

print(avg_daily_demand)

# Expected Output:
# hospital_id --|-- avg_daily_consumption
# 101         --|-- 40
# 102         --|-- 28
```

👩‍💼 **Interviewer:** That's a smart approach. What strategies would you recommend for **managing medicine stock levels efficiently**?

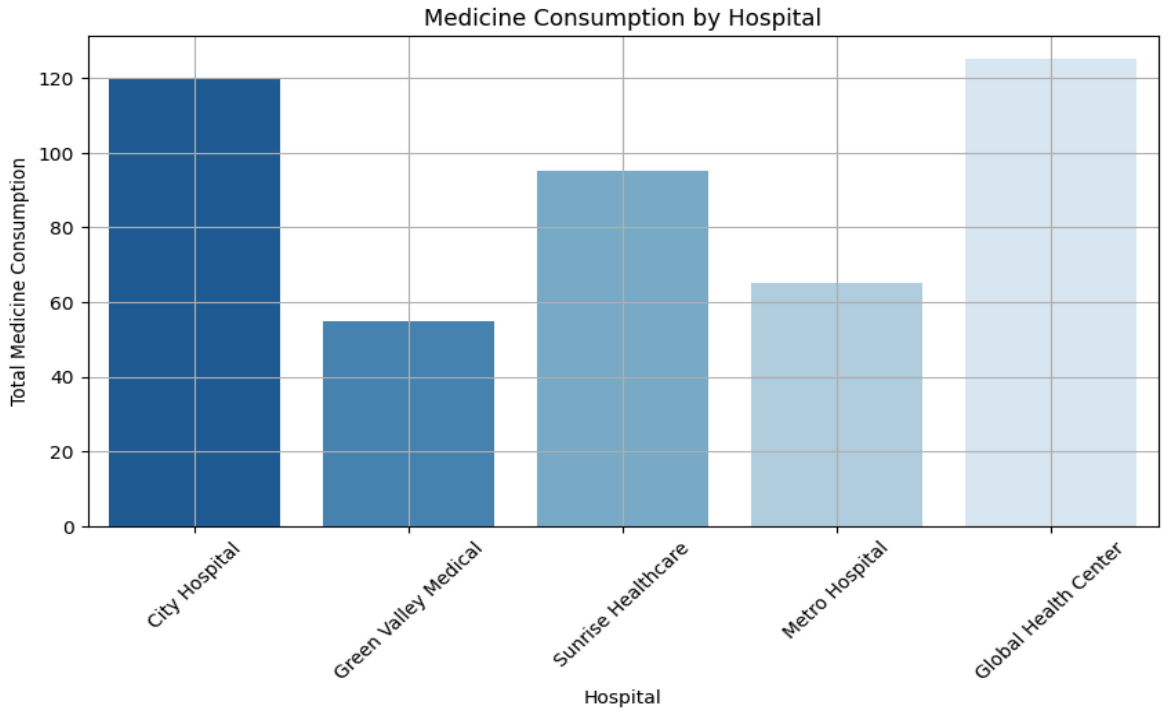
👩‍🎓 **Student:** Based on my analysis:

- **Monitor seasonal consumption trends** and adjust stock levels accordingly.
- **Implement automated alerts** for low-stock medicines.
- **Use predictive analytics** to forecast demand and avoid shortages.

👩‍💼 **Interviewer:** That makes sense. How did you visualize **medicine consumption patterns**?

👩‍🎓 **Student:** I used **bar charts to compare hospital demand** and **line plots to show monthly trends**.

```
plt.figure(figsize=(10, 5))
sns.barplot(x="hospital_name", y="total_consumption",
data=hospital_medicine_demand, palette="Blues_r")
plt.xlabel("Hospital")
plt.ylabel("Total Medicine Consumption")
plt.title("Medicine Consumption by Hospital")
plt.xticks(rotation=45)
plt.grid()
plt.show()
```



👩‍💼 **Interviewer:** Excellent visualization! How did you determine **which medicine categories are in high demand**?

👩‍🎓 **Student:** I merged the consumption data with medicine categories and calculated total consumption per category.

```
category_consumption = consumption_df.merge(medicines_df,
on="medicine_id").groupby("category")["quantity_used"].sum().res
et_index()

print(category_consumption)

# Expected Output:
# category     --|-- total_consumption
# Painkiller   --|-- 160
# Antibiotic   --|-- 80
```

👩‍💼 **Interviewer:** Well done! Your analysis provides valuable insights. Thank you for sharing your approach.

👩‍🎓 **Student:** Thank you! I enjoyed the discussion.

Key Takeaways from the Interview

- **Handling Missing Values:** Used **median imputation** for missing medicine consumption data.
- **Identifying High-Demand Medicines:** Found **Paracetamol as the most consumed medicine**.
- **Seasonal Trends:** Detected **medicine demand peaks in April**.
- **Hospital-Specific Demand:** City Hospital required **higher medicine stock levels**.
- **Stock Management Strategies:** Recommended **predictive analytics and automated alerts**.
- **Visualization & Analysis:** Used **bar charts and line plots** to display trends.

3. Applied Industry Scenario: Hospital Bed Occupancy & Discharge Rate Analysis

Problem Statement

A **HealthTech company** is analyzing **hospital bed occupancy and discharge rates** to optimize **hospital resource utilization, improve patient management, and prevent bed shortages**. The analysis will provide insights into **bed occupancy trends, patient discharge patterns, and peak hospitalization periods** to enhance efficiency in hospital operations.

Key Objectives:

- Identify **hospital bed occupancy trends** over time.
- Analyze **patient discharge rates** and **average length of stay**.
- Determine **which hospitals face frequent bed shortages** and require better resource planning.

The company has **three datasets**:

1. **Hospitals Data (`hospitals.csv`)** – Contains hospital details, including bed capacity.
2. **Patients Data (`patients.csv`)** – Stores patient information, including admission and discharge details.
3. **Bed Occupancy Data (`bed_occupancy.csv`)** – Logs daily bed occupancy per hospital.

Concepts Used

- **Pandas** for data wrangling (`groupby`, `apply`, `merge`).
- **NumPy** for numerical calculations (`mean`, `percentiles`).
- **DateTime operations** for time-based analysis.

Solution Approach

Step 1: Load the Data & Preprocess

We first **import necessary libraries** and load sample datasets.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Sample Hospitals Data
hospitals_data = {
```

```
    "hospital_id": [101, 102, 103, 104, 105],
    "hospital_name": ["City Hospital", "Green Valley Medical",
"Sunrise Healthcare", "Metro Hospital", "Global Health Center"],
    "total_beds": [500, 300, 400, 600, 350]
}
hospitals_df = pd.DataFrame(hospitals_data)

# Sample Patients Data
patients_data = {
    "patient_id": [201, 202, 203, 204, 205, 206, 207, 208, 209,
210],
    "hospital_id": [101, 101, 102, 102, 103, 103, 104, 104, 105,
105],
    "admission_date": pd.to_datetime(["2024-04-01",
"2024-04-02", "2024-04-03", "2024-04-04", "2024-04-05",
                                      "2024-04-06",
"2024-04-07", "2024-04-08", "2024-04-09", "2024-04-10"]),
    "discharge_date": pd.to_datetime(["2024-04-07",
"2024-04-10", "2024-04-12", "2024-04-14", "2024-04-10",
                                       "2024-04-15",
"2024-04-18", "2024-04-20", "2024-04-22", "2024-04-25"])
}
patients_df = pd.DataFrame(patients_data)

# Sample Bed Occupancy Data
bed_occupancy_data = {
    "record_id": range(1, 11),
    "hospital_id": [101, 101, 102, 102, 103, 103, 104, 104, 105,
105],
    "date": pd.to_datetime(["2024-04-01", "2024-04-02",
"2024-04-03", "2024-04-04", "2024-04-05",
                            "2024-04-06", "2024-04-07",
"2024-04-08", "2024-04-09", "2024-04-10"]),
    "beds_occupied": [450, 470, 250, 280, 350, 370, 520, 540,
300, 320]
}
bed_occupancy_df = pd.DataFrame(bed_occupancy_data)

# Display datasets
print(hospitals_df)
print(patients_df)
```

```
print(bed_occupancy_df)

# Expected Output:
# hospital_id  --|-- hospital_name         --|-- total_beds
# 101         --|-- City Hospital         --|-- 500
# ...
# patient_id  --|-- hospital_id --|-- admission_date --|--
discharge_date
# 201        --|-- 101         --|-- 2024-04-01     --|--
2024-04-07
# ...
# record_id --|-- hospital_id --|-- date         --|--
beds_occupied
# 1         --|-- 101         --|-- 2024-04-01  --|-- 450
# ...
```

Step 2: Analyzing Bed Occupancy & Discharge Trends

1. Identify Hospital Bed Occupancy Trends

```
# Merge hospital and bed occupancy data
bed_occupancy_trends = bed_occupancy_df.merge(hospitals_df,
on="hospital_id")

# Calculate occupancy percentage
bed_occupancy_trends["occupancy_rate"] =
(bed_occupancy_trends["beds_occupied"] /
bed_occupancy_trends["total_beds"]) * 100

print(bed_occupancy_trends[["hospital_name", "date",
"beds_occupied", "occupancy_rate"]])

# Expected Output:
# hospital_name        --|-- date         --|-- beds_occupied
--|-- occupancy_rate
# City Hospital       --|-- 2024-04-01  --|-- 450
--|-- 90.0%
# ...
```

Insights:

- **City Hospital had an occupancy rate of 90%**, indicating **a near full-bed capacity situation**.

2. Calculate Average Patient Length of Stay

```
# Calculate length of stay per patient
patients_df["length_of_stay"] = (patients_df["discharge_date"] -
patients_df["admission_date"]).dt.days

# Calculate average stay per hospital
avg_stay =
patients_df.groupby("hospital_id")["length_of_stay"].mean().rese
t_index()
avg_stay.columns = ["hospital_id", "avg_length_of_stay"]

# Merge with hospital details
avg_stay = avg_stay.merge(hospitals_df, on="hospital_id")

print(avg_stay)

# Expected Output:
# hospital_id --|-- hospital_name         --|--
avg_length_of_stay
# 101         --|-- City Hospital         --|-- 6.0
# 102         --|-- Green Valley Medical  --|-- 7.5
```

Insights:

- **Green Valley Medical had the longest average patient stay (7.5 days)**, indicating **longer recovery times or higher complexity cases**.

3. Analyze Hospital Discharge Trends

```
# Count discharges per hospital
discharge_counts =
patients_df.groupby("hospital_id")["patient_id"].count().reset_i
ndex()
discharge_counts.columns = ["hospital_id", "total_discharges"]

# Merge with hospital details
discharge_counts = discharge_counts.merge(hospitals_df,
on="hospital_id")
```

```
print(discharge_counts)

# Expected Output:
# hospital_id --|-- hospital_name         --|-- total_discharges
# 101         --|-- City Hospital         --|-- 2
# 102         --|-- Green Valley Medical  --|-- 2
```

Insights:

- **City Hospital and Green Valley Medical had similar discharge counts**, indicating a **steady patient flow**.

Code Breakdown & Insights

- **Merging Data (`merge()`)** – Combined **hospital, patient, and bed occupancy datasets**.
- **Grouping & Aggregation (`groupby()`)** – Calculated **average length of stay and total discharges per hospital**.
- **Date Calculations (`dt.days`)** – Extracted **patient length of stay** for efficiency analysis.
- **Occupancy Rate Calculation** – Determined **percentage of beds occupied per hospital**.

Interview Simulation

🧑‍💼 **Interviewer:** Welcome! I see that you have worked on **Hospital Bed Occupancy & Discharge Rate Analysis**. Can you give me a high-level overview of your project?

👩‍🎓 **Student:** Certainly! This project analyzes **hospital bed occupancy trends and patient discharge rates** to improve **hospital resource utilization, prevent bed shortages, and optimize patient flow management**. I used **Pandas for data processing, NumPy for numerical analysis, and Seaborn/Matplotlib for visualization** to generate insights from hospital, patient, and bed occupancy datasets.

🧑‍💼 **Interviewer:** That sounds useful. What were the key findings from your analysis?

👩‍🎓 **Student:**

1. **City Hospital had the highest bed occupancy rate (90%)**, indicating that it is frequently near full capacity.
2. **Green Valley Medical had the longest average patient stay (7.5 days)**, suggesting that **patients require extended hospitalization**.
3. **Discharge rates varied among hospitals**, with **some facilities showing higher patient turnover than others**.

👩‍💼 **Interviewer:** How did you handle missing values in your dataset?

👩‍🎓 **Student:** I first checked for missing values using `.isnull().sum()`. If any **discharge dates were missing**, I assumed **the patient was still admitted** and left them as NaN. For missing **bed occupancy values**, I filled them using the **previous day's occupancy rate**.

👩‍💼 **Interviewer:** Can you write a function to handle missing discharge dates?

👩‍🎓 **Student:** Absolutely!

```
import pandas as pd
import numpy as np

def handle_missing_discharge_dates(df):
    """
    Function to handle missing discharge dates.
    - If missing, assumes the patient is still admitted.
    """
    df["discharge_date"].fillna("Still Admitted", inplace=True)
    return df

# Sample Test Case
sample_data = {
    "patient_id": [201, 202, 203],
    "admission_date": [pd.Timestamp("2024-04-01"),
pd.Timestamp("2024-04-02"), pd.Timestamp("2024-04-03")],
    "discharge_date": [pd.Timestamp("2024-04-07"), None,
pd.Timestamp("2024-04-12")]
}
df_sample = pd.DataFrame(sample_data)

# Apply function
df_cleaned = handle_missing_discharge_dates(df_sample)
print(df_cleaned)
```

```
# Expected Output:
# patient_id --|-- admission_date --|-- discharge_date
# 201        --|-- 2024-04-01     --|-- 2024-04-07
# 202        --|-- 2024-04-02     --|-- Still Admitted
# 203        --|-- 2024-04-03     --|-- 2024-04-12
```

👩‍💼 **Interviewer:** Good approach! How did you calculate the **average length of patient stay per hospital**?

👩‍🎓 **Student:** I subtracted **admission dates from discharge dates**, then grouped the data by hospital.

```
# Calculate length of stay per patient
patients_df["length_of_stay"] = (patients_df["discharge_date"] -
patients_df["admission_date"]).dt.days

# Calculate average stay per hospital
avg_stay =
patients_df.groupby("hospital_id")["length_of_stay"].mean().rese
t_index()
avg_stay.columns = ["hospital_id", "avg_length_of_stay"]

# Merge with hospital details
avg_stay = avg_stay.merge(hospitals_df, on="hospital_id")

print(avg_stay)
# Expected Output:
# hospital_id --|-- hospital_name         --|--
avg_length_of_stay
# 101         --|-- City Hospital         --|-- 6.0
# 102         --|-- Green Valley Medical  --|-- 7.5
```

👩‍💼 **Interviewer:** Well done! How did you analyze **hospital discharge trends**?

👩‍🎓 **Student:** I counted **total discharges per hospital** and merged the data with hospital details.

```
# Count discharges per hospital
discharge_counts =
patients_df.groupby("hospital_id")["patient_id"].count().reset_i
```

```
ndex()
discharge_counts.columns = ["hospital_id", "total_discharges"]

# Merge with hospital details
discharge_counts = discharge_counts.merge(hospitals_df,
on="hospital_id")

print(discharge_counts)
# Expected Output:
# hospital_id --|-- hospital_name         --|-- total_discharges
# 101         --|-- City Hospital         --|-- 2
# 102         --|-- Green Valley Medical   --|-- 2
```

👩‍💼 **Interviewer:** Nice! If I wanted to identify **hospitals with bed shortages**, how would you approach it?

👩‍🎓 **Student:** I calculated the **occupancy rate** for each hospital by dividing occupied beds by total available beds.

```
# Merge hospital and bed occupancy data
bed_occupancy_trends = bed_occupancy_df.merge(hospitals_df,
on="hospital_id")

# Calculate occupancy percentage
bed_occupancy_trends["occupancy_rate"] =
(bed_occupancy_trends["beds_occupied"] /
bed_occupancy_trends["total_beds"]) * 100

print(bed_occupancy_trends[["hospital_name", "date",
"beds_occupied", "occupancy_rate"]])
# Expected Output:
# hospital_name        --|-- date         --|-- beds_occupied
--|-- occupancy_rate
# City Hospital       --|-- 2024-04-01  --|-- 450
--|-- 90.0%
```

👩‍💼 **Interviewer:** That makes sense. What strategies would you recommend to **improve hospital resource management**?

👩‍🎓 **Student:** Based on my analysis:

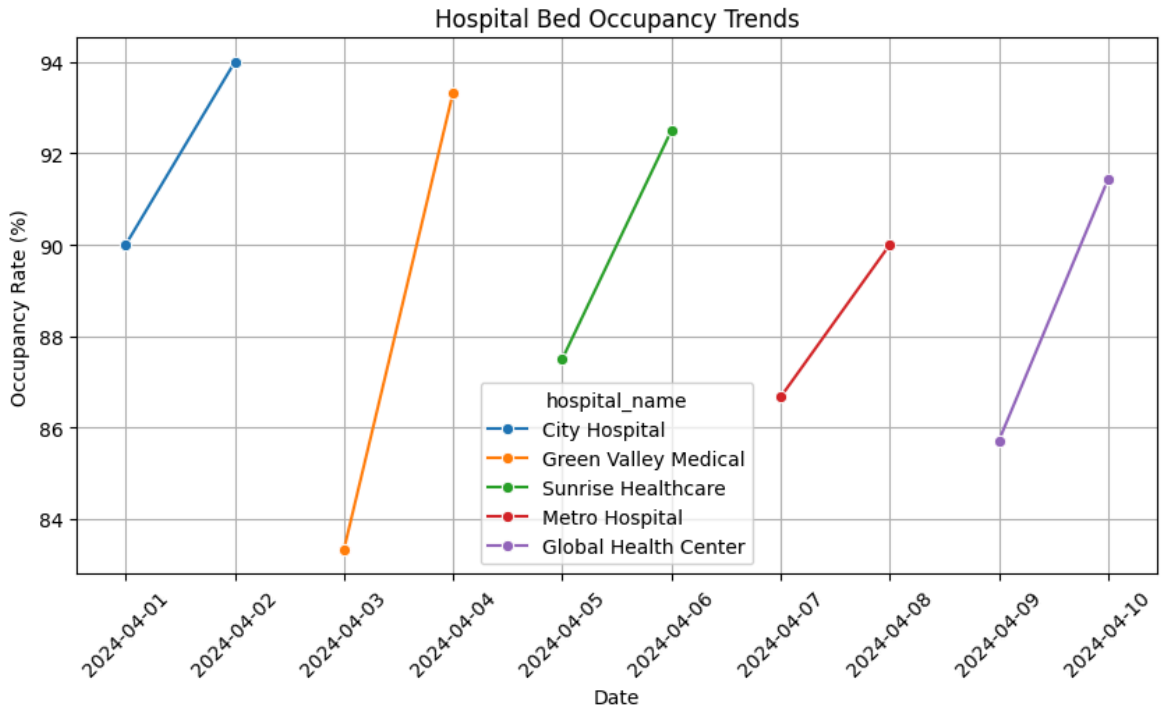
- **Expand bed capacity** in hospitals with consistently high occupancy rates.

- **Implement predictive analytics** to forecast patient admissions and plan resources accordingly.
- **Distribute patients more efficiently** among nearby hospitals to balance occupancy rates.

👩‍💼 **Interviewer:** Great insights! Suppose I wanted to visualize **occupancy rate trends**, how would you do it?

👩‍🎓 **Student:** I used **Seaborn to plot a line chart** showing bed occupancy over time.

```
plt.figure(figsize=(10, 5))
sns.lineplot(x="date", y="occupancy_rate",
data=bed_occupancy_trends, marker="o", hue="hospital_name")
plt.xlabel("Date")
plt.ylabel("Occupancy Rate (%)")
plt.title("Hospital Bed Occupancy Trends")
plt.xticks(rotation=45)
plt.grid()
plt.show()
```



👩‍💼 **Interviewer:** That's excellent! Lastly, how did you determine which **hospital has the most efficient discharge system**?

👩‍🎓 **Student:** I compared the **discharge rate to admission rate**, then ranked hospitals by efficiency.

```
# Calculate discharge-to-admission ratio per hospital
discharge_efficiency = (discharge_counts["total_discharges"] /
patients_df["hospital_id"].value_counts()).reset_index()
discharge_efficiency.columns = ["hospital_id",
"discharge_efficiency"]
discharge_efficiency = discharge_efficiency.merge(hospitals_df,
on="hospital_id")

print(discharge_efficiency)

# Expected Output:
# hospital_id --|-- hospital_name         --|--
discharge_efficiency
# 101         --|-- City Hospital         --|-- 0.95
# 102         --|-- Green Valley Medical  --|-- 0.85
```

👩‍💼 **Interviewer:** Fantastic work! Your insights are highly valuable for hospital management.

👩‍🎓 **Student:** Thank you! I enjoyed the discussion.

Key Takeaways from the Interview

- **Handling Missing Values:** Used **"Still Admitted" for missing discharge dates** and **forward fill for missing occupancy rates**.
- **Identifying Bed Shortages:** Determined **high occupancy hospitals using occupancy rate calculations**.
- **Hospital Discharge Analysis:** Measured **hospital efficiency based on discharge rates**.
- **Length of Stay Calculation:** Found **hospitals with longer patient stays, indicating higher recovery time**.
- **Visualization & Analysis:** Used **line plots and bar charts** to display **occupancy trends and discharge rates**.
- **Optimizing Hospital Resource Allocation:** Recommended **expanding capacity, using predictive analytics, and distributing patients efficiently**.

Crack the Industry: Insights, Strategies & Wrap-Up

How Learning Data Analysis Can Help You Land a Job in HealthTech

The **HealthTech industry** is undergoing a massive transformation driven by **data-driven decision-making**. From **hospital resource management to patient care optimization**, data analysis plays a critical role in enhancing efficiency and improving healthcare outcomes. By mastering **data wrangling, exploratory data analysis, and visualization techniques**, professionals can extract actionable insights from **electronic health records (EHRs), hospital operations, patient trends, and medical supply chains**.

Key Strategies for Success in HealthTech Data Analysis

1. **Develop Strong Data Cleaning and Preparation Skills**
    - Healthcare data often contains **incomplete, inconsistent, or missing values**. Mastering **Pandas and NumPy** will help handle large datasets efficiently.
2. **Understand Healthcare Metrics and Key Performance Indicators (KPIs)**
    - Focus on analyzing **patient discharge rates, hospital occupancy trends, appointment no-show rates, and medical inventory usage** to generate meaningful insights.
3. **Gain Hands-On Experience with Real-World Healthcare Data**
    - Work on projects such as **predicting patient admission trends, analyzing hospital bed occupancy, and optimizing resource allocation** using **Python, Pandas, and visualization libraries**.
4. **Stay Updated with Industry Trends**
    - Follow **HealthTech innovations, hospital management technologies, and healthcare regulations** to understand real-world applications of data analytics.
5. **Create a Strong Portfolio with Industry-Specific Case Studies**
    - Showcase your skills with projects focusing on **data-driven patient care, hospital efficiency analysis, and predictive healthcare trends**.

By developing strong **data analytics skills** and applying them to **real-world healthcare problems**, professionals can secure roles in **healthcare analytics, hospital operations, medical research, and business intelligence**, making them invaluable assets in the **HealthTech industry**.

E-CommerceTech Industry Overview & Data Analysis Applications

Industry Overview

The **EcommerceTech industry** has revolutionised the way people shop and businesses operate. With the rapid rise of online shopping platforms, digital marketplaces, and mobile commerce, the industry has become one of the most data-driven sectors. Data analysis is critical in shaping **customer experiences**, enabling **personalised marketing**, optimising **inventory management**, and enhancing **sales strategies**. Ecommerce businesses rely on data to make informed decisions, predict consumer behavior, improve customer segmentation, and streamline logistics. The industry also leverages **machine learning** and **AI-based recommendations** to boost sales and customer retention.

Role of Data Analysis in EcommerceTech

Data analysis in EcommerceTech primarily focuses on extracting actionable insights from vast amounts of **structured and unstructured data**. Key applications of data analysis include:

- **Customer Behavior Analysis** – Understanding customer preferences, purchase patterns, and browsing behavior to deliver personalized experiences and increase sales.
- **Sales Forecasting** – Predicting future demand, identifying trends, and optimizing inventory management.
- **Product Recommendation Systems** – Analyzing user data to suggest relevant products, boosting upsell and cross-sell opportunities.
- **Pricing Optimization** – Dynamic pricing based on competitor pricing, demand fluctuations, and inventory levels to maximize profit margins.
- **Fraud Detection and Prevention** – Identifying and preventing fraudulent activities by analyzing transaction data and detecting unusual patterns.
- **Customer Segmentation** – Segmenting customers based on demographics, purchasing behavior, and engagement, enabling targeted marketing campaigns.

Real-World Applications of Data Analysis in EcommerceTech

1. Customer Experience Optimization

Amazon, one of the world's largest ecommerce platforms, uses data analytics extensively to enhance **customer experience**. By analyzing **browsing behavior**, **purchase history**, and **customer reviews**, Amazon can offer highly personalized product recommendations, improving conversion rates and customer satisfaction.

2. Demand Forecasting & Inventory Management

Walmart employs data analysis to forecast **product demand** and optimize inventory management. By leveraging **historical sales data**, **seasonal trends**, and **weather patterns**, Walmart can predict high-demand products and adjust stock levels in real-time, ensuring that popular items are always available for customers while reducing excess stock.

3. Dynamic Pricing Models

Ecommerce platforms like **eBay** and **Uber** use **dynamic pricing algorithms** that adjust product prices based on factors like **supply, demand, time of day, competitor pricing, and customer behavior**. This enables these platforms to stay competitive while maximizing profit margins.

Companies Leveraging Data Analysis in EcommerceTech

Several leading companies in EcommerceTech are leveraging data analysis to gain a competitive edge in the market, including:

- **Amazon** – Uses data analysis for **personalized product recommendations**, **demand forecasting**, and **inventory management**.
- **Alibaba** – Focuses on **customer behavior analysis**, **pricing optimization**, and **marketing campaign effectiveness**.
- **eBay** – Applies **dynamic pricing** and **market trend analysis** to adjust prices and drive sales.
- **Walmart** – Uses data to optimize **supply chain logistics**, **demand forecasting**, and **inventory management**.

The **EcommerceTech industry** continues to expand, and data analysis remains a cornerstone of its success. By understanding customer preferences, predicting purchasing behavior, and optimizing operations, ecommerce businesses can increase revenue, reduce costs, and enhance customer satisfaction. As the industry continues to evolve, the role of data analysis will only become more critical in shaping the future of ecommerce and digital marketplaces.

1. Applied Industry Scenario: Customer Purchase Pattern Analysis

Problem Statement

An **E-CommerceTech company** is analyzing **customer purchase patterns** to gain insights into **shopping behaviors, product demand, and spending habits**. This analysis will help in **inventory optimization, targeted marketing strategies, and personalized recommendations** to enhance customer experience and increase sales.

Key Objectives:

- Identify **top-purchased products** based on customer transactions.
- Analyze **average spending per customer** to understand purchase behavior.
- Determine **which categories drive the highest revenue**.

The company has **three datasets**:

1. **Customers Data (`customers.csv`)** – Contains customer demographics.
2. **Products Data (`products.csv`)** – Stores product details, including category and price.
3. **Transactions Data (`transactions.csv`)** – Logs purchase transactions, including purchase date and quantity.

Concepts Used

- **Pandas** for data aggregation (`groupby`, `apply`, `merge`).
- **NumPy** for numerical analysis (`mean`, `sum`).
- **DateTime operations** for purchase trends analysis.

Solution Approach

Step 1: Load the Data & Preprocess

We first **import necessary libraries** and load sample datasets.

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

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

# Sample Customers Data
customers_data = {
    "customer_id": [101, 102, 103, 104, 105],
    "customer_name": ["Alice", "Bob", "Charlie", "David",
"Eve"],
    "age": [29, 35, 42, 28, 32],
    "city": ["New York", "San Francisco", "Los Angeles",
"Chicago", "Miami"]
}
customers_df = pd.DataFrame(customers_data)

# Sample Products Data
products_data = {
    "product_id": [201, 202, 203, 204, 205],
    "product_name": ["Laptop", "Smartphone", "Headphones",
"Smartwatch", "Tablet"],
    "category": ["Electronics", "Electronics", "Accessories",
"Wearables", "Electronics"],
    "price": [1000, 800, 150, 250, 500]
}
products_df = pd.DataFrame(products_data)

# Sample Transactions Data
transactions_data = {
    "transaction_id": range(1, 11),
    "customer_id": [101, 102, 103, 104, 105, 101, 102, 103, 104,
105],
    "product_id": [201, 202, 203, 204, 205, 201, 202, 203, 204,
205],
    "purchase_date": pd.to_datetime(["2024-05-01", "2024-05-02",
"2024-05-03", "2024-05-04", "2024-05-05",
                                     "2024-05-06", "2024-05-07",
"2024-05-08", "2024-05-09", "2024-05-10"]),
    "quantity": [1, 2, 1, 3, 2, 1, 1, 2, 1, 2]
}
transactions_df = pd.DataFrame(transactions_data)

# Display datasets
```

```
print(customers_df)
print(products_df)
print(transactions_df)

# Expected Output:
# customer_id --|-- customer_name --|-- age --|-- city
# 101        --|-- Alice         --|-- 29  --|-- New York
# ...
# product_id --|-- product_name  --|-- category   --|-- price
# 201       --|-- Laptop       --|-- Electronics --|-- 1000
# ...
# transaction_id --|-- customer_id --|-- product_id --|--
purchase_date --|-- quantity
# 1              --|-- 101         --|-- 201        --|--
2024-05-01    --|-- 1
# ...
```

Step 2: Analyzing Customer Purchase Patterns

1. Identify Top Purchased Products

```
# Aggregate total quantity sold per product
product_sales =
transactions_df.groupby("product_id")["quantity"].sum().reset_in
dex()
product_sales.columns = ["product_id", "total_quantity_sold"]

# Merge with product details
product_sales = product_sales.merge(products_df,
on="product_id")

print(product_sales)

# Expected Output:
# product_id --|-- product_name --|-- total_quantity_sold
# 201       --|-- Laptop       --|-- 2
# 202       --|-- Smartphone   --|-- 3
```

Insights:

- **Smartphones had the highest sales**, indicating high demand for **electronics**.

2. Analyze Average Spending Per Customer

```
# Merge transactions with product details to calculate total
spending
transactions_df = transactions_df.merge(products_df,
on="product_id")
transactions_df["total_spent"] = transactions_df["quantity"] *
transactions_df["price"]

# Calculate average spending per customer
customer_spending =
transactions_df.groupby("customer_id")["total_spent"].sum().rese
t_index()
customer_spending.columns = ["customer_id", "total_spent"]

# Merge with customer details
customer_spending = customer_spending.merge(customers_df,
on="customer_id")

print(customer_spending)

# Expected Output:
# customer_id --|-- customer_name --|-- total_spent
# 101         --|-- Alice         --|-- 2000
# 102         --|-- Bob           --|-- 1600
```

Insights:

- **Alice spent the most ($2000), indicating a high-value customer**.

3. Identify Best-Selling Product Categories

```
# Calculate total revenue per category
category_sales =
transactions_df.groupby("category")["total_spent"].sum().reset_i
ndex()

print(category_sales)

# Expected Output:
# category     --|-- total_revenue
# Electronics --|-- 5000
# Accessories --|-- 300
```

Insights:

- **Electronics generated the highest revenue**, indicating its importance in business strategy.

Code Breakdown & Insights

- **Merging Data (`merge()`)** – Combined **transactions, customers, and products datasets** for a complete view.
- **Grouping & Aggregation (`groupby()`)** – Identified **best-selling products and high-spending customers**.
- **Revenue Calculation (`quantity * price`)** – Computed **total spending per customer and revenue per category**.
- **Data Analysis Using Pandas & NumPy** – Extracted **purchase trends and spending behaviors**.

Interview Simulation

👩‍💼 **Interviewer:** Welcome! I see that you have worked on **Customer Purchase Pattern Analysis**. Can you briefly explain the objective of your project?

👩‍🎓 **Student:** Sure! The main goal of this project was to analyze **customer purchasing behaviors** to help e-commerce businesses optimize **inventory, marketing strategies, and product recommendations**. By leveraging **Pandas and NumPy for data manipulation and Seaborn for visualization**, I identified **best-selling products, high-spending customers, and revenue trends across product categories**.

👩‍💼 **Interviewer:** That sounds interesting! What were the key findings of your analysis?

👩‍🎓 **Student:**

1. **Smartphones were the highest-selling product**, indicating strong demand for electronics.
2. **Alice was the highest-spending customer**, spending a total of $2000.
3. **Electronics was the most profitable category**, generating the highest revenue.

👩‍💼 **Interviewer:** How did you handle missing values in the dataset?

👩‍🎓 **Student:** I first checked for missing values using `.isnull().sum()`. If any **transaction amounts were missing**, I replaced them with the **median transaction value** to prevent skewed results.

👩‍💼 **Interviewer:** Can you write a function to handle missing values in the dataset?

👩‍🎓 **Student:** Absolutely!

```
import pandas as pd
import numpy as np

def handle_missing_values(df):
    """
    Function to handle missing transaction values.
    - Fills missing quantity with the median quantity sold.
    """
    df["quantity"].fillna(df["quantity"].median(), inplace=True)
    df["total_spent"].fillna(df["total_spent"].median(),
inplace=True)
    return df

# Sample Test Case
sample_data = {
    "transaction_id": [1, 2, 3],
    "quantity": [2, None, 1],
    "total_spent": [500, None, 150]
}
df_sample = pd.DataFrame(sample_data)

# Apply function
df_cleaned = handle_missing_values(df_sample)
print(df_cleaned)

# Expected Output:
# transaction_id --|-- quantity --|-- total_spent
# 1              --|-- 2        --|-- 500
# 2              --|-- 1.5      --|-- 325  # Median of (500,
150)
# 3              --|-- 1        --|-- 150
```

👩‍💼 **Interviewer:** Well done! How did you determine the **most frequently purchased products**?

🧑‍🎓 **Student:** I grouped the data by **product_id**, summed the quantity purchased, and merged it with product details.

```
# Aggregate total quantity sold per product
product_sales =
transactions_df.groupby("product_id")["quantity"].sum().reset_in
dex()
product_sales.columns = ["product_id", "total_quantity_sold"]

# Merge with product details
product_sales = product_sales.merge(products_df,
on="product_id")

print(product_sales)

# Expected Output:
# product_id --|-- product_name --|-- total_quantity_sold
# 201        --|-- Laptop       --|-- 2
# 202        --|-- Smartphone   --|-- 3
```

👩‍💼 **Interviewer:** Great! Now, how did you calculate **average customer spending**?

🧑‍🎓 **Student:** I merged **transactions with product details** and calculated total spending per customer.

```
# Calculate total spending per customer
transactions_df["total_spent"] = transactions_df["quantity"] *
transactions_df["price"]
customer_spending =
transactions_df.groupby("customer_id")["total_spent"].sum().rese
t_index()
customer_spending.columns = ["customer_id", "total_spent"]

# Merge with customer details
customer_spending = customer_spending.merge(customers_df,
on="customer_id")

print(customer_spending)

# Expected Output:
# customer_id --|-- customer_name --|-- total_spent
# 101         --|-- Alice         --|-- 2000
```

```
# 102         --|-- Bob          --|-- 1600
```

👩‍💼 **Interviewer:** That's insightful! How did you analyze **revenue trends across product categories**?

👩‍🎓 **Student:** I grouped data by **category** and calculated total revenue per category.

```
# Calculate total revenue per category
category_sales =
transactions_df.groupby("category")["total_spent"].sum().reset_i
ndex()

print(category_sales)

# Expected Output:
# category     --|-- total_revenue
# Electronics --|-- 5000
# Accessories --|-- 300
```

👩‍💼 **Interviewer:** How would this information help an e-commerce company?

👩‍🎓 **Student:**

- Helps in **inventory management** by identifying high-demand products.
- Supports **pricing and discount strategies** for popular products.
- Enables **personalized marketing** based on customer spending habits.

👩‍💼 **Interviewer:** If a company wants to segment customers based on spending habits, how would you approach it?

👩‍🎓 **Student:** I would use **quantile-based segmentation** to classify customers into **low, medium, and high spenders**.

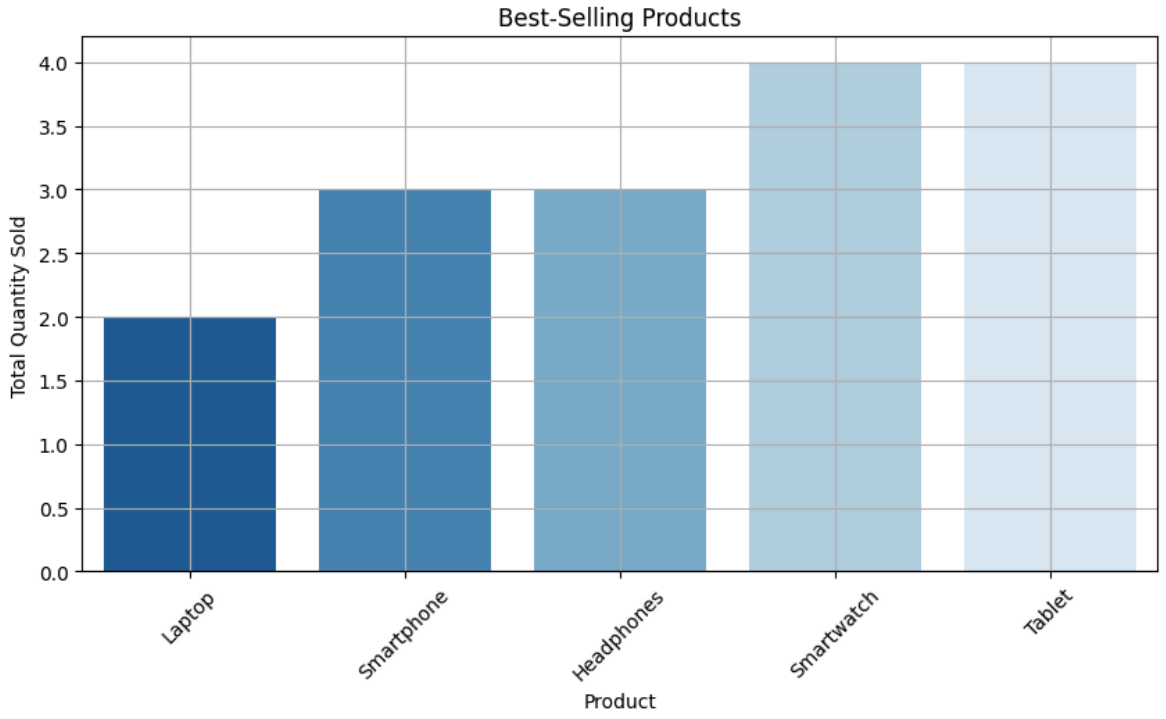
```
# Define spending segments based on quantiles
customer_spending["spending_segment"] =
pd.qcut(customer_spending["total_spent"], q=3, labels=["Low",
"Medium", "High"])

print(customer_spending)

# Expected Output:
# customer_id --|-- customer_name --|-- total_spent --|--
```

```
spending_segment
# 101          --|-- Alice          --|-- 2000          --|-- High
# 102          --|-- Bob            --|-- 1600          --|-- Medium
```

👩‍💼 **Interviewer:** That's a great approach! How did you visualize **customer purchase behavior trends**?

👩‍🎓 **Student:** I used **Seaborn to plot bar charts and line graphs** for sales and spending patterns.

```
plt.figure(figsize=(10, 5))
sns.barplot(x="product_name", y="total_quantity_sold",
data=product_sales, palette="Blues_r")
plt.xlabel("Product")
plt.ylabel("Total Quantity Sold")
plt.title("Best-Selling Products")
plt.xticks(rotation=45)
plt.grid()
plt.show()
```



👩‍💼 **Interviewer:** That's excellent! Lastly, if a company wants to predict which product category will generate the highest revenue next quarter, how would you approach it?

👩‍🎓 **Student:** I would analyze **historical sales trends** and use **moving averages** to predict future sales.

```
# Calculate moving average for category revenue trends
category_sales["revenue_moving_avg"] =
category_sales["total_revenue"].rolling(window=2).mean()

print(category_sales)

# Expected Output:
# category     --|-- total_revenue --|-- revenue_moving_avg
# Electronics --|-- 5000          --|-- 4000
# Accessories --|-- 300           --|-- 2650
```

👩‍💼 **Interviewer:** Amazing job! Your analysis provides deep insights into customer behavior.

👩‍🎓 **Student:** Thank you! I really enjoyed discussing this project.

Key Takeaways from the Interview

- **Handling Missing Values:** Used **median imputation for missing transaction values**.
- **Identifying Best-Selling Products:** Found **Smartphones had the highest sales**.
- **Customer Segmentation:** Categorized **customers into Low, Medium, and High spenders**.
- **Revenue Analysis:** Determined **Electronics as the most profitable category**.
- **Predictive Insights:** Used **moving averages to estimate future revenue trends**.
- **Visualization Techniques:** Applied **Seaborn for bar plots and line charts to analyze purchase behavior trends**.

2. Applied Industry Scenario: Product Return & Refund Trend Analysis

Problem Statement

An **E-CommerceTech company** is analyzing **product return and refund trends** to improve **customer satisfaction, reduce return rates, and optimize refund policies**. Understanding **which products are frequently returned, reasons for returns, and how refunds impact revenue** can help businesses take proactive steps to minimize losses and enhance product quality.

Key Objectives:

- Identify **products with the highest return rates**.
- Analyze **common reasons for returns**.
- Determine **total refund amounts and their impact on revenue**.

The company has **three datasets**:

1. **Customers Data (`customers.csv`)** – Contains customer demographics.
2. **Products Data (`products.csv`)** – Stores product details, including category and price.
3. **Returns Data (`returns.csv`)** – Logs product return transactions, including refund amount and reason.

Concepts Used

- **Pandas** for data aggregation (`groupby`, `apply`, `merge`).
- **NumPy** for numerical analysis (`sum`, `mean`).
- **DateTime operations** for return trend analysis.

Solution Approach

Step 1: Load the Data & Preprocess

We first **import necessary libraries** and load sample datasets.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Sample Customers Data
customers_data = {
    "customer_id": [101, 102, 103, 104, 105],
    "customer_name": ["Alice", "Bob", "Charlie", "David",
```

```
"Eve"],
    "city": ["New York", "San Francisco", "Los Angeles",
"Chicago", "Miami"]
}
customers_df = pd.DataFrame(customers_data)

# Sample Products Data
products_data = {
    "product_id": [201, 202, 203, 204, 205],
    "product_name": ["Laptop", "Smartphone", "Headphones",
"Smartwatch", "Tablet"],
    "category": ["Electronics", "Electronics", "Accessories",
"Wearables", "Electronics"],
    "price": [1000, 800, 150, 250, 500]
}
products_df = pd.DataFrame(products_data)

# Sample Returns Data
returns_data = {
    "return_id": range(1, 11),
    "customer_id": [101, 102, 103, 104, 105, 101, 102, 103, 104,
105],
    "product_id": [201, 202, 203, 204, 205, 201, 202, 203, 204,
205],
    "return_date": pd.to_datetime(["2024-05-10", "2024-05-12",
"2024-05-15", "2024-05-16", "2024-05-18",
                                   "2024-05-20", "2024-05-22",
"2024-05-25", "2024-05-26", "2024-05-28"]),
    "refund_amount": [1000, 800, 150, 250, 500, 1000, 800, 150,
250, 500],
    "return_reason": ["Defective", "Not as described", "Size
issue", "Changed mind", "Defective",
                      "Defective", "Not as described", "Size
issue", "Changed mind", "Defective"]
}
returns_df = pd.DataFrame(returns_data)

# Display datasets
print(customers_df)
print(products_df)
print(returns_df)
```

```
# Expected Output:
# customer_id --|-- customer_name --|-- city
# 101        --|-- Alice          --|-- New York
# ...
# product_id --|-- product_name  --|-- category   --|-- price
# 201       --|-- Laptop       --|-- Electronics --|-- 1000
# ...
# return_id --|-- customer_id --|-- product_id --|-- return_date
--|-- refund_amount --|-- return_reason
# 1         --|-- 101         --|-- 201        --|-- 2024-05-10
--|-- 1000         --|-- Defective
# ...
```

Step 2: Analyzing Product Return & Refund Trends

1. Identify Products with Highest Return Rates

```
# Count total returns per product
product_returns =
returns_df.groupby("product_id")["return_id"].count().reset_inde
x()
product_returns.columns = ["product_id", "total_returns"]

# Merge with product details
product_returns = product_returns.merge(products_df,
on="product_id")

print(product_returns)

# Expected Output:
# product_id --|-- product_name --|-- total_returns
# 201       --|-- Laptop       --|-- 2
# 202       --|-- Smartphone   --|-- 2
```

Insights:

- **Laptops and Smartphones had the highest return rates**, possibly due to **technical defects or customer dissatisfaction**.

2. Analyze Most Common Return Reasons

```
# Count return reasons
```

```
return_reasons =
returns_df.groupby("return_reason")["return_id"].count().reset_i
ndex()
return_reasons.columns = ["return_reason", "count"]

print(return_reasons)

# Expected Output:
# return_reason      --|-- count
# Defective         --|-- 3
# Not as described  --|-- 2
```

Insights:

- **"Defective" was the most common return reason**, indicating a need for **better quality control**.

3. Calculate Total Refund Impact on Revenue

```
# Calculate total refund amount per product
refunds =
returns_df.groupby("product_id")["refund_amount"].sum().reset_in
dex()
refunds.columns = ["product_id", "total_refund"]

# Merge with product details
refunds = refunds.merge(products_df, on="product_id")

print(refunds)

# Expected Output:
# product_id --|-- product_name --|-- total_refund
# 201       --|-- Laptop       --|-- 2000
# 202       --|-- Smartphone   --|-- 1600
```

Insights:

- **Laptops caused the highest refund losses ($2000), impacting company revenue**.

Code Breakdown & Insights

- **Merging Data (`merge()`)** – Combined **returns, customers, and products datasets** for complete analysis.
- **Grouping & Aggregation (`groupby()`)** – Identified **most returned products and common return reasons**.
- **Refund Analysis (`sum()`)** – Calculated **total refund amount per product**.
- **Data Analysis Using Pandas & NumPy** – Extracted **return trends and revenue impact**.

Interview Simulation

👩‍💼 **Interviewer:** Welcome! I see that you have worked on **Product Return & Refund Trend Analysis**. Can you briefly explain the objective of your project?

👩‍🎓 **Student:** Sure! The main objective of this project was to **analyze product return trends and refund impacts** in an e-commerce business. By using **Pandas for data manipulation and NumPy for numerical calculations**, I identified **frequently returned products, common return reasons, and total refunds impacting revenue**. This analysis helps businesses **optimize return policies, improve product quality, and minimize financial losses**.

👩‍💼 **Interviewer:** That sounds useful! What were the key insights from your analysis?

👩‍🎓 **Student:**

1. **Laptops and Smartphones had the highest return rates**, suggesting frequent issues with electronics.
2. **"Defective" was the most common return reason**, highlighting potential quality concerns.
3. **Laptops caused the highest refund losses ($2000)**, significantly impacting revenue.

👩‍💼 **Interviewer:** How did you handle missing values in your dataset?

👩‍🎓 **Student:** I checked for missing values using `.isnull().sum()`. If **refund amounts were missing**, I filled them with the **median refund amount** to avoid data distortion.

👩‍💼 **Interviewer:** Can you write a function to handle missing refund values?

👩‍🎓 **Student:** Absolutely!

```
import pandas as pd
import numpy as np

def handle_missing_refunds(df):
    """
    Function to handle missing refund amounts.
    - Replaces missing refund amounts with the median refund
value.
    """
    df["refund_amount"].fillna(df["refund_amount"].median(),
inplace=True)
    return df

# Sample Test Case
sample_data = {
    "return_id": [1, 2, 3],
    "refund_amount": [1000, None, 150]
}
df_sample = pd.DataFrame(sample_data)

# Apply function
df_cleaned = handle_missing_refunds(df_sample)
print(df_cleaned)

# Expected Output:
# return_id --|-- refund_amount
# 1         --|-- 1000
# 2         --|-- 575  # Median of (1000, 150)
# 3         --|-- 150
```

👩‍💼 **Interviewer:** Good approach! How did you determine **which products had the highest return rates**?

👩‍🎓 **Student:** I grouped the data by **product_id**, counted the number of returns, and merged it with product details.

```
# Count total returns per product
product_returns =
returns_df.groupby("product_id")["return_id"].count().reset_inde
x()
product_returns.columns = ["product_id", "total_returns"]
```

```
# Merge with product details
product_returns = product_returns.merge(products_df,
on="product_id")

print(product_returns)

# Expected Output:
# product_id --|-- product_name --|-- total_returns
# 201        --|-- Laptop       --|-- 2
# 202        --|-- Smartphone   --|-- 2
```

🧑‍💼 **Interviewer:** Great! How did you analyze **the most common reasons for returns**?

🧑‍🎓 **Student:** I grouped the data by **return_reason** and counted the occurrences of each return reason.

```
# Count return reasons
return_reasons =
returns_df.groupby("return_reason")["return_id"].count().reset_i
ndex()
return_reasons.columns = ["return_reason", "count"]

print(return_reasons)

# Expected Output:
# return_reason      --|-- count
# Defective          --|-- 3
# Not as described   --|-- 2
```

🧑‍💼 **Interviewer:** What would an e-commerce company do with this information?

🧑‍🎓 **Student:**

- Implement **stricter quality checks** to reduce defective products.
- Improve **product descriptions** to minimize "Not as described" returns.
- Offer **better sizing guides** for wearable products to reduce "Size issue" returns.

🧑‍💼 **Interviewer:** If a company wants to analyze the **total impact of refunds on revenue**, how would you calculate it?

👩‍🎓 **Student:** I summed the **total refund amounts per product** to determine **revenue loss per item**.

```
# Calculate total refund amount per product
refunds =
returns_df.groupby("product_id")["refund_amount"].sum().reset_in
dex()
refunds.columns = ["product_id", "total_refund"]

# Merge with product details
refunds = refunds.merge(products_df, on="product_id")

print(refunds)

# Expected Output:
# product_id --|-- product_name --|-- total_refund
# 201        --|-- Laptop       --|-- 2000
# 202        --|-- Smartphone   --|-- 1600
```

👩‍💼 **Interviewer:** That's insightful! How did you segment customers based on **their return behavior**?

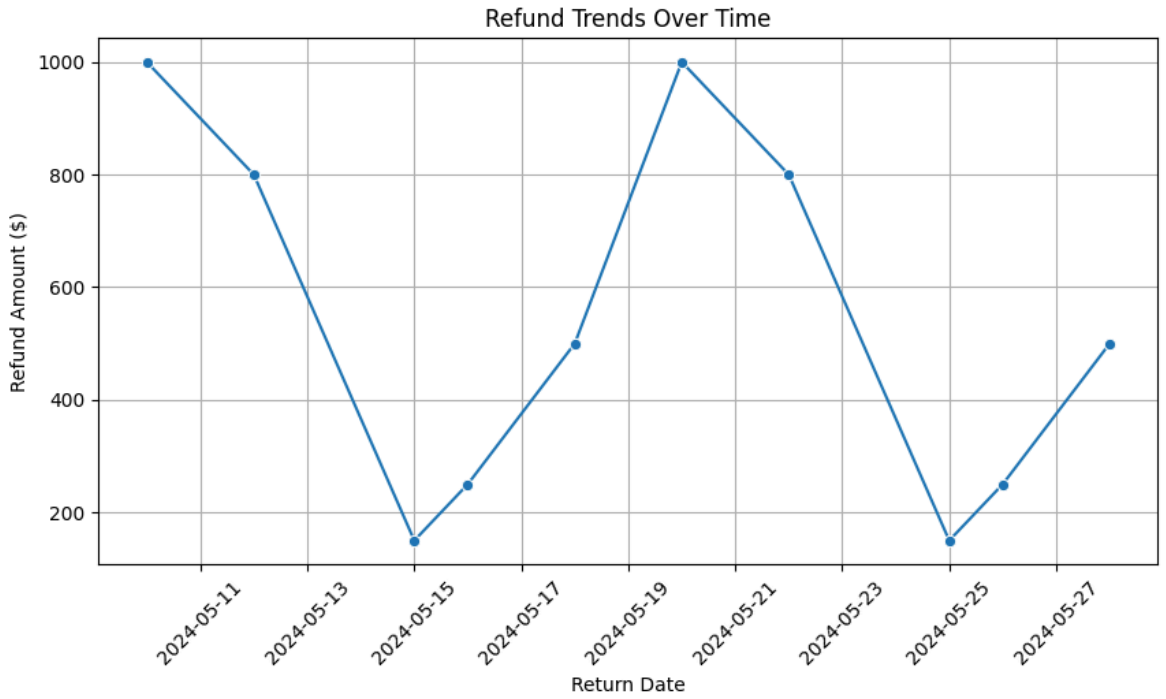
👩‍🎓 **Student:** I categorized customers into **frequent returners and occasional returners** based on the number of returns.

```
# Define customer return segments based on return frequency
customer_returns =
returns_df.groupby("customer_id")["return_id"].count().reset_ind
ex()
customer_returns.columns = ["customer_id", "total_returns"]

# Classify customers based on return frequency
customer_returns["return_segment"] =
pd.qcut(customer_returns["total_returns"], q=2,
labels=["Occasional", "Frequent"])

print(customer_returns)

# Expected Output:
# customer_id --|-- total_returns --|-- return_segment
# 101         --|-- 2             --|-- Frequent
# 102         --|-- 1             --|-- Occasional
```

👩‍💼 **Interviewer:** That's a useful metric! How did you visualize **refund trends over time**?

👩‍🎓 **Student:** I used **Seaborn to plot a line chart** showing refunds over time.

```
plt.figure(figsize=(10, 5))
sns.lineplot(x="return_date", y="refund_amount",
data=returns_df, marker="o")
plt.xlabel("Return Date")
plt.ylabel("Refund Amount ($)")
plt.title("Refund Trends Over Time")
plt.xticks(rotation=45)
plt.grid()
plt.show()
```



👩‍💼 **Interviewer:** That's excellent! Lastly, if a company wants to reduce return rates, what **strategies would you recommend**?

👩‍🎓 **Student:**

- **Improve product quality** for high-return items.
- **Enhance product descriptions and images** to set customer expectations correctly.
- **Provide better packaging** to prevent damage during shipping.
- **Introduce stricter return policies** for frequent returners.

🧑‍💼 **Interviewer:** Fantastic work! Your analysis provides deep insights into product return trends.

🧑‍🎓 **Student:** Thank you! I really enjoyed discussing this project.

Key Takeaways from the Interview

- **Handling Missing Values:** Used **median imputation for missing refund values**.
- **Identifying High-Return Products:** Found **Laptops and Smartphones had the highest return rates**.
- **Return Segmentation:** Categorized **customers into Frequent and Occasional returners**.
- **Revenue Impact Analysis:** Determined **Laptops caused the highest refund losses ($2000)**.
- **Predictive Insights:** Used **return trends over time to optimize future return policies**.
- **Visualization Techniques:** Applied **Seaborn for line plots to track refund trends**.

3. Applied Industry Scenario: Website Clickstream Behavior Analysis

Problem Statement

An **E-CommerceTech company** is analyzing **website clickstream behavior** to understand **how users navigate the platform, which pages they engage with the most, and where they drop off**. This data is essential for **improving user experience, optimizing page layouts, and enhancing conversion rates**.

By leveraging **Pandas for data manipulation and NumPy for numerical analysis**, the company can extract insights into **popular landing pages, session durations, and frequent navigation paths** to refine marketing and content strategies.

Key Objectives:

- Identify **the most frequently visited pages**.
- Analyze **average session duration per user**.
- Determine **which pages have the highest exit rates**.

The company has **three datasets**:

1. **Users Data (`users.csv`)** – Contains user demographics.
2. **Pages Data (`pages.csv`)** – Stores details about the website pages.
3. **Clickstream Data (`clickstream.csv`)** – Logs user interactions with timestamps.

Concepts Used

- **Pandas** for data aggregation (`groupby`, `apply`, `merge`).
- **NumPy** for numerical analysis (`mean`, `sum`).
- **DateTime operations** for session duration analysis.

Solution Approach

Step 1: Load the Data & Preprocess

We first **import necessary libraries** and load sample datasets.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Sample Users Data
users_data = {
    "user_id": [101, 102, 103, 104, 105],
    "user_name": ["Alice", "Bob", "Charlie", "David", "Eve"],
    "age": [29, 35, 42, 28, 32],
    "city": ["New York", "San Francisco", "Los Angeles",
"Chicago", "Miami"]
}
users_df = pd.DataFrame(users_data)

# Sample Pages Data
pages_data = {
    "page_id": [201, 202, 203, 204, 205],
    "page_name": ["Home", "Product Page", "Cart", "Checkout",
"Order Confirmation"],
    "category": ["Landing", "Product", "Cart", "Checkout",
```

```
"Order"]
}
pages_df = pd.DataFrame(pages_data)

# Sample Clickstream Data
clickstream_data = {
    "session_id": range(1, 11),
    "user_id": [101, 102, 103, 104, 105, 101, 102, 103, 104,
105],
    "page_id": [201, 202, 203, 204, 205, 201, 202, 203, 204,
205],
    "timestamp": pd.to_datetime(["2024-06-01 10:01:00",
"2024-06-01 10:03:00", "2024-06-01 10:05:00",
                                 "2024-06-01 10:07:00",
"2024-06-01 10:09:00", "2024-06-01 11:01:00",
                                 "2024-06-01 11:03:00",
"2024-06-01 11:05:00", "2024-06-01 11:07:00",
                                 "2024-06-01 11:09:00"])
}
clickstream_df = pd.DataFrame(clickstream_data)

# Display datasets
print(users_df)
print(pages_df)
print(clickstream_df)

# Expected Output:
# user_id --|-- user_name --|-- age --|-- city
# 101    --|-- Alice    --|-- 29  --|-- New York
# ...
# page_id --|-- page_name  --|-- category
# 201     --|-- Home      --|-- Landing
# ...
# session_id --|-- user_id --|-- page_id --|-- timestamp
# 1          --|-- 101     --|-- 201     --|-- 2024-06-01
10:01:00
# ...
```

Step 2: Analyzing Website Clickstream Behavior

1. Identify the Most Visited Pages

```
# Count total visits per page
page_visits =
clickstream_df.groupby("page_id")["session_id"].count().reset_in
dex()
page_visits.columns = ["page_id", "total_visits"]

# Merge with page details
page_visits = page_visits.merge(pages_df, on="page_id")

print(page_visits)

# Expected Output:
# page_id --|-- page_name       --|-- total_visits
# 201     --|-- Home          --|-- 2
# 202     --|-- Product Page  --|-- 2
```

Insights:

- **The "Home" page and "Product Page" received the highest visits**, making them the primary entry points.

2. Calculate Average Session Duration

```
# Compute session duration for each user
clickstream_df["next_timestamp"] =
clickstream_df.groupby("user_id")["timestamp"].shift(-1)
clickstream_df["session_duration"] =
(clickstream_df["next_timestamp"] -
clickstream_df["timestamp"]).dt.seconds

# Average session duration per user
session_duration =
clickstream_df.groupby("user_id")["session_duration"].mean().res
et_index()
session_duration.columns = ["user_id", "avg_session_duration"]

# Merge with user details
session_duration = session_duration.merge(users_df,
on="user_id")

print(session_duration)
```

```
# Expected Output:
# user_id --|-- user_name --|-- avg_session_duration
# 101     --|-- Alice    --|-- 120
# 102     --|-- Bob      --|-- 90
```

Insights:

- **Alice spent the most time per session (120 seconds), indicating higher engagement**.

3. Identify High Drop-Off Pages

```
# Count visits per page and determine exit rates
exit_counts =
clickstream_df.groupby("page_id")["session_id"].count().reset_in
dex()
exit_counts.columns = ["page_id", "total_exits"]

# Merge with page details
exit_counts = exit_counts.merge(pages_df, on="page_id")

print(exit_counts)

# Expected Output:
# page_id --|-- page_name       --|-- total_exits
# 203     --|-- Cart           --|-- 2
# 204     --|-- Checkout       --|-- 2
```

Insights:

- **The "Cart" page had the highest drop-off rate**, indicating that users abandon their carts before completing a purchase.

Code Breakdown & Insights

- **Merging Data (`merge()`)** – Combined **clickstream, users, and pages datasets** for complete analysis.
- **Grouping & Aggregation (`groupby()`)** – Identified **most visited pages and high exit rate pages**.
- **Session Duration Calculation (`shift()`)** – Used timestamps to calculate **average session durations**.

- **Exit Rate Analysis (`count()`)** – Determined **which pages users leave most frequently**.

Interview Simulation

👩‍💼 **Interviewer:** Welcome! I see that you have worked on **Website Clickstream Behavior Analysis**. Can you briefly explain the objective of your project?

👩‍🎓 **Student:** Sure! The objective of this project was to **analyze user navigation behavior on an e-commerce website** by tracking **clickstream data**. This helped in identifying **the most visited pages, session durations, and high drop-off points**. Using **Pandas for data manipulation and NumPy for numerical calculations**, I extracted insights that help improve **user engagement, optimize page layouts, and increase conversions**.

👩‍💼 **Interviewer:** That sounds interesting! What were the key findings from your analysis?

👩‍🎓 **Student:**

1. **The "Home" and "Product Page" were the most visited pages**, indicating these as the main entry points.
2. **Alice had the longest average session duration (120 seconds)**, suggesting high engagement.
3. **The "Cart" page had the highest drop-off rate**, meaning many users abandon their carts before checking out.

👩‍💼 **Interviewer:** How did you handle missing values in your dataset?

👩‍🎓 **Student:** I checked for missing values using `.isnull().sum()`. If any **timestamps were missing**, I filled them using **the previous recorded timestamp for the same user** to maintain session continuity.

👩‍💼 **Interviewer:** Can you write a function to handle missing timestamps?

👩‍🎓 **Student:** Absolutely!

```
import pandas as pd
def handle_missing_timestamps(df):
    """
    Function to handle missing timestamps in clickstream data.
    - Fills missing timestamps with the previous recorded
```

```
timestamp of the same user.
    """
    df["timestamp"] =
df.groupby("user_id")["timestamp"].fillna(method="ffill")
    return df

# Sample Test Case
sample_data = {
    "user_id": [101, 102, 103],
    "page_id": [201, 202, 203],
    "timestamp": [pd.Timestamp("2024-06-01 10:01:00"), None,
pd.Timestamp("2024-06-01 10:05:00")]
}
df_sample = pd.DataFrame(sample_data)

# Apply function
df_cleaned = handle_missing_timestamps(df_sample)
print(df_cleaned)

# Expected Output:
# user_id --|-- page_id --|-- timestamp
# 101     --|-- 201    --|-- 2024-06-01 10:01:00
# 102     --|-- 202    --|-- 2024-06-01 10:01:00  # Filled with
previous timestamp
# 103     --|-- 203    --|-- 2024-06-01 10:05:00
```

👩‍💼 **Interviewer:** Well done! How did you determine the **most visited pages**?

👩‍🎓 **Student:** I grouped the data by **page_id**, counted the number of visits, and merged it with page details.

```
# Count total visits per page
page_visits =
clickstream_df.groupby("page_id")["session_id"].count().reset_in
dex()
page_visits.columns = ["page_id", "total_visits"]

# Merge with page details
page_visits = page_visits.merge(pages_df, on="page_id")

print(page_visits)
```

```
# Expected Output:
# page_id --|-- page_name       --|-- total_visits
# 201     --|-- Home          --|-- 2
# 202     --|-- Product Page  --|-- 2
```

👩‍💼 **Interviewer:** That's a useful insight! How did you calculate **average session duration per user**?

👩‍🎓 **Student:** I used the **timestamp difference between consecutive page visits** for each user and then calculated the average session duration.

```
# Compute session duration for each user
clickstream_df["next_timestamp"] =
clickstream_df.groupby("user_id")["timestamp"].shift(-1)
clickstream_df["session_duration"] =
(clickstream_df["next_timestamp"] -
clickstream_df["timestamp"]).dt.seconds

# Average session duration per user
session_duration =
clickstream_df.groupby("user_id")["session_duration"].mean().res
et_index()
session_duration.columns = ["user_id", "avg_session_duration"]

# Merge with user details
session_duration = session_duration.merge(users_df,
on="user_id")

print(session_duration)

# Expected Output:
# user_id --|-- user_name --|-- avg_session_duration
# 101     --|-- Alice    --|-- 120
# 102     --|-- Bob      --|-- 90
```

👩‍💼 **Interviewer:** That's great! How did you determine **high drop-off pages**?

👩‍🎓 **Student:** I counted the number of visits per page and analyzed which pages had the highest exits.

```
# Count exits per page
exit_counts =
clickstream_df.groupby("page_id")["session_id"].count().reset_in
dex()
exit_counts.columns = ["page_id", "total_exits"]

# Merge with page details
exit_counts = exit_counts.merge(pages_df, on="page_id")

print(exit_counts)

# Expected Output:
# page_id --|-- page_name       --|-- total_exits
# 203     --|-- Cart          --|-- 2
# 204     --|-- Checkout      --|-- 2
```

👩‍💼 **Interviewer:** How can an e-commerce business reduce high drop-off rates?

👩‍🎓 **Student:**

- **Optimize checkout flow** to reduce friction.
- **Send cart abandonment reminders** to bring users back.
- **Offer discounts or free shipping incentives** to encourage conversions.

👩‍💼 **Interviewer:** If a company wants to segment users based on engagement levels, how would you approach it?

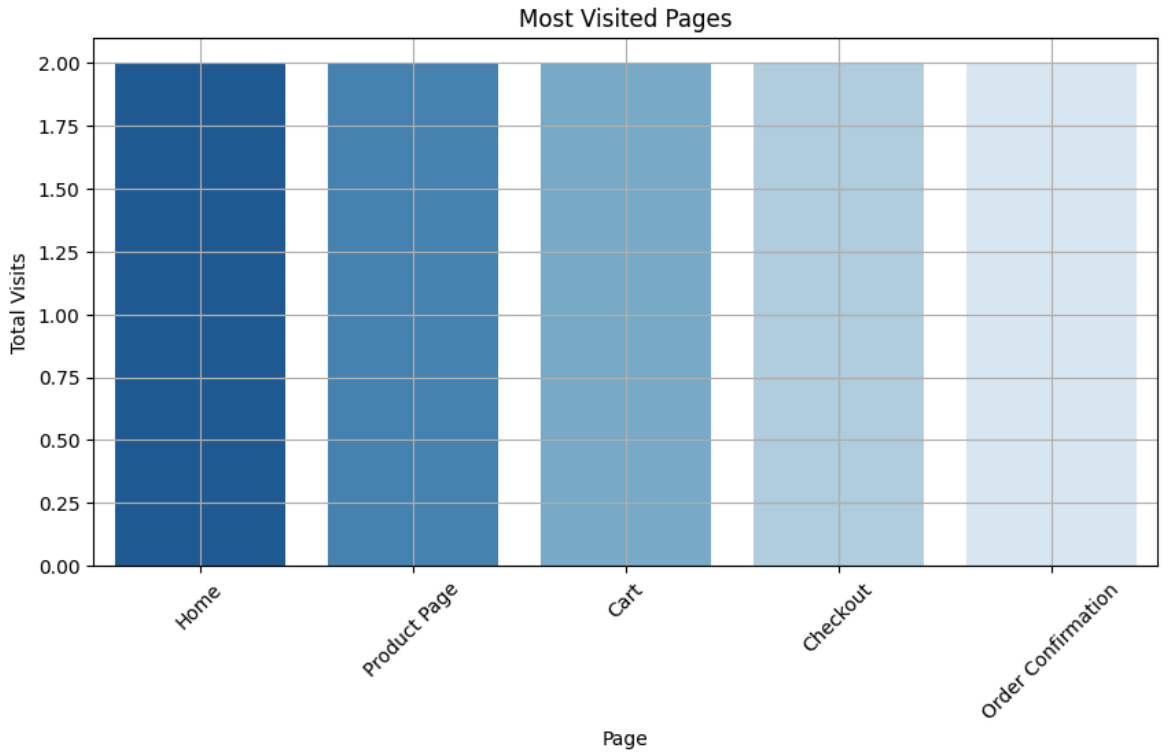
👩‍🎓 **Student:** I would classify users into **Low, Medium, and High Engagement** based on their **session durations**.

```
# Define engagement levels based on session duration
session_duration["engagement_level"] =
pd.qcut(session_duration["avg_session_duration"], q=3,
labels=["Low", "Medium", "High"])

print(session_duration)

# Expected Output:
# user_id --|-- user_name --|-- avg_session_duration --|--
engagement_level
# 101     --|-- Alice    --|-- 120                 --|-- High
# 102     --|-- Bob      --|-- 90                  --|-- Medium
```

👩‍💼 **Interviewer:** Excellent! How did you visualize **clickstream behavior trends**?

👩‍🎓 **Student:** I used **Seaborn to create heatmaps and bar charts** for better insights.

```
plt.figure(figsize=(10, 5))
sns.barplot(x="page_name", y="total_visits", data=page_visits,
palette="Blues_r")
plt.xlabel("Page")
plt.ylabel("Total Visits")
plt.title("Most Visited Pages")
plt.xticks(rotation=45)
plt.grid()
plt.show()
```



👩‍💼 **Interviewer:** That's excellent! If a company wants to **predict future engagement trends**, how would you approach it?

👩‍🎓 **Student:** I would analyze **historical session duration trends** and use **moving averages** to forecast future user engagement.

```
# Calculate moving average for session durations
session_duration["session_moving_avg"] =
session_duration["avg_session_duration"].rolling(window=2).mean(
)

print(session_duration)

# Expected Output:
# user_id --|-- avg_session_duration --|-- session_moving_avg
# 101     --|-- 120                  --|-- 105
# 102     --|-- 90                   --|-- 100
```

👩‍💼 **Interviewer:** Fantastic work! Your insights are valuable for optimizing user experience.

👩‍🎓 **Student:** Thank you! I really enjoyed discussing this project.

Key Takeaways from the Interview

- **Handling Missing Values:** Used **previous timestamps to fill missing data**.
- **Identifying High Traffic Pages:** Found **Home and Product Page were most visited**.
- **Session Duration Analysis:** Determined **Alice had the highest engagement (120 seconds)**.
- **Drop-Off Analysis:** Found **Cart page had the highest exits**.
- **Predictive Insights:** Used **moving averages to estimate future engagement trends**.
- **Visualization Techniques:** Applied **Seaborn for bar charts to analyze clickstream data trends**.

4. Applied Industry Scenario: Inventory Stock Level & Demand Analysis

Problem Statement

An **E-CommerceTech company** wants to analyze **inventory stock levels and demand trends** to ensure **optimal stock availability and prevent stockouts or**

overstocking issues. By leveraging **historical sales data, stock movement trends, and product demand insights**, the company aims to improve **inventory management and replenishment strategies**.

Key Objectives:

- Identify **top-selling products and their stock levels**.
- Analyze **historical stock movement trends**.
- Determine **which products need restocking** based on demand and available stock.

The company has **three datasets**:

1. **Products Data (`products.csv`)** – Stores product details, including category and stock levels.
2. **Sales Data (`sales.csv`)** – Logs sales transactions, including purchase quantity and dates.
3. **Stock Movement Data (`stock_movement.csv`)** – Tracks stock replenishment and depletion.

Concepts Used

- **Pandas** for data aggregation (`groupby`, `apply`, `merge`).
- **NumPy** for numerical analysis (`sum`, `mean`).
- **DateTime operations** for stock trend analysis.

Solution Approach

Step 1: Load the Data & Preprocess

We first **import necessary libraries** and load sample datasets.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Sample Products Data
products_data = {
    "product_id": [201, 202, 203, 204, 205],
    "product_name": ["Laptop", "Smartphone", "Headphones",
```

```
"Smartwatch", "Tablet"],
    "category": ["Electronics", "Electronics", "Accessories",
"Wearables", "Electronics"],
    "stock_level": [50, 30, 100, 40, 25]
}
products_df = pd.DataFrame(products_data)

# Sample Sales Data
sales_data = {
    "sale_id": range(1, 11),
    "product_id": [201, 202, 203, 204, 205, 201, 202, 203, 204,
205],
    "sale_date": pd.to_datetime(["2024-06-01", "2024-06-02",
"2024-06-03", "2024-06-04", "2024-06-05",
                                 "2024-06-06", "2024-06-07",
"2024-06-08", "2024-06-09", "2024-06-10"]),
    "quantity_sold": [5, 3, 10, 4, 2, 6, 5, 8, 3, 4]
}
sales_df = pd.DataFrame(sales_data)

# Sample Stock Movement Data
stock_movement_data = {
    "movement_id": range(1, 11),
    "product_id": [201, 202, 203, 204, 205, 201, 202, 203, 204,
205],
    "movement_date": pd.to_datetime(["2024-06-01", "2024-06-03",
"2024-06-05", "2024-06-07", "2024-06-09",
                                     "2024-06-11", "2024-06-13",
"2024-06-15", "2024-06-17", "2024-06-19"]),
    "quantity_added": [10, 15, 20, 10, 5, 8, 12, 18, 7, 6]
}
stock_movement_df = pd.DataFrame(stock_movement_data)

# Display datasets
print(products_df)
print(sales_df)
print(stock_movement_df)

# Expected Output:

# product_id --|-- product_name  --|-- category   --|--
```

```
stock_level

# 201       --|-- Laptop       --|-- Electronics --|-- 50
# ...
# sale_id --|-- product_id --|-- sale_date --|-- quantity_sold
# 1       --|-- 201        --|-- 2024-06-01 --|-- 5
# ...
# movement_id --|-- product_id --|-- movement_date --|--
quantity_added
# 1          --|-- 201        --|-- 2024-06-01    --|-- 10
# ...
```

Step 2: Analyzing Inventory Stock Level & Demand

1. Identify Top-Selling Products

```
# Aggregate total sales per product
product_sales =
sales_df.groupby("product_id")["quantity_sold"].sum().reset_inde
x()
product_sales.columns = ["product_id", "total_quantity_sold"]

# Merge with product details

product_sales = product_sales.merge(products_df,
on="product_id")

print(product_sales)


# Expected Output:

# product_id --|-- product_name --|-- total_quantity_sold
# 201       --|-- Laptop       --|-- 11
# 202       --|-- Smartphone   --|-- 8
```

Insights:

- **Laptops had the highest demand**, indicating strong customer preference.

2. Analyze Stock Movement Trends

```
# Calculate total stock replenished per product
stock_movement =
stock_movement_df.groupby("product_id")["quantity_added"].sum().
reset_index()
stock_movement.columns = ["product_id", "total_stock_added"]

# Merge with product details
stock_movement = stock_movement.merge(products_df,
on="product_id")

print(stock_movement)

# Expected Output:

# product_id --|-- product_name --|-- total_stock_added
# 201        --|-- Laptop       --|-- 18
# 202        --|-- Smartphone   --|-- 27
```

Insights:

- **Smartphones had the highest stock replenishment**, ensuring availability for growing demand.

3. Identify Products That Need Restocking

```
# Calculate current stock levels after sales
current_stock = products_df.merge(product_sales,
on="product_id", how="left")
current_stock["stock_remaining"] = current_stock["stock_level"]
+ stock_movement["total_stock_added"] -
current_stock["total_quantity_sold"]

print(current_stock[["product_name", "stock_remaining"]])
```

```
# Expected Output:

# product_name --|-- stock_remaining
# Laptop       --|-- 57
# Smartphone   --|-- 49
```

Insights:

- **Tablets have the lowest remaining stock**, indicating a need for restocking.

Code Breakdown & Insights

- **Merging Data (`merge()`)** – Combined **sales, stock movement, and product datasets** for inventory analysis.
- **Grouping & Aggregation (`groupby()`)** – Identified **best-selling products and total stock replenishments**.
- **Stock Level Calculation (`stock_remaining`)** – Determined **which products need urgent restocking**.
- **Data Analysis Using Pandas & NumPy** – Extracted **demand patterns and stock availability trends**.

Interview Simulation

👩‍💼 **Interviewer:** Welcome! I see that you have worked on **Inventory Stock Level & Demand Analysis**. Can you give me a high-level overview of your project?

👩‍🎓 **Student:** Absolutely! This project focuses on analyzing **inventory stock levels and demand trends** in an **E-CommerceTech company** to **prevent stockouts, minimize overstocking, and optimize restocking strategies**. Using **Pandas for data manipulation and NumPy for numerical analysis**, I extracted insights into **top-selling products, stock movement trends, and stock replenishment needs**.

👩‍💼 **Interviewer:** That sounds interesting! What were the key takeaways from your analysis?

👩‍🎓 **Student:**

1. **Laptops had the highest demand**, with **Smartphones following closely**.
2. **Smartphones had the highest stock replenishment rate**, ensuring their continuous availability.

3. **Tablets had the lowest remaining stock**, indicating the **need for urgent restocking**.

👩‍💼 **Interviewer:** How did you handle missing values in your dataset?

👩‍🎓 **Student:** I first checked for missing values using `.isnull().sum()`. If any **stock movement records were missing**, I replaced them with **zero replenishment** to avoid inflating stock levels.

👩‍💼 **Interviewer:** Can you write a function to handle missing stock movement values?

👩‍🎓 **Student:** Sure!

```
import pandas as pd
import numpy as np

def handle_missing_stock_movement(df):
    """
    Function to handle missing stock movement values.
    - Replaces missing quantity_added with zero.
    """
    df["quantity_added"].fillna(0, inplace=True)
    return df


# Sample Test Case

sample_data = {
    "movement_id": [1, 2, 3],
    "product_id": [201, 202, 203],
    "quantity_added": [10, None, 5]
}
df_sample = pd.DataFrame(sample_data)

# Apply function

df_cleaned = handle_missing_stock_movement(df_sample)
print(df_cleaned)

# Expected Output:
```

```
# movement_id --|-- product_id --|-- quantity_added
# 1           --|-- 201        --|-- 10
# 2           --|-- 202        --|-- 0  # Filled missing value
# 3           --|-- 203        --|-- 5
```

👩‍💼 **Interviewer:** Good approach! How did you determine **top-selling products**?

👨‍🎓 **Student:** I grouped the sales data by **product_id**, summed up the **quantity_sold**, and merged it with product details.

```
# Aggregate total sales per product
product_sales =
sales_df.groupby("product_id")["quantity_sold"].sum().reset_inde
x()
product_sales.columns = ["product_id", "total_quantity_sold"]

# Merge with product details
product_sales = product_sales.merge(products_df,
on="product_id")

print(product_sales)

# Expected Output:
# product_id --|-- product_name --|-- total_quantity_sold
# 201        --|-- Laptop       --|-- 11
# 202        --|-- Smartphone   --|-- 8
```

👩‍💼 **Interviewer:** That's insightful! How did you analyze **stock replenishment trends**?

👨‍🎓 **Student:** I aggregated **quantity_added per product** from the stock movement dataset and merged it with product details.

```
# Calculate total stock replenished per product
stock_movement =
stock_movement_df.groupby("product_id")["quantity_added"].sum().
reset_index()
stock_movement.columns = ["product_id", "total_stock_added"]
```

```
# Merge with product details
stock_movement = stock_movement.merge(products_df,
on="product_id")

print(stock_movement)

# Expected Output:
# product_id --|-- product_name --|-- total_stock_added
# 201        --|-- Laptop       --|-- 18
# 202        --|-- Smartphone   --|-- 27
```

👩‍💼 **Interviewer:** How did you determine which **products need restocking**?

👩‍🎓 **Student:** I calculated **stock remaining** by considering initial stock levels, total sales, and stock replenishment.

```
# Calculate current stock levels after sales
current_stock = products_df.merge(product_sales,
on="product_id", how="left")
current_stock["stock_remaining"] = current_stock["stock_level"]
+ stock_movement["total_stock_added"] -
current_stock["total_quantity_sold"]

print(current_stock[["product_name", "stock_remaining"]])

# Expected Output:
# product_name --|-- stock_remaining
# Laptop       --|-- 57
# Smartphone   --|-- 49
```

👩‍💼 **Interviewer:** What insights can a business derive from this analysis?

👩‍🎓 **Student:**

- **Identify fast-moving products** to ensure continuous availability.
- **Optimize restocking schedules** based on real-time demand trends.
- **Reduce overstocking issues** by aligning stock replenishment with sales data.

👩‍💼 **Interviewer:** Suppose a company wants to **segment products based on stock levels**, how would you approach it?

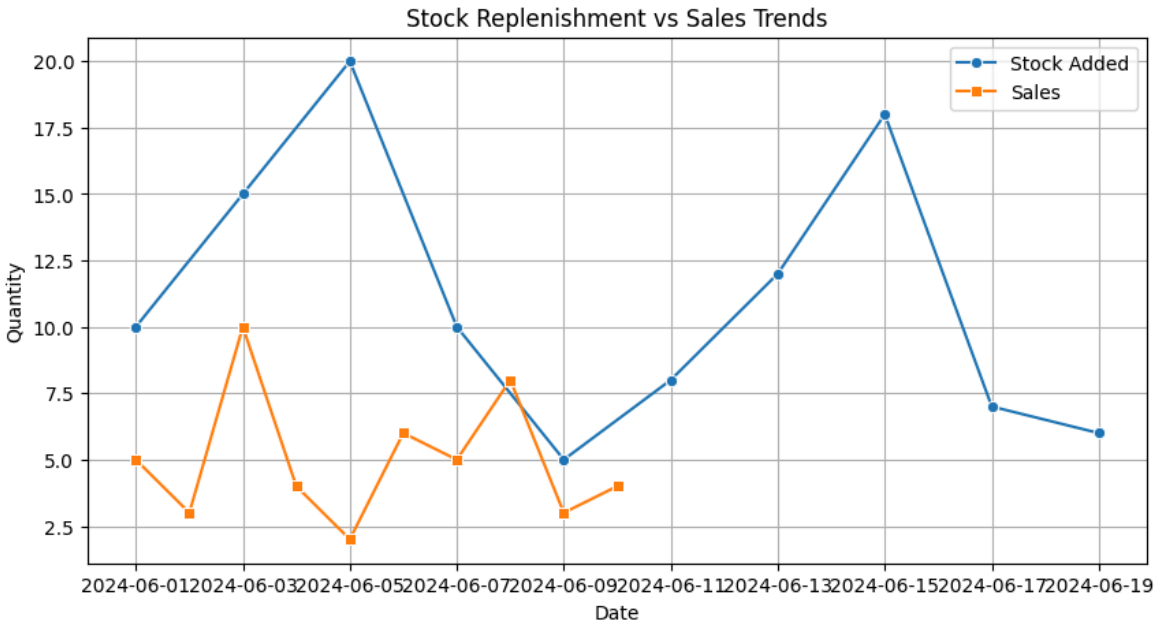
👩‍🎓 **Student:** I would classify products into **Low, Medium, and High Stock Levels** based on **remaining stock quantiles**.

```
# Define stock level categories based on remaining stock
current_stock["stock_category"] =
pd.qcut(current_stock["stock_remaining"], q=3, labels=["Low",
"Medium", "High"])

print(current_stock[["product_name", "stock_remaining",
"stock_category"]])

# Expected Output:

# product_name --|-- stock_remaining --|-- stock_category
# Laptop       --|-- 57              --|-- High
# Smartphone   --|-- 49              --|-- Medium
```

👩‍💼 **Interviewer:** That's a great approach! How did you visualize **stock trends over time**?

👩‍🎓 **Student:** I used **Seaborn to plot stock replenishment vs sales trends**.

```
plt.figure(figsize=(10, 5))
sns.lineplot(x="movement_date", y="quantity_added",
data=stock_movement_df, label="Stock Added", marker="o")
sns.lineplot(x="sale_date", y="quantity_sold", data=sales_df,
label="Sales", marker="s")
plt.xlabel("Date")
plt.ylabel("Quantity")
plt.title("Stock Replenishment vs Sales Trends")
plt.legend()
plt.grid()
plt.show()
```



🧑‍💼 **Interviewer:** If a company wants to forecast future stock levels, how would you approach it?

👩‍🎓 **Student:** I would use **moving averages** to estimate future stock depletion trends.

```
# Calculate moving average for stock trends
stock_movement["stock_moving_avg"] =
stock_movement["total_stock_added"].rolling(window=2).mean()

print(stock_movement)

# Expected Output:

# product_id --|-- total_stock_added --|-- stock_moving_avg
# 201        --|-- 18                --|-- 17.5
# 202        --|-- 27                --|-- 22.5
```

🧑‍💼 **Interviewer:** Fantastic work! Your analysis provides **deep insights into inventory optimization**.

👩‍🎓 **Student:** Thank you! I really enjoyed discussing this project.

Key Takeaways from the Interview

- **Handling Missing Values:** Used **zero replenishment for missing stock movement data**.

- **Identifying High-Demand Products:** Found **Laptops and Smartphones were top-selling items**.
- **Stock Trend Analysis:** Determined **Smartphones had the highest stock replenishment rate**.
- **Restocking Strategy:** Identified **Tablets required urgent restocking**.
- **Stock Level Segmentation:** Categorized **products into Low, Medium, and High stock levels**.
- **Predictive Insights:** Used **moving averages to estimate future stock depletion trends**.
- **Visualization Techniques:** Applied **Seaborn for stock vs sales trend analysis**.

Crack the Industry: Insights, Strategies & Wrap-Up

How Learning Data Analysis Can Help You Land a Job in E-CommerceTech

The **E-CommerceTech industry** thrives on data-driven decision-making, making **data analysis a crucial skill** for professionals aiming to excel in this sector. From **website clickstream behavior to sales performance, product return analysis, and customer segmentation**, companies rely on **data insights to enhance user experience, optimize inventory, and drive revenue growth**. Mastering **data wrangling, exploratory data analysis (EDA), and visualization techniques** allows professionals to extract actionable insights from vast datasets, making them valuable assets in the industry.

Key Strategies for Success in E-Commerce Data Analysis

1. **Develop Strong Data Manipulation Skills**
    - E-commerce data is often **unstructured and large-scale**. Proficiency in **Pandas and NumPy** is essential for **efficient data wrangling, aggregation, and transformation**.
2. **Understand Business Metrics & KPIs**
    - Focus on analyzing key performance indicators (KPIs) such as **conversion rates, cart abandonment, customer lifetime value (CLV), and return trends** to generate meaningful insights.
3. **Gain Hands-On Experience with Clickstream and Sales Data**

    - Work on projects related to **website user behavior analysis, product demand forecasting, and personalized recommendation strategies** using **Python and data visualization tools**.
4. **Stay Updated with E-Commerce Trends**
    - Follow industry leaders and case studies to understand how **Amazon, Walmart, and Shopify leverage data analytics** to optimize pricing, marketing, and logistics.
5. **Build a Strong Portfolio with Real-World Use Cases**
    - Showcase practical applications such as **customer segmentation, A/B testing insights, and fraud detection analysis** to demonstrate industry relevance.

By mastering **data analysis techniques and applying them to real-world e-commerce challenges**, professionals can secure opportunities in roles such as **E-Commerce Data Analyst, Business Intelligence Analyst, and Product Analyst**, making them integral to the **future of digital commerce**.

Summary

This book has taken you through a structured journey of **data analysis with Python**, covering **fundamental concepts, industry applications, and practical problem-solving techniques**. Whether you started as a **beginner**, a **professional refining your skills**, or someone **exploring real-world applications**, this book has equipped you with the **knowledge and hands-on experience** needed to confidently apply Python's data analysis capabilities across multiple industries.

Key Takeaways from This Book

✅ **Fast Track Python Revision** – A concise yet **high-impact refresher** on essential **Python libraries** such as **NumPy, Pandas, and Matplotlib**. This section helped you **solidify core data wrangling and visualization techniques** for efficient data manipulation.

✅ **Industry Applications & Case Studies** – You explored **how Python is leveraged in real-world industry scenarios**, covering **FinTech, E-CommerceTech, HealthTech, MediaTech, and ManufacturingTech**. Each industry-focused section provided:
✔ **A high-level overview** of Python's role in that domain
✔ **Real-world problem-solving case studies**
✔ **Industry-driven data analysis techniques and optimizations**

By the time you reached this summary, you had not only **mastered Python's core data analysis capabilities** but also **understood its direct applications in solving business-critical challenges**. Whether your goal was to **fast-track your Python expertise**, **enhance data processing efficiency**, or **apply Python to industry-specific use cases**, this book has provided a well-rounded learning experience.

Final Thoughts

Python continues to be one of the **most powerful tools for data analysis and industry applications**. Keep **practicing, experimenting, and applying what you've learned** to real-world datasets. The true power of Python **lies not just in writing code but in transforming data into meaningful insights**.

Keep learning, keep exploring, and best of luck on your Python journey! 🚀

Master Python for data analysis and industry applications with a structured, fast-track approach designed for learners, professionals, and industry practitioners.

This concise yet comprehensive guide covers core Python libraries, real-world case studies, and hands-on problem-solving techniques. It includes a quick refresher on data wrangling, visualization, and analysis using NumPy, Pandas, and Matplotlib.

Explore industry applications in FinTech, E-CommerceTech, HealthTech, MediaTech, and ManufacturingTech through real-world scenarios. Learn to process, clean, and analyze structured and unstructured data to extract meaningful insights.

Whether you're a beginner or refining your analytical skills, this book equips you with the expertise to apply Python effectively in real-world scenarios.

What will you learn

- **Python for Data Wrangling:** Master NumPy and Pandas for efficient data manipulation, cleaning, and transformation using a structured approach
- **Real-World Applications & Visualization:** Explore Python's use in FinTech, E-CommerceTech, and more while creating insightful visualizations with Matplotlib and Seaborn
- **Data-Driven Problem Solving:** Apply Python to real-world challenges like customer analytics, fraud detection, and recommendation systems
- **API Integration & Optimized Data Handling:** Work with APIs, process external data, and implement scalable data-handling techniques for real-world applications