UNIT - 1

Data science Introduction

Data Science is a multidisciplinary field that combines domain expertise, programming skills, and knowledge of mathematics and statistics to extract meaningful insights from data.

Key components of Data Science:

- Problem Definition
- Data Collection – Gathering raw data from various sources.
- Data Cleaning – Removing errors, handling missing values, etc.
- Exploratory Data Analysis (EDA) – Understanding patterns, trends, and relationships.
- Modelling – Applying algorithms to make predictions or classifications.
- Deployment & Monitoring – Implementing the model into production and tracking performance.

What is Exploratory Data Analysis (EDA)?

- **Exploratory Data Analysis (EDA)** is the process of analysing datasets to summarize their key characteristics using visual and quantitative techniques.
- It is typically performed **before** applying any machine learning models and serves as a **bridge** between data collection and modelling
- Understand the structure of the data
- Detect outliers and anomalies
- Identify relationships among variables
- Generate hypotheses for further analysis

Uses of EDA in the Data Science Life Cycle

1. **Understanding the Data**
    - Check Data type ,dimensions and column description
    - Understanding the range and distribution of features
    - Detect missing or invalid data

2. **Detecting Pattern and Trends**
    - Visualize the relationship(correlation between features)
    - Identify the cluster or groups in data
    - Spot seasonality or trends in time series data
3. **Detecting Pattern and Trends**
    - Uses box plot ,scatter plot etc
    - Outliers can effect the model performance if not handled
4. **Detecting Pattern and Trends**
    - Helps identify which features are useful
    - Suggest possible transformations
5. **Data Cleaning and Preprocessing**
    - Detect null values ,duplicates and inconsistent categories
    - Guides decision on how to impute or correct the issues
6. **Model selection Preparation**
    - Determine if data is suitable for certain type of models (eg: linear vs nonlinear)
    - Guide target variable distribution analysis for classification or regression

EDA Techniques Include:

Common EDA Techniques

Technique	Purpose
Descriptive Statistics	**Mean, median, mode, standard deviation, skewness, etc.**
Data Visualization	**Histograms, box plots, scatter plots, bar charts, heatmaps**
Correlation Analysis	**Understanding relationships between variables**
Missing Value Analysis	**Identifying and handling missing data**
Outlier Detection	**Identifying abnormal data points that may affect modelling**

What is Anaconda?

Anaconda is a **Python distribution** – it includes:

- **Conda** is a **package manager and environment manager**. It is used to:
- Install and manage Python (and other language) packages.
- Create **isolated environments** for different projects with specific versions of Python and libraries.
- Avoid conflicts between different packages or projects.

🔹 **Why use Anaconda?**

- Easy to install and manage packages
- Ideal for Data Science, Machine Learning, and Scientific Computing
- Comes with all the tools in one place
- You are working on two projects: Project A needs Python 3.8 with TensorFlow 2.9 Project B needs Python 3.11 with PyTorch 2.1 Conda allows you to run both projects independently without errors.

What is Jupyter Notebook?

Jupyter Notebook is a **web-based tool** that allows you to:

- Write Python code in small blocks (called "cells")
- Run one block at a time (great for learning & testing)
- Add explanations using Markdown (text formatting)
- Combine code, comments, formulas, and charts in one document.

Think of it like a notebook where:

- Each cell can contain **code** or **text**
- You can **see results immediately** below each code block

🔹 **Example Use:**

- Data cleaning
- Visualization
- Machine learning experiments
- Teaching/learning Python

Why Use Them Together?

- Conda gives you a clean, controlled environment with exactly the libraries you need.
- Jupyter runs inside that environment, using the correct versions of Python and packages.
- This ensures:
    - No version mismatches
    - Reproducible results
    - Stable development environment

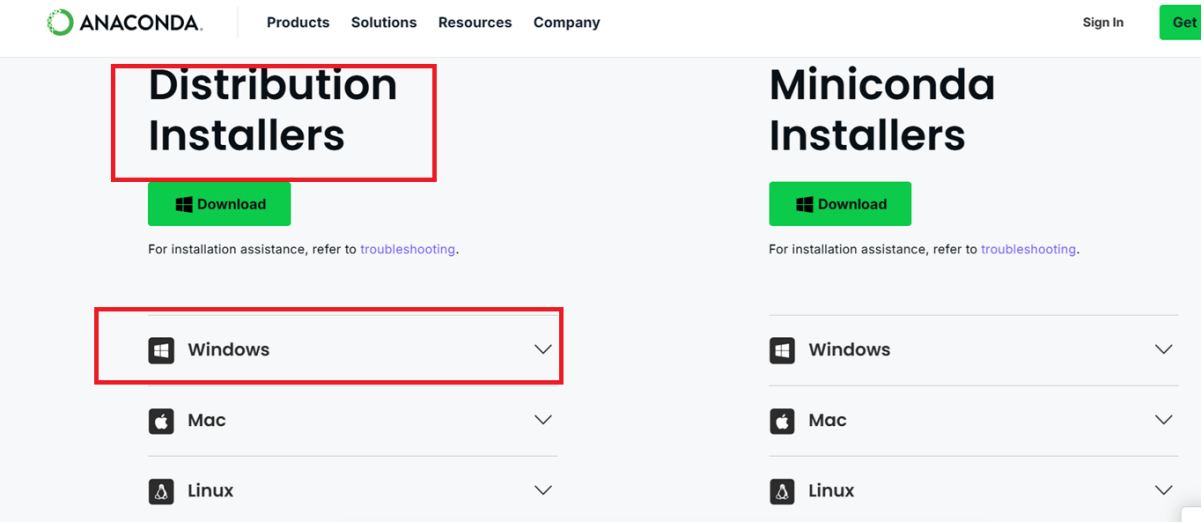
VS Code (Visual Studio Code) is a **powerful code editor** developed by Microsoft. It supports:

- Writing and running Python scripts
- Syntax highlighting and auto-completion
- Debugging tools
- Extensions like:
    - Python
    - Jupyter
    - Git

VS Code is used for **more advanced Python projects**, compared to Jupyter which is more for interactive use.

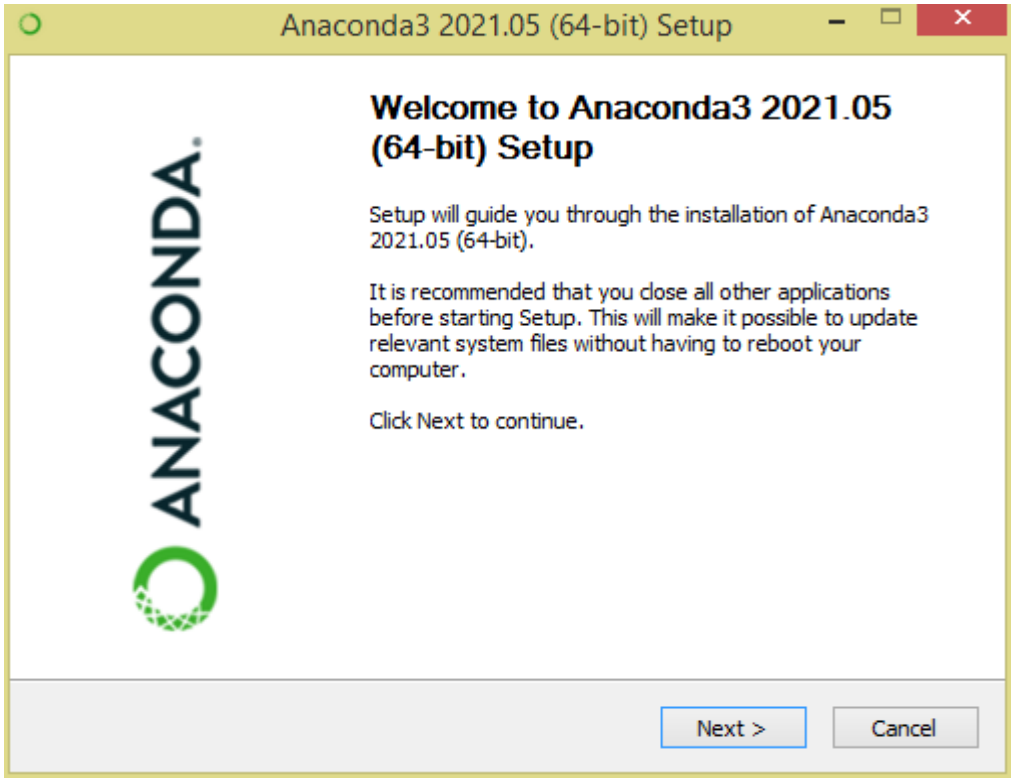
1.1 Download Anaconda Distribution

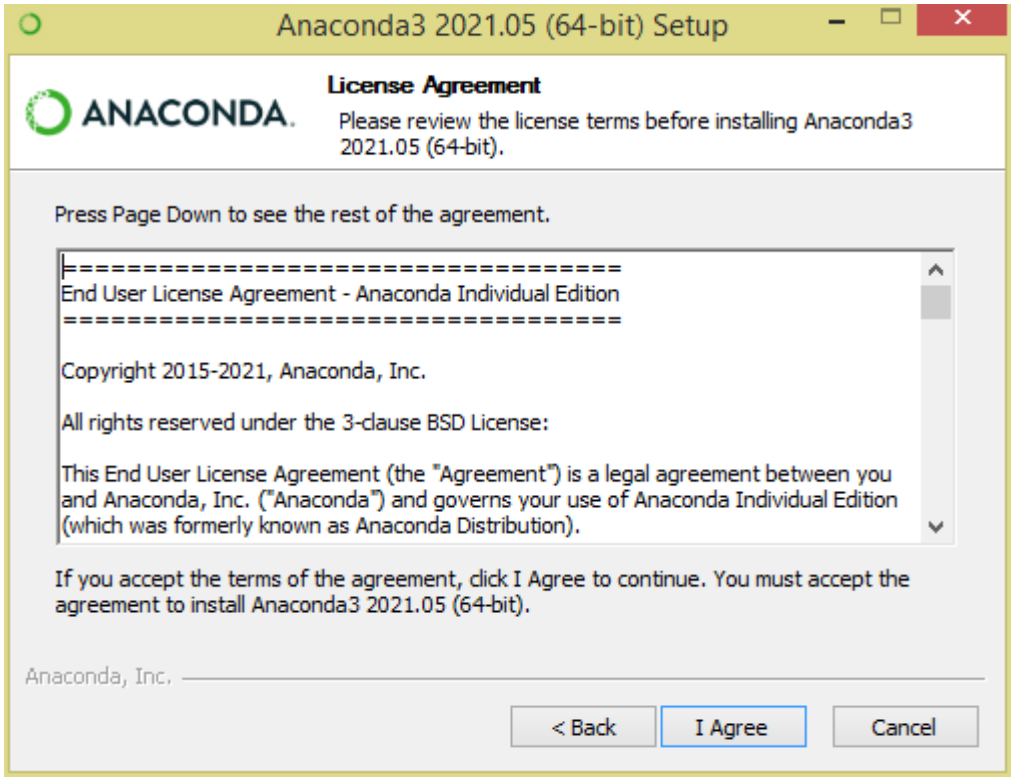
Go to https://www.anaconda.com/products/distribution and select **Anaconda Distributionin installer** to download the latest version of Anaconda. This downloads the .exe file to the windows download folder.

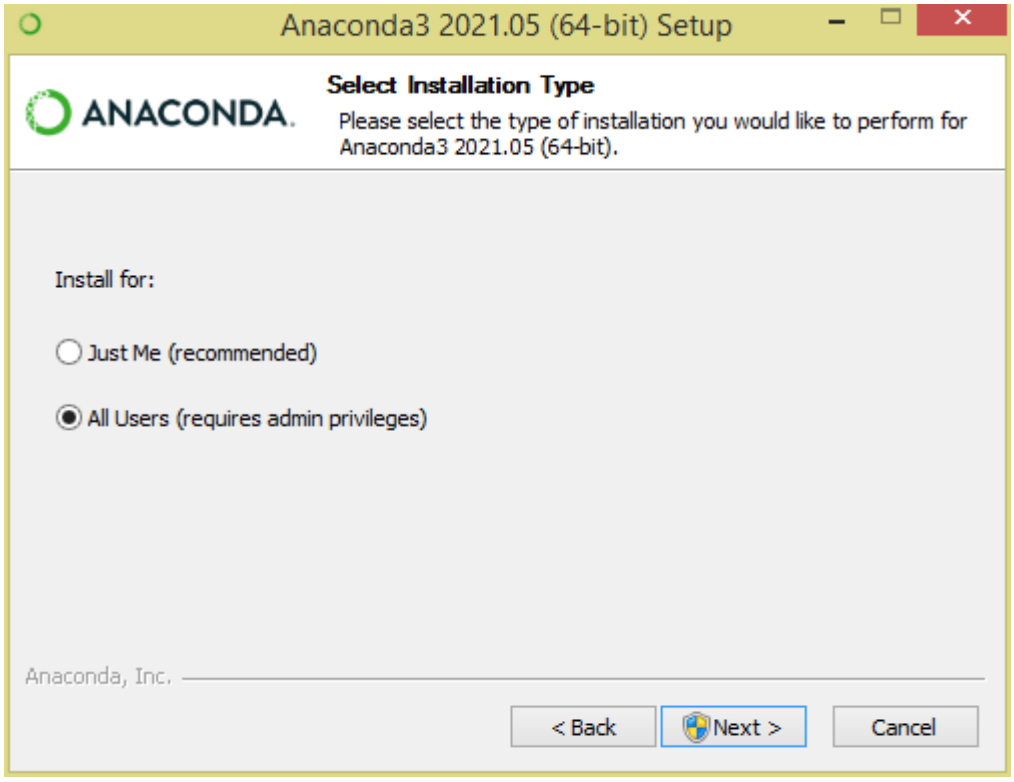


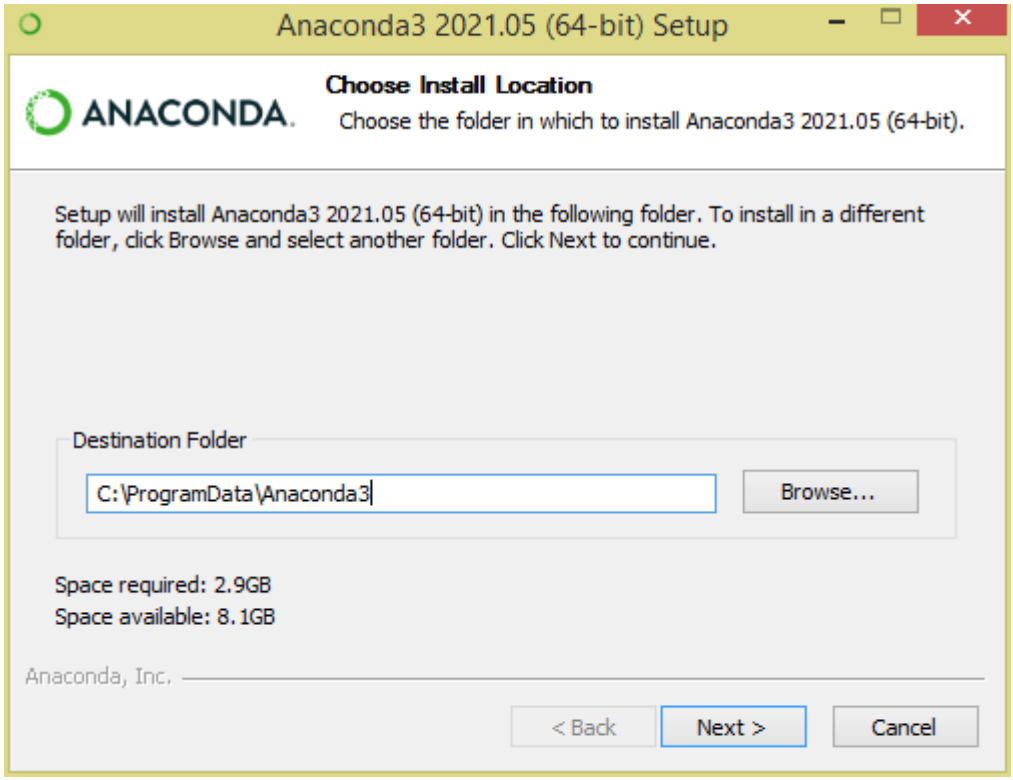
1.2 Install Anaconda

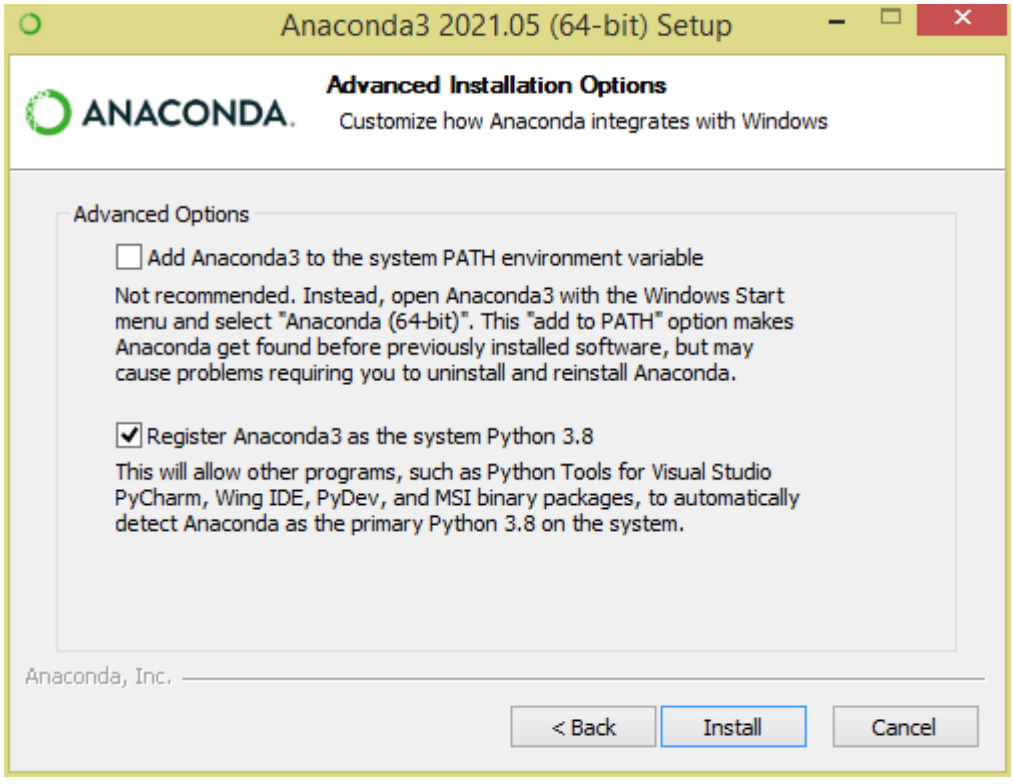
By double-clicking the .exe file starts the Anaconda installation. Follow the below screen shot's and complete the installation

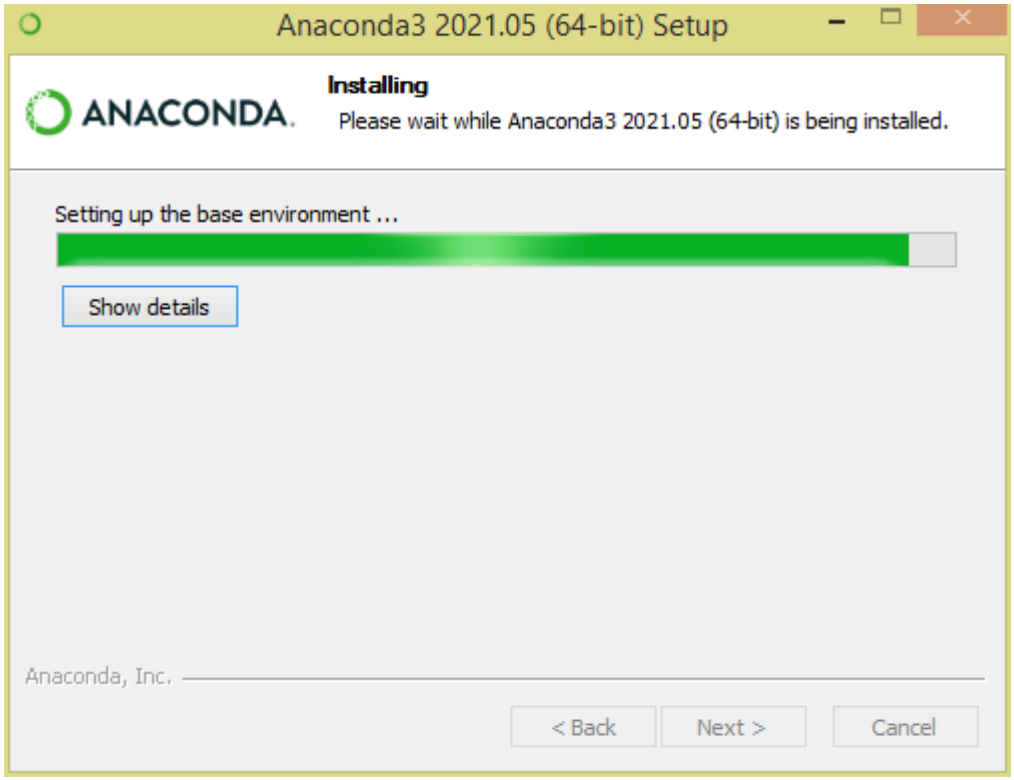


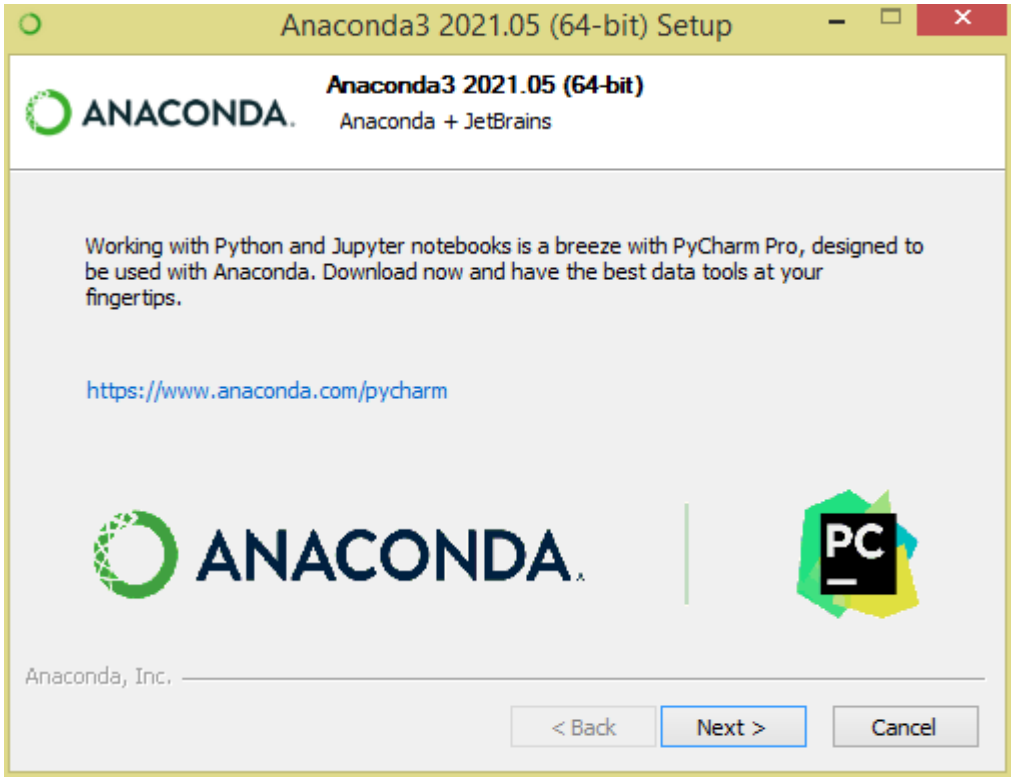

Anaconda3 2021.05 (64-bit) Setup
ANACONDA.
License Agreement
Please review the license terms before installing Anaconda3 2021.05 (64-bit).
Press Page Down to see the rest of the agreement.
==========
End User License Agreement - Anaconda Individual Edition
==========
Copyright 2015-2021, Anaconda, Inc.
All rights reserved under the 3-clause BSD License:
This End User License Agreement (the "Agreement") is a legal agreement between you and Anaconda, Inc. ("Anaconda") and governs your use of Anaconda Individual Edition (which was formerly known as Anaconda Distribution).
If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install Anaconda3 2021.05 (64-bit).
Anaconda, Inc.
< Back
I Agree
Cancel

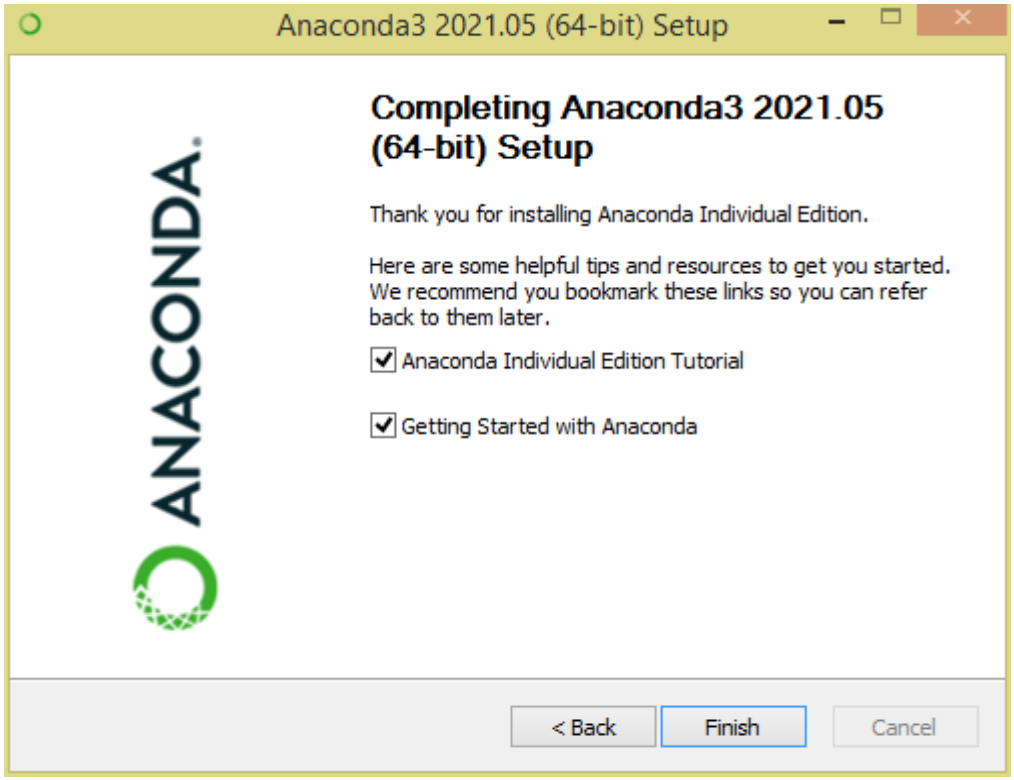

Anaconda3 2021.05 (64-bit) Setup
ANACONDA.
Select Installation Type
Please select the type of installation you would like to perform for Anaconda3 2021.05 (64-bit).
Install for:
Just Me (recommended)
All Users (requires admin privileges)
Anaconda, Inc.
< Back
Next >
Cancel


Anaconda3 2021.05 (64-bit) Setup
ANACONDA.
Choose Install Location
Choose the folder in which to install Anaconda3 2021.05 (64-bit).
Setup will install Anaconda3 2021.05 (64-bit) in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.
Destination Folder
C:\ProgramData\Anaconda3
Browse...
Space required: 2.9GB
Space available: 8.1GB
Anaconda, Inc.
< Back
Next >
Cancel


Anaconda3 2021.05 (64-bit) Setup
ANACONDA.
Advanced Installation Options
Customize how Anaconda integrates with Windows
Advanced Options
Add Anaconda3 to the system PATH environment variable
Not recommended. Instead, open Anaconda3 with the Windows Start menu and select "Anaconda (64-bit)". This "add to PATH" option makes Anaconda get found before previously installed software, but may cause problems requiring you to uninstall and reinstall Anaconda.
Register Anaconda3 as the system Python 3.8
This will allow other programs, such as Python Tools for Visual Studio PyCharm, Wing IDE, PyDev, and MSI binary packages, to automatically detect Anaconda as the primary Python 3.8 on the system.
Anaconda, Inc.
< Back
Install
Cancel


Anaconda3 2021.05 (64-bit) Setup
ANACONDA.
Installing
Please wait while Anaconda3 2021.05 (64-bit) is being installed.
Setting up the base environment ...
Show details
Anaconda, Inc.
< Back
Next >
Cancel


Anaconda3 2021.05 (64-bit) Setup
ANACONDA.
Anaconda3 2021.05 (64-bit)
Anaconda + JetBrains
Working with Python and Jupyter notebooks is a breeze with PyCharm Pro, designed to be used with Anaconda. Download now and have the best data tools at your fingertips.
https://www.anaconda.com/pycharm
ANACONDA.
PC
Anaconda, Inc.
< Back
Next >
Cancel



This finishes the installation of Anaconda distribution, now let's see how to create an environment and install Jupyter Notebook.

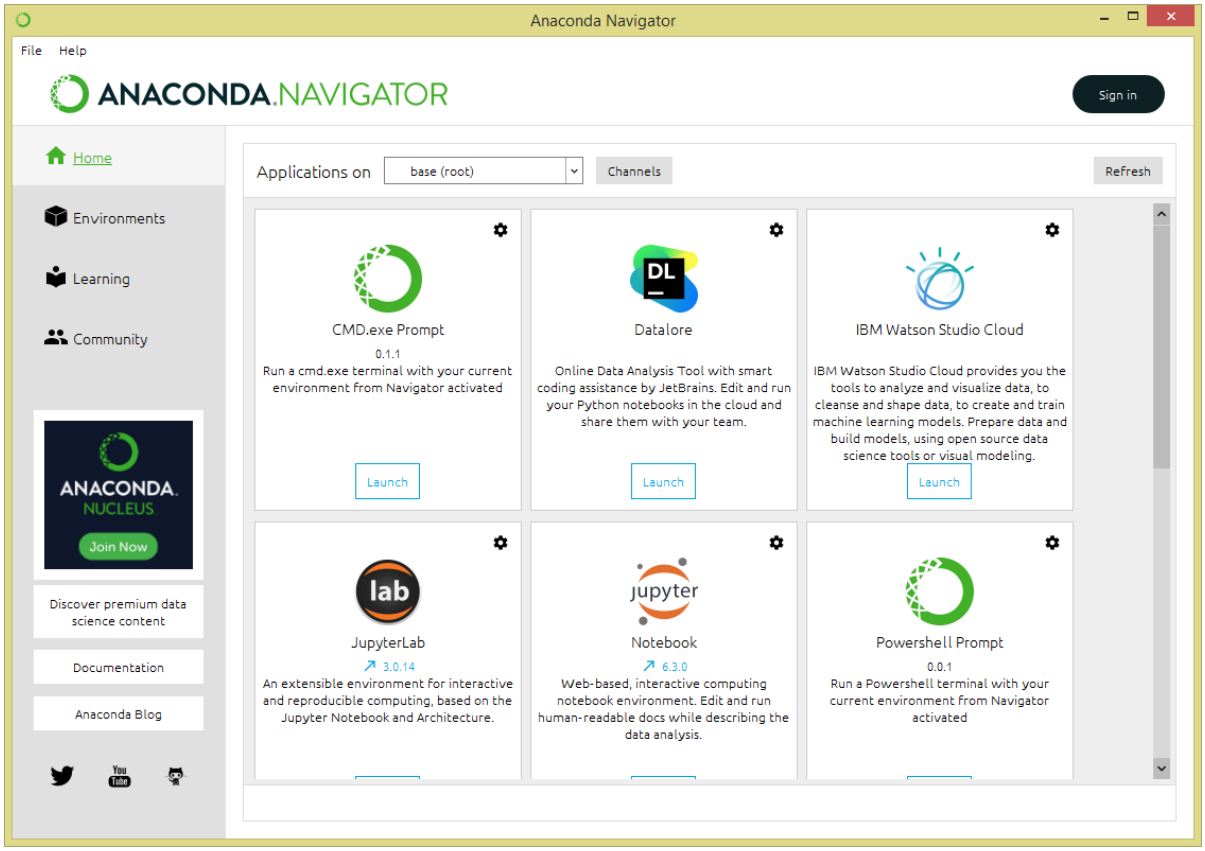
2. Create Anaconda Environment from Navigator

A conda environment is ***a directory that contains a specific collection of conda packages that you have installed****. For example, you may have one environment with NumPy 1.7 and its dependencies, and another environment with NumPy 1.6 for legacy testing.*

https://conda.io/docs/using/envs.html

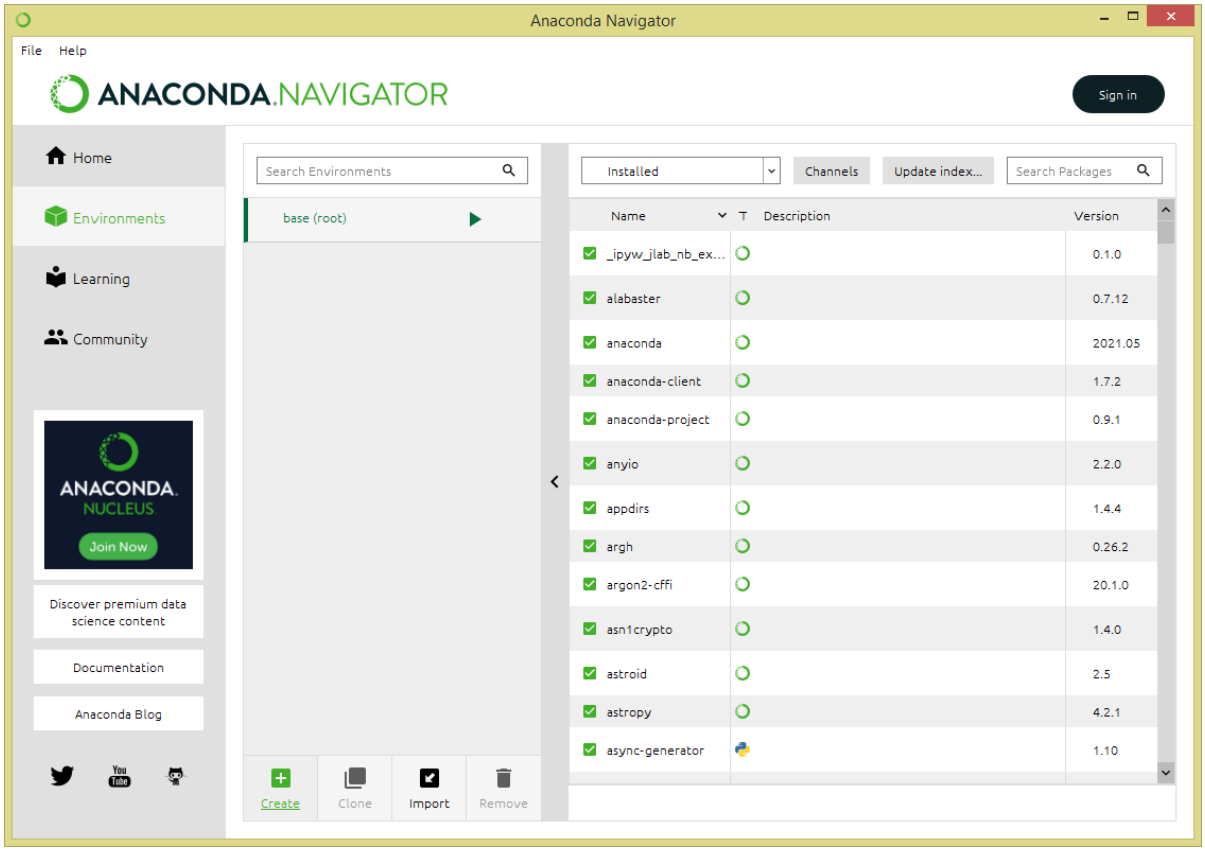
2.1 Open Anaconda Navigator

Open Anaconda Navigator from windows start or by searching it. Anaconda Navigator is a UI application where you can control the Anaconda packages, environment e.t.c

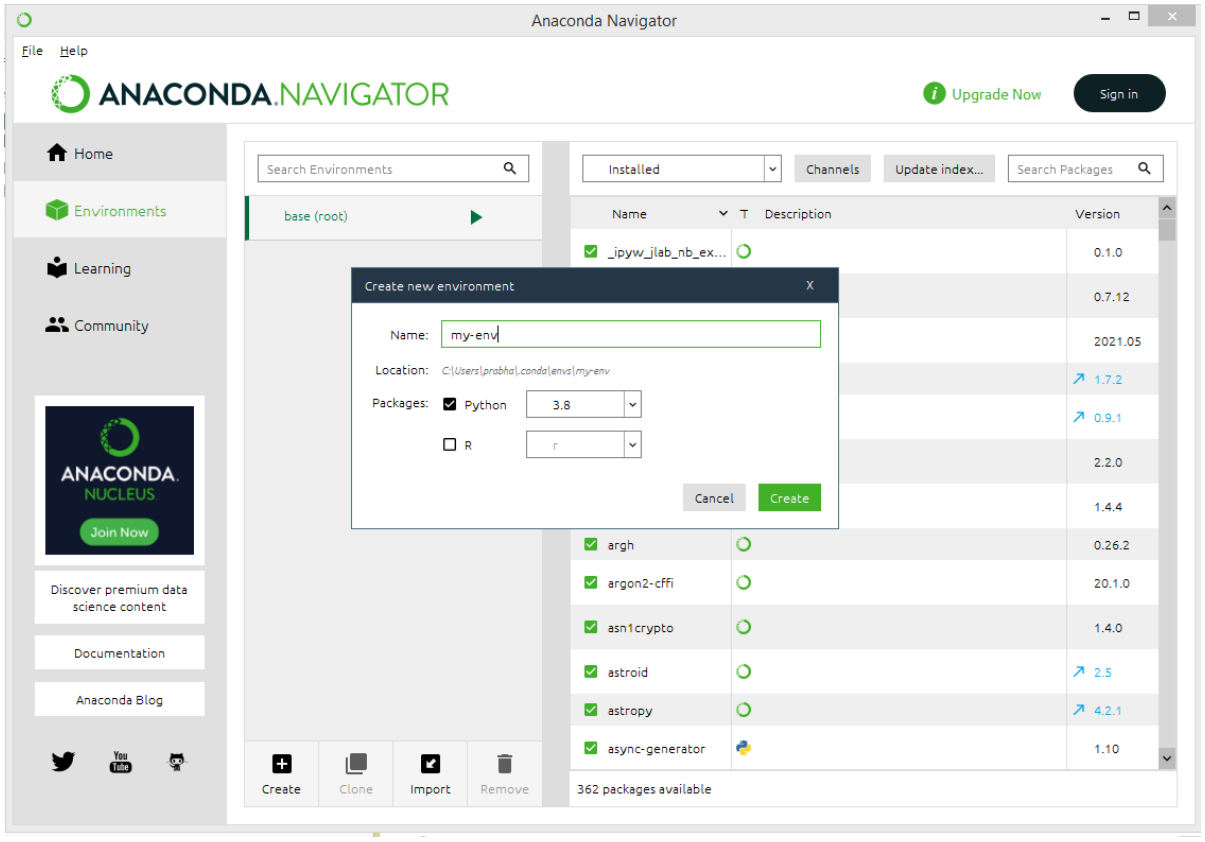


2.2 Create an Environment to Run Jupyter Notebook

This is optional but recommended to create an environment before you proceed. This gives complete segregation of different package installs for different projects you would be working on. If you already have an environment, you can use it too.

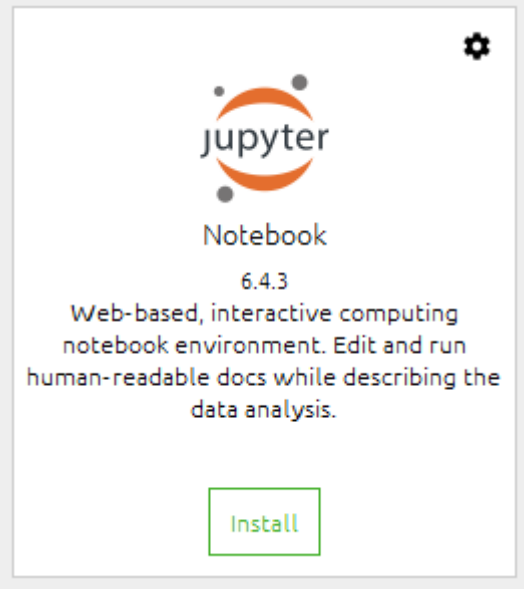


select + Create icon at the bottom of the screen to create an Anaconda environment.

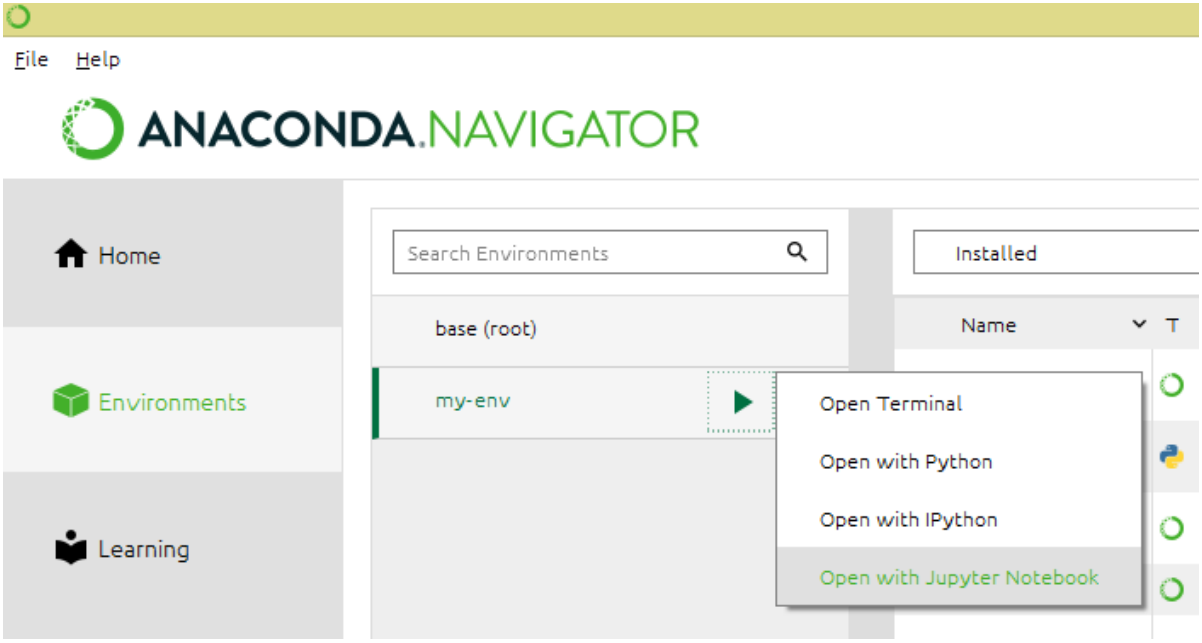


3. Install and Run Jupyter Notebook

Once you create the anaconda environment, go back to the Home page on Anaconda Navigator and install Jupyter Notebook from an application on the right panel.

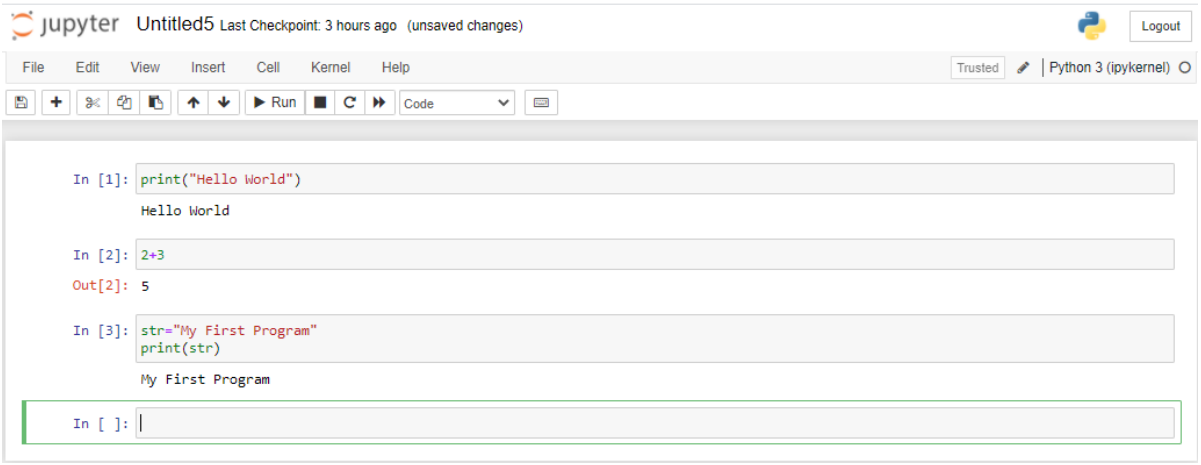


It will take a few seconds to install Jupyter to your environment, once the install completes, you can open Jupyter from the same screen or by accessing **Anaconda Navigator** -> **Environments** -> **your environment** (mine pandas-tutorial) -> select **Open With Jupyter Notebook**.



This opens up Jupyter Notebook in the default browser.

Now select **New** -> **PythonX** and enter the below lines and select **Run**. On Jupyter, each cell is a statement, so you can run each cell independently when there are no dependencies on previous cells.



Running Python Code in Visual Studio Code (VS Code)

Visual Studio Code (VS Code) is a lightweight yet powerful code editor that is widely used for Python development. It provides an excellent environment with features like syntax highlighting, intelligent code completion, integrated terminal, and powerful extensions. Once Python and the necessary extensions are set up, you can easily write, edit, and run your Python scripts directly from VS Code.

Why Use VS Code to Run Python Scripts?

- Easy to set up and use

- Integrated terminal for running scripts

- Support for virtual environments (including Conda)

- Built-in debugger and output viewer

- Rich extension marketplace

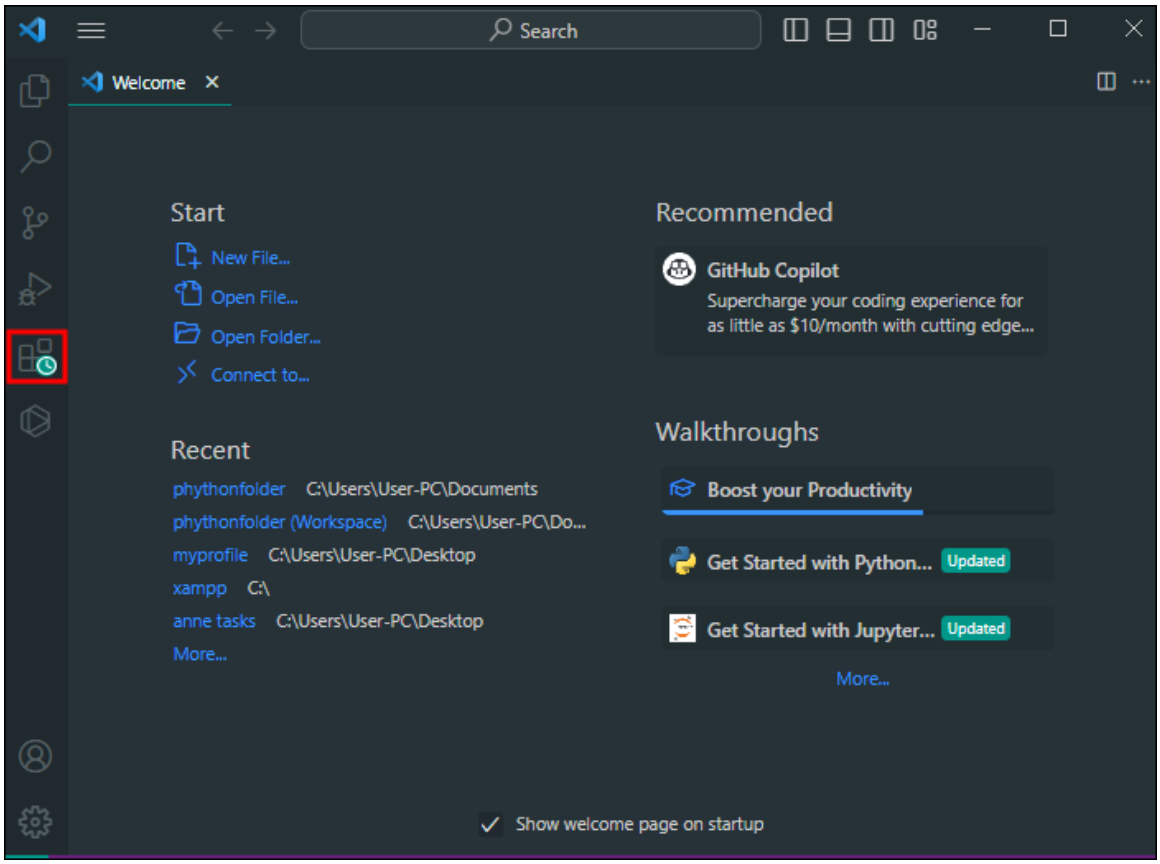
Prerequisites

Before running Python scripts, ensure the following are installed:

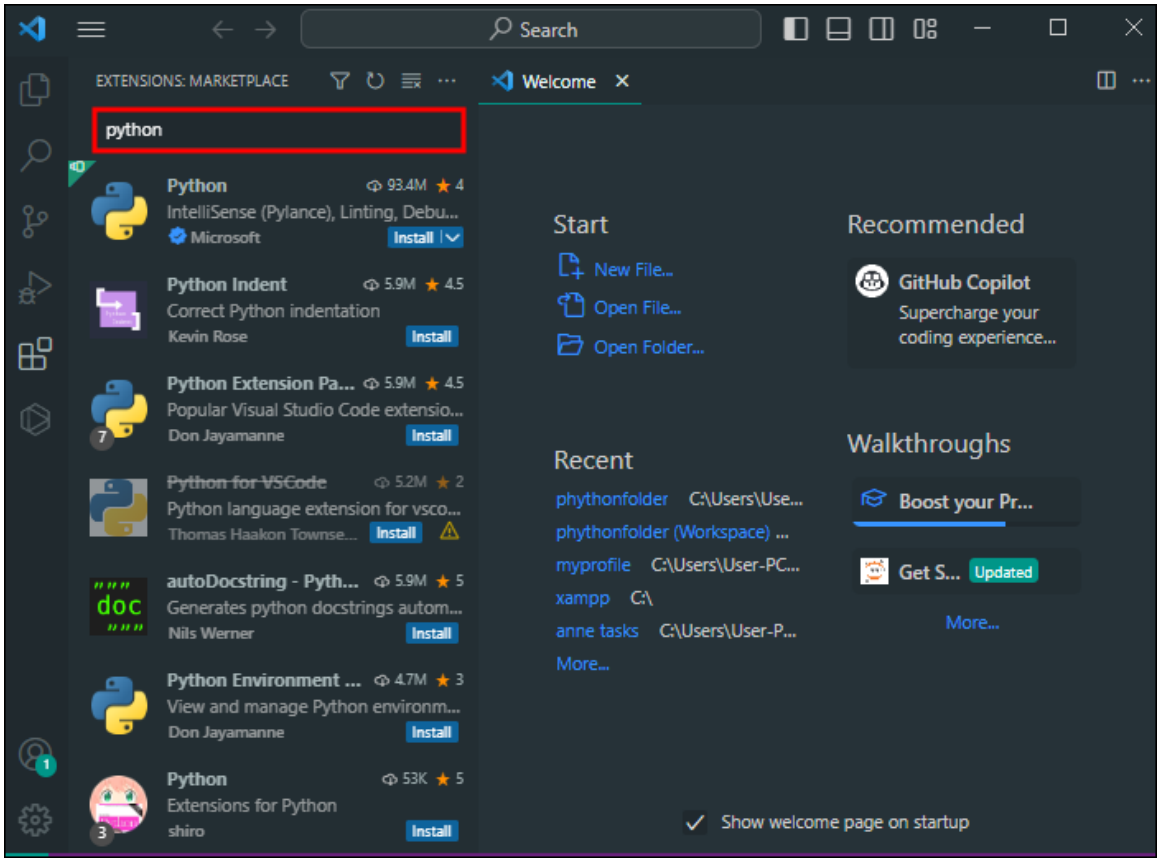
- Visual Studio Code (VS Code) url : https://code.visualstudio.com/download

- Python Extension in VS Code (provided by Microsoft)

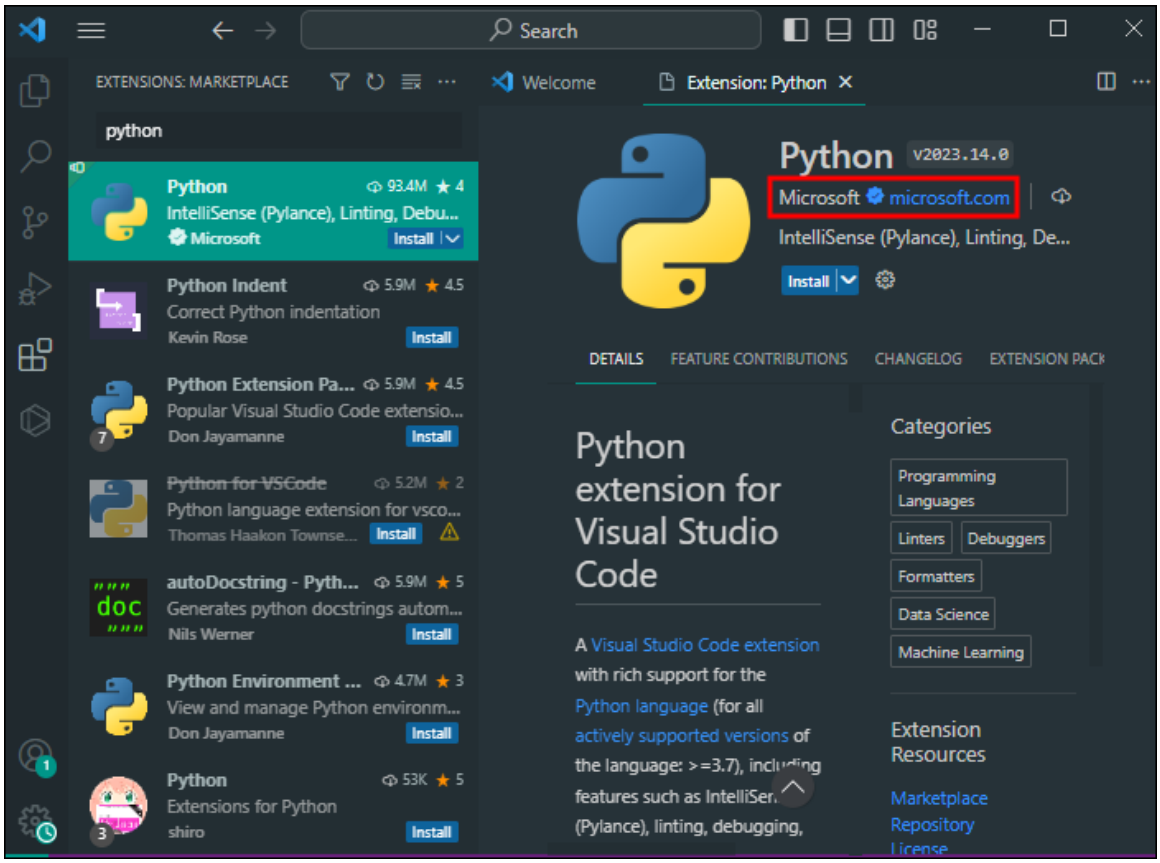
Click on the Extension tab. You can find it on the left side of the window, denoted by a four-squared icon.



Type “Python” into the extension search bar.



Select “Python from Microsoft” from the results.



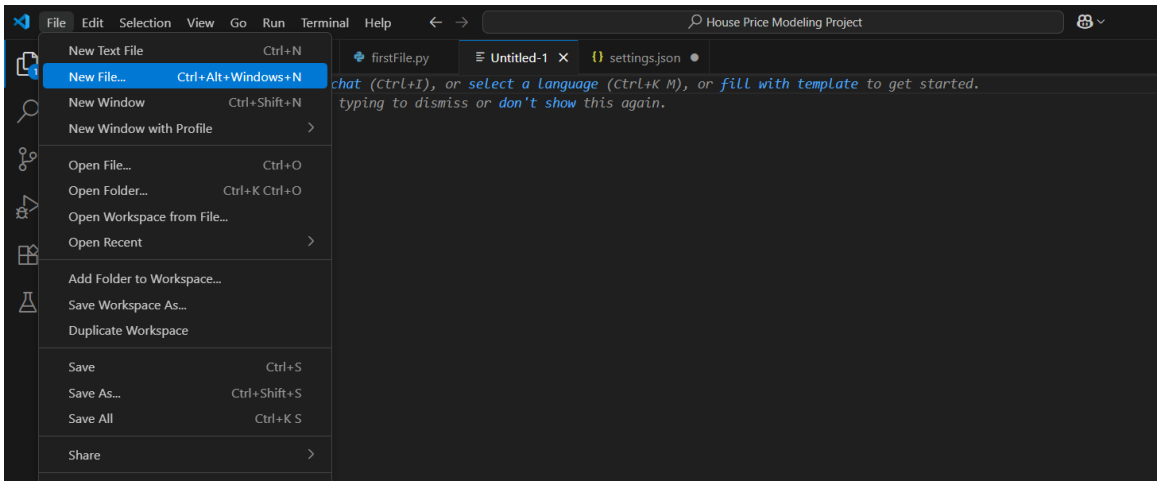
Steps to Run Python Script in VS Code

Step 1: Open VS Code

Start VS Code by clicking its icon from the Start menu or taskbar.

Step 2: Create or Open a Python File

Go to File → New File and select language as python and save the file



Step 3: Write Python Code

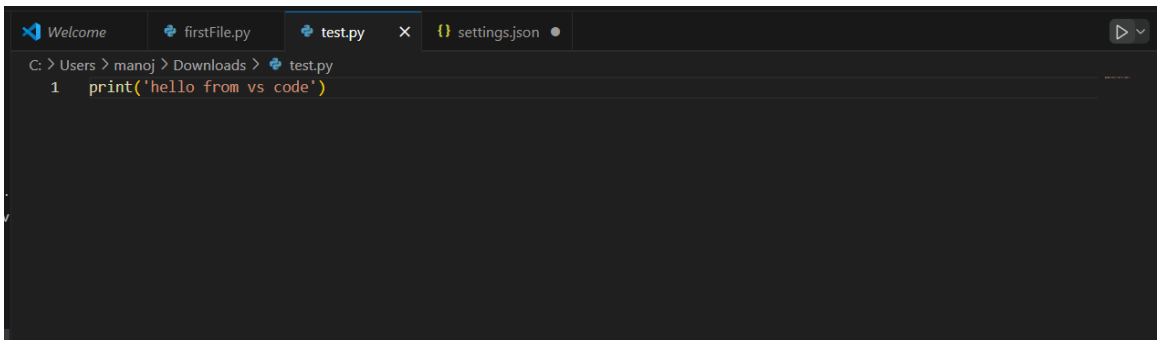
Example code:

print("Hello from VS Code!")

Step 5: Run the Python Script

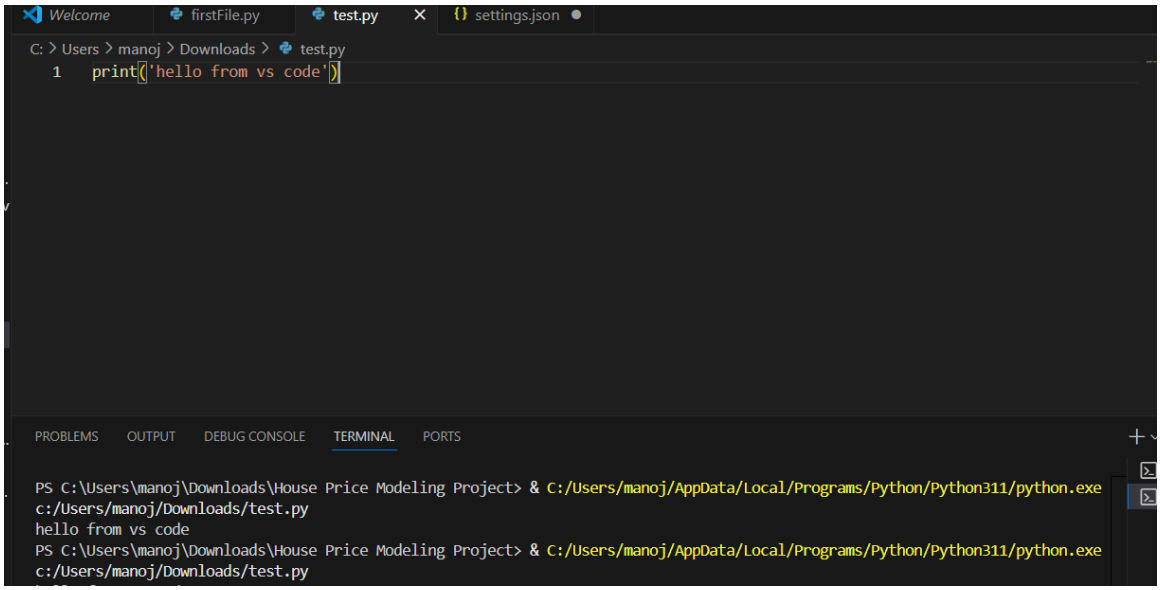
You can run the script in three ways:

1. Click the ▶ Play button in the top right corner.
2. Right-click inside the editor and choose 'Run Python File in Terminal'.



Step 6: View the Output

The output will appear in the integrated terminal at the bottom of the VS Code window.



Conclusion

VS Code offers a seamless experience for running and debugging Python code. Once set up, you can focus on development without worrying about switching tools or windows. It's an excellent choice for students, professionals, and hobbyists alike.

Structured Data: NumPy's Structured Arrays

While often our data can be well represented by a homogeneous array of values, sometimes this is not the case. This section demonstrates the use of NumPy's structured arrays and record arrays, which provide efficient storage for compound, heterogeneous data

Imagine that we have several categories of data on a number of people (say, name, age, and weight), and we'd like to store these values for use in a Python program. It would be possible to store these in three separate arrays:

name = ['Alice', 'Bob', 'Cathy', 'Doug'] age = [25, 45, 37, 19] weight = [55.0, 85.5, 68.0,61.5]

But this is a bit clumsy. There's nothing here that tells us that the three arrays are related; it would be more natural if we could use a single structure to store all of this data. NumPy can handle this through structured arrays, which are arrays with com- pound data types

Recall that previously we created a simple array using an expression like this:

x = np.zeros(4, dtype=int)

Creates an array of 4 records (rows).

Output

array([0, 0, 0, 0])

Fills all values initially with zero (or zero-equivalents for each field type).

We can similarly create a structured array using a compound data type specification:

data = np.zeros(4, dtype={'names':('name', 'age', 'weight'), 'formats':('U10', 'i4', 'f8')})

print(data.dtype)

Defines a **structured data type** (i.e., similar to a table with columns of different types).

Field	Type	Explanation
name	'U10'	Unicode string of up to 10 characters
age	'i4'	4-byte (32-bit) integer
weight	'f8'	8-byte (64-bit) float

So, each row in this array has:

- a name (string up to 10 characters),
- an age (integer),
- and a weight (float).

Why to use ?

- Structured arrays are useful when you want to work with **tabular data** (like a lightweight DataFrame).

  You can access fields like: data = np.zeros(4, dtype={'names': ('name', 'age', 'weight'), 'formats': ('U10', 'i4', 'f8')})

  data['name'] # Access all names

  data['age'] # Access all ages

Set Values

data[0] = ('Alice', 25, 55.5)

data[1] = ('Bob', 30, 72.0)

Get Values

first_row = data[0]

print("First row:", first_row)

output : First row: ('Alice', 25, 55.5)

You can also access individual fields like this:

```
print("Name:", first_row['name'])
print("Age:", first_row['age'])
print("Weight:", first_row['weight'])
```

Filter: Get all records where age > 25

```
mask = data['age'] > 25
result = data[mask]
```

More Advanced Compound Types

- **Complex queries (Boolean masking + multiple conditions)**
- **Nested dtypes (fields inside fields)**
- **Fancy indexing by field**
- **Sorting and filtering based on one field**

Example

```
import numpy as np
data = np.array([
    ('Alice', 25, 55.5),
    ('Bob', 30, 72.0),
    ('Carol', 22, 60.2),
    ('David', 35, 68.4)
], dtype=[('name', 'U10'), ('age', 'i4'), ('weight', 'f8')])
```

Boolean Masking + Multiple Conditions

```
# People over 25 years and weighing less than 70
filtered = data[(data['age'] > 25) & (data['weight'] < 70)]
print("Filtered:\n", filtered)
```

Sorting by a Field

```
sorted_by_age = np.sort(data, order='age')
print("Sorted by age:\n", sorted_by_age)
```

Nested Data Types (Twist!)

```
nested_dtype = np.dtype([
    ('name', 'U10'),
    ('age', 'i4'),
    ('metrics', [('weight', 'f4'), ('height', 'f4')])    # Nested field
])

data2 = np.array([
    ('Alice', 25, (55.5, 160.0)),
    ('Bob', 30, (72.0, 175.2))
], dtype=nested_dtype)

print("Nested structure:\n", data2)
print("Access Alice's height:", data2[0]['metrics']['height'])
```

Output

```
Nested structure:
 [('Alice', 25, (55.5, 160. )) ('Bob', 30, (72. , 175.2))]
  Access Alice's height: 160.0
```

Fancy Field Indexing

```
ages = data['age']
print("All ages:", ages)
```

The Pandas Series Object

The Series is the simplest data structure in Pandas. It's a one-dimensional labelled array capable of holding any data type — integers, floats, strings, Python objects, etc.

Key Characteristics:

- One-dimensional
- Has index (labels for each element)

- Can hold any data type
- Built on top of NumPy array

Example :

```
import pandas as pd
data = [10, 20, 30, 40]
s = pd.Series(data)
print(s)
```

Output

```
0    10
1    20
2    30
3    40
dtype: int64
```

With custom index:

```
s = pd.Series([10, 20, 30], index=['a', 'b', 'c'])
print(s)
```

Output

```
a    10
b    20
c    30
dtype: int64
```

From a dictionary:

```
data = {'a': 1, 'b': 2, 'c': 3}
s = pd.Series(data)
```

Output

```
a    1
b    2
```

c 3

Accessing Data in a Series

```
s['a']      # Access by label
s[0]        # Access by position
s[0:2]      # First two elements
```

```
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                    'Illinois': 12882135}
population = pd.Series(population_dict)
```

Output

California 38332521

Florida 19552860

Illinois 12882135

New York 19651127

Texas 26448193

Pandas Series - Explanation and Operations

What is a Pandas Series?

A **Series** in pandas is a **1-dimensional labelled array** that can hold any data type like integers, floats, strings, etc. It is similar to a single column in an Excel sheet.

Features of Series:

- One-dimensional data structure
- Can hold different data types (int, float, string, etc.)

- Has labelled index
- Supports element-wise operations

Creating a Series

```
import pandas as pd


# Simple Series
s = pd.Series([10, 20, 30, 40])
print(s)
```

Output:

```
0    10
1    20
2    30
3    40
dtype: int64
```

Custom Index Series

```
s = pd.Series([90, 80, 70], index=['Math', 'Science', 'English'])
print(s)
```

Output:

```
Math       90
Science    80
English    70
dtype: int64
```

Accessing Elements

```
print(s['Math'])      # Access by label
print(s[1])           # Access by position
```

Output:

90

80

Arithmetic Operations

```
s = pd.Series([10, 20, 30])
print(s + 5)        # Add 5 to each element
print(s * 2)        # Multiply each element by 2
```

Output:

```
0    15
1    25
2    35
dtype: int64

0    20
1    40
2    60
dtype: int64
```

Statistical Operations

```
s = pd.Series([4, 8, 15, 16, 23, 42])
print("Max:", s.max())
print("Min:", s.min())
print("Mean:", s.mean())
print("Sum:", s.sum())
```

Output:

Max: 42

Min: 4

Mean: 18.0

Sum: 108

Filtering / Conditional Selection

```
s = pd.Series([45, 90, 60, 30, 80], index=['A', 'B', 'C', 'D', 'E'])
print(s[s > 60])
```

Output:

```
B    90
E    80
dtype: int64
```

Updating Values

```
s = pd.Series([10, 20, 30], index=['x', 'y', 'z'])
s['y'] = 99
print(s)
```

Output:

```
x    10
y    99
z    30
dtype: int64
```

Appending Two Series

```
s1 = pd.Series([1, 2], index=['a', 'b'])
s2 = pd.Series([3, 4], index=['c', 'd'])
s3 = s1.append(s2)
print(s3)
```

Output:

```
a    1
```

b 2

c 3

d 4

dtype: int64

Sorting Values and Index

```
s = pd.Series([30, 10, 50], index=['a', 'b', 'c'])
print(s.sort_values())        # By value
print(s.sort_index())         # By index
```

Output (Sorted by Values):

b 10

a 30

c 50

dtype: int64

Output (Sorted by Index):

a 30

b 10

c 50

dtype: int64

Checking for Missing Values

```
s = pd.Series([10, None, 20, None])
print(s.isnull())          # Check missing (NaN)
print(s.notnull())         # Check non-missing
```

Output:

0 False

1 True

2 False

3 True

dtype: bool

0 True

1 False

2 True

3 False

dtype: bool

Summary

- Pandas Series is a powerful tool for handling labeled 1D data.
- Supports indexing, filtering, statistical analysis, and arithmetic operations.
- Useful for tasks like time series, single-column data analysis, and quick filtering.

Introduction to DataFrames in Python

A DataFrame is a 2-dimensional labeled data structure provided by the pandas library in Python. It's similar to a table in a database or an Excel spreadsheet. Each column can have a different data type, making it a powerful tool for data analysis.

Creating a DataFrame

```
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Department': ['HR', 'IT', 'Finance']
}

df = pd.DataFrame(data)
print(df)
```

Output:

	Name	Age	Department
0	Alice	25	HR
1	Bob	30	IT
2	Charlie	35	Finance

Common Operations on DataFrames

- **df.info()**

  Provides a concise summary of the DataFrame.
  It helps in understanding
  - Total number of entries (rows)
  - Data types of each column
  - Non-null values per column
  - Memory usage

  Output

  ```
  <class 'pandas.core.frame.DataFrame'>
  Data columns (total 3 columns):
  ```

#	Column	Non-Null Count	Dtype
0	Name	3 non-null	object
1	Age	3 non-null	int64
2	Department	3 non-null	object

df.describe()

- Gives statistical summary of **numeric** columns by default.
  Includes:
  - Count, Mean, Std (standard deviation)
  - Min, Max
  - 25%, 50%, 75% (percentiles)

  Output:

	Age

```
count    3.0
mean    30.0
std      5.0
min     25.0
25%     27.5
50%     30.0
75%     32.5
max     35.0
```

df.shape

- Returns the dimensions of the DataFrame in the form (rows, columns)

```
df.shape
# Output: (3, 3)
    df.columns          # List of column names
    df.head(3)          # First 3 rows
    df.tail(2)          # Last 2 rows
```

Selecting Data

- Select a single column- df['Name']

  Output:

```
0      Alice
1        Bob
2    Charlie
      Name: Name, dtype: object
```

- Select multiple columns - df[['Name', 'Age']]

- Iloc

.iloc is a Pandas indexer used to select data by row and column numbers (integer-location based indexing)

syntax
df.iloc[row_index, column_index]

Example:

1. Get 2nd row 3rd column value

   df.iloc[1, 2]

   Output : 'IT'
2. Get first two rows df.iloc[0:2]

   Output :

	Name	Age	Department
0	Alice	25	HR
1	Bob	30	IT

3. Get specific rows and columns

   df.iloc[[0, 2], [0, 1]] # rows 0 & 2, columns 0 & 1

	Name	Department
0	Alice	HR
2	Charlie	Finance

4. All rows, only column at index 1

   df.iloc[:, 1]

   Output:

   0 25
   1 30
   2 35
   Name: Age, dtype: int64

- .loc is a **label-based indexer** used to **select rows and columns by their labels** (i.e., names).
Syntax : df.loc[row_label, column_label]

Example :

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

```
    'Age': [25, 30, 35],
    'City': ['Delhi', 'Mumbai', 'Chennai']
}

df = pd.DataFrame(data, index=['a', 'b', 'c'])
```

1. Get row with index 'a'

```
df.loc['a']
```

Output:

```
Name       Alice
Age           25
City       Delhi
Name: a, dtype: object
```

2. Get age of row 'b'

```
df.loc['b', 'Age']      # Output: 30
```

3. Get multiple rows

```
df.loc[['a', 'c']]
```

	Name	Age	City
a	Alice	25	Delhi
c	Charlie	35	Chennai

4. Get specific rows and columns

```
df.loc[['a', 'c'], ['Name', 'City']]
```

Output :

	Name	City
a	Alice	Delhi
c	Charlie	Chennai

5. All rows, only 'City' column

```
df.loc[:, 'City']
```

Output :

```
a          Delhi
b         Mumbai
```

```
c       Chennai
Name: City, dtype: object
```

Filtering Data

```
df[df['Age'] > 28]                    # Rows where Age > 28
df[df['Department'] == 'IT']          # Rows in IT department
```

Adding/Modifying Columns

- df['Salary'] = [50000, 60000, 70000] – Adds a new column.
- df['AgePlus10'] = df['Age'] + 10 – Adds a calculated column.

Deleting/Sorting Data

- df.drop('AgePlus10', axis=1, inplace=True) – Deletes a column.
- df.sort_values(by='Age', ascending=False) – Sorts by age descending.

Grouping and Aggregation

- df.groupby('Department')['Age'].mean() – Groups by department and calculates average age.

Exporting Data

```
df.to_csv('output.csv', index=False)
df.to_excel('output.xlsx', index=False)
```

How to read Excel using DataFrame

```
import pandas as pd
# If your file is in the same directory
df = pd.read_excel("your_file.xlsx")
```

```
# If you have multiple sheets
df = pd.read_excel("your_file.xlsx", sheet_name='Sheet1')
df.head()            # Shows first 5 rows
df.tail()            # Shows last 5 rows
df.shape             # Shows (rows, columns)
df.info()            # Summary of data types and non-null values
df.describe()        # Summary statistics
```

Handling Missing Data

df.isnull() - **used to detect missing (null/NaN) values** in a DataFrame

df.fillna(0) - .fillna() is used to **replace missing values (NaN) with a specified value or method**.

```
df.dropna()                  # Drop rows with missing values
```

Selecting Data

- df['Name'] – Selects a single column.
- df[['Name', 'Age']] – Selects multiple columns.
- df.iloc[1] – Accesses the second row by position.
- df.loc[1] – Accesses the second row by index label.

UNIT -2

Handling Missing Values

Missing values are a common challenge in machine learning and data analysis. They occur when certain data points are missing for specific variables in a dataset. These gaps in information can take the form of blank cells, null values or special symbols like "NA", "NaN" or "unknown." If not addressed properly, missing values can harm the accuracy and reliability of our models. They can reduce the sample size, introduce bias and make it difficult to apply certain analysis techniques that require complete data. Efficiently handling missing values is important to ensure our machine learning models produce accurate and unbiased results, we'll see more about the methods and strategies to deal with missing data effectively.

	School ID	Name	Address	City	Subject	Marks	Rank	Grade
0	101.0	Alice	123 Main St	Los Angeles	Math	85.0	2	B
1	102.0	Bob	456 Oak Ave	New York	English	92.0	1	A
2	103.0	Charlie	789 Pine Ln	Houston	Science	78.0	4	C
3	NaN	David	101 Elm St	Los Angeles	Math	89.0	3	B
4	105.0	Eva	NaN	Miami	History	NaN	8	D
5	106.0	Frank	222 Maple Rd	NaN	Math	95.0	1	A
6	107.0	Grace	444 Cedar Blvd	Houston	Science	80.0	5	C
7	108.0	Henry	555 Birch Dr	New York	English	88.0	3	B

Importance of Handling Missing Values

Handling missing values is important for ensuring the accuracy and reliability of data analysis and machine learning models. Key reasons include:

- **Improved Model Accuracy:** Addressing missing values helps avoid incorrect predictions and boosts model performance.
- **Increased Statistical Power:** Imputation or removal of missing data allows the use of more analysis techniques, maintaining the sample size.
- **Bias Prevention:** Proper handling ensures that missing data doesn’t introduce systematic bias, leading to more reliable results.

- **Better Decision-Making:** A clean dataset leads to more informed, trustworthy decisions based on accurate insights.

Challenges Posed by Missing Values

Missing values can introduce several challenges in data analysis including:

- **Reduce sample size:** If rows or data points with missing values are removed, it reduces the overall sample size which may decrease the reliability and accuracy of the analysis.
- **Bias in Results:** When missing data is not handled carefully, it can introduce bias. This is especially problematic when the missingness is not random, leading to misleading conclusions.
- **Difficulty in Analysis:** Many statistical techniques and machine learning algorithms require complete data for all variables. Missing values can cause certain analyses or models inapplicable, limiting the methods we can use.

Understanding Different Types of 'Missing' Data

None: Pythonic Missing Data

In pure Python, the built-in constant None is used to represent the absence of a value or missing data. It's often the default for variables that haven't been initialized or when a function returns nothing.

Example : data = [10, 15, None, 25]

```
for value in data:
    if value is None:
        print("Missing value found!")
    else:
        print("Value:", value)
```

Output :

Value: 10

Value: 15

Missing value found!

Value: 25

Example :2

```
import pandas as pd
```

```
df = pd.DataFrame({
    'name': ['Alice', 'Bob', None],
    'age': [25, None, 30]
})
print(df)
```

Output

```
    name   age
0  Alice  25.0
1    Bob   NaN
2   None  30.0
```

Operating on Null Values

As we have seen, Pandas treats None and NaN as essentially interchangeable for indicating missing or null values. To facilitate this convention, there are several useful methods for detecting, removing, and replacing null values in Pandas data structures. They are:

- isnull()
  Generate a Boolean mask indicating missing values
- notnull()
  Opposite of isnull()
- dropna()
  Return a filtered version of the data
- fillna()
  Return a copy of the data with missing values filled or imputed

Detecting null values

Pandas data structures have two useful methods for detecting null data: isnull() and notnull(). Either one will return a Boolean mask over the data. For example

```
In[13]: data = pd.Series([1, np.nan, 'hello', None])

In[14]: data.isnull()

Out[14]: 0    False
         1     True
         2    False
         3     True
         dtype: bool

In[15]: data[data.notnull()]

Out[15]: 0        1
         2    hello
         dtype: object
```

The isnull() and notnull() methods produce similar Boolean results for Data Frames.

Dropping null values

In addition to the masking used before, there are the convenience methods, dropna() (which removes NA values) and fillna() (which fills in NA values). For a Series, the result is straightforward:

For a DataFrame, there are more options. Consider the following DataFrame:

```
In[17]: df = pd.DataFrame([[1,      np.nan, 2],
                           [2,      3,      5],
                           [np.nan, 4,      6]])
        df

Out[17]:      0    1  2
          0  1.0  NaN  2
          1  2.0  3.0  5
          2  NaN  4.0  6
```

We cannot drop single values from a DataFrame; we can only drop full rows or full columns. Depending on the application, you might want one or the other, so dropna() gives a number of options for a DataFrame.

By default, dropna() will drop all rows in which any null value is present:

```
In[18]: df.dropna()

Out[18]:      0    1  2
          1  2.0  3.0  5
```

Alternatively, you can drop NA values along a different axis; axis=1 drops all columns containing a null value:

```
In[19]: df.dropna(axis='columns')

Out[19]:    2
         0  2
         1  5
         2  6
```

But this drops some good data as well; you might rather be interested in dropping rows or columns with all NA values, or a majority of NA values. This can be specified through the how or thresh parameters, which allow fine control of the number of nulls to allow through.

The default is how='any', such that any row or column (depending on the axis key- word) containing a null value will be dropped. You can also specify how='all', which will only drop rows/columns that are all null values:

```
In[20]: df[3] = np.nan
        df

Out[20]:      0    1  2   3
         0  1.0  NaN  2 NaN
         1  2.0  3.0  5 NaN
         2  NaN  4.0  6 NaN

In[21]: df.dropna(axis='columns', how='all')

Out[21]:      0    1  2
         0  1.0  NaN  2
         1  2.0  3.0  5
         2  NaN  4.0  6
```

For finer-grained control, the thresh parameter lets you specify a minimum number of non-null values for the row/column to be kept:

```
In[22]: df.dropna(axis='rows', thresh=3)

Out[22]:      0    1  2   3
         1  2.0  3.0  5 NaN
```

thresh=3 **Keep** only rows with **at least 3 non-null values**

Filling null values

Sometimes rather than dropping NA values, you'd rather replace them with a valid value. This value might be a single number like zero, or it might be some sort of imputation or interpolation from the good values. You could do this in-place using the isnull() method as a mask, but because it is such a common operation Pandas provides the fillna() method, which returns a copy of the array with the null values replaced.

data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))

We can fill NA entries with a single value, such as zero:

```
In[24]: data.fillna(0)

Out[24]: a    1.0
         b    0.0
         c    2.0
         d    0.0
         e    3.0
         dtype: float64
```

We can specify a forward-fill to propagate the previous value forward:

```
In[25]: # forward-fill
        data.fillna(method='ffill')

Out[25]: a    1.0
         b    1.0
         c    2.0
         d    2.0
         e    3.0
         dtype: float64
```

Or we can specify a back-fill to propagate the next values backward

```
In[26]: # back-fill
        data.fillna(method='bfill')

Out[26]: a    1.0
         b    2.0
         c    2.0
         d    3.0
         e    3.0
         dtype: float64
```

For DataFrames, the options are similar, but we can also specify an axis along which the fills take place:

```
In[27]: df

Out[27]:      0    1  2   3
         0  1.0  NaN  2 NaN
         1  2.0  3.0  5 NaN
         2  NaN  4.0  6 NaN

In[28]: df.fillna(method='ffill', axis=1)

Out[28]:      0    1    2    3
         0  1.0  1.0  2.0  2.0
         1  2.0  3.0  5.0  5.0
         2  NaN  4.0  6.0  6.0
```

Trade-Offs in Missing Data Conventions

There are several strategies to handle missing data, such as filling with mean, median, or mode, interpolation, or dropping rows entirely. Each strategy has its own advantages and disadvantages, and choosing the right one involves making trade-offs.

1. Filling Missing Values with Mean

Description:

The mean (average) is a commonly used method for replacing missing values in numeric columns.

Pros:

- Easy to compute and implement.
- Preserves the dataset size (no rows dropped).
- Useful when data is symmetrically distributed.

Cons:

- Sensitive to outliers, which can distort the average.
- Can underestimate or overestimate values if the distribution is skewed.
- Reduces data variance and can lead to biased models.

Real-World Example:

In a student performance dataset, if one student's math score is missing, we might fill it with the average score of the class. However, if a few students have extremely high or low scores, the average might not reflect a fair value.

Example : import pandas as pd

```
import numpy as np


data = {'student': ['A', 'B', 'C', 'D'],
        'marks': [80, 85, np.nan, 90]}
df = pd.DataFrame(data)
# Fill missing marks with mean
df['marks'].fillna(df['marks'].mean(), inplace=True)
```

2. Filling Missing Values with Median

Description:

The median is the middle value when all values are sorted. It is a better measure of central tendency for skewed data.

Pros:

- Not affected by outliers.
- Better for skewed data distributions (e.g., income).

Cons:

- Ignores distribution shape in normal data.
- Doesn’t consider the relationships between data points.

Real-World Example:

In a financial dataset containing household income, if one value is missing, replacing it with the median income avoids the problem of a few very rich households skewing the results.

```
import pandas as pd
import numpy as np
data = {'customer': ['A', 'B', 'C', 'D', 'E'],
        'income': [30000, 35000, np.nan, 40000, 1000000]}     # Notice the outlier (1000000)
df = pd.DataFrame(data)
```

df['income'].fillna(median_income, inplace=True)

Output :

customer		income	
0	A	30000.0	
1	B	35000.0	
2	C	37500.0	← filled with median
3	D	40000.0	
4	E	1000000.0	

3. Filling Missing Values with Mode

Description:

The mode is the most frequent value in a column. This method is especially useful for categorical data.

Pros:

- Best suited for categorical variables.
- Maintains the most common class or category.

Cons:

- Doesn’t work well with numeric or continuous variables.
- Can introduce bias if the missing data is not actually most similar to the mode.

Real-World Example:

In a survey dataset, if the gender column is missing for some entries, and the most frequent gender is 'Female', we might fill the missing entries with 'Female'. However, this can lead to overrepresentation of that category.

```
import pandas as pd

import numpy as np

data = {'customer': [1, 2, 3, 4, 5],

        'gender': ['Male', 'Female', np.nan, 'Male', np.nan]}

df = pd.DataFrame(data)

df['gender'].fillna(df['gender'].mode()[0], inplace=True)

df
```

	customer	gender	
0	1	Male	
1	2	Female	
2	3	Male	← filled with mode
3	4	Male	
4	5	Male	← filled with mode

4. Interpolation

Description:

Estimates missing values by using the values before and after the missing one. Commonly used for time series data.

Pros:

- Maintains trends and patterns.
- Preserves time-sequence data.

Cons:

- Can create artificial trends.
- Not suitable for unordered or categorical data.

Real-World Example:

In a weather dataset, if the temperature reading is missing for a single day, it can be estimated by taking the average of the temperature from the previous and next day.

```
import pandas as pd

import numpy as np

# Temperature data with one missing value

data = {'day': ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'],

        'temperature': [28.0, 30.0, np.nan, 32.0, 34.0]}

df = pd.DataFrame(data)

df['temperature'] = df['temperature'].interpolate()
```

Output :

	day	temperature
0	Mon	28.0

1 Tue 30.0

2 Wed 31.0 ← interpolated between Tue (30.0) and Thu (32.0)

3 Thu 32.0

4 Fri 34.0

Data Duplication Removal from Dataset

Duplicates are a common issues in real-world datasets that can negatively impact our analysis. They occur when identical rows or entries appear multiple times in a dataset. Although they may seem harmless but they can cause problems in analysis if not fixed. Duplicates could happen due to:

- Data entry errors: When the same information is recorded more than once by mistake.
- Merging datasets: When combining data from different sources can lead to overlapping of data that can create duplicates.

Why Duplicates Can Cause Problems?

- **Skewed Analysis**: Duplicates can affect our analysis results which leads to misleading conclusions such as an wrong average salary.
- **Inaccurate Models:** It can cause machine learning models to overfit which reduces their ability to perform well on new data.
- **Increased Computational Costs:** It consume extra computational power which slows down analysis and impacts workflow.
- **Data Redundancy and Complexity:** It make it harder to maintain accurate records and organize data and adds unnecessary complexity.

Identifying Duplicates

- **To manage duplicates the first step is identifying them in the dataset. Pandas offers various functions which are helpful to spot and remove duplicate rows**

```
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Alice', 'Charlie', 'Bob', 'David'],
    'Age': [25, 30, 25, 35, 30, 40],
    'City': ['New York', 'Los Angeles', 'New York', 'Chicago', 'Los Angeles', 'San Francisco']
}

df = pd.DataFrame(data)
df
Output
```

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
2	Alice	25	New York
3	Charlie	35	Chicago
4	Bob	30	Los Angeles
5	David	40	San Francisco

1. Using duplicated() Method

- **The duplicate() method helps to identify duplicate rows in a dataset. It returns a boolean Series indicating whether a row is a duplicate of a previous row.**
  duplicates = df.duplicated()
  duplicates
  Output:

	0
0	False
1	False
2	True
3	False
4	True
5	False

dtype: bool

Removing Duplicates

Duplicates may appear in one or two columns instead of the entire dataset. In such cases, we can choose specific columns to check for duplicates.

1. Based on Specific Columns

Here we will specify columns i.e name and city to remove duplicates
using drop_duplicates().

df_no_duplicates_columns = df.drop_duplicates(subset=['Name', 'City'])

df_no_duplicates_columns

Output

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
3	Charlie	35	Chicago
5	David	40	San Francisco

2. Keeping the First or Last Occurrence

By default drop_duplicates() keeps the first occurrence of each duplicate row. However, we can adjust it to keep the last occurrence instead

df_keep_last = df.drop_duplicates(keep='last')

df_keep_last

Output

	Name	Age	City
2	Alice	25	New York
3	Charlie	35	Chicago
4	Bob	30	Los Angeles
5	David	40	San Francisco

Outliers

Outliers are data points that deviate significantly from other data points in a dataset. They can arise from a variety of factors such as measurement errors, rare events or natural variations in the data. If left unchecked it can distort data analysis, skew statistical results and impact machine learning model performance

Methods for Detecting and Removing Outliers

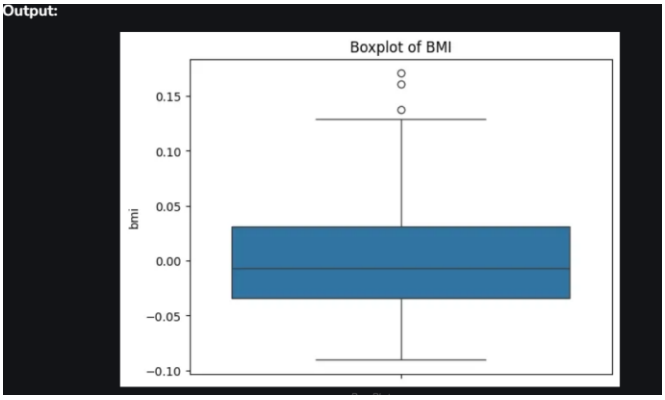
There are several ways to detect and handle outliers in Python. We can use visualization techniques or statistical methods depending on the nature of our data Each method serves different purposes and is suited for specific types of data. Here we will be using Pandas and Matplotlib libraries on the Diabetes dataset which is preloaded in the Sckit-learn library.

1. **Visualizing and Removing Outliers Using Box Plots**

```
import sklearn

from sklearn.datasets import load_diabetes

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

diabetes = load_diabetes()

column_name = diabetes.feature_names

df_diabetics = pd.DataFrame(diabetes.data, columns=column_name)

sns.boxplot(df_diabetics['bmi'])
```

```
plt.title('Boxplot of BMI')

plt.show()
```

Output:

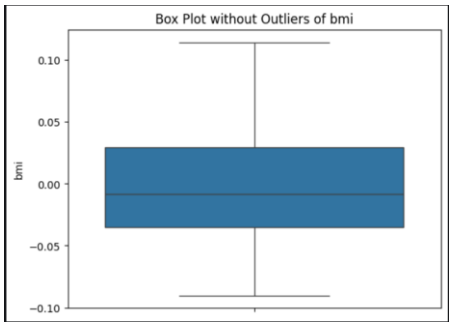


in the boxplot, outliers appear as points outside the whiskers. These values are much higher or lower than the rest of the data. For example, bmi values above 0.12 could be identified as outliers.

To remove outliers, we can define a threshold value and filter the data.

```
def removal_box_plot(df, column, threshold):
    removed_outliers = df[df[column] <= threshold]
    sns.boxplot(removed_outliers[column])
    plt.title(f'Box Plot without Outliers of {column}')
    plt.show()
    return removed_outliers
threshold_value = 0.12
no_outliers = removal_box_plot(df_diabetics, 'bmi', threshold_value)
```

Output



2. IQR Method (Interquartile Range)

This method is a widely used and reliable technique for detecting outliers. It is robust to skewed data and helps identify extreme values based on quartiles and it most trusted approach used in the research field. The IQR is calculated as the difference between the third quartile (Q3) and the first quartile (Q1):

$IQR = Q3 - Q1$

```
Q1 = df['bmi'].quantile(0.25)

Q3 = df['bmi'].quantile(0.75)
```

To define the outlier base value is defined above and below dataset's normal range namely Upper and Lower bounds define the upper and the lower bound (1.5*IQR value is considered i.e:

- $upper = Q3 + 1.5 * IQR$ $upper = Q3 + 1.5 * IQR$
- $lower = Q1 - 1.5 * IQR$ $lower = Q1 - 1.5 * IQR$

In the above formula the 0.5 scale-up of IQR (new_IQR = IQR + 0.5*IQR) is taken to consider all the data between 2.7 standard deviations in the Gaussian Distribution.

```
upper = Q3+1.5*IQR

upper_array = np.array(df_diabetics['bmi'] >= upper)

print("Upper Bound:", upper)

print(upper_array.sum())


lower = Q1-1.5*IQR

lower_array = np.array(df_diabetics['bmi'] <= lower)

print("Lower Bound:", lower)

print(lower_array.sum())
```

Output:

```
Upper Bound: 0.12879000811776306
3
Lower Bound: -0.13204051376139045
0
```

Now lets detect and remove outlier using the interquartile range (IQR).

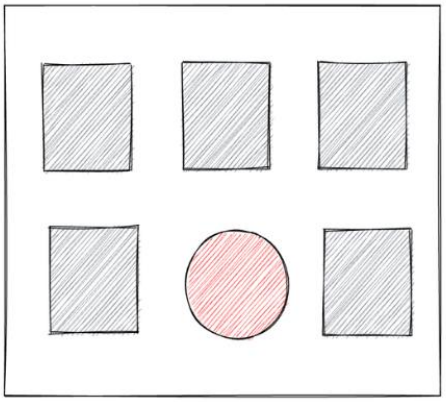
Here we are using the interquartile range (IQR) method to detect and remove outliers in the 'bmi' column of the diabetes dataset. It calculates the upper and lower limits based on the IQR it identifies outlier indices using Boolean arrays and then removes the corresponding rows from the DataFrame which

results in a new DataFrame with outliers excluded. The before and after shapes of the DataFrame are printed for comparison.

```
    import sklearn
from sklearn.datasets import load_diabetes
import pandas as pd
diabetes = load_diabetes()

column_name = diabetes.feature_names
df_diabetes = pd.DataFrame(diabetes.data)
df_diabetes .columns = column_name
df_diabetes .head()
print("Old Shape: ", df_diabetes.shape)



Q1 = df_diabetes['bmi'].quantile(0.25)
Q3 = df_diabetes['bmi'].quantile(0.75)
IQR = Q3 - Q1
lower = Q1 - 1.5*IQR
upper = Q3 + 1.5*IQR

upper_array = np.where(df_diabetes['bmi'] >= upper)[0]
lower_array = np.where(df_diabetes['bmi'] <= lower)[0]

df_diabetes.drop(index=upper_array, inplace=True)
df_diabetes.drop(index=lower_array, inplace=True)

print("New Shape: ", df_diabetes.shape)
```

Output:

Old Shape: (442, 10)
New Shape: (439, 10)



What are Anomalies

An anomaly is any data point, event, or pattern that does not conform to the expected behavior of the dataset.

- Sometimes, anomalies are bad data (errors, noise).
- Sometimes, anomalies are critical insights (fraud, failures, rare events).

Types of Anomalies

1. Point Anomalies (Global Anomalies)

Definition:
A single data point that is significantly different from the majority of other data points in the dataset. This is the most common and widely recognized anomaly type.

Real-World Examples:

- Credit Card Fraud: Suddenly seeing a $10,000 charge on your credit card, when your typical bill is around $2,000.

    - Healthcare: A patient's heart rate suddenly spikes to 200 bpm for a single measurement, even though previous and following readings are normal.
    - Network Security: A sudden, singular spike in network traffic that could indicate a potential DDoS attack.
- Energy Usage: A household records an abnormally high electricity usage for one day, suggesting a faulty appliance or unauthorized use.

2. Contextual Anomalies (Conditional Anomalies)

Definition:
A data point that is only anomalous in a specific context (such as time, season, or environment) but might appear normal otherwise.

Real-World Examples:

- **Website Traffic**: High web traffic during a marketing campaign is expected, but a similar traffic spike on a regular night might indicate bot activity or a technical glitch.
- **Energy Consumption**: Elevated energy use is normal during daytime for offices, but high power usage late at night may be suspicious and could indicate problems.
- **Healthcare Monitoring:** An increased heart rate during exercise is normal, but the same elevated rate during rest is a contextual anomaly.

3. Collective Anomalies

Definition:
A collection or sequence of data points that, as a group, represents anomalous behavior, even if individual points are not anomalous.

Real-World Examples:

- Cybersecurity: Multiple failed login attempts from the same IP address in a short period indicate a brute-force attack, even if a single failed attempt is normal.

- Finance: A customer makes several small transactions in rapid succession to evade detection—together, these form a collective anomaly and potential money laundering case.
- Healthcare: A slowly and steadily declining vital sign (such as blood pressure dropping a little every hour) may indicate an emerging medical crisis.
- Traffic Management: While low traffic volumes on certain days are normal, a week-long, sustained dip could indicate construction or an event affecting traffic flow.

Anomalies in EDA (Exploratory Data Analysis)

In EDA, anomalies are often treated as outliers. The purpose is to understand data quality, distribution, and patterns.

Common EDA Techniques to Detect Anomalies

Visualization-based

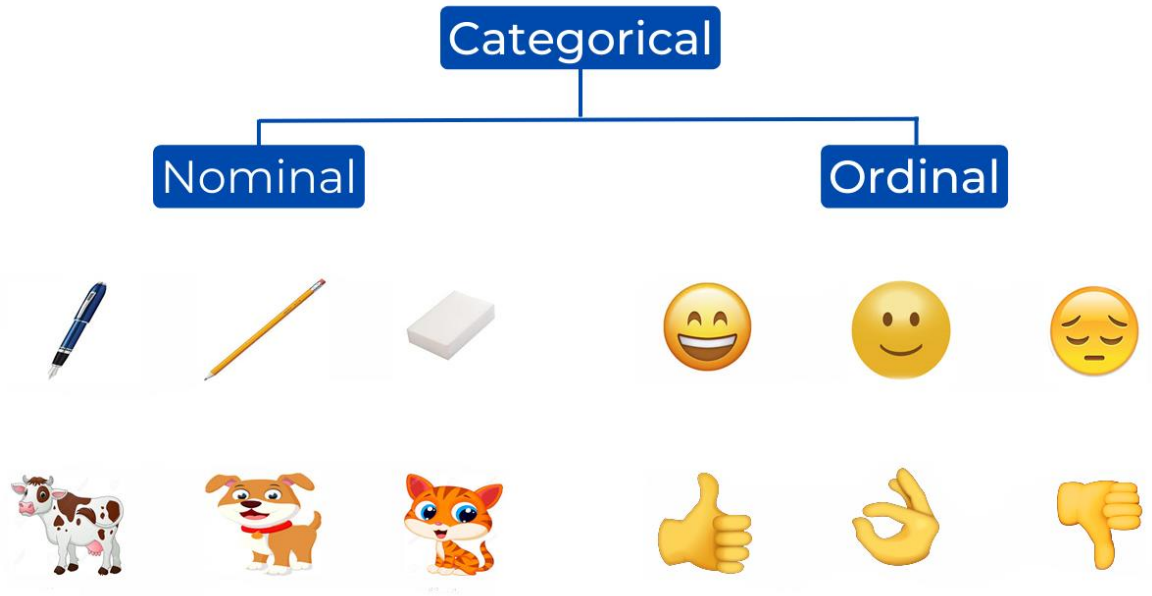
- Box plots → find points beyond whiskers.
- Scatter plots → find points away from clusters.
- Histogram/Distribution plots → identify long tails.

Statistical methods

- Z-score
- IQR (Interquartile Range) → points beyond Q1 - 1.5×IQR or Q3 + 1.5×IQR.

Encoding Categorical Variables

Encoding categorical variables is an essential step in preparing data for machine learning models, which often require numerical input. The two most common methods are **Label Encoding and One-hot Encoding**, and they serve different purposes based on the nature of your data.



Label encoding:

Label encoding is a technique used in machine learning to convert categorical data into numerical data by assigning a unique integer to each category. This allows machine learning algorithms to process and analyze categorical variables that originally contain non-numerical data. Typically, each unique category is assigned a number starting from 0 upwards.

KeyPoints:

- It assigns a distinct numerical label to each category within a variable.
- It is especially useful for ordinal data where categories have a meaningful order.
- For nominal data (categories without order), label encoding can sometimes mislead models into assuming an inherent ranking, which might affect performance.

Categorical data is broadly divided into two types:

- **Nominal Data:** Categories without inherent order (e.g., colors: red, blue, green).

- **Ordinal Data:** Categories with a natural order (e.g., satisfaction levels: low, medium, high).

Sample code

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder

# Severity levels
severity = ['Low', 'Medium', 'High', 'Medium', 'Low']
df = pd.DataFrame({'Severity': severity})

# Label Encoding
label_encoder = LabelEncoder()
df['Severity_encoded'] = label_encoder.fit_transform(df['Severity'])
print(df)
```

Output :

	Severity	Severity_encoded
0	Low	1
1	Medium	2
2	High	0
3	Medium	2
4	Low	1

Other examples of ordinal categorical data that can be label encoded include:

1. Education Levels: “High School” < “Some College” < “Bachelor’s Degree” < “Master’s Degree” < “Ph.D.”

2. Income Levels: “Low” < “Medium” < “High”
3. Rating Scales: “Poor” < “Average” < “Good” < “Excellent”

Now we are able to convert that numerical but algorithms like linear regression may misinterpret numbers as ordinal (Red=2 > Green=1).

For non-ordinal categorical variables, use One-Hot Encoding instead.

One-hot encoding:

One-Hot Encoding is a method of converting categorical variables into a binary vector representation.

- Each unique category in the original column becomes a new column.
- A value of 1 in a column indicates the presence of that category, while 0 indicates its absence.
- This encoding is used to make categorical data machine-readable for algorithms that require numerical input.

Original Column: ['Apple', 'Banana', 'Mango']

After applying One-Hot Encoded:

Apple	**Banana**	**Mango**
1	0	0
0	1	0
0	0	1

Example 1 :

```
fruits = ['Apple', 'Banana', 'Mango', 'Banana', 'Apple', 'Orange']

import pandas as pd


df = pd.DataFrame({'Fruit': fruits})
```

```
# One-Hot Encoding
one_hot = pd.get_dummies(df, columns=['Fruit'])
print(one_hot)
```

Output :

	Fruit_Apple	Fruit_Banana	Fruit_Mango	Fruit_Orange
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	1	0	0
4	1	0	0	0
5	0	0	0	1

Example : 2

```
from sklearn.preprocessing import OneHotEncoder
import pandas as pd

df = pd.DataFrame({'Fruit': ['Apple', 'Banana', 'Mango']})

encoder = OneHotEncoder(sparse=False)    # sparse=False returns a NumPy array
encoded_array = encoder.fit_transform(df[['Fruit']])

encoded_df = pd.DataFrame(encoded_array, columns=encoder.get_feature_names_out(['Fruit']))
print(encoded_df)
```

	Fruit_Apple	Fruit_Banana	Fruit_Mango
0	1.0	0.0	0.0
1	0.0	1.0	0.0
2	0.0	0.0	1.0

Data Transformation

Data Transformation is the process of converting raw data into a more useful format for analysis or machine learning.

It helps in:

- Improving data **quality and consistency**
- Preparing data for **ML models**
- Handling **different scales, distributions, or data types**

Normalization

Normalization is a technique in Machine Learning applied during data preparation to change the values of numeric columns in the dataset to use a common scale between 0 to 1 . It is not necessary for all datasets in a model. It is required only when features of machine learning models have different ranges.

Mathematically, we can calculate normalization with the below formula:

1. Xn = (X - Xminimum) / (Xmaximum - Xminimum)

Example: Let's assume we have a model dataset having maximum and minimum values of feature. To normalize the machine learning model, values are shifted and rescaled so their range can vary between 0 and 1. This technique is also known as **Min-Max scaling**. In this scaling technique, we will change the feature values as follows:

Values: [10, 20, 30, 40, 50] - Min = 10, Max = 50

For 30: [x' = 0.5]

Normalized values: [0, 0.25, 0.5, 0.75, 1]

```
import numpy as np
from sklearn.preprocessing import MinMaxScaler
# Sample data
X = np.array([[10], [20], [30], [40], [50]])
# Apply Min-Max Normalization to [0,1]
scaler = MinMaxScaler()
X_normalized = scaler.fit_transform(X)
print("Original Data:\n", X.ravel())
print("Normalized Data (0-1 range):\n", X_normalized)
```

Output

```
Original Data:
[10 20 30 40 50]
Normalized Data (0-1 range):
[0.   0.25 0.5  0.75 1.  ]
```

Binning

Binning is the process of dividing a continuous variable into a set of discrete intervals or bins. The intervals can be of equal or unequal size, and can be defined using different methods, such as

- **Fixed Width Binning:** Dividing the data into a fixed number of equally sized bins. For example, dividing a range of values from 0 to 100 into 10 bins of width 10.
- **Fixed Frequency Binning:** Dividing the data into a fixed number of bins with approximately the same number of data points in each bin. For example, dividing a dataset of 1000 data points into 10 bins with 100 data points in each bin.
- **Custom binning :** Bins are defined manually based on domain knowledge.

Example: Age groups = Child (0-12), Teen (13-19), Adult (20-59), Senior (60+).

Example with Age Data

ages = [5, 12, 18, 24, 30, 35, 40, 50, 60, 70, 80, 90]

Equal-width Binning (3 bins)

We want **3 bins**.

- Range of data = 90 − 5 = 85
- Bin width = 85 ÷ 3 ≈ 28.3

So bins are approximately:

- **Bin 1**: 5–33
- **Bin 2**: 33–61
- **Bin 3**: 61–90

Ages grouped:

- Bin 1 → [5, 12, 18, 24, 30]
- Bin 2 → [35, 40, 50, 60]
- Bin 3 → [70, 80, 90]

Equal-frequency Binning (3 bins)

We want **3 bins with equal number of people**.

- 12 people ÷ 3 bins = **4 people per bin**

So bins are:

- **Bin 1**: [5, 12, 18, 24]
- **Bin 2**: [30, 35, 40, 50]
- **Bin 3**: [60, 70, 80, 90]

Notice here:

- Ranges are uneven (Bin 1: 5–24, Bin 2: 30–50, Bin 3: 60–90)
- But **each bin has exactly 4 people**

Python Example :

```
import pandas as pd
```

```
ages = [5, 12, 18, 24, 30, 35, 40, 50, 60, 70, 80, 90]
df = pd.DataFrame(ages, columns=['Age'])

# Equal-width binning (3 bins)
df['Equal_Width_Bin'] = pd.cut(df['Age'], bins=3)

# Equal-frequency binning (3 bins)
df['Equal_Freq_Bin'] = pd.qcut(df['Age'], q=3)

print(df)
```

	Age	Equal_Width_Bin	Equal_Freq_Bin
0	5	(4.915, 33.3]	(4.999, 24.0]
1	12	(4.915, 33.3]	(4.999, 24.0]
2	18	(4.915, 33.3]	(4.999, 24.0]
3	24	(4.915, 33.3]	(4.999, 24.0]
4	30	(4.915, 33.3]	(29.999, 50.0]
5	35	(33.3, 61.7]	(29.999, 50.0]
6	40	(33.3, 61.7]	(29.999, 50.0]
7	50	(33.3, 61.7]	(29.999, 50.0]
8	60	(33.3, 61.7]	(59.999, 90.0]
9	70	(61.7, 90.0]	(59.999, 90.0]
10	80	(61.7, 90.0]	(59.999, 90.0]
11	90	(61.7, 90.0]	(59.999, 90.0]

Equal-width → fixed-size ranges, but bins may have unequal counts.

Equal-frequency → bins have equal counts, but ranges vary.

Example for Custom Binning

```
import pandas as pd

ages = [5, 12, 18, 24, 30]
```

```
df = pd.DataFrame(ages, columns=['Age'])

bins = [0, 12, 19, 35]      # intervals: (0-12], (12-19], (19-35]
labels = ['Child', 'Teen', 'Young Adult']

df['Age_Group'] = pd.cut(df['Age'], bins=bins, labels=labels)
print(df)
```

Output

	Age	Age_Group	
0	5	Child	→ falls into (0–12]
1	12	Child	→ falls into (0–12]
2	18	Teen	→ falls into (12–19]
3	24	Young Adult	→ falls into (19–35]
4	30	Young Adult	→ falls into (19–35]

Data Types Conversion

Data Type Conversion is the process of changing one data type into another, so the data can be used correctly in analysis

Example: "25" (string) → 25 (integer)

Example: 25 (integer) → 25.0 (float)

Why Data Type Conversion is Needed

Memory Efficiency

- Large datasets can be optimized by converting types.
- Example: Converting float64 → float32 reduces memory usage.

Data Cleaning

- Data loaded from CSV/Excel might come as object (string) even if values are numeric.
- Conversion ensures proper type before analysis.

Correct Calculations

- Wrong type leads to wrong results:
  - "25" + "5" → "255" (string concatenation)
  - 25 + 5 → 30 (integer addition)

Types of Data Type Conversion

Numeric Conversions

- Integer ↔ Float
- Example: int(25.9) → 25
- Example: float(25) → 25.0

String ↔ Numeric

- Example: "100" → 100 (string → integer)
- Example: 100 → "100" (integer → string)

Object ↔ Category

- Example: "Male"/"Female" → 0/1 (category encoding)

```
import pandas as pd

# Sample dataset
data = {'Age': ['25', '30', '35'],
        'Salary': ['40000', '50000', '60000'],
        'Gender': ['Male', 'Female', 'Male']}
df = pd.DataFrame(data)
print("Before Conversion:")
print(df.dtypes)
# Convert Age and Salary from string → integer
df['Age'] = df['Age'].astype(int)
df['Salary'] = df['Salary'].astype(float)
```

```
# Convert Gender to category
df['Gender'] = df['Gender'].astype('category')
print("\nAfter Conversion:")
print(df.dtypes)
```

Data Type Casting

Data Type Casting is the process of forcibly changing one data type into another, even if it may lead to data loss or truncation

Why Casting is Important in AI/ML?

1. **Memory Optimization**
   - Large datasets → reduce precision from float64 → float32 to save memory.
2. **Speed**
   - Smaller types (int32 instead of int64) improve training performance.
3. **Compatibility**
   - Some ML models or libraries expect specific types (float32 for TensorFlow/PyTorch).
4. **Data Cleaning**
   - When values are not in the desired type, casting ensures they become usable.

Example :

1.
```
x = 10
y = float(x)
print(y, type(y))
```
Output: 10.0 <class 'float'

2.
```
x = 10.75
y = int(x)      # Casting removes decimal part
print(y, type(y))
```
Output: 10 <class 'int'> (lost .75

3.
```
x = "25"
y = int(x)
print(y, type(y))
```

UNIT – 3

Measures of Central Tendency and Dispersion

---

1. Measures of Central Tendency

Central tendency is a fundamental statistical concept that represents the "center" or typical value around which data points in a dataset cluster. It helps summarize a dataset with a single representative value, giving an idea about the average or most common outcome.

The three main measures of central tendency

1.1 Mean (Arithmetic Average)

Definition: The sum of all values divided by the number of values.
Formula: Mean = Σx / n

Example: Predicting house prices using features like average square footage in a neighborhood.

In datasets, mean values summarize typical feature values, helping to fill missing data (mean imputation) or standardize scales before feeding into ML models.

Use case: If 10% of house records lack square footage, replacing missing values with the mean square footage avoids data loss and improves model stability.

1.2 Median (Middle Value)

Definition: The middle value when data is arranged in order.

Example:

- Income prediction models often handle skewed income data with heavy outliers (few very high earners).
- Using the median instead of mean to impute missing income values ensures the imputation is not skewed by extreme values.

- **Use case:** This preserves model accuracy by maintaining a representative “typical” income value during training.

.

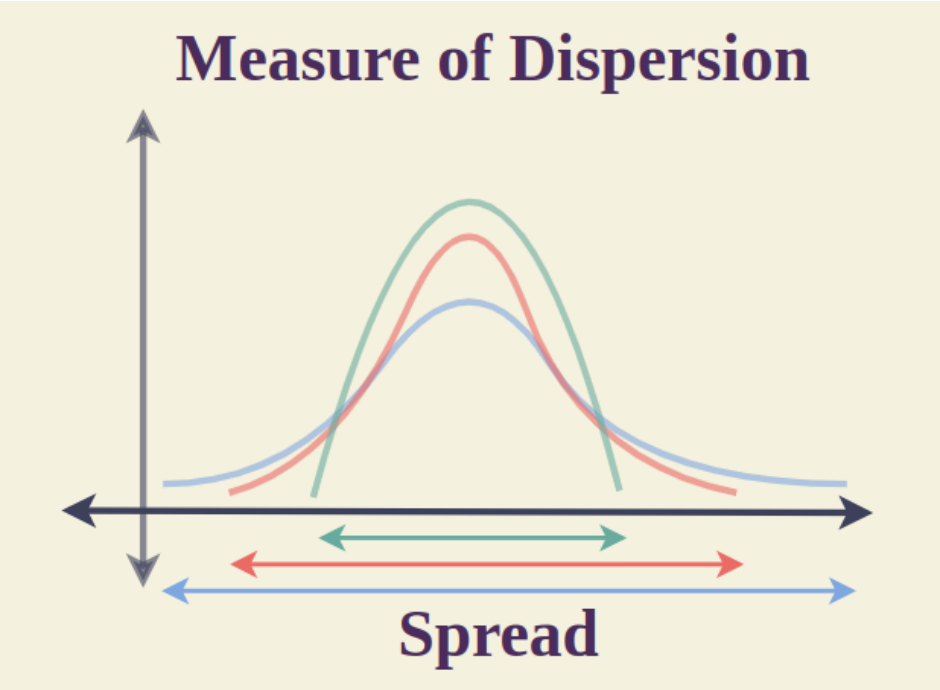
1.3 Mode (Most Frequent Value)

Definition: The value that occurs most frequently.

- Example: In customer churn prediction, categorical features like “preferred communication channel” (email, phone, app) may have missing values.
- Imputing missing categories with the mode (most frequent category) helps keep dataset integrity.
- **Use case:** Mode imputation avoids introducing rare categories and preserves dominant customer behavior patterns in ML models.

2. Measures of Dispersion

Dispersion in statistics is a way to describe how spread out or scattered the data is around an average value. It helps to understand if the data points are close together or far apart. It shows the variability or consistency in a set of data.



Common Measures of Dispersion

2.1 Range

Definition: Difference between maximum and minimum values.
Formula: Range = Max – Min

Example: Given {20, 42, 13, 71, 54, 93, 15, 16}

- Largest value = 93
- Smallest value = 13
- **Range = 93 - 13 = 80**

2.2 Variance

Definition: Average of squared differences from the mean.
Formula: Variance = Σ(x - mean)^2 / n

Example: Test scores: 60, 65, 70, 75, 80

- **Mean ($\bar{X}$) = (2+6+12+15)/4 = 8.75**

  Variance = ((60−70)2+(65−70)2+(70−70)2+(75−70)2+(80−70)2)÷5

  = (100 + 25 + 0 + 25 + 100) ÷ 5 = 50

2.3 Standard Deviation (SD)

Definition: Square root of variance; shows average deviation from mean.
Formula: SD = √Variance

Example: Dataset = {1, 3, 6, 7, 12}

- Mean = 5.8
- Find squared differences: (1-5.8)^2 = 23.04, (3-5.8)^2 = 7.84, (6-5.8)^2 = 0.04, (7-5.8)^2 = 1.44, (12-5.8)^2 = 38.44
  Sum = 70.8
  Variance = 70.8 / 5 = 14.16
  Standard Deviation = 14.16≈3.7614.16≈3.76

AI/ML Use Cases:
- Used in feature scaling (z-score normalization)
- Important for Gaussian distribution assumptions in ML models.

Real-World Example: In height measurement, if average height = 170 cm with SD = 10 cm, then most people are between 160–180 cm.

2.4 Interquartile Range (IQR)

Definition: Spread of middle 50% of data.
Formula: IQR = Q3 - Q1
where,

- Q1 = First quartile (25th percentile), the median of the lower half of the data,
- Q3 = Third quartile (75th percentile), the median of the upper half of the data.

Here are real-world scenarios where measures of dispersion (range, standard deviation, MAD, IQR) are used in AI/ML, analytics, and everyday contexts, each with concrete examples that can be brought into the classroom:

1. Sales Revenue Variability

Scenario:
Two retail stores each report an average monthly revenue of ₹1,00,000.

- Store A: Monthly figures range from ₹98,000 to ₹1,02,000 (SD = ₹1,500).
- Store B: Figures fluctuate from ₹50,000 to ₹1,50,000 (SD = ₹24,000).

Application:
Although means are identical, measures of dispersion reveal that Store B's income is far less consistent, indicating higher financial risk—critical for planning and forecasting.

2. Stock Price Volatility

Scenario:
Suppose two stocks average the same closing price (₹500) over a month:

- Stock X: Prices are ₹499, ₹501, ₹500, ₹502, ₹498 (low SD).
- Stock Y: Prices are ₹400, ₹600, ₹460, ₹540, ₹500 (high SD).

Application:
Investors use standard deviation to judge investment risk—lower SD suggests more stable stocks, guiding portfolio decisions.

3. Weather Data Monitoring

Scenario:

- City 1: Daily temperatures over a week are 30°C, 31°C, 32°C, 30°C, 29°C, 30°C, 31°C (low SD: weather is consistent).
- City 2: Temperatures are 20°C, 35°C, 31°C, 42°C, 24°C, 28°C, 15°C (high SD: weather is unpredictable).

Application:
For event planning or agriculture, knowing the dispersion helps in risk assessment and strategy planning—greater dispersion means preparation for more variability.

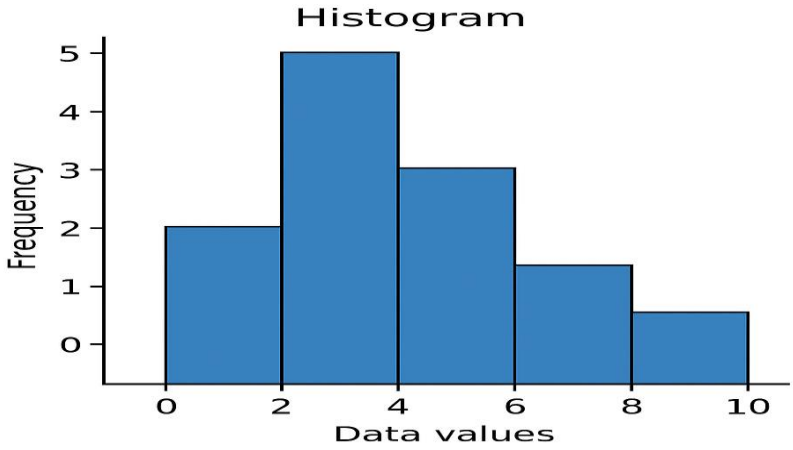
Takeaway

Concept	**What It Shows**	**Common Measures**	**Use Cases**
Central Tendency	Typical or average value in data	Mean, Median, Mode	Data summary, baseline models, imputation
Dispersion	Variability or spread of data	Range, SD, IQR, Variance	Outlier detection, scaling, risk analysis

Histogram

A histogram is a type of graphical representation used in statistics to show the distribution of numerical data. It looks somewhat like a bar chart, but unlike bar graphs, which are used for categorical data, histograms are designed for continuous data, grouping it into logical ranges, which are also known as "bins."

A histogram helps in visualizing the distribution of data across a continuous interval or period which makes the data more understandable and also highlights the trends and patterns

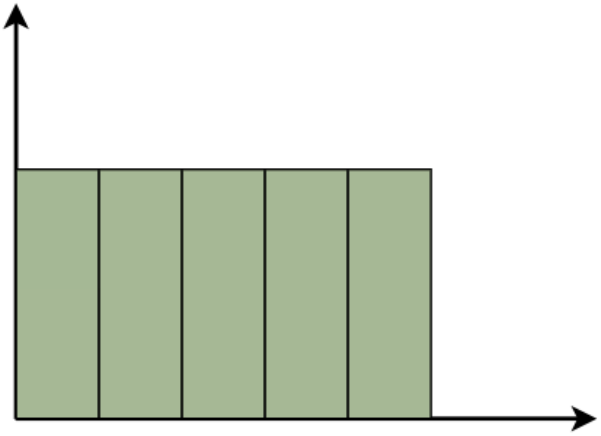


Types of Histogram

There are various variations of the histograms based on their shapes:

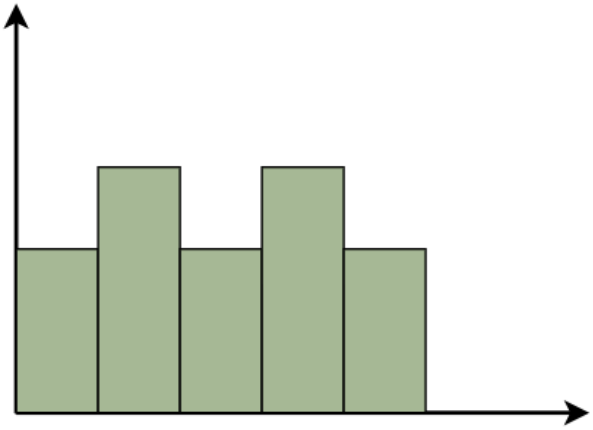
Uniform Histogram

A Uniform Histogram shows uniform distribution means that the data is uniformly distributed among the classes, with each having a same number of elements. It may display many peaks, suggesting varying degrees of incidence.



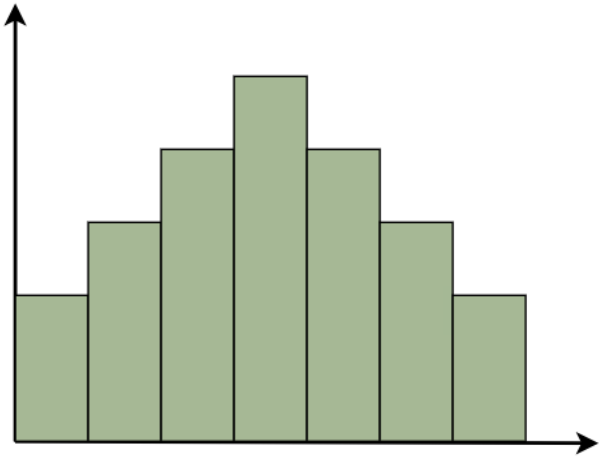
Bimodal Histogram

A histogram is called bimodal if it has two distinct peaks. This implies that the data consists of observations from two distinct groups or categories, with notable variations between them.



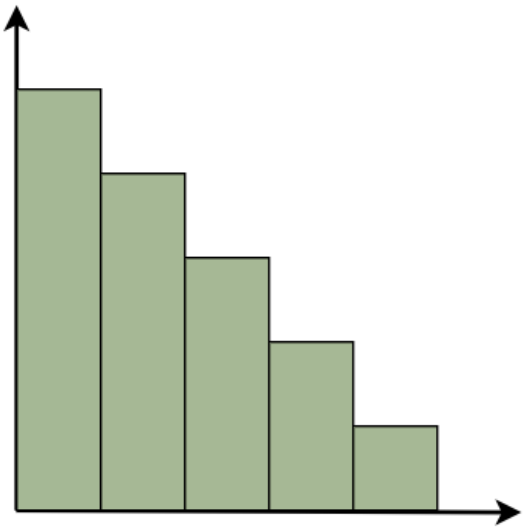
Symmetric Histogram

Symmetric Histogram is also known as a bell-shaped histogram, it has perfect symmetry when divided vertically down the centre, with both sides matching each other in size and shape. The balance reflects a steady distribution pattern.



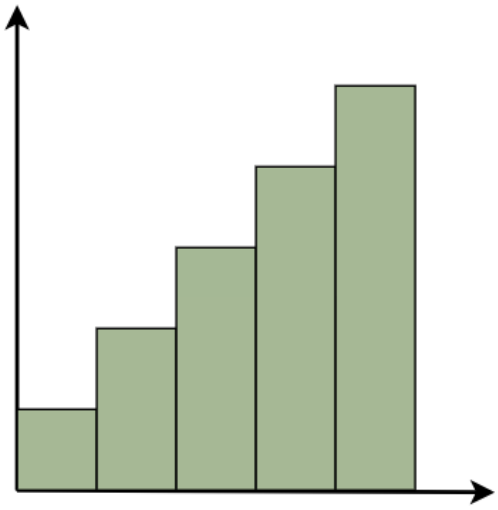
Right-Skewed Histogram

A right-skewed histogram shows bars leaning towards the right side. This signifies that the majority of the data points are on the left side, with a few outliers reaching to the right. Consider a histogram showing the distribution of family earnings. A right-skewed histogram occurs when the majority of families are in lower income groups, but a small number of highly rich households skew the average income.



Left-Skewed Histogram

A left-skewed histogram shows bars that lean towards the left side. This means that the majority of the data points are on the right side, with a few exceptionally low values extending to the left. Consider a histogram reflecting the distribution of test scores in a classroom. A left-skewed histogram occurs when the majority of students receive excellent grades but a few do badly, resulting in an average that is dragged to the left.



Sudo Code

```
import matplotlib.pyplot as plt
```

```
import numpy as np

scores = np.random.normal(loc=70, scale=10, size=130)   # mean=70, std=10
# Create Histogram
plt.figure(figsize=(10,10))
plt.hist(scores, bins=10, color="skyblue", edgecolor="black")

# Labels and Title
plt.title("Histogram of Student Exam Scores")
plt.xlabel("Score Ranges")
plt.ylabel("Number of Students")

# Show grid
plt.grid(axis='y', linestyle='--', alpha=0.7)

plt.show()
```

```
scores = np.random.normal(loc=70, scale=10, size=130)   # mean=70, std=10
```

loc=70

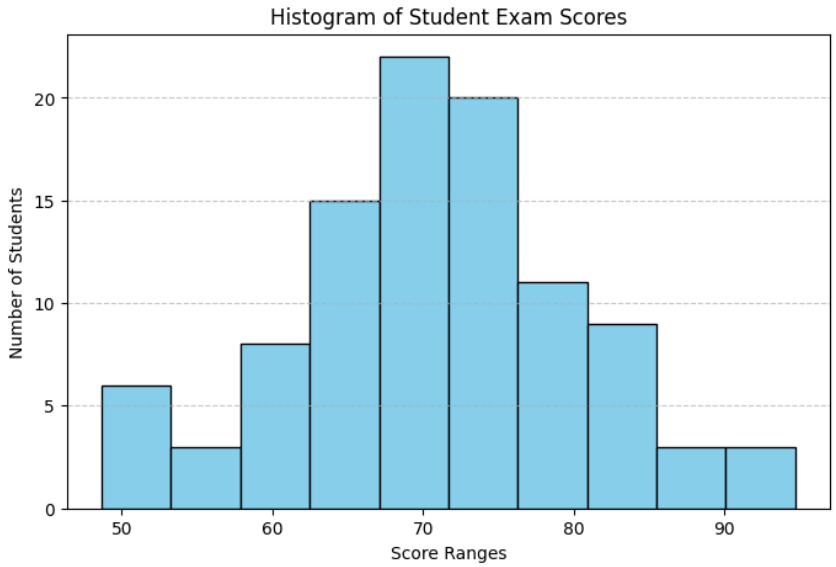
- This is the **mean (center)** of the distribution.
- The generated values will be centered around **70**.
- Example: Think of it as the "average score" in an exam.

scale=10

- This is the **standard deviation (spread)** of the distribution.
- A higher scale means values will spread out more from the mean.
- Here, most values will fall within **70 ± 10 → between 60 and 80**.

size=130

- This is the **number of random samples** to generate.
- It will create an array with **130 numbers** following that distribution.



Visualization Techniques in Python: Bar Charts, Count Plots, and Pie Charts

1. Bar Charts

🔹 **Concept**

- A **bar chart** uses rectangular bars to represent **numerical values** for different **categories**.
- The **length/height** of the bar corresponds to the value.
- Useful when comparing **discrete categories** (e.g., sales by product, students by grade).

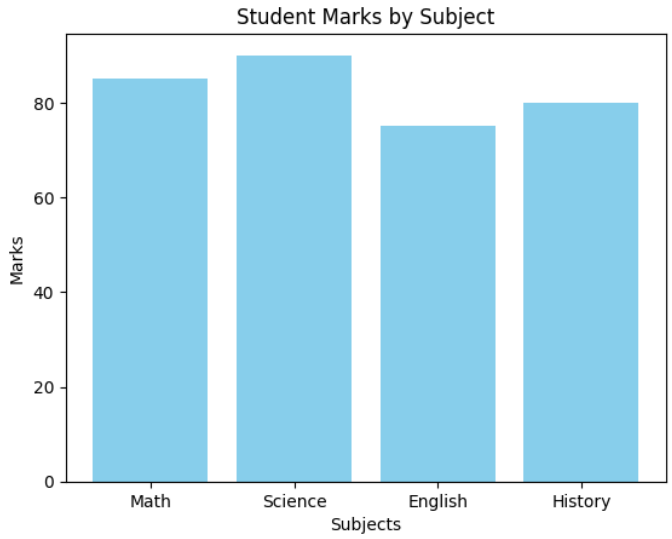
🔹 **Example Use Case**

- Showing the **average marks of students** in different subjects.

🔹 **Python Pseudo Code**

```
import matplotlib.pyplot as plt

subjects = ["Math", "Science", "English", "History"]

marks = [85, 90, 75, 80]
```

```
plt.bar(subjects, marks, color='skyblue')
plt.title("Student Marks by Subject")
plt.xlabel("Subjects")
plt.ylabel("Marks")
plt.show()
```



---

2. Count Plots

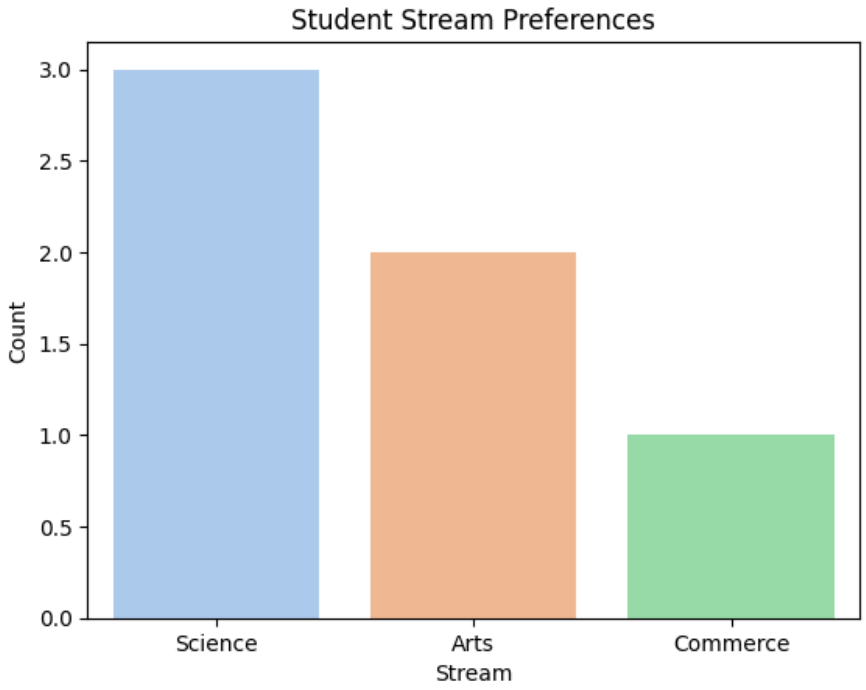
- **Concept**
  - A **count plot** (Seaborn) is a special kind of bar chart.
  - Instead of giving values manually, it **counts the number of occurrences** of each category in a dataset.
  - Useful for **categorical data frequency visualization**.
- **Example Use Case**
  - Counting how many students chose **Science stream vs Arts vs Commerce**.
- **Python Pseudo Code**

```
import seaborn as sns
import matplotlib.pyplot as plt
stream_choices = ["Science", "Arts", "Commerce", "Science", "Arts", "Science"]
sns.countplot(x=stream_choices, palette="pastel")
plt.title("Student Stream Preferences")
```

plt.xlabel("Stream")

plt.ylabel("Count")

plt.show()



---

3. Pie Charts

- **Concept**
  - A **pie chart** shows proportions of categories as slices of a circle.
  - Good for showing **percentage distribution**.
  - Should be used when you want to highlight **parts of a whole**.

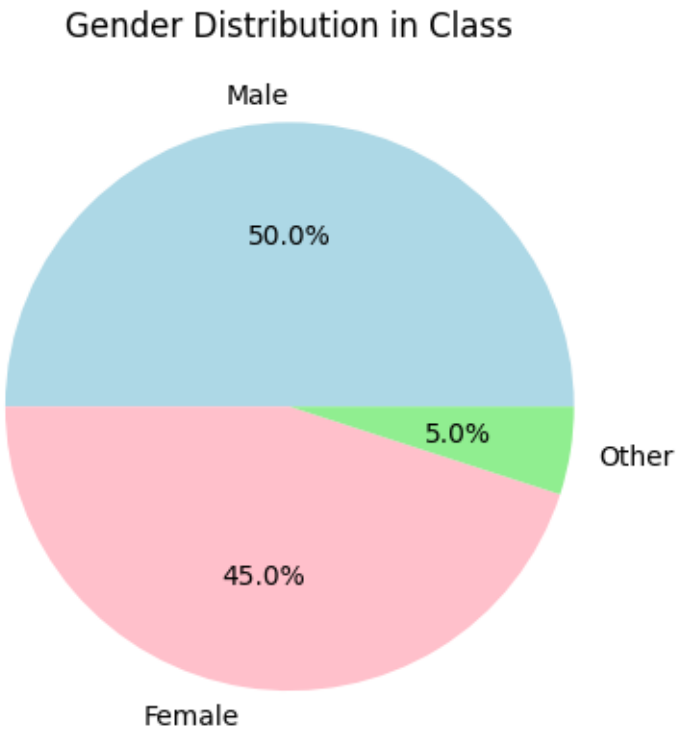
Note: Avoid too many categories (>6), otherwise the chart becomes confusing.

- **Example Use Case**
  - Distribution of students by **gender** in a class.
- **Python Pseudo Code**

import matplotlib.pyplot as plt

Categories and values

```
genders = ["Male", "Female", "Other"]
counts = [50, 45, 5]
plt.pie(counts, labels=genders, autopct='%1.1f%%', colors=["lightblue", "pink", "lightgreen"])
plt.title("Gender Distribution in Class")
plt.show()
```



When to use Histograms, Boxplots, KDE, Bar Charts, Count Plots, Pie Charts?

1. Histogram

A **histogram** is used to display the **distribution of a continuous numerical variable**. The data is grouped into intervals (called bins), and the height of each bar shows how many values fall into that interval.

- **When to use**: When you want to see how data is spread across different ranges.
- **Example**: If you plot exam scores of 100 students, a histogram will show how many students scored between 0–10, 10–20, 20–30, and so on.
- **Good for**: Detecting skewness, spread, and common ranges.

---

2. Boxplot

A **boxplot** shows the **summary of a dataset**: median, quartiles (25%, 50%, 75%), and outliers. The "box" represents the middle 50% of the data, while the "whiskers" show the spread of the rest. Outliers are shown as dots.

- **When to use**: When comparing distributions across groups or when you want to check for outliers.
- **Example**: Comparing salaries of employees in different departments. A boxplot will quickly show which department has higher salaries and where outliers exist.
- **Good for**: Spotting outliers, comparing groups.

---

3. KDE (Kernel Density Estimate Plot)

A **KDE plot** is like a smooth version of a histogram. Instead of showing bins, it creates a continuous curve that represents the probability distribution of the data.

- **When to use**: When you want to visualize the shape of the distribution (whether it looks normal, skewed, or has multiple peaks).
- **Example**: Plotting the heights of students in a class. A KDE curve will show if most students are around a certain height and whether the distribution is normal or skewed.
- **Good for**: Understanding the shape of data distribution.

---

4. Bar Chart

A **bar chart** is used to compare **numerical values across categories**. Each category has a bar, and the length/height of the bar represents its value.

- **When to use**: When comparing averages or totals across different groups.
- **Example**: Showing the **average marks in different subjects** (Math, Science, English, History). A bar chart makes it clear which subject has the highest or lowest marks.
- **Good for**: Comparing categories side by side.

---

5. Count Plot

A **count plot** is a special type of bar chart used when the data is **categorical**. Instead of providing numerical values, it automatically counts how many times each category occurs in the dataset.

- **When to use**: When you want to see the frequency of categories in your data.

- **Example**: Counting how many students chose **Science, Arts, or Commerce** as their stream. The plot will show which stream is most popular.
- **Good for**: Frequency analysis of categorical variables.

---

6. Pie Chart

A **pie chart** shows data as slices of a circle, where each slice represents the proportion of a category relative to the whole.

- **When to use**: When you want to show **percentage share** of categories.
- **Example**: Showing the **gender distribution** in a classroom (Male, Female, Other). Each slice shows what percentage of the class belongs to that category.
- **Good for**: Displaying proportions or parts of a whole.
- **Note**: Works best with 3–6 categories; too many slices make it confusing.

Box Plot

A **Box Plot (**or **Whisker plot)** display the summary of a data set, including minimum, first quartile, median, third quartile and maximum. it consists of a box from the first quartile to the third quartile, with a vertical line at the median. the x-axis denotes the data to be plotted while the y-axis shows the frequency distribution. The matplotlib.pyplot module of matplotlib library provides boxplot() function with the help of which we can create box plots.

Syntax

matplotlib.pyplot.boxplot(data)

Example:

```
import matplotlib.pyplot as plt
import numpy as np
np.random.seed(10)
d = np.random.normal(100, 20, 200)
```

// generates 200 random values from a normal distribution with mean = 100 and standard deviation = 20.

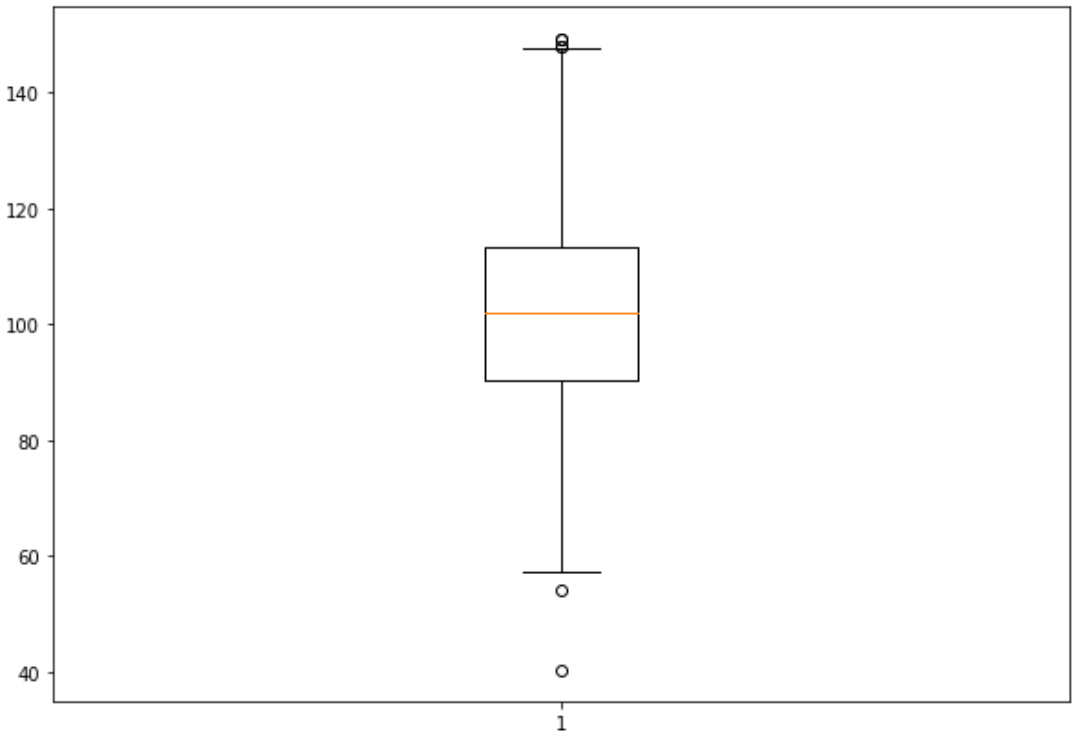
//Most values will lie within ±20 of the mean

// ~68% of values between **80 and 120** (μ ± 1σ)

// ~95% of values between **60 and 140** (μ ± 2σ)

//~99.7% of values between **40 and 160** (μ ± 3σ)

```
fig = plt.figure(figsize =(10, 7))
plt.boxplot(d)
plt.show()
```

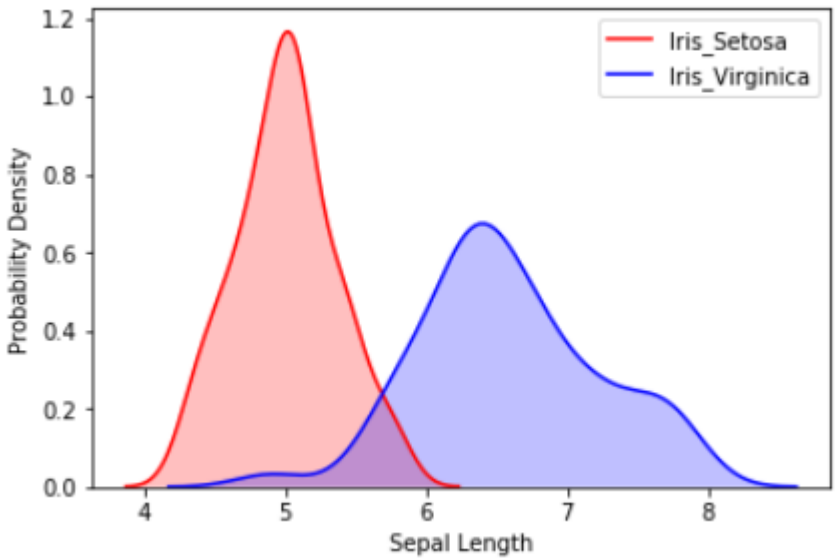


KDE Plot

What is KDE Plot?

KDE Plot described as Kernel Density Estimate is used for visualizing the Probability Density of a continuous variable. It depicts the probability density at different values in a continuous variable. We can also plot a single graph for multiple samples which helps in more efficient data visualization. It provides a smoothed representation of the underlying distribution of a dataset.

The KDE plot visually represents the distribution of data, providing insights into its shape, central tendency, and spread. It is particularly useful when dealing with continuous data or when you want to explore the distribution without making assumptions about a specific parametric form (e.g., assuming the data follows a normal distribution). KDE plots are commonly used in statistical software packages and libraries for data visualization, such as Seaborn and Matplotlib in Python.



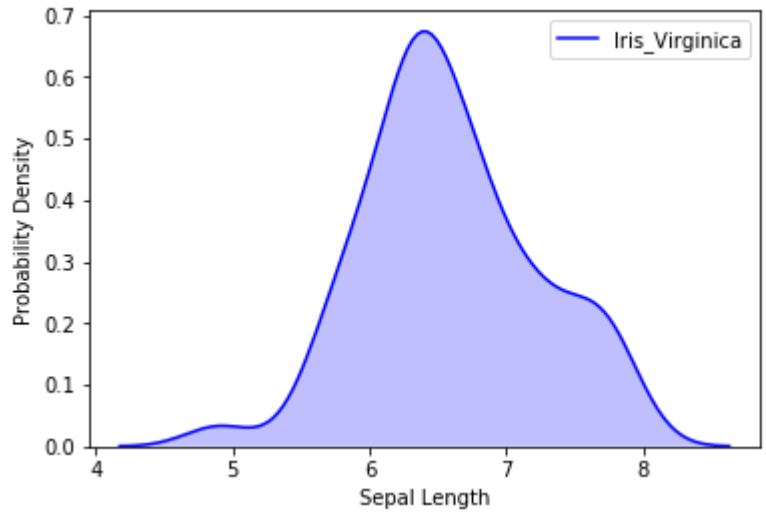
Creating a Univariate Seaborn KDE Plot

To start our exploration, we delve into the creation of a Univariate Seaborn KDE plot, visualizing the probability distribution of a single continuous attribute.

We can visualize the probability distribution of a sample against a single continuous attribute.

```
from sklearn import datasets
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Setting up the Data Frame
iris = datasets.load_iris()
iris_df = pd.DataFrame(iris.data, columns=['Sepal_Length',
                                           'Sepal_Width', 'Petal_Length', 'Petal_Width'])
iris_df['Target'] = iris.target
iris_df['Target'].replace([0], 'Iris_Setosa', inplace=True)
iris_df['Target'].replace([1], 'Iris_Vercicolor', inplace=True)
iris_df['Target'].replace([2], 'Iris_Virginica', inplace=True)

# Plotting the KDE Plot
sns.kdeplot(iris_df.loc[(iris_df['Target']=='Iris_Virginica'),
                        'Sepal_Length'], color='b', shade=True, label='Iris_Virginica')

# Setting the X and Y Label
plt.xlabel('Sepal Length')
plt.ylabel('Probability Density')
```

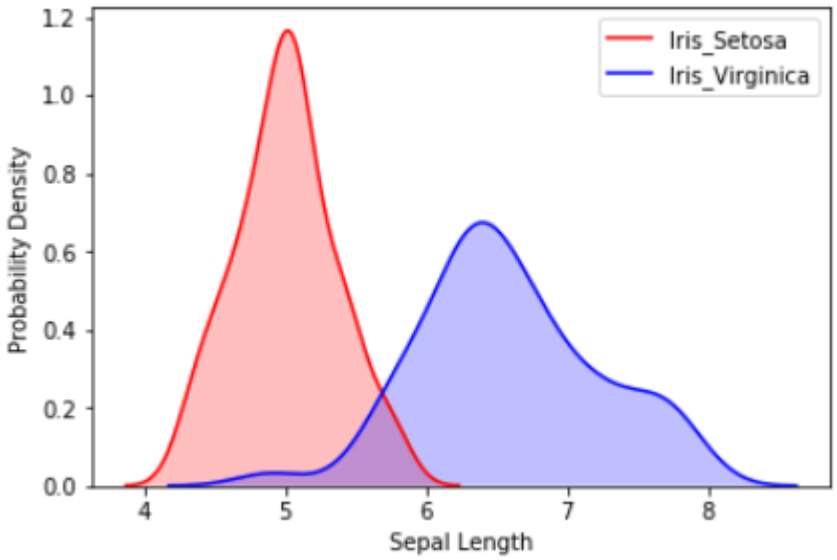
Output:



We can also visualize the probability distribution of multiple samples in a single plot.

```
sns.kdeplot(iris_df.loc[(iris_df['Target']=='Iris_Setosa'),
                        'Sepal_Length'], color='r', shade=True, label='Iris_Setosa')


sns.kdeplot(iris_df.loc[(iris_df['Target']=='Iris_Virginica'),
                        'Sepal_Length'], color='b', shade=True, label='Iris_Virginica')


plt.xlabel('Sepal Length')

plt.ylabel('Probability Density')
```

Output:



Bivariate Analysis

Bivariate Analysis is the statistical technique used to study the **relationship between two variables** in a dataset. It helps us understand whether the variables are related, how strongly they are related, and in what direction.

- Understanding dependencies between features and target.
- Detecting patterns, trends, and correlations.
- Deciding which features are important for predictive modeling.

---

1. Scatter Plots

🔹 **Definition**

A scatter plot is a type of data visualization that displays values for two numerical variables as points on a two-dimensional graph. Each point represents an observation in the dataset, with:

- The x-axis showing one variable
- The y-axis showing another variable

This helps to visually identify patterns, relationships, correlations, clusters, and outliers between the two variables.

🔹 **When to Use**

- To check if two variables are correlated (positive, negative, or no correlation).

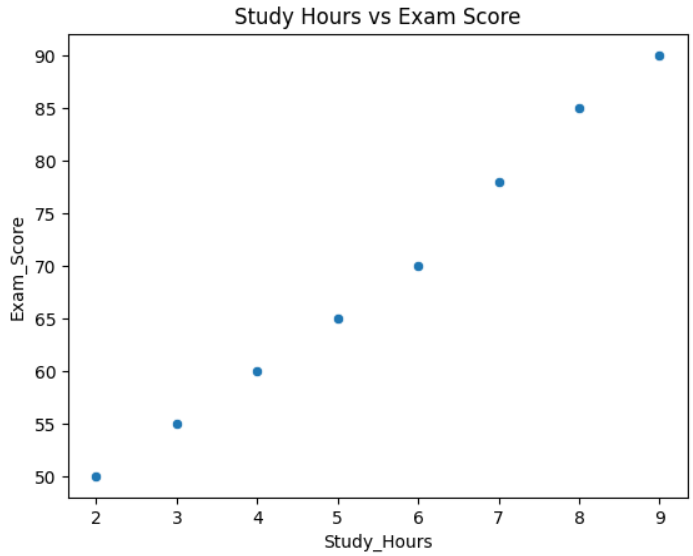
- To observe **linear or non-linear relationships**.
- When you need to check for unusual data points that don't follow the general pattern

🔹 **Example**

- **Study Hours vs Exam Score** → More study hours → Higher exam scores.

🔹 **Python Pseudo Code**

```
import seaborn as sns

import matplotlib.pyplot as plt

import pandas as pd


# Sample data

data = pd.DataFrame({

    "Study_Hours": [2, 3, 4, 5, 6, 7, 8, 9],

    "Exam_Score": [50, 55, 60, 65, 70, 78, 85, 90]

})


# Scatter Plot

sns.scatterplot(x="Study_Hours", y="Exam_Score", data=data)

plt.title("Study Hours vs Exam Score")

plt.show()
```



---

2. Pair Plots

- **Definition**

A **pair plot** is a data visualization that shows the **pairwise relationships** between multiple numerical variables in a dataset. It creates a **grid of scatter plots** for every combination of variables, along with **histograms (or KDE plots)** on the diagonal to show the distribution of each variable..

- **When to Use**
  - To analyze **all pairwise relationships** at once.
  - To identify **clusters, trends, or separations** in data.
  - To explore data with multiple features before modeling.
- **Example**
  - **Iris dataset** → See how sepal and petal measurements are related for different species.
- **Python Pseudo Code**

from sklearn.datasets import load_iris

Load dataset

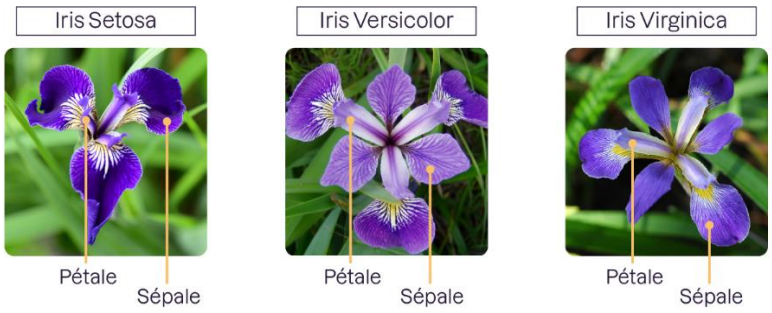
iris = load_iris()

Loads the **Iris dataset** from scikit-learn.

This dataset is widely used in ML tutorials.

It contains **150 flower samples** from 3 species:

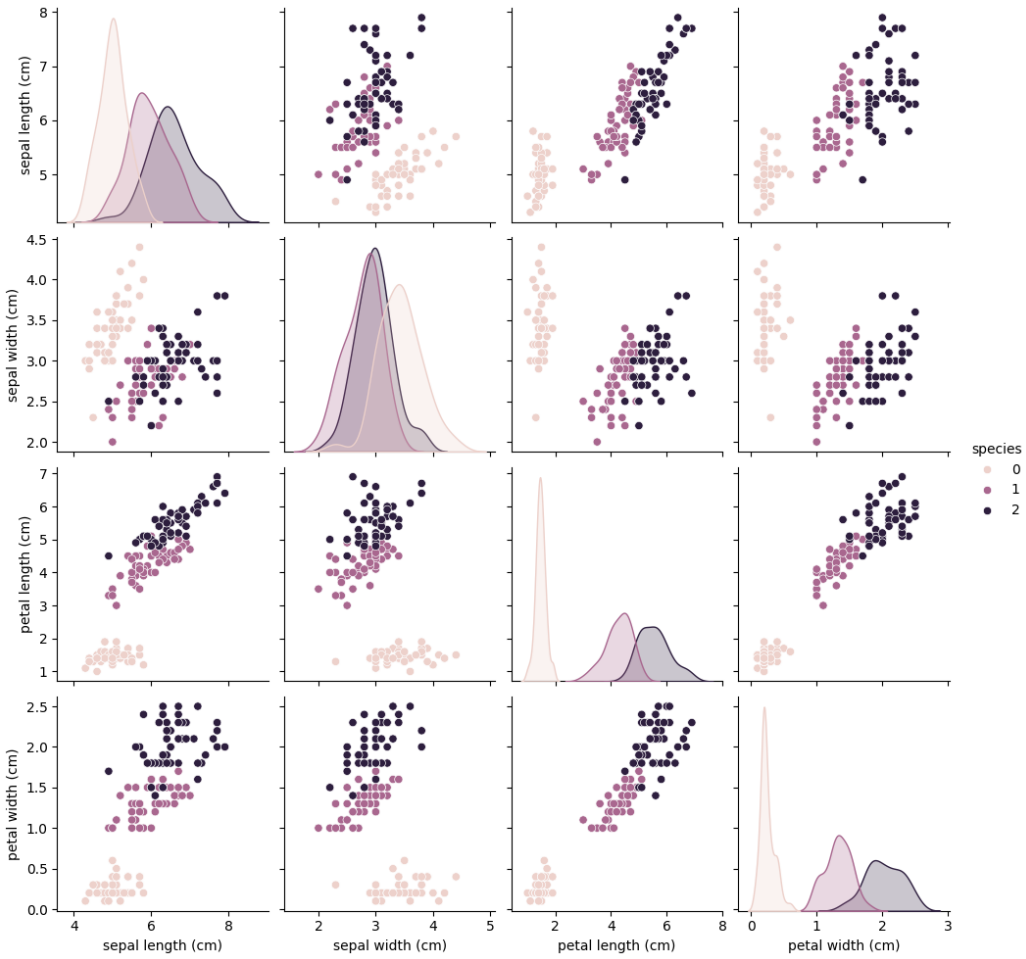
- Setosa
- Versicolor
- Virginica



Each sample has **4 features**:

- Sepal length
- Sepal width
- Petal length
- Petal width

```
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df["species"] = iris.target


# Pair Plot
sns.pairplot(df, hue="species")    # hue adds color by class
plt.show()
```



---

3. Heatmaps

🔹 Definition

A **heat map** is a data visualization technique that represents values in a matrix or table using **colour gradients**. The intensity of the colour shows the magnitude of the value, making it easy to spot patterns, correlations, and outliers at a glance, it is often used to visualize **correlation values** between features.

🔹 When to Use

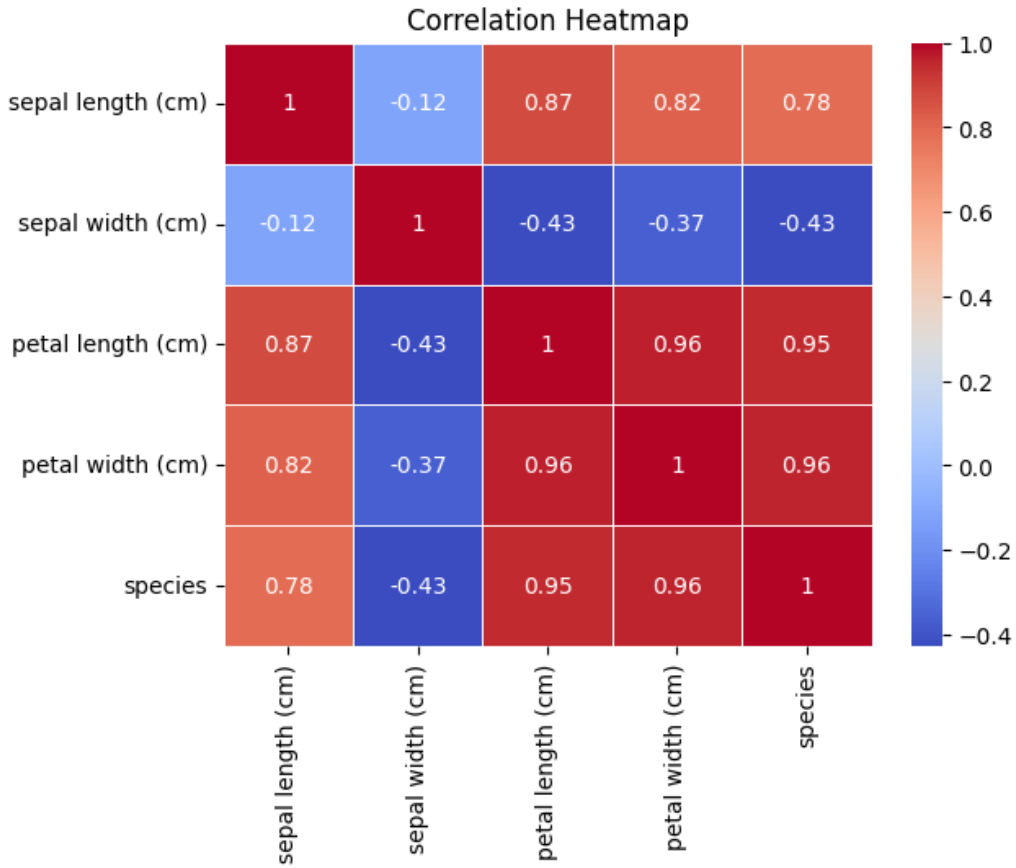
- To see correlation patterns among variables.
- To quickly detect which features are strongly related to the target.
- To identify multicollinearity (when two features are highly correlated).

🔹 Example

- Correlation matrix of house features (size, price, number of rooms).

🔹 Python Pseudo Code

```
# Correlation matrix
corr = df.corr()

# Heatmap
sns.heatmap(corr, annot=True, cmap="coolwarm", linewidths=0.5)
plt.title("Correlation Heatmap")
plt.show()
```



---

4. Correlation Analysis

- **Definition**

Correlation Analysis is a statistical method used to measure the strength and direction of the relationship between two variables. It tells us whether an increase (or decrease) in one variable is associated with an increase or decrease in another variable.

- Values range from **-1 to +1**:
    - **+1** → Perfect positive relationship (as one increases, the other increases).
    - **-1** → Perfect negative relationship (as one increases, the other decreases).
    - **0** → No relationship.

🔹 **When to Use**

- To quantify the relationship strength between variables.
- To select features that are strongly related to the target variable.

🔹 **Example**

- **Height vs Weight** → Strong positive correlation.
- **Temperature vs Sales of Ice Cream** → Positive correlation.

🔹 **Python Pseudo Code**

```
# Correlation value between two variables

correlation = data["Study_Hours"].corr(data["Exam_Score"])

print("Correlation:", correlation)
```

Output :

Correlation: 0.9966669694751458

---

5. Covariance Analysis

🔹 **Definition**

Covariance Analysis is a statistical method that measures the degree to which two variables change together. It indicates whether an increase in one variable corresponds to an increase or decrease in another variable.

- **Positive covariance** → Variables increase together.
- **Negative covariance** → One increases while the other decreases.
- Unlike correlation, covariance is **not standardized** (depends on scale).

🔹 **When to Use**

- To check the **direction of relationship** (but not strength).
- Used as a foundation for **correlation** (since correlation = normalized covariance).

🔹 **Example**

- **Height and Weight** → Positive covariance (both increase together).
- **Temperature and Heater Usage** → Negative covariance (as temperature increases, heater usage decreases).

🔹 **Python Pseudo Code**

```
# Covariance matrix

cov_matrix = data.cov()

print(cov_matrix)
```

	Study_Hours	Exam_Score
Study_Hours	6.000000	34.928571
Exam_Score	34.928571	204.696429

📌 **Summary Table**

Technique	Purpose	When to Use	Example
Scatter Plot	Visualize relationship between 2 numeric variables	Check correlation, trends, outliers	Study hours vs Exam score
Pair Plot	Visualize all pairwise relationships	Explore datasets with multiple numeric features	Iris dataset features
Heatmap	Show correlation matrix visually	Identify strong/weak feature relationships	House features correlation
Correlation	Strength + direction of relationship (-1 to +1)	Feature selection, dependency check	Height vs Weight
Covariance	Direction of relationship (scale-dependent)	Foundation for correlation analysis	Temp vs Heater usage

---

UNIT -4

Matplotlib

Matplotlib is a powerful and versatile open-source plotting library for Python, designed to help users visualize data in a variety of formats. Developed by John D. Hunter in 2003, it enables users to graphically represent data, facilitating easier analysis and understanding

- Matplotlib is a low level graph plotting library in python that serves as a visualization utility.
- Matplotlib was created by John D. Hunter.
- Matplotlib is open source and we can use it freely.

Installation

Before you can start using Matplotlib, you need to install it. You can do this easily using

pip install matplotlib

Seaborn

is a high-level data visualization library in Python built on top of Matplotlib. It helps you create stunning statistical graphics using just a few lines of code. Seaborn is especially useful when you are working with datasets and want to explore relationships between variables. The library comes with a variety of built-in themes and colour palettes, which means your graphs will look professional without needing much customization

Installation

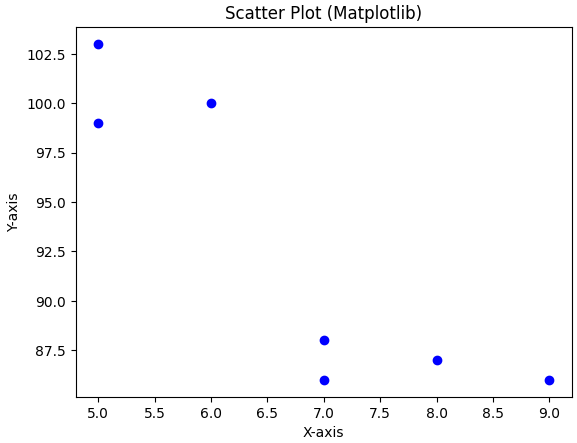
pip install seaborn

Visualization using Matplotlib

```
import matplotlib.pyplot as plt
x = [5, 7, 8, 7, 6, 9, 5]
y = [99, 86, 87, 88, 100, 86, 103]
plt.scatter(x, y, color="blue")
plt.title("Scatter Plot (Matplotlib)")
plt.xlabel("X-axis")
```

plt.ylabel("Y-axis")

plt.show()



Using Seaborn

import seaborn as sns

import matplotlib.pyplot as plt

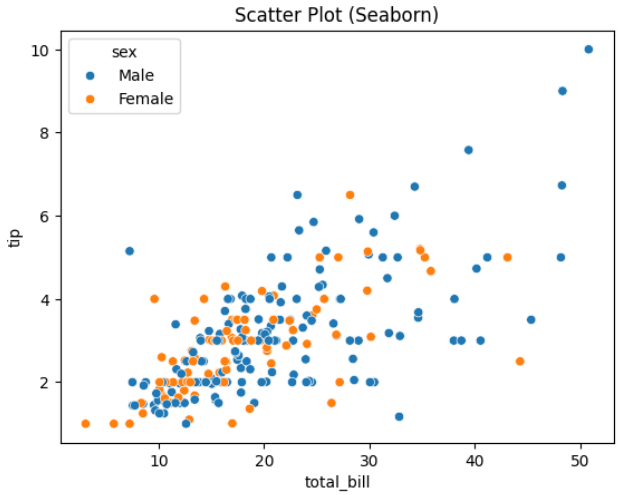
Sample dataset

tips = sns.load_dataset("tips")

sns.scatterplot(x="total_bill", y="tip", hue="sex", data=tips)

plt.title("Scatter Plot (Seaborn)")

plt.show()



Another example

import matplotlib.pyplot as plt

Sample data

students = ['Alice', 'Bob', 'Charlie', 'David', 'Eva']

math_scores = [85, 90, 78, 92, 88]

english_scores = [80, 85, 82, 88, 90]

Bar plot for Math and English scores

plt.bar(students, math_scores, color='skyblue', label='Math')

plt.bar(students, english_scores, color='lightgreen', label='English', alpha=0.7)// alpha ranges from **0 to 1**:

// 0 → completely **transparent** (invisible)

//1 → completely **opaque** (solid color)

//0.7 → **70% opaque, 30% transparent**

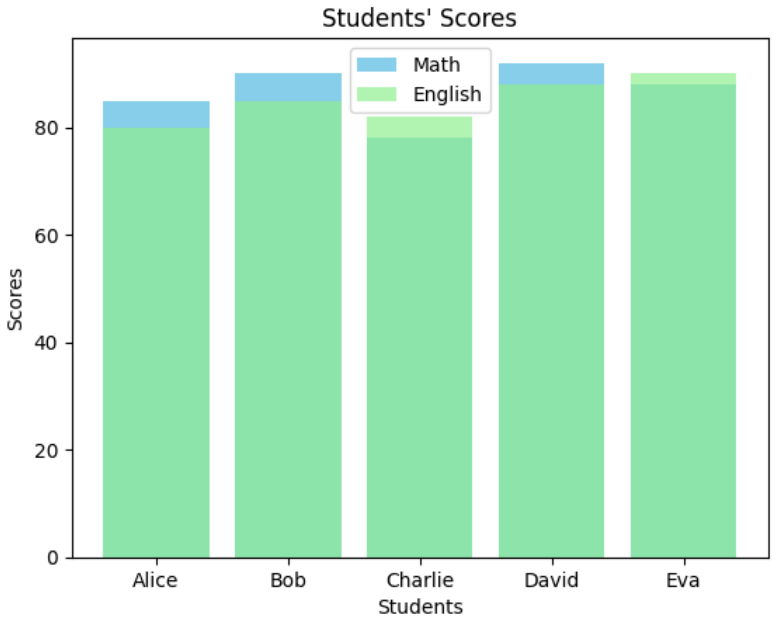
plt.title("Students' Scores")

plt.xlabel("Students")

plt.ylabel("Scores")

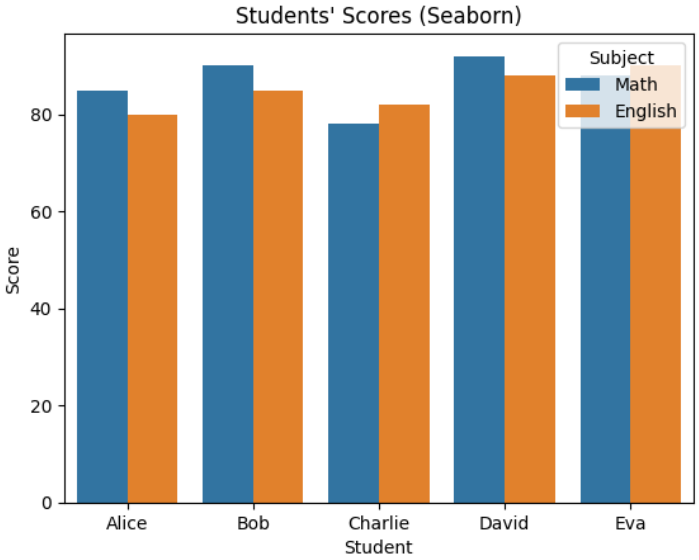
plt.legend()

plt.show()



Seaborn

```
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
# Create DataFrame
df = pd.DataFrame({
    'Student': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
    'Math': [85, 90, 78, 92, 88],
    'English': [80, 85, 82, 88, 90]
})
# Melt DataFrame for seaborn
df_melt = df.melt(id_vars='Student', var_name='Subject', value_name='Score')
# Bar plot
sns.barplot(x='Student', y='Score', hue='Subject', data=df_melt)
plt.title("Students' Scores (Seaborn)")
plt.show()
```



What is df.melt()?

melt() **converts a DataFrame from wide format to long format**.

- **Wide format**: Each variable is in a separate column.

Student	Math	English
Alice	85	80
Bob	90	85

Long format: All values of variables are stacked in one column, with another column indicating the variable type

Student	Subject	Score
Alice	Math	85
Alice	English	80
Bob	Math	90
Bob	English	85

Customization in plotting refers to **enhancing visual clarity and presentation** by adding **titles, axis labels, legends, and themes**.
These elements make a plot easier to understand and more professional.

---

1. Adding Titles

Definition

A **title** describes the purpose of the graph — what information it shows.

Syntax

plt.title("Your Title", fontsize=14, color='blue', loc='center')

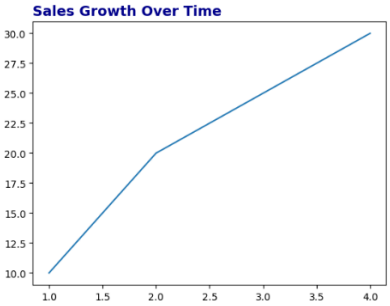
Example

import matplotlib.pyplot as plt

plt.plot([1, 2, 3, 4], [10, 20, 25, 30])

plt.title("Sales Growth Over Time", fontsize=14, color='darkblue', fontweight='bold',loc='left')

plt.show()



Parameters

Parameter	Description	Example
label	Title text	"Sales Growth"
fontsize	Title size	fontsize=14
color	Title color	color='red'
loc	Location ('left', 'center', 'right')	loc='left'

---

2. Adding Axis Labels

Definition

Axis labels tell us **what data each axis represents** — like Time, Sales, etc.

Syntax

plt.xlabel("X-axis Label", fontsize=12)

plt.ylabel("Y-axis Label", fontsize=12)

Example

plt.plot([1, 2, 3, 4], [10, 20, 25, 30])

plt.xlabel("Quarter", fontsize=12, color='purple')

plt.ylabel("Sales (in $1000s)", fontsize=12, color='green')

plt.title("Quarterly Sales Performance")

plt.show()

---

3. Adding Legends

Definition

A **legend** identifies different datasets or lines in a single plot.

Syntax

plt.legend(title="Legend Title", loc="upper right", fontsize=10)

Example

plt.plot([1, 2, 3, 4], [10, 20, 25, 30], label='Product A')

plt.plot([1, 2, 3, 4], [15, 18, 22, 28], label='Product B')

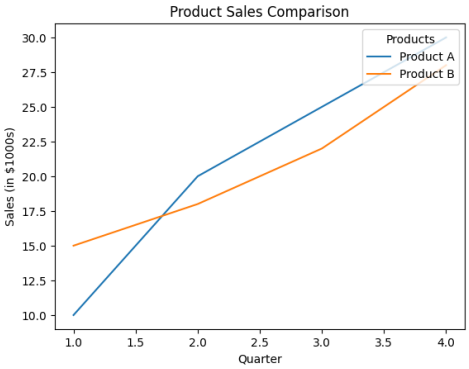
plt.title("Product Sales Comparison")

plt.xlabel("Quarter")

plt.ylabel("Sales (in $1000s)")

plt.legend(title="Products", loc='upper right’)

plt.show()



---

4. Applying Themes (Styles)

Definition

Themes (also called **styles**) control the overall look of the plot — colors, gridlines, backgrounds, and fonts.

Syntax

plt.style.use("stylename")

Example

plt.style.use("ggplot")

plt.plot([1, 2, 3, 4], [10, 20, 25, 30], label='Product A', marker='o')

plt.plot([1, 2, 3, 4], [15, 18, 22, 28], label='Product B', linestyle='--')

plt.title("Styled Sales Graph", fontsize=14)

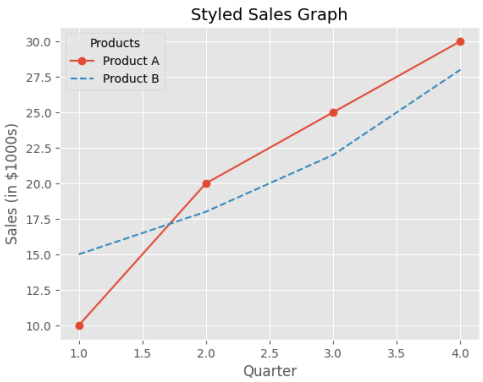
plt.xlabel("Quarter")

plt.ylabel("Sales (in $1000s)")

plt.legend(title="Products", loc="upper left")

plt.grid(True)

plt.show()

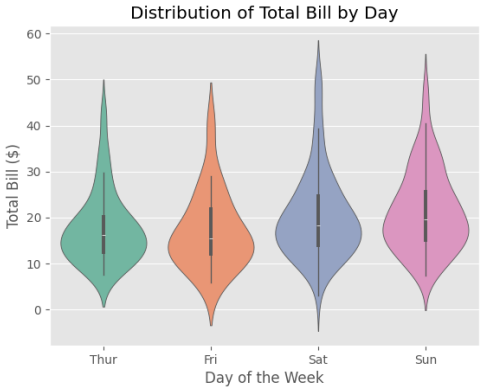


Violin Plots

A **violin plot** is a combination of a **boxplot** and a **kernel density plot**.
It displays the **probability density** of the data at different values (the width of the "violin" shows how frequent values are) along with key summary statistics such as the **median** and **quartiles**.

The symmetrical "violin" shape represents the distribution of the data — wider areas indicate higher data density, while narrower parts show fewer observations. Violin plots are especially useful for **comparing multiple distributions** side-by-side.

```
import seaborn as sns

import matplotlib.pyplot as plt

# Load example dataset

tips = sns.load_dataset("tips")

# Create a violin plot

sns.violinplot(x="day", y="total_bill", data=tips, palette="Set2")

plt.title("Distribution of Total Bill by Day")

plt.xlabel("Day of the Week")

plt.ylabel("Total Bill ($)")

plt.show()
```

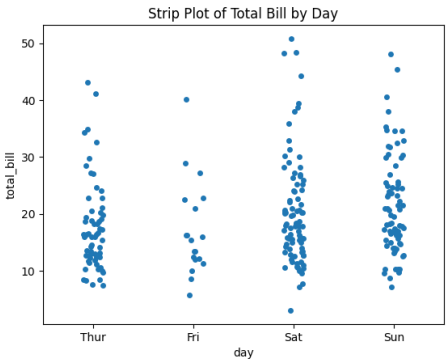


Strip Plots

Definition:
A **Strip Plot** is a categorical scatter plot where **individual data points** are plotted along a single axis corresponding to a **categorical variable**.

- Each **dot represents one observation**.
- Dots may **overlap**, but adding **jitter** spreads them out for better visibility.
- Mainly used to **visualize distribution and spread** of data points across categories.

```
import seaborn as sns

import matplotlib.pyplot as plt

tips = sns.load_dataset("tips")

sns.stripplot(x="day", y="total_bill", data=tips, jitter=True)

plt.title("Strip Plot of Total Bill by Day")

plt.show()
```



Swarm Plots

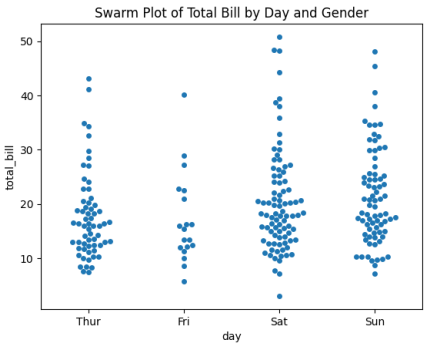
Definition:
A **Swarm Plot** is similar to a strip plot but **automatically adjusts the position of points** to **avoid overlapping**.

- Each dot represents **one observation**, but the points are **spaced out horizontally or vertically** for clarity.
- Provides a **clear view of data distribution**, especially when many points are clustered.

Key Points:

- Prevents **overlapping of points** (unlike strip plots).
- Best for **small to medium datasets** (large datasets can be slow).
- Can be combined with **violin or boxplots** for distribution + points visualization.

```
import seaborn as sns

import matplotlib.pyplot as plt

sns.swarmplot(x="day", y="total_bill", data=tips)

plt.title("Swarm Plot of Total Bill by Day and Gender")

plt.show()
```

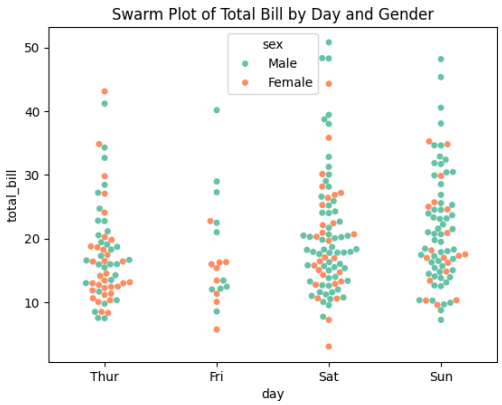


adding palette and hue for the above graph

```
import seaborn as sns

import matplotlib.pyplot as plt

sns.swarmplot(x="day", y="total_bill", data=tips, hue="sex", palette="Set2")
```

plt.title("Swarm Plot of Total Bill by Day and Gender")

plt.show()



Note :

palette="Set2" means Use a set of light pastel colours to differentiate categories in the plot.

Multivariate Visualization

Multivariate Visualization is the process of graphically representing **three or more variables** in a dataset **simultaneously** to understand how they **relate, interact, and influence each other**.

It **visualizing multiple variables together** to discover patterns, correlations, trends

Use

- To explore complex relationships among multiple features
- To detect interactions, clusters, or outliers

- To understand dependencies before building predictive models

Example :

If you're studying sales data with these columns:

- Sales Amount
- Advertising Spend
- Customer Age
- Region

Pair Plot – shows pairwise relationships between variables

- **Heatmap** – shows correlations among many numeric features
- **3D Scatter Plot** – visualizes relationships among three variables
- **Parallel Coordinates Plot** – compares several features across samples
- **Facet Grid** – shows the effect of multiple categorical variables

3D Scatter Plot

A **3D Scatter Plot** is a type of data visualization that displays the **relationship among three numerical variables** using a **three-dimensional coordinate system (X, Y, Z axes)**.

- Each point in the plot represents one observation, positioned according to its values on those three variables.

Example :

Imagine a dataset of customers with:

- **X-axis:** Age
- **Y-axis:** Income
- **Z-axis:** Spending Score

A 3D scatter plot will show how customers of different ages and incomes vary in spending behavior — helping in **customer segmentation**.

```
import plotly.express as px

import pandas as pd


# Create a small dataset

data = {

    'Age': [23, 45, 30, 55, 27, 38, 50, 22, 35, 48],

    'Annual_Income': [35000, 70000, 60000, 90000, 40000, 65000, 85000, 30000, 58000, 75000],

    'Spending_Score': [80, 40, 75, 20, 90, 60, 30, 85, 65, 45],

    'Customer_Type': ['High', 'Low', 'High', 'Low', 'High', 'Medium', 'Low', 'High', 'Medium', 'Low']

}


df = pd.DataFrame(data)


# Create 3D Scatter Plot

fig = px.scatter_3d(df,
```

```
                    x='Age',
                    y='Annual_Income',
                    z='Spending_Score',
                    color='Customer_Type',
                    symbol='Customer_Type',
                    size='Spending_Score',
                    title="Customer Segmentation in 3D: Age vs Income vs Spending Score")

fig.show()
```

color

- color='Customer_Type' means **the points will be colored differently based on the customer type**

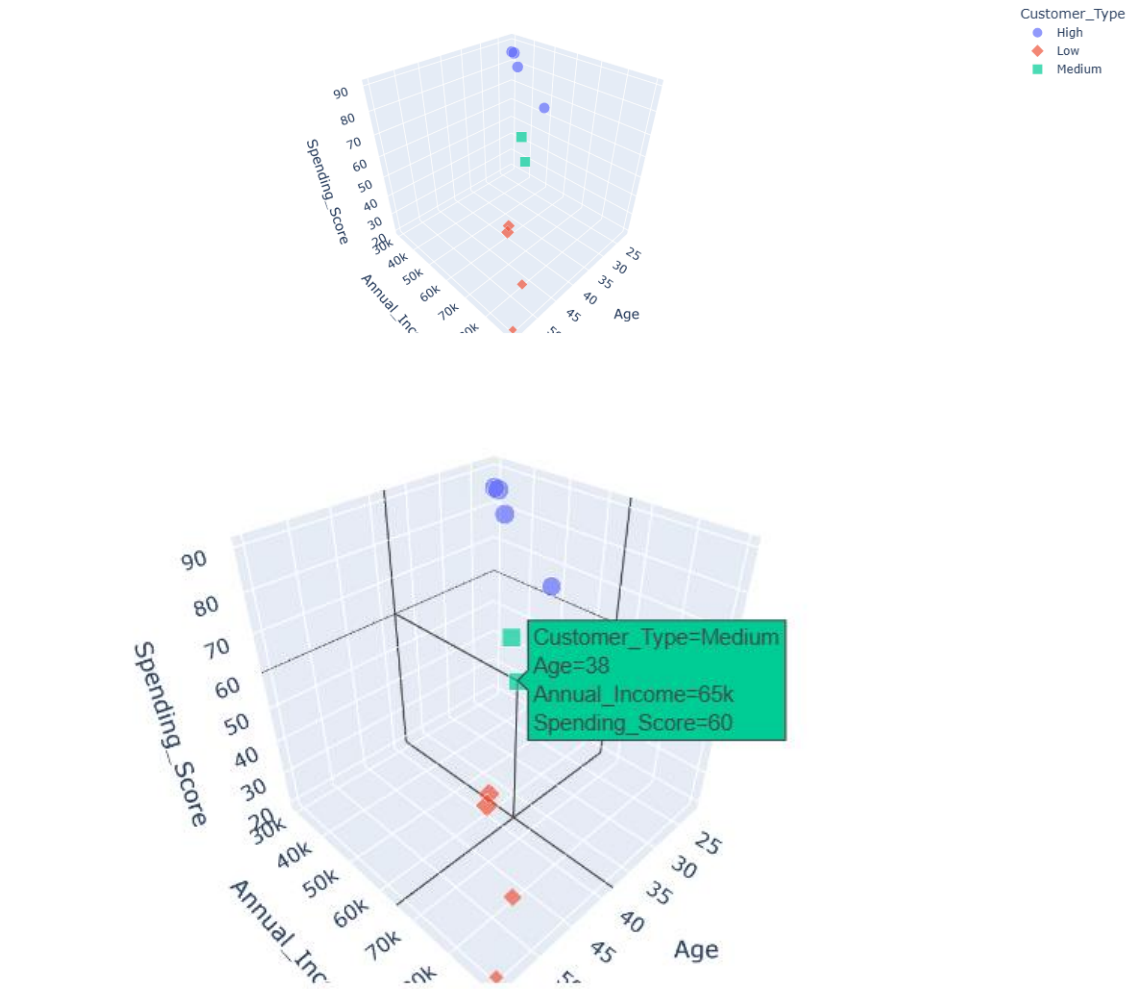
symbol

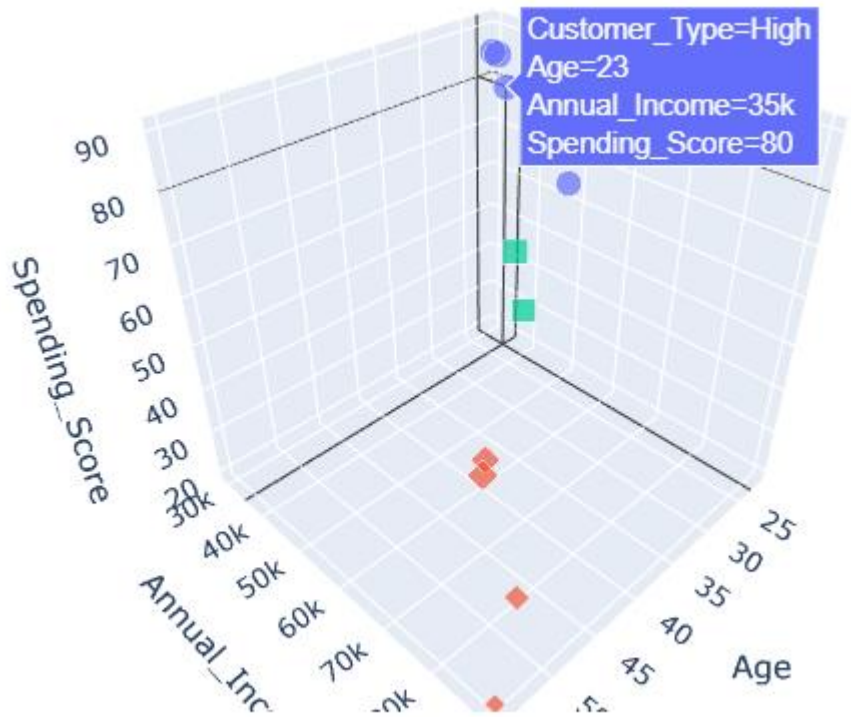
- symbol='Customer_Type' means **the marker shape changes based on customer type**.

size

- size='Spending_Score' means **the marker size varies according to the spending score**.

Customer Segmentation in 3D: Age vs Income vs Spending Score





Parallel Coordinates

A **Parallel Coordinates Plot** is a way to **compare multiple variables at the same time**.

- Each **variable** has its **own vertical line (axis)**.
- Each **data point** (like a student, customer, or car) is drawn as a **line connecting its values** across all the vertical axes.

Example:

Imagine students have **three scores**:

Student	Math	Science	English
Alice	90	85	70
Bob	60	75	80
Carol	80	90	95

Student Math Science English

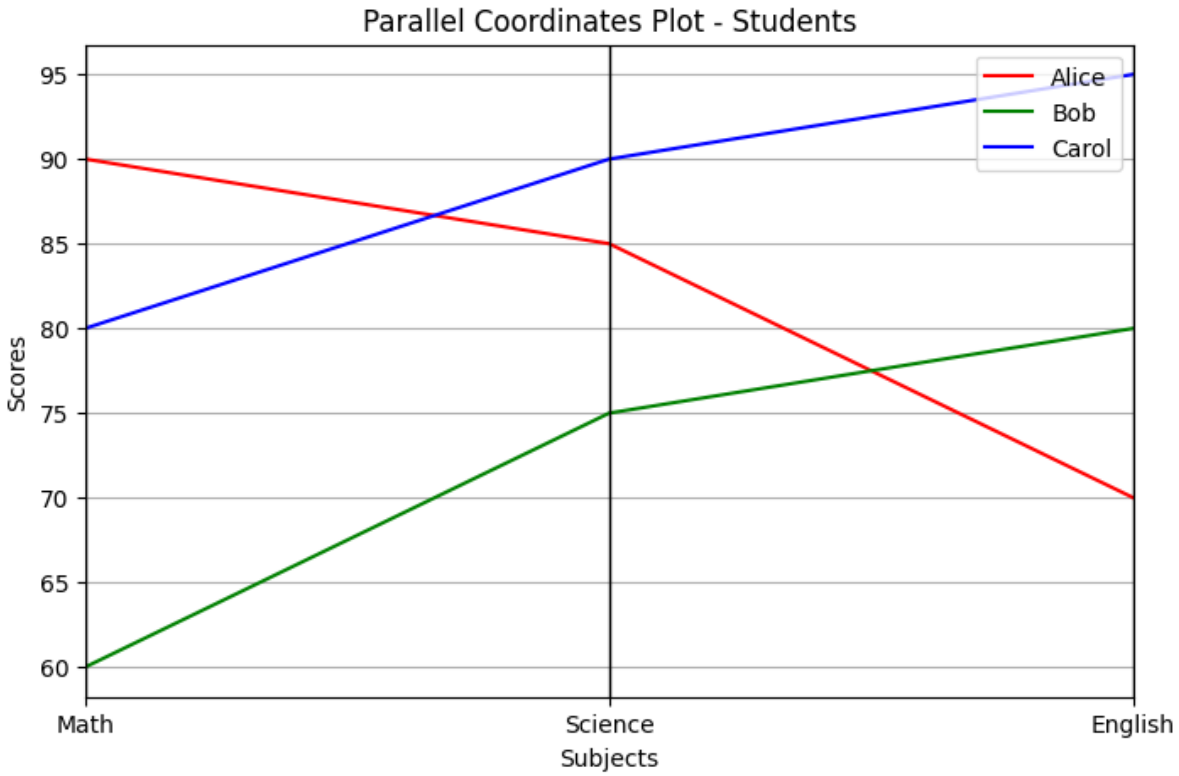
Draw **three vertical axes**: Math, Science, English.

For **Alice**, put a dot at **90 on Math**, **85 on Science**, **70 on English**, and **connect the dots** with a line.

Repeat for Bob and Carol.

```
import pandas as pd
from pandas.plotting import parallel_coordinates
import matplotlib.pyplot as plt
# Simple dataset
data = {
    'Student': ['Alice', 'Bob', 'Carol'],
    'Math': [90, 60, 80],
    'Science': [85, 75, 90],
    'English': [70, 80, 95]
}
df = pd.DataFrame(data)
# Parallel Coordinates Plot
plt.figure(figsize=(8,5))
parallel_coordinates(df, 'Student', color=['r','g','b'])
plt.title("Parallel Coordinates Plot - Students")
plt.xlabel("Subjects")
plt.ylabel("Scores")
```

plt.show()



Patterns across subjects:

- Alice is strong in Math & Science, weaker in English.
- Carol is strong in all subjects.

Compare multiple students at once:

- See who has similar performance patterns.

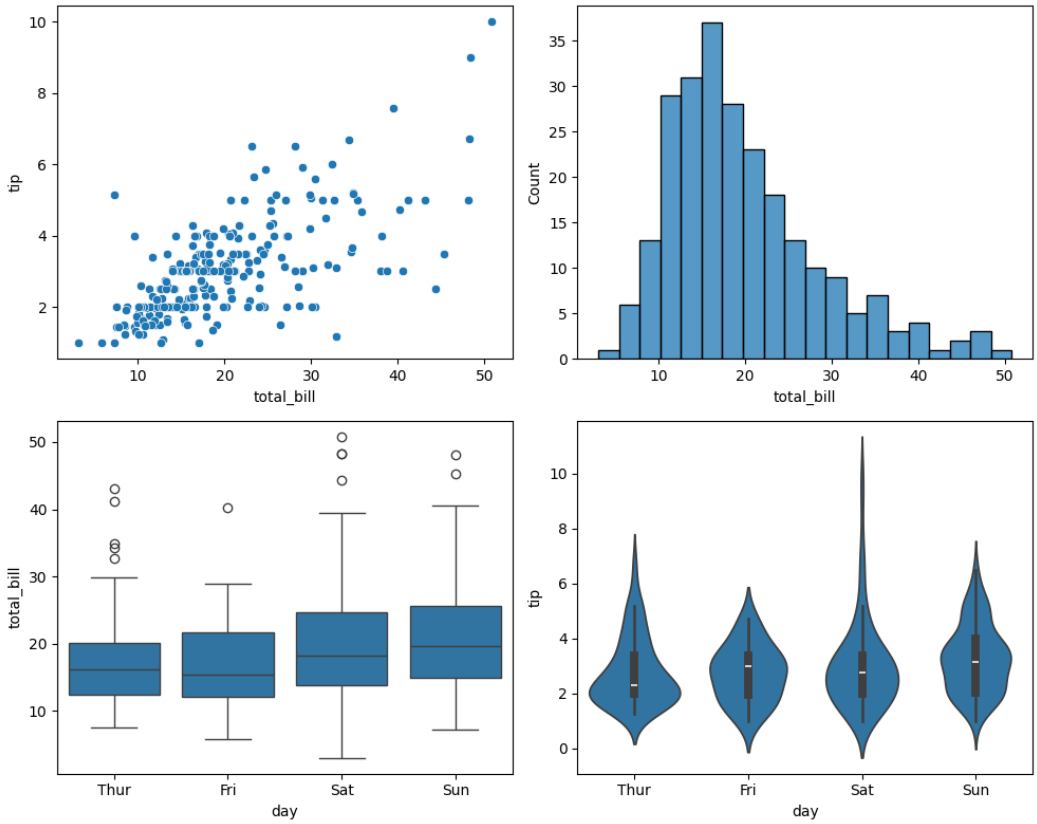
Identify extremes/outliers:

- Lines that are very high or low stand out.

Subplots

Subplots are **multiple plots displayed within a single figure**. Instead of creating separate figures for each graph, you can arrange **multiple plots in a grid** (rows × columns) inside **one figure**

```
import matplotlib.pyplot as plt
import seaborn as sns
# Load dataset
tips = sns.load_dataset("tips")
# Create subplots: 2 rows × 2 columns
fig, axes = plt.subplots(2, 2, figsize=(10, 8))
# Top-left plot
sns.scatterplot(data=tips, x="total_bill", y="tip", ax=axes[0, 0])
# Top-right plot
sns.histplot(data=tips, x="total_bill", bins=20, ax=axes[0, 1])
# Bottom-left plot
sns.boxplot(data=tips, x="day", y="total_bill", ax=axes[1, 0])
# Bottom-right plot
sns.violinplot(data=tips, x="day", y="tip", ax=axes[1, 1])

plt.tight_layout()    # Adjust spacing
plt.show()
```



Plotly

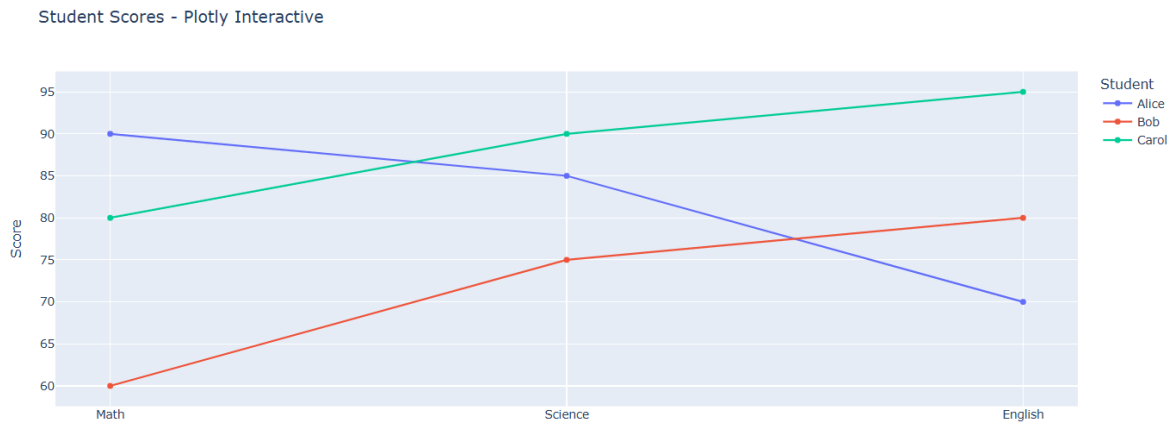
Plotly is a **Python library for interactive data visualization**.

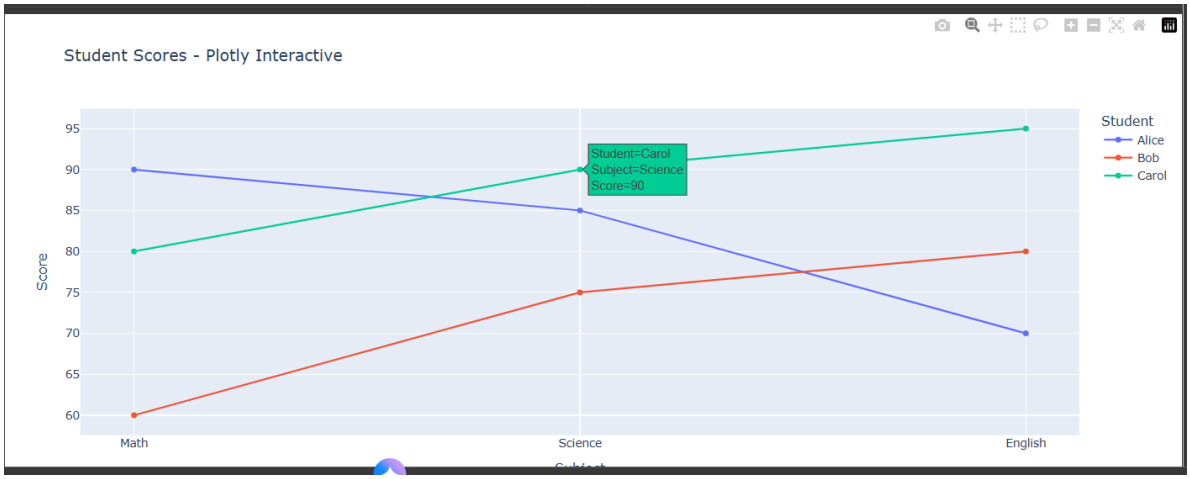
- Unlike **Matplotlib** or **Seaborn**, which produce **static plots**, Plotly allows users to **interact** with plots.
- You can **zoom, pan, hover**, and **rotate 3D plots**.
- It works for **both simple charts and complex dashboards**, and integrates easily with Pandas dataframes

Example :

import plotly.express as px

```
# Data in long format for Plotly
df_long = pd.DataFrame({
    'Student':
['Alice','Alice','Alice','Bob','Bob','Bob','Carol','Carol','Carol'],
    'Subject': ['Math','Science','English']*3,
    'Score': [90,85,70,60,75,80,80,90,95]
})
fig = px.line(df_long, x='Subject', y='Score', color='Student',
markers=True,
              title="Student Scores - Plotly Interactive")
fig.show()
```





What Are Interactive Visualizations?

Interactive visualizations are plots or charts that allow users to **interact with the data** instead of just viewing it as a static image.

- Users can **zoom, pan, filter, hover, or rotate** to explore data.
- They make it easier to **discover patterns, trends, and outliers** dynamically.

Uses :

Explore Data Dynamically

- Zoom into a specific region or filter categories
- Hover to see exact values for each data point

Handle Large Datasets

- Avoid clutter by exploring only the area you need

Better Presentations

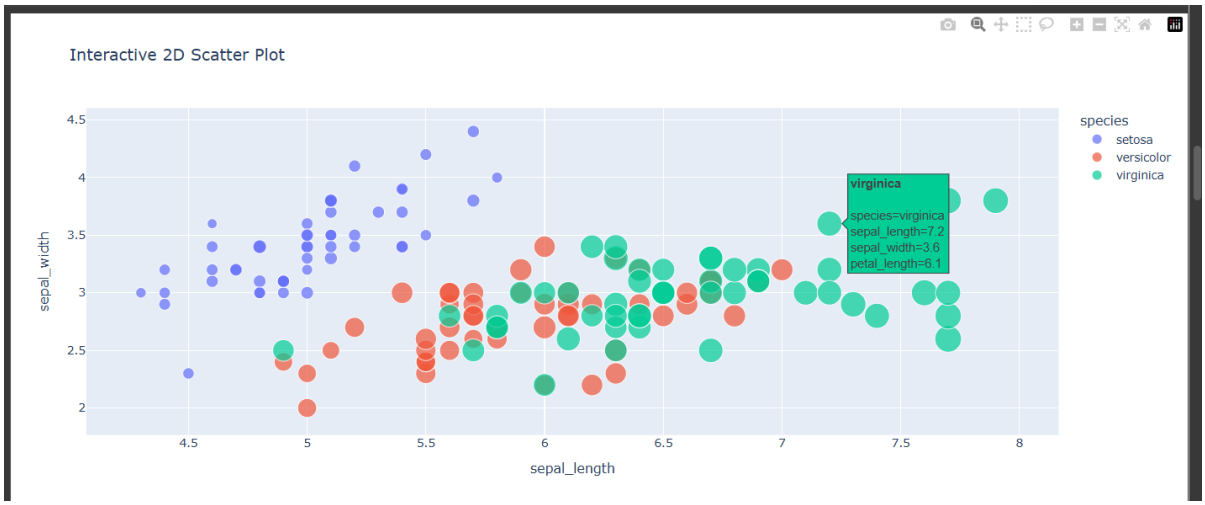
- Makes charts more engaging for reports, dashboards, or web apps

Multivariate Data

- Can visualize relationships between 3+ variables using color, size, shape, or 3D axes

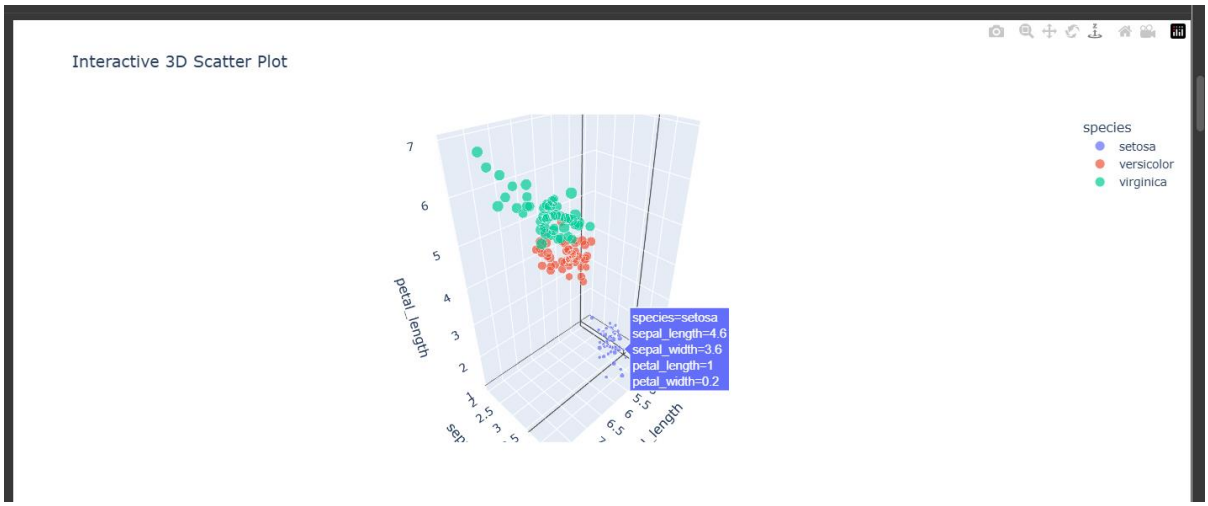
With 2D plot

```
import plotly.express as px


# Sample dataset
df = px.data.iris()


fig = px.scatter(df,
                 x='sepal_length',
                 y='sepal_width',
                 color='species',
                 size='petal_length',
                 hover_name='species',
                 title="Interactive 2D Scatter Plot")
fig.show()
```



With 3D

```
fig = px.scatter_3d(df,
                    x='sepal_length',
                    y='sepal_width',
                    z='petal_length',
                    color='species',
                    size='petal_width',
                    title="Interactive 3D Scatter Plot")
fig.show()
```



UNIT – 5

Step-by-step EDA on Titanic Datasets

Overview of the Titanic Dataset Columns

Before diving into data visualization and feature engineering, it's essential to understand the context and meaning of each column in the Titanic dataset. Here's a brief overview of the columns:

1. **PassengerId**: A unique identifier for each passenger.
2. **Survived:** Indicates whether the passenger survived (1) or not (0).
3. **Pclass**: Passenger's class (1 = 1st class, 2 = 2nd class, 3 = 3rd class). This is a proxy for socio-economic status (SES).
4. **Name**: The full name of the passenger.
5. **Sex**: The gender of the passenger (male or female).
6. **Age**: The age of the passenger in years. Some entries contain fractional values to represent ages less than one year. If the age is estimated, it is in the form of xx.5.
7. **SibSp**: Number of siblings and spouses aboard the Titanic.

- **Sibling** = brother, sister, stepbrother, stepsister
- **Spouse** = husband, wife (mistresses and fiancés were ignored)

8. **Parch**: Number of parents and children aboard the Titanic.

- **Parent** = mother, father
- **Child** = daughter, son, stepdaughter, stepson
- Some children travelled only with a nanny, therefore Parch=0 for them.

9. **Ticket**: The ticket number.

10. **Fare**: The amount of money paid for the ticket.

11. **Cabin**: The cabin number where the passenger stayed.

12. **Embarked**: The port where the passenger boarded the ship (C = Cherbourg; Q = Queenstown; S = Southampton).

Load the Data and show basic information about the data

1. titanic.head(): Provides top 5 rows in dataset,

2. titanic.info(): Offers a concise summary of the dataset, highlighting the number of entries, non-null counts, data types, and memory usage. It acts as a fact sheet for our DataFrame, outlining its structure and alerting us to potential data quality issues**.** We note missing values in the Age, Cabin, and Embarked columns, which need to be addressed before analysis.

3. titanic.describe(): Provides descriptive statistics that summarize the central tendency, dispersion, and shape of a dataset's distribution, excluding NaN values. It offers a quick statistical summary of numerical columns which helps in understanding the distribution, scale, and spread of data.

We can observe that about 38% of the passengers survived (the mean of the Survived column is 0.38).

We can also observe the **median of Fares is 14.454 but the mean is 32.2**, thus indicating a number of very high-paying passengers.

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

titanic = sns.load_dataset('titanic')

titanic.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
actions															
	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

titanic.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
 #   Column       Non-Null Count  Dtype   
---  ------       --------------  -----   
 0   survived     891 non-null    int64   
 1   pclass       891 non-null    int64   
 2   sex          891 non-null    object  
 3   age          714 non-null    float64 
 4   sibsp        891 non-null    int64   
 5   parch        891 non-null    int64   
 6   fare         891 non-null    float64 
 7   embarked     889 non-null    object  
 8   class        891 non-null    category
 9   who          891 non-null    object  
 10  adult_male   891 non-null    bool    
 11  deck         203 non-null    category
 12  embark_town  889 non-null    object  
 13  alive        891 non-null    object  
 14  alone        891 non-null    bool    
dtypes: bool(2), category(2), float64(2), int64(4), object(5)
memory usage: 80.7+ KB
```

titanic.describe())

	survived	pclass	age	sibsp	parch	fare
count	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

Now, let’s get an overview of missing values:

```
missing_values = titanic.isnull().sum()
print(missing_values)
```

	0
survived	0
pclass	0
sex	0
age	177
sibsp	0
parch	0
fare	0
embarked	2
class	0
who	0
adult_male	0
deck	688
embark_town	2
alive	0
alone	0

dtype: int64

Filling in the Missing Values

When addressing the issue of missing values in the Titanic dataset, different strategies can be applied to fill the data based on the nature of the data in each column, let's look at the basic approach:

1. Age:

The median can be used to fill missing values in the Age column because:

- The **median is robust against outliers which means it preserves the original distribution of ages** without being skewed by outliers, which can occur with the mean if outliers are present.
- It is more representative of the **central tendency** of the dataset.

2. Embarked:

The **mode** is used to fill the two missing values in the Embarked column because:

- **The data is nominal**, meaning no category inherently ranks above another.
- Only two values are missing, making the mode a straightforward and justifiable choice given its simplicity and the small number of imputations needed.

3. Cabin:

- Due to the large number of missing values in the cabin column, (687 out of 891), and very small sample data, we will drop this column.
- However, we will use predictive modelling for Cabin prediction and filling the column in other articles of this series. **(Surprise!)**

median_age = titanic["age"].median()

```
median_age = titanic["age"].median()
median_age
```

```
28.0
```

titanic["Age"].fillna(median_age,inplace = True)

mode_point = titanic["Embarked"].mode()[0]

```
titanic["embark_town"].mode()[0]
```

```
'Southampton'
```

titanic["Embarked"].fillna(mode_point,inplace = True)

```
# Drop the original 'Cabin' column
titanic.drop(columns=['Cabin'], inplace=True)
```

Data Visualization

Before diving into feature engineering and scaling, it's crucial to understand the initial distribution and relationships within the data. First of all **we will plot histograms to see what the distribution of continuous numerical features(ie. data from Age and Fare column) is like.**

To see any outliers within the Age and Fare column, which will help us in deciding what scaling algorithm to use, we also make boxplots.

We make count plots to view the distribution within our categorical variables.

You can make as many plots as you please and ask GPT to explain the code to you line by line, hell, it will plot it for you and explain to you the visualizations, what are we even doing, anyway, let's keep this existential crisis away for now and focus here.

```
# 1. Histogram and KDE Plots
plt.figure(figsize=(12, 6))

# Age Distribution

plt.subplot(1, 2, 1)

sns.histplot(data=titanic, x='age', bins=30, kde=True, color='skyblue')

plt.title('Histogram & KDE of Age', fontsize=14)

plt.xlabel('Age')

plt.ylabel('Frequency')


# Fare Distribution

plt.subplot(1, 2, 2)

sns.histplot(data=titanic, x='fare', bins=30, kde=True, color='lightcoral')

plt.title('Histogram & KDE of Fare', fontsize=14)

plt.xlabel('Fare')
```

plt.ylabel('Frequency')

plt.tight_layout()#it fits to the screen

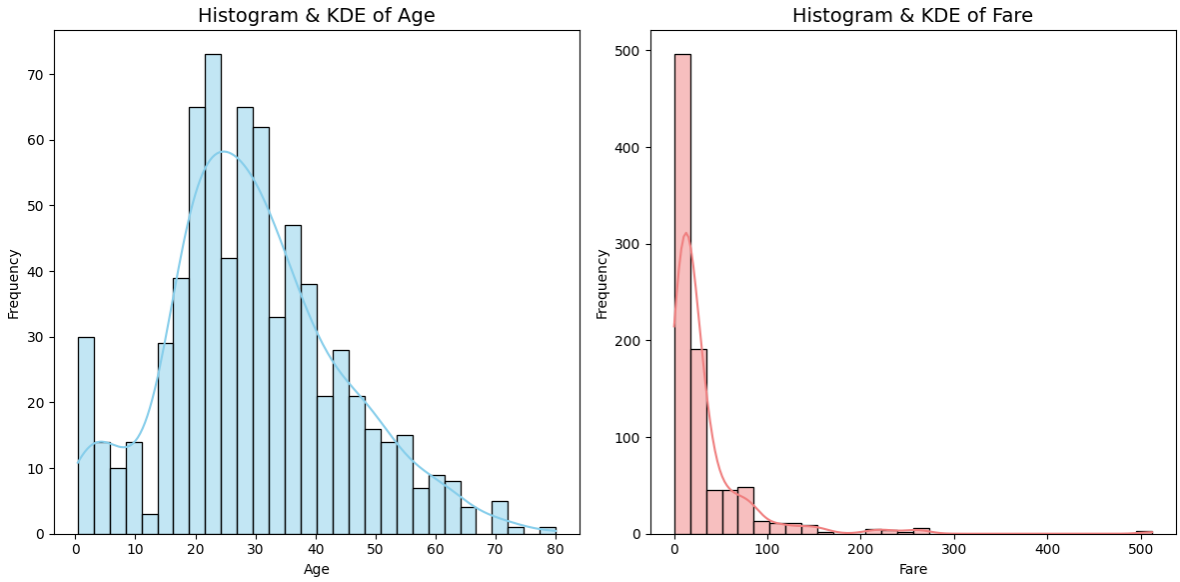
plt.show()

plt.subplot(1, 2, 1)

1You want **1 row** of plots.

#2You want **2 columns** of plots (side by side).

#1This is the **first plot** in that grid.



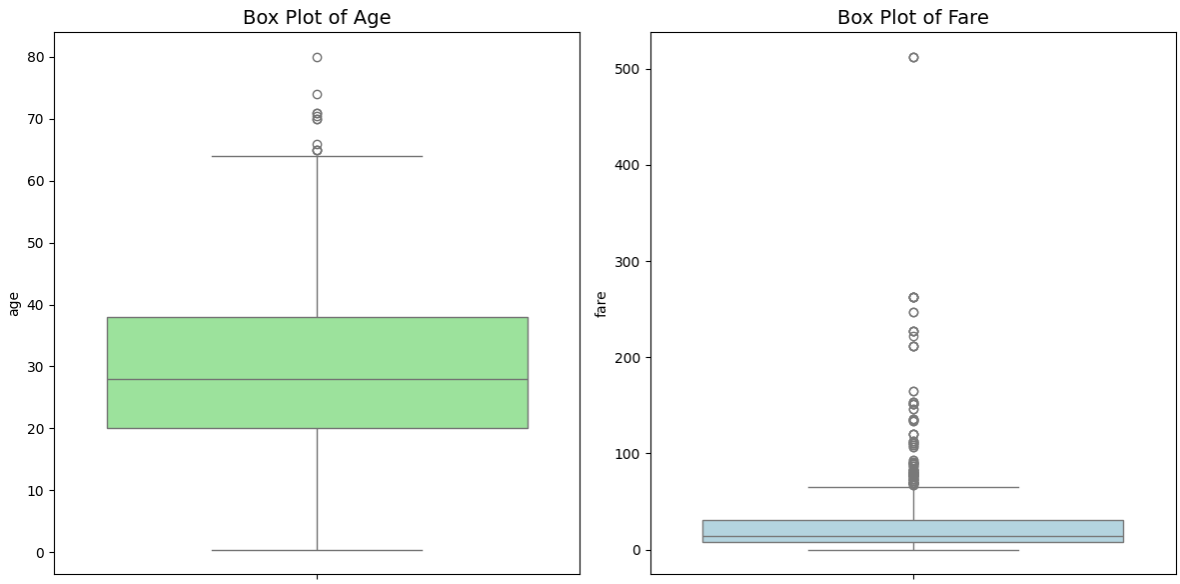
Histograms of Age and Fare, respectively

Histograms for ‘Age’ and ‘Fare’ features reveal their distributions.

The ‘Age’ histogram shows a roughly **normal distribution (or is it ? More on it at Feature Scaling Part)** with a peak around 30 years old, slightly right-skewed, indicating a few older passengers.

The ‘Fare’ histogram is highly right-skewed, with most values at the lower end and significant outliers extending up to 500.

```
# 2. Box Plots
plt.figure(figsize=(12, 6))

# Age Boxplot
plt.subplot(1, 2, 1)
sns.boxplot(data=titanic, y='age', color='lightgreen')
plt.title('Box Plot of Age', fontsize=14)

# Fare Boxplot
plt.subplot(1, 2, 2)
sns.boxplot(data=titanic, y='fare', color='lightblue')
plt.title('Box Plot of Fare', fontsize=14)

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

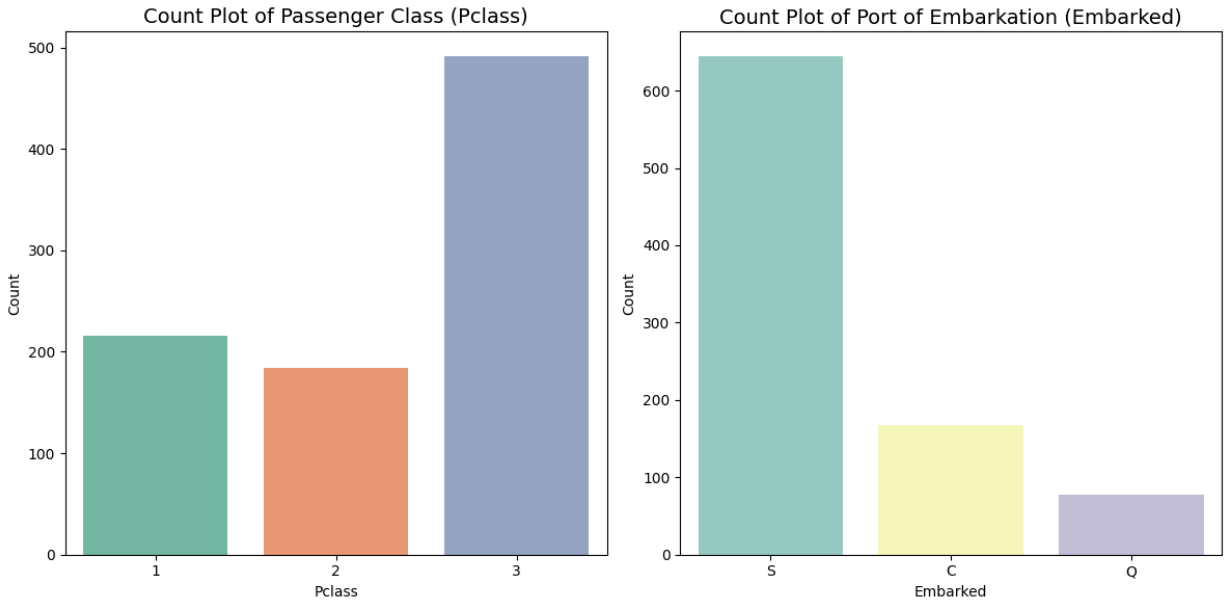


Box Plots of Age and Fare

The ‘Age’ box plot shows a median around 30, with outliers beyond 60, indicating elderly passengers.

The ‘Fare’ box plot highlights a low median fare with numerous outliers above 300. This confirms the presence of high-value outliers seen in the histogram, underscoring the need for robust scaling, especially for ‘Fare’.

```
# 3. Count Plots for Categorical Features
plt.figure(figsize=(12, 6))

# Passenger Class Count
plt.subplot(1, 2, 1)
sns.countplot(data=titanic, x='pclass', palette='Set2')
plt.title('Count Plot of Passenger Class (Pclass)', fontsize=14)
plt.xlabel('Pclass')
plt.ylabel('Count')

# Embarked Port Count
plt.subplot(1, 2, 2)
sns.countplot(data=titanic, x='embarked', palette='Set3')
plt.title('Count Plot of Port of Embarkation (Embarked)', fontsize=14)
plt.xlabel('Embarked')
plt.ylabel('Count')
plt.tight_layout()
plt.show()
```



Count plot PClass(Passenger Class) and Embarked

Count plots for ‘Pclass’ and ‘Embarked’ reveal distributions in categorical features.

Most passengers are in the third class, followed by the first and second classes, indicating socio-economic disparities.

For ‘Embarked’, most passengers embarked from port ‘S’ (Southampton), followed by ‘C’ (Cherbourg) and ‘Q’ (Queenstown). This shows Southampton was the most common embarkation point, correlating with class distribution and fare.

Feature Engineering

Feature engineering is a crucial step in preparing data for machine learning models. **It involves transforming raw data into features that better represent the underlying problem to the predictive models(like Logistic Regression).**

In this step, we’ll convert **categorical variables into numerical values**, which is essential for most machine learning algorithms.

Mapping for binary value categorical data

First, we'll convert the Sex column into numerical values. This is because many machine learning algorithms require numerical input. **We'll map the values 'male' to 0 and 'female' to 1.**

```
# Convert 'Sex' to numerical values
titanic['Sex'] = titanic['Sex'].map({'male': 0, 'female': 1})
```

One-hot encoding for multi-value categorical data

Next, we'll convert the Embarked column into numerical values using a technique called one-hot encoding.

```
# Convert 'Embarked' to numerical values using one-hot encoding
titanic = pd.get_dummies(titanic, columns=['Embarked'])
```

One-hot encoding is a method of converting categorical data into numerical data by creating new columns for each unique category. Each column corresponds to a category and contains binary values (0 or 1) indicating the presence or absence of the category in a particular row.

One-hot encoding is used for nominal categorical data, where the categories do not have an inherent order. Examples of such data include:

- Colours (e.g., 'red', 'blue', 'green')
- Cities (e.g., 'New York', 'Los Angeles', 'Chicago')
- Types of fruit (e.g., 'apple', 'banana', 'cherry')

Example of One-Hot Encoding

Let's illustrate one-hot encoding with the Embarked column:

Original Embarked column:

Embarked
C
Q
S
C
S

After one-hot encoding:

Embarked_C	Embarked_Q	Embarked_S
1	0	0
0	1	0
0	0	1
1	0	0
0	0	1

Feature Scaling

We through titanic.describe() saw that the Ages vary from 0 to 80 and the Fares vary from £ 7 to £ 512.

All other columns do not have such variety, and we need to scale these features to **prevent dominance by these features and ensure equal contribution by all features.**

Here, comes the nice part, we have three kinds of feature scaling techniques:

Standardization (Z-score Normalization)

Standardization scales **the features to have a mean of zero and a standard deviation of one**. It is useful when the features follow a **Gaussian distribution.**

MinMaxScaler (Normalization)

Min-Max scaling **transforms the features to a fixed range, typically [0, 1]**. It is useful when **the features are required to be within a specific range.**

Robust Scaling

Robust scaling **uses the median and the interquartile range (IQR) for scaling**. It is useful for datasets with outliers, as it reduces the impact of outliers.

Fit and transform the selected features

from sklearn.preprocessing import StandardScaler

age_scaler = StandardScaler()

```
fare_scaler = StandardScaler()


titanic['Age'] = age_scaler.fit_transform(titanic[['Age']])
titanic['Fare'] = fare_scaler.fit_transform(titanic[['Fare']])

# Display the first few rows of the modified dataset
print(titanic.head())
```

```
   survived  pclass  sex       age  sibsp  parch      fare  class    who  \
0         0       3    0 -0.530377      1      0 -0.502445  Third    man
1         1       1    1  0.571831      1      0  0.786845  First  woman
2         1       3    1 -0.254825      0      0 -0.488854  Third  woman
3         1       1    1  0.365167      1      0  0.420730  First  woman
4         0       3    0  0.365167      0      0 -0.486337  Third    man

   adult_male deck alive  alone  embarked_Q  embarked_S  \
0        True  NaN    no  False       False        True
1       False    C   yes  False       False       False
2       False  NaN   yes   True       False        True
3       False    C   yes  False       False        True
4        True  NaN    no   True       False        True

   embark_town_Queenstown  embark_town_Southampton
0                   False                    True
1                   False                   False
2                   False                    True
3                   False                    True
4                   False                    True
```

Correlation Analysis

Now, the insights from this will help us do predictive modelling on our dataset, but with a ton of insight on why we do what we do.

So, to apply correlation analysis here we will first identify what kind of data is within our columns, first, we observe that the data types are as follows:

- ***Numerical Continuous***: Data that can take any value within a range and has a meaningful order and interval (e.g., Age, Fare)
- ***Ordinal***: Categorical data with a meaningful order but not necessarily equidistant between categories (e.g., Pclass)
- ***Nominal***: Categorical data without a meaningful order or ranking (e.g., Sex, Embarked, Survived)

- ***Binary (treated as nominal)***: A special case of nominal data with only two categories, often represented as 0 and 1 (e.g., Sex, Survived).

Then we see what are some of the correlation measures we could use that are appropriate to the above data types:

- ***Numerical Continuous:*** Pearson or Spearman correlation
- ***Ordinal***: Spearman or Kendall's Tau correlation
- ***Nominal:*** Cramer's V, Chi-Square Test of Independence

Here is a brief overview of what these words mean because we are trying to understand the fundamentals, here:

1. **Pearson Correlation:**

- **Definition:** Measures the linear relationship between two continuous variables.
- **Values:** Range from -1 (perfect negative linear relationship) to 1 (perfect positive linear relationship), with 0 indicating no linear relationship.

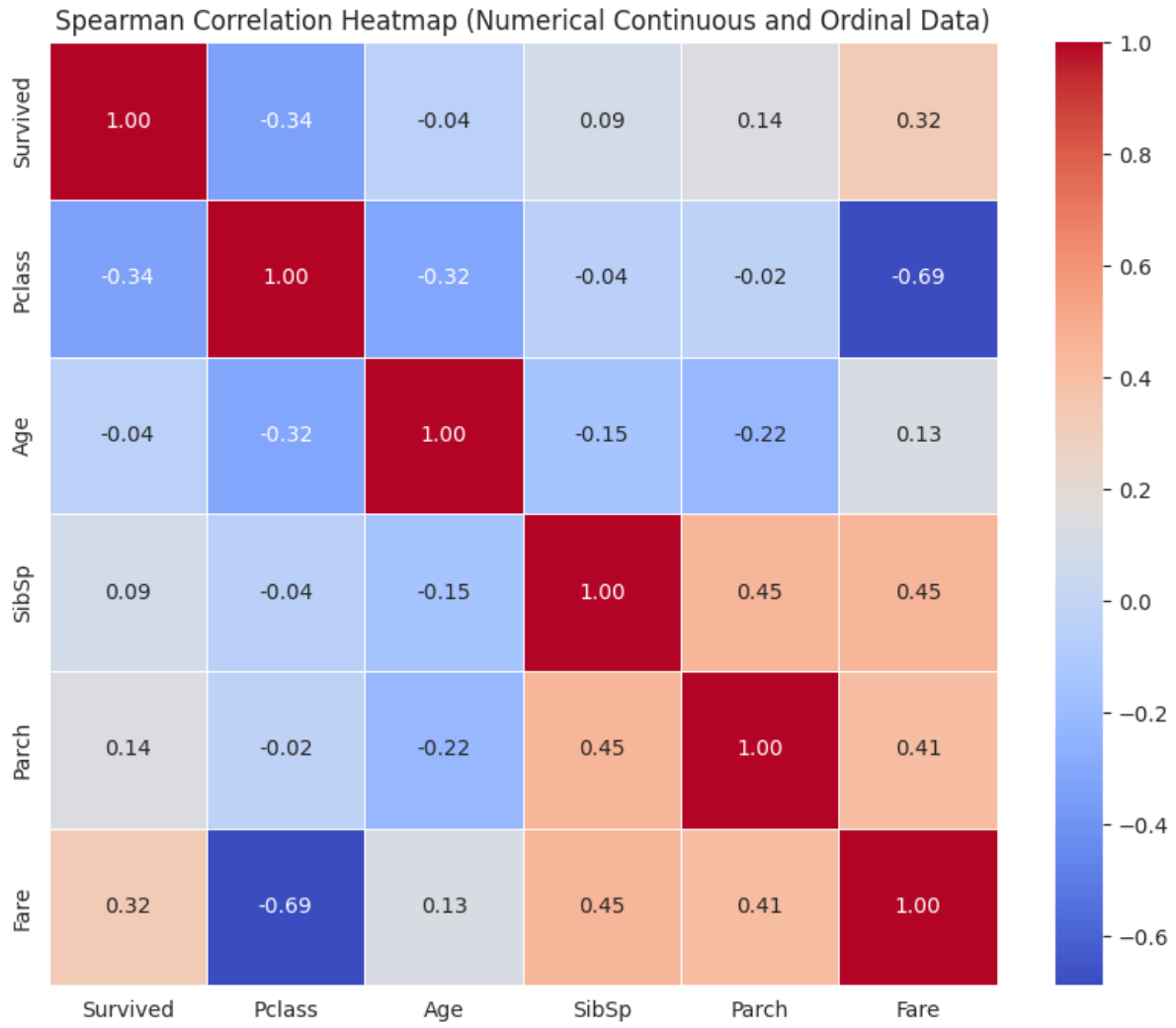
2. **Spearman Rank Correlation:**

- **Definition:** Measures the monotonic relationship between two continuous or ordinal variables.
- **Values:** Range from -1 (perfect negative monotonic relationship) to 1 (perfect positive monotonic relationship), with 0 indicating no monotonic relationship.

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

# Calculate Spearman correlation matrix for numerical continuous and ordinal variables
spearman_corr = titanic[['survived', 'pclass', 'age', 'sibsp', 'parch', 'fare']].corr(method='spearman')

# Plot heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(spearman_corr, annot=True, fmt=".2f", cmap='coolwarm',
```

linewidths=0.5)
plt.title('Spearman Correlation Heatmap (Numerical Continuous and Ordinal Data)')
plt.show()

Press enter or click to view image in full size



Spearman Coefficient based Heatmap for continuous and ordinal variables

Key Takeaways for the survival column we intend to predict:

- ***Positively Correlated with Fare (0.32)***:
- Passengers who paid higher fares tend to have higher survival rates.
- ***Negatively Correlated with Pclass (-0.34):***
- Lower class passengers (higher Pclass value) tend to have lower survival rates.

- ***Weak Positive Correlation with SibSp (0.09):***
- Passengers with more siblings/spouses aboard have a slightly higher chance of survival.
- ***Weak Positive Correlation with Parch (0.14):***
- Passengers with more parents/children aboard also have a slightly higher chance of survival.

Feature Selection

features = ['pclass', 'sex', 'age', 'sibsp', 'parch', 'fare']

X = titanic[features]

y = titanic['survived']

print(X)

	pclass	sex	age	sibsp	parch	fare
0	3	0	-0.530377	1	0	-0.502445
1	1	1	0.571831	1	0	0.786845
2	3	1	-0.254825	0	0	-0.488854
3	1	1	0.365167	1	0	0.420730
4	3	0	0.365167	0	0	-0.486337
...	...	...	...	...	...	...
886	2	0	-0.185937	0	0	-0.386671
887	1	1	-0.737041	0	0	-0.044381
888	3	1	NaN	1	2	-0.176263
889	1	0	-0.254825	0	0	-0.044381
890	3	0	0.158503	0	0	-0.492378

891 rows × 6 columns

Train-Test Split

Split the available data in to train and test dataset

from sklearn.model_selection import train_test_split

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

```
X_train.shape
```

```
(712, 6)
```

```
X_test.shape
```

```
(179, 6)
```

Now we can use the model and train the model with our dataset here we are using logisticregression model and train the model on our train data set

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(random_state=42)
model.fit(X_train, y_train)
```

for validation we use the test dataset

```
y_pred = model.predict(X_test)
```