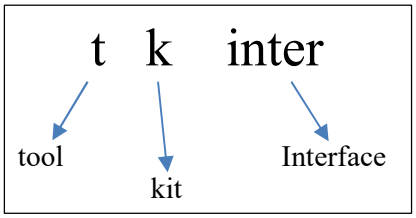
UNIT 4

GU Interface

GUI stands for **Graphical User Interface**.
GUI in Python is a **visual interface** that allows users to interact with a program using **graphical elements** such as **windows, buttons, text boxes, menus, and icons**, instead of typing text commands.

Python provides several libraries for developing GUI applications, with **Tkinter** being the **standard and built-in GUI toolkit** used to create desktop applications easily.

Tkinter module



Tkinter is built-in **standard Graphical User Interface (GUI) Library** in Python.
It provides a set of tools and **widgets** to create **desktop applications** such as windows, buttons, labels, text boxes, and menus.

Features of Tkinter

1. **Built-in module** – No need for separate installation.
2. **Easy to learn and use** – Ideal for beginners.
3. **Cross-platform** – Works on Windows, macOS, and Linux.
4. **Lightweight** – Uses very little system resources.
5. **Customizable widgets** – Supports labels, buttons, menus, etc.

Importing tkinter library

```
import tkinter
```

Window and Widgets in Tkinter

Window: In Tkinter, a **window** is the **main graphical area** where all the components (like buttons, labels, text boxes, etc.) are placed.
The window acts as the **base or container** for all GUI elements.

Syntax for Creating a Window in Tkinter

```
from tkinter import *
window_variable = Tk()
window_variable.title("Window Title")
window_variable.geometry("width x height")
window_variable.mainloop()
```

Example:

```
from tkinter import *
root = Tk()
root.title("My First GUI")
root.geometry("400x300")
root.mainloop()
```

Command	Description
Tk()	Creates the main window (root window).
title("...")	Sets the title of the window.
geometry("WxH")	Sets width and height of window (e.g., "400x300").
mainloop()	Runs the GUI event loop and keeps window open.

Widgets

Widgets are the **GUI elements or components** that appear inside the window, such as **Labels, Buttons, Text boxes, Menus, etc.**
Each widget performs a specific function and helps in building interactive applications.

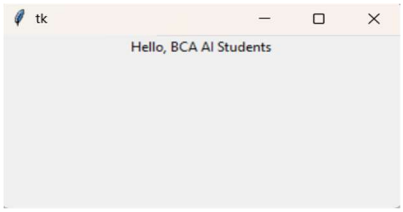
Tkinter Common Widgets

1. Label

A widget used to **display text** in a GUI window.

Example:

```
from tkinter import *
root = Tk()
l=Label(root, text="Hello, Tkinter!")
l.pack()
root.mainloop()
```

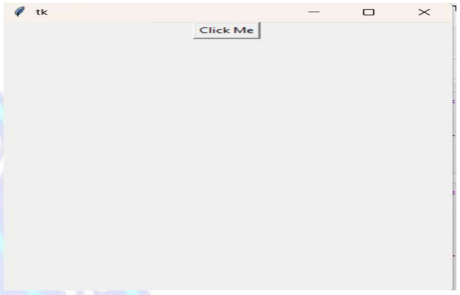


2. Button

A widget that creates a **clickable button** to execute a function when pressed.

Example:

```
from tkinter import *
root = Tk()
b = Button(root, text="Click Me")
b.pack()
root.mainloop()
```

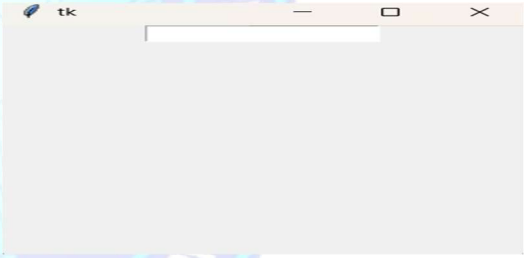


3. Entry

A widget that provides a **single-line text input** field.

Example:

```
from tkinter import *
root = Tk()
e = Entry(root)
e.pack()
root.mainloop()
```



4. Text

A widget that provides a **multi-line text area** for input or display.

Example:

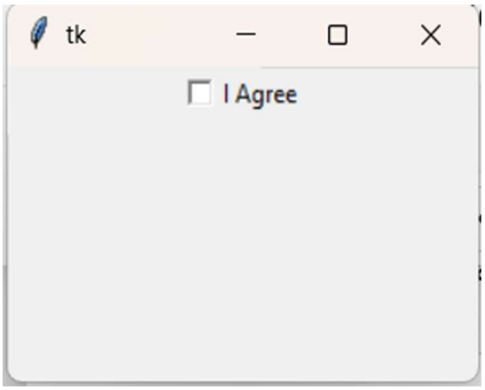
```
from tkinter import *
root = Tk()
t = Text(root, height=5, width=30)
t.pack()
root.mainloop()
```

5. Checkbutton

A widget that displays a **checkbox** to select or deselect an option.

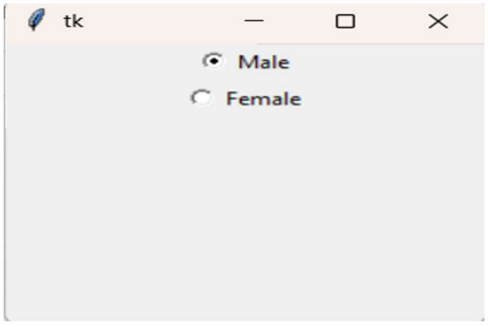
Example:

```
from tkinter import *
root = Tk()
c = Checkbutton(root, text="I Agree")
c.pack()
root.mainloop()
```



6. Radiobutton

A widget that allows the user to **select one option** from multiple choices.



Example:

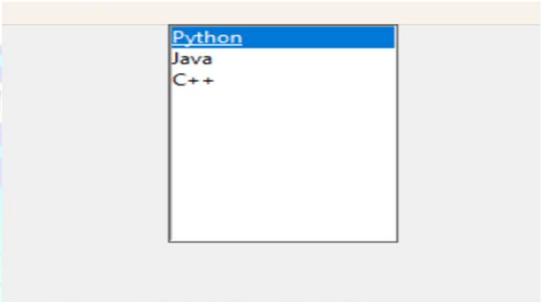
```
from tkinter import *
root = Tk()
r1 = Radiobutton(root, text="Male", value=1)
r1.pack()
r2 = Radiobutton(root, text="Female", value=2)
r2.pack()
root.mainloop()
```

7. Listbox

A widget that displays a **list of items** for selection.

Example:

```
from tkinter import *
root = Tk()
l = Listbox(root)
l.insert(1, "Python")
l.insert(2, "Java")
l.insert(3, "C++")
l.pack()
root.mainloop()
```



Layout Management

Layout Management in Tkinter is the process of **organizing widgets (buttons, labels, entries, etc.) inside a window** so that they are displayed in a structured, readable, and user-friendly way.
Without proper layout management, widgets may **overlap, be misplaced, or appear outside the window**.

Tkinter provides three **layout managers** to control widget layout:

1. **pack()**
2. **grid()**
3. **place()**

pack( ) Geometry/Layout Manager

The **pack() geometry manager** in Tkinter is used to organize and arrange widgets in a **block-wise manner** inside their parent container.

It places widgets **relative to each other**, either **vertically (top/bottom)** or **horizontally (left/right)**, by packing them in the specified order and direction.

Syntax

```
widget_var_name.pack(option1=value1, option2=value2)
```

Common Options

Option	Description
`side`	Where to place the widget (TOP, BOTTOM, LEFT, RIGHT)
`fill`	How widget fills space (X, Y, BOTH, NONE)
`padx, pady`	Adds padding around the widget

Example

```
a=Tk()
a.title("welcome")
a.geometry("500x500")
b1=Button(a,text="Login")
b1.pack(side=LEFT)
b2=Button(a,text="Submit")
b2.pack(side=RIGHT)
a.mainloop()
```

grid( ) Layout Manager

The `grid()` manager arranges widgets in a **tabular structure**, defined by **rows and columns**. Used for **forms, calculators, or complex layouts**.

Syntax

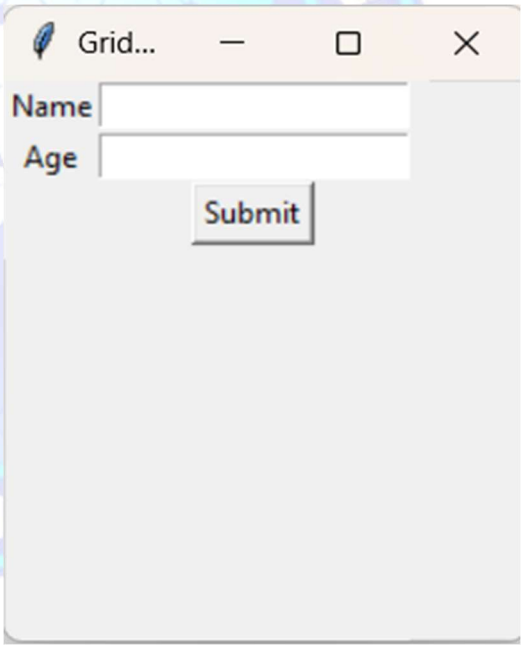
```
widget_var_name.grid(row=row_number, column=column_number, **options)
```

Common Options

Option	Description
row	Row position of widget (starts at 0)
column	Column position of widget (starts at 0)
padx, pady	Padding inside/outside the cell

Example

```
from tkinter import *

root = Tk()
root.title("Grid Example")

l1 = Label(root, text="Name")
l1.grid(row=0, column=0)
e1 = Entry(root)
e1.grid(row=0, column=1)

l2 = Label(root, text="Age")
l2.grid(row=1, column=0)
e2 = Entry(root)
e2.grid(row=1, column=1)

b = Button(root, text="Submit")
b.grid(row=2, column=1)

root.mainloop()
```



place( ) Layout Manager

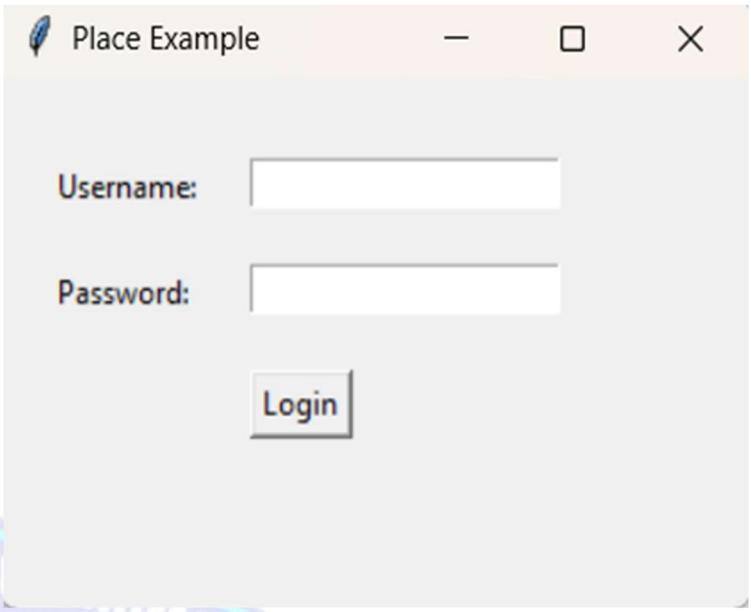
The **place() manager** in Tkinter positions widgets in a window using **absolute coordinates (x, y)** or **relative coordinates**.

Syntax

```
Widget_var_name.place(x=x_coord, y=y_coord, width=w, height=h)
```

Example

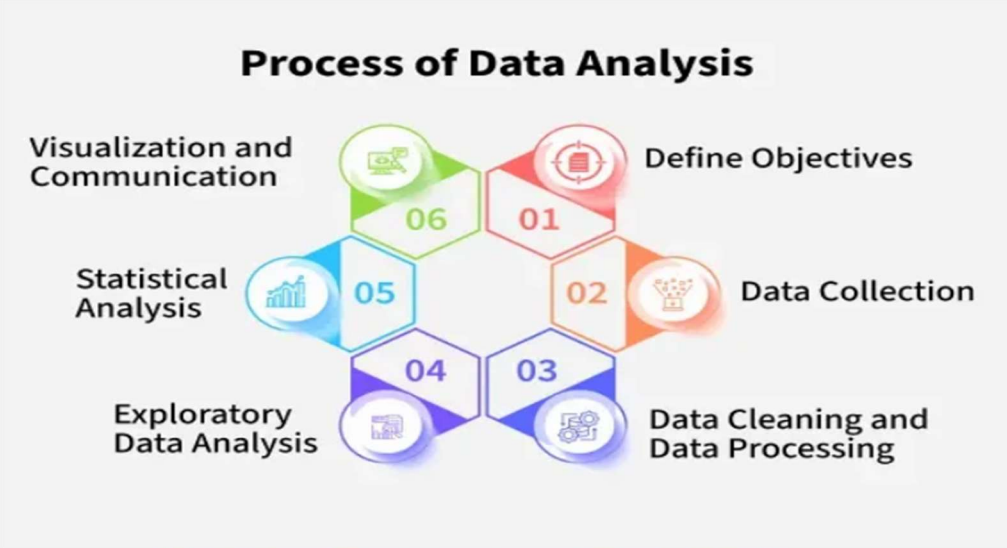
```
from tkinter import *

root = Tk()
root.title("Place Example")
root.geometry("300x200")

l1 = Label(root, text="Username:")
l1.place(x=20, y=30)
e1 = Entry(root)
e1.place(x=100, y=30)

l2 = Label(root, text="Password:")
l2.place(x=20, y=70)
e2 = Entry(root)
e2.place(x=100, y=70)

b = Button(root, text="Login")
b.place(x=100, y=110)

root.mainloop()
```



Note:
Use ***grid()*** for structured layouts, ***pack()*** for simple stacking, ***place()*** for absolute positioning.

Data Analysis

Data Analysis is the technique of collecting, transforming and organizing data to make future predictions and informed data-driven decisions.
It also helps to find possible solutions for a business problem.



Introduction to NumPy

NumPy

NumPy (Numerical Python) is a powerful Python library used for **scientific computing**, **data analysis**, and **numerical operations**.
It supports **multi-dimensional arrays (ndarray)** and performs operations **faster than Python lists** due to optimized C (Programming Language) - based implementation.

Features:

- N-dimensional array objects
- Mathematical, statistical, and algebraic operations
- Broadcasting and vectorization
- Integration with scientific libraries (Pandas, Matplotlib)

NumPy is **not a built-in library** in Python.
That means it doesn't come pre-installed when you install Python.
So, we need to **install it manually** using the command:

```
pip install numpy
```

Here:

- **pip** → is the Python package manager used to install external libraries.
- **install** → tells pip to download and set up the package.
- **numpy** → is the name of the library we want to install.

NumPy Arrays

A **NumPy array** (ndarray) is a **homogeneous** collection of elements stored in **contiguous memory** and used for numerical operations.

Array Creation in NumPy

`np.array()` – Creating Array from List/Tuple

This function is used to create a NumPy array from an existing Python list or tuple.

Syntax:

```
np.array(sequence)
```

```
sequence: list or tuple or set
```

Example:

```
import numpy as np
a = np.array([1, 2, 3, 4])
print(a)
print(type(a))
```

Output:
```
[1 2 3 4]
<class 'numpy.ndarray'>
```

Note: In general, lists are commonly used for creating arrays.

Creating a Two-Dimensional Array from List of Lists/Tuples
This function is also used to create a **2D NumPy array** from an existing list (or tuple) of lists (or tuples).

Syntax:
```
np.array([sequence1, sequence2, ...])
```

Example:

```
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a)
print(type(a))
```

Output:
```
[[1 2 3]
 [4 5 6]]
<class 'numpy.ndarray'>
```

Note: We can create arrays of any **number of dimensions (n-dimensional arrays)** using **NumPy**

`np.zeros()` – Array of Zeros
The **`np.zeros()`** function creates a new array of a specified shape filled entirely with the value zero. It is commonly used to initialize arrays before assigning values or performing computations.

Syntax:
```
np.zeros(shape)
```

shape means: row_value and column_value

Example

```
import numpy as np
z = np.zeros((2,3))
print(z)
```

Output
```
[[0. 0. 0.]
 [0. 0. 0.]]
```

Note: We can also give the shape value using tuples, lists, or sets, but commonly we use tuples.

We can also pass only the shape value **2**, and we get:

```
import numpy as np
z = np.zeros(2)
print(z)
```

Output:
```
[0. 0.]
```

np.ones() – Array of Ones

This function is used to generate an array of a specified **shape** where all the elements are initialized to one. is particularly useful when a default value of one is needed for computations.

Syntax
```
np.ones(shape)
```

Example

```
import numpy as np
o = np.ones(5)
print(o)
```

Output
[1. 1. 1. 1. 1.]

np.full() – Array with Constant Value

The np.full() function creates an array of the specified shape and fills it with a constant value provided by the user. This is useful when initializing arrays with a specific repeated value.

Syntax

```
np.full(shape, value)
```

Example

```
import numpy as np
f = np.full((2,2), 7)
print(f)
```

Output

```
[[7 7]
 [7 7]]
```

Operations on Arrays in NumPy

Arithmetic Operations

Arithmetic operations perform mathematical calculations element-wise on arrays (each element is operated with the corresponding element of the other array).

Syntax

```
array1 + array2
array1 - array2
array1 * array2
array1 / array2
array1 // array2
array1 % array2
array1 ** array2
```

Element-wise Operation	Operator	NumPy Function
Addition	+	np.add(a,b)
Subtraction	-	np.subtract(a,b)
Multiplication	*	np.multiply(a,b)
Division	/	np.divide(a,b)
Exponentiation	**	np.power(a,b)
Modulus	%	np.mod(a,b)

Example

```
import numpy as np
a = np.array([10, 20, 30])
b = np.array([1, 2, 3])
print("Addition:", a + b)
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b)
print("Floor Division:", a // b)
print("Modulus:", a % b)
print("Power:", a ** b)
```

Output:

```
Addition: [11 22 33]
Subtraction: [ 9 18 27]
Multiplication: [10 40 90]
Division: [10. 10. 10.]
Floor Division: [10 10 10]
Modulus: [0 0 0]
Power: [ 10 400 27000]
```

Note:
All arithmetic operations in NumPy are **element-wise**, meaning they are applied individually to each corresponding element of the arrays.

Relational (Comparison) Operations

Relational operations compare two arrays element-wise and return a Boolean array.

Syntax

```
array1 > array2
array1 < array2
array1 == array2
array1 != array2
```

Example

```
a = np.array([10, 20, 30])
b = np.array([15, 20, 25])
print(a > b)
print(a == b)
```

Output

```
[False False  True]
[False  True False]
```

Logical Operations

Logical operations evaluate Boolean expressions on arrays and return True/False results.

Syntax

```
np.logical_and(array1, array2)
np.logical_or(array1, array2)
np.logical_not(array)
```

Example

```
a = np.array([True, False, True])
b = np.array([False, False, True])
print(np.logical_and(a, b))
print(np.logical_or(a, b))
print(np.logical_not(a))
```

Output

```
[False False  True]
[True False  True]
[False  True False]
```

Statistical Operations

Aggregate functions compute summary values for the entire array (like sum, mean, max, etc.)

Syntax

```
np.sum(array)
np.mean(array)
np.max(array)
np.min(array)
np.std(array)
np.median(array)
```

Example

```
a = np.array([10, 20, 30, 40])
print(np.sum(a))
print(np.mean(a))
print(np.max(a))
print(np.min(a))
```

Output

```
100
25.0
40
10
```

Array Indexing & Slicing

Indexing accesses a single element; slicing retrieves a range of elements from an array.

Syntax

```
array[index]
array[start:end]
array[start:end:step]
```

Example

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

print(a[2])
print(a[1:4])
```

Output

```
30
[20 30 40]
```

Array Manipulations in NumPy

Array manipulation in NumPy refers to operations that **change the shape, structure, or orientation** of an array without modifying its data. These operations help in reshaping, flattening, transposing, joining, and splitting arrays for efficient data handling.

shape

The shape attribute returns the dimensions (rows & columns) of an array as a tuple, indicating the arrangement of elements.

Syntax

```
array_name.shape
```

Example

```
import numpy as np
arr = np.array([[1,2,3],[4,5,6]])
print(arr.shape)
```

Output

```
(2, 3)
```

reshape()

reshape() changes the shape of an array to a specified dimension without changing the original data. The total number of elements must remain the same.

Syntax

```
array_name.reshape(new_rows, new_columns)
```

Example

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

Output

```
[[1 2 3]
 [4 5 6]]
```

flatten()

flatten() converts a multi-dimensional array into a one-dimensional array and returns a **new copy** of the data.

Syntax

```
array_name.flatten()
```

Example

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

Output

```
[1 2 3 4]
```

4. ravel()

ravel() converts a multi-dimensional array into a one-dimensional array similar to flatten(), but it returns a **view (reference)** of the original array if possible.

Syntax

```
array_name.ravel()
```

Example

```
arr = np.array([[10,20],[30,40]])
rv = arr.ravel()
print(rv)
```

Output

```
[10 20 30 40]
```

5. transpose()

transpose() reverses the rows and columns of an array. It is widely used in matrix operations and data transformation.

Syntax

```
array_name.transpose()
```

Example

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

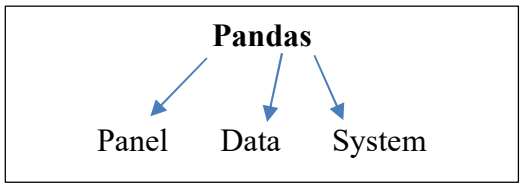
Output

```
[[1 4]
 [2 5]
 [3 6]]
```

Pandas

Pandas is a powerful **external Python library** used for data analysis, data manipulation, and data cleaning. It provides easy-to-use data structures like **Series** and **DataFrame** for handling and analyzing structured data efficiently.

The name **"pandas"** comes from the term **"Panel Data"**, which refers to multidimensional structured data used in statistics and econometrics.



Pandas is not a built-in Python library, so you need to **install it manually** before using it.

Command to Install Pandas

You can install it using **pip** (Python install Package):

```
pip install pandas
```

After Installation, Import It

```
import pandas as pd
```

History of Pandas

- Pandas was created by **Wes McKinney**.
- He developed it in the year **2008**.
- Wes McKinney is an **American software developer and data scientist**.
- He created Pandas while working at **Tesco Capital Management**, a financial firm.
- The main goal of creating Pandas was to **make data analysis and manipulation easier in Python**.
- He is also the **author of the book "Python for Data Analysis."**

Features of Pandas

1. **Handling Large Data**
   Pandas can efficiently handle and process large volumes of data.
2. **Data Cleaning**
   It helps in cleaning data by filling empty cells, correcting wrong entries, and removing duplicates.
3. **Data Grouping and Visualization**
   Supports grouping of data and visualizing it easily using libraries like Matplotlib.
4. **Merging and Joining Datasets**
   Allows combining multiple datasets based on common columns or keys.
5. **Reading and Writing Data**
   Supports reading from and writing to various file formats such as CSV, Excel, and SQL.

Applications of Pandas

1. **Financial Analysis**
   Used for analysing financial data like stock prices, profits, and expenses.
2. **Machine Learning**
   Helps in data preprocessing and preparing datasets for training models.
3. **Data Visualization**
   Used to visualize data patterns and trends using graphs and charts.
4. **Business and Marketing Analysis**
   Useful for analysing sales, customer behaviour, and marketing performance.
5. **Healthcare Data Analysis**
   Helps in managing and analysing medical data for research and treatment insights.

Series in Pandas

A **Series** in Pandas is a **one-dimensional labeled array** that can hold data of any type (integers, floats, strings, etc.), with each element having an associated **index label**.

It is similar to a **column in an Excel sheet** or a **single list** in Python, but with an added **index** for each element means label.

Syntax for Creating a Series:

```
pandas.Series(data, index)
```

Where:

- **data** → The data to be stored (list, array, and dictionary).
- **index** → (Optional) Labels for each element.

Example 1: Creating a Series from a List

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

Output:

```
0    10
1    20
2    30
3    40
dtype: int64
<class 'pandas.core.series.Series'>
```

Here, `0, 1, 2, 3` are **index labels** automatically assigned by Pandas.

Example 2: Creating a Series with Custom Index

```
import pandas as pd
data = [100, 200, 300]
s = pd.Series(data, index=['a', 'b', 'c'])
print(s)
```

Output:

```
a    100
b    200
c    300
dtype: int64
```

Here, `a, b, c` are **index labels** assigned by using `index` argument are passed in list.

We can also create series using dictionary

Example 3: Creating a Series from a Dictionary

```
import pandas as pd
data = {'Name': "Raju", 'Age': 24, 'Course': 'BCA AI'}
s = pd.Series(data)
print(s)
```

Output:

```
Name      Raju
Age         24
Course  BCA AI
dtype: Object
```

DataFrame in Pandas

A **DataFrame** is a **two-dimensional labeled data structure** in Pandas that stores data in rows and columns, where each column can contain different data types such as integers, floats, or strings.

Syntax:

```
pandas.DataFrame(data, index, columns)
```

Where:

- **data** → Data can come from a dictionary, list, array, or another DataFrame.
- **index** → (Optional) Row labels.
- **columns** → (Optional) Column labels.

Examples

We can create DataFrame using Dictionary, List and Tuples.

1. From a Dictionary

```
import pandas as pd
data = {'Name': ['Raju', 'Anita', 'Ravi'],
        'Age': [25, 28, 22],
        'City': ['Mysore', 'Bangalore', 'Mandya']}
df = pd.DataFrame(data)
print(df)
print(type(df))
```

Output:

```
    Name  Age       City
0   Raju   25     Mysore
1  Anita   28  Bangalore
2   Ravi   22     Mandya
pandas.core.frame.DataFrame
```

Easy and most commonly used method.

2. From a List of Lists

```
data = [['Raju', 25, 'Mysore'],
        ['Anita', 28, 'Bangalore'],
        ['Ravi', 22, 'Mandya']]

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

Output:

```
    Name  Age       City
0   Raju   25     Mysore
1  Anita   28  Bangalore
2   Ravi   22     Mandya
```

Useful when data is organized row-wise.

3. From a Tuple of Tuples

```
data = (('Raju', 25, 'Mysore'),
        ('Anita', 28, 'Bangalore'),
        ('Ravi', 22, 'Mandya'))
df = pd.DataFrame(data, columns=['Name', 'Age',
'City'])
print(df)
```

Output:

```
    Name  Age       City
0   Raju   25     Mysore
1  Anita   28  Bangalore
2   Ravi   22     Mandya
```

Creating a DataFrame from Excel file

Excel file

An **Excel file** is a **spreadsheet file** used to store, organize, and analyze data in tabular form. It consists of **rows and columns**, where data can include text, numbers, or formulas. Excel files usually have extensions like **.xls** or **.xlsx**.

Syntax for Creating DataFrame from Excel file

```
pd.read_excel("file_name.xlsx", sheet_name='Sheet_No')
```

Where,

file_name.xlsx → name or path of Excel file
sheet_name → optional; specifies which sheet to read (default is the first one)

Consider for Excel file employee.xlsx

The below excel file contains employee details with six columns — **Name, Age, Department, Salary, Experience (Years), and City.**
Each row represents an employee's record.

	A	B	C	D	E	F
1	**Name**	**Age**	**Department**	**Salary**	**Experience (Years)**	**City**
2	Manoj	27	Operations	33438	6	Chennai
3	Manoj V	34	IT	50940	7	Hyderabad
4	Pooja	29	HR	50612	7	Pune
5	Asha	21	Operations	36595	7	Bangalore
6	Ravi	30	HR	30515	10	Chennai
7	Pooja	25	Support	35581	3	Pune
8	Kiran	32	Operations	41701	4	Chennai
9	Sanjay	35	IT	54745	4	Bangalore
10	Pooja	35	Sales	41151	7	Chennai
11	Rohit	35	IT	59299	9	Pune
12	Nisha	23	Operations	43460	5	Chennai
13	Gaurav	32	Marketing	57092	9	Bangalore
14	Priya	21	Marketing	44675	7	Hyderabad
15	Divya	29	Operations	37199	6	Mysore
16	Harish	25	Support	42226	8	Chennai
17	Manoj	30	Marketing	51512	6	Hyderabad
18	Kavya	30	HR	35857	2	Chennai
19	Nisha	27	Finance	36018	8	Bangalore
20	Kavya	27	Support	42180	9	Pune
21	Rohit	27	Marketing	51290	6	Mysore
22	Vijay	31	IT	41962	5	Chennai
23	Kavya	24	Finance	32619	5	Pune

Example

```
import pandas as pd
df = pd.read_excel("Employee.xlsx")
print(df)
```

Output:

```
        Name  Age  Department  Salary  Experience (Years)       City
0      Manoj   27  Operations   33438                   6    Chennai
1    Manoj V   34          IT   50940                   7  Hyderabad
2      Pooja   29          HR   50612                   7       Pune
3       Asha   21  Operations   36595                   7  Bangalore
4       Ravi   30          HR   30515                  10    Chennai
5      Pooja   25     Support   35581                   3       Pune
6      Kiran   32  Operations   41701                   4    Chennai
7     Sanjay   35          IT   54745                   4  Bangalore
8      Pooja   35       Sales   41151                   7    Chennai
9      Rohit   35          IT   59299                   9       Pune
10     Nisha   23  Operations   43460                   5    Chennai
11    Gaurav   32   Marketing   57092                   9  Bangalore
```

Creating DataFrame using CSV file

CSV file

A **CSV file** (Comma-Separated Values file) is a **plain text file** used to store tabular data such as numbers and text. Each line in the file represents a row, and the values in each row are separated by **commas**.

Consider for CSV file student.csv

The below CSV file consists of **50 student records** with columns — **Name, Age, Course, and City.**

```
Name,Age,Course,City
Manoj,21,BCA,Mysore
Pooja,22,BSc,Bangalore
Ravi,23,BCom,Chennai
Nisha,20,BA,Hyderabad
Kiran,24,BBA,Pune
Divya,21,BCA,Mangalore
Asha,22,BSc,Coimbatore
Rohit,23,BCom,Delhi
Kavya,20,BA,Mumbai
Sanjay,25,BBA,Chennai
Gaurav,24,BCA,Bangalore
Harish,22,BSc,Mysore
Priya,23,BCom,Hyderabad
Vijay,21,BA,Pune
Sneha,20,BBA,Chennai
Ajay,24,BCA,Coimbatore
Meena,25,BSc,Bangalore
Ankit,22,BCom,Delhi
Swathi,23,BA,Mangalore
Rakesh,21,BBA,Pune
Deepa,24,BCA,Mysore
Vivek,22,BSc,Hyderabad
Chitra,25,BCom,Chennai
```

Syntax for Creating DataFrame from CSV file

```
pd.read_csv("file_name.csv")
```

Where,

file_name.csv → name or path of csv file

Example

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

Output:

	Name	Age	Course	City
0	Manoj	21	BCA	Mysore
1	Pooja	22	BSc	Bangalore
2	Ravi	23	BCom	Chennai
3	Nisha	20	BA	Hyderabad
4	Kiran	24	BBA	Pune
5	Divya	21	BCA	Mangalore
6	Asha	22	BSc	Coimbatore
7	Rohit	23	BCom	Delhi
8	Kavya	20	BA	Mumbai
9	Sanjay	25	BBA	Chennai

A DataFrame in pandas can be easily saved as an Excel or CSV file using the **to_excel()** and **to_csv()** methods. This means we can easily create **Excel** and **CSV** files using a **DataFrame** by following methods.

1. to_excel()

- Used to **save a DataFrame to an Excel file** (.xlsx format).

Syntax:

```
dataframe_name.to_excel("file_name.xlsx", index=False)
```

- **Parameters**:
  - ***file_name*** → name or path of the Excel file.
  - ***index=False*** → prevents saving row numbers.

Example:

```
import pandas as pd
data = {'Name': ['Raju', 'Anita'], 'Age': [25, 28]}
df = pd.DataFrame(data)
df.to_excel("students.xlsx", index=False)
```

2. to_csv()

- Used to **save a DataFrame to a CSV file** (.csv format).

Syntax:

```
Dataframe_name.to_csv("file_name.csv", index=False)
```

- **Parameters**:
  - ***file_name*** → name or path of the CSV file.
  - ***index=False*** → prevents saving row numbers.

Example:

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

Note:

Both methods allow you to **easily export DataFrames** to external files for sharing, reporting, or further processing.

Operations on DataFrame

Consider an Example DataFrame

```
import pandas as pd
df = pd.DataFrame({
    'Name': ['Asha', 'Ravi', 'Manu'],
    'Age': [23, 27, 25],
    'City': ['Mysore', 'Bangalore', 'Mysore']
})

print(df)
```

Output

```
   Name  Age       City
0  Asha   23     Mysore
1  Ravi   27  Bangalore
2  Manu   25     Mysore
```

1. Viewing Data

a. df.head(n) – Shows first *n* rows

Syntax:

```
df.head(2)
```

Output:

```
   Name  Age     City
0  Asha   23   Mysore
1  Ravi   27  Bangalore
```

b. df.tail(n) – Shows last *n* rows

Syntax:

df.tail(2)

Output:

```
   Name  Age     City
1  Ravi   27  Bangalore
2  Manu   25     Mysore
```

c. df.shape – Returns number of rows & columns

Syntax:

```
df.shape
```

Output:

```
(3, 3)
```

d. df.info() – Summary of DataFrame

Syntax:

```
df.info()
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (3 columns):
 Name  3 non-null object
 Age   3 non-null int64
```

```
 City  3 non-null object
```

e. df.describe() – Statistical summary

Syntax:

```
df.describe()
```

Output:

```
       Age
count   3.0
mean   25.0
min    23.0
max    27.0
```

2. Selecting Data
a. Select column

Syntax:

```
df['Age']
```

```
Output:
0    23
1    27
2    25
Name: Age, dtype: int64
```

b. Select multiple columns

Syntax:

```
df[['Name', 'City']]
```

Output:

```
  Name       City
0  Asha     Mysore
1  Ravi  Bangalore
2  Manu     Mysore
```

c. Select row using index

Syntax:

```
df.iloc[1]     # by position
```

Output:

```
Name        Ravi
Age           27
City    Bangalore
Name: 1, dtype: object
```

3. Filtering Data
a. Single condition

Syntax:

```
df[df['Age'] > 25]
```

Output:

```
  Name  Age       City
1  Ravi   27  Bangalore
```

b. Multiple conditions

Syntax:

```
df[(df['Age'] > 25) & (df['City'] == 'Mysore')]
```

Output:

```
Empty DataFrame
Columns: [Name, Age, City]
Index: []
```

4. Adding / Modifying Columns
a. Modify column

Syntax:

```
df['Age'] = df['Age'] + 1
print(df)
```

Output:

```
  Name  Age       City
0  Asha   24     Mysore
1  Ravi   28  Bangalore
2  Manu   26     Mysore
```

b. Add new column

Syntax:

```
df['Salary'] = df['Age'] * 1000
print(df)
```

Output:

```
   Name  Age       City  Salary
0  Asha   24     Mysore   24000
1  Ravi   28  Bangalore   28000
2  Manu   26     Mysore   26000
```

5. Deleting Columns / Rows
Delete column

Syntax:

```
df.drop('Salary', axis=1)
```

Output:

```
  Name  Age       City
0  Asha   24     Mysore
1  Ravi   28  Bangalore
2  Manu   26     Mysore
```

6. Sorting Data
Sort by Age
Syntax:

```
df.sort_values(by='Age', ascending=True)
```

Output:

```
  Name  Age       City
0  Asha   24     Mysore
2  Manu   26     Mysore
1  Ravi   28  Bangalore
```

7. Statistical / Aggregation Functions
Common functions

```
df['Age'].mean()
```

Output:

```
26.0
```

```
df['Age'].sum()
```

```
Output:
78
```

```
df['Age'].min()
```

```
Output:
24
```

```
df['Age'].max()
```

```
Output:
28
```

Value counts

```
df['City'].value_counts()
```

```
Output:
Mysore      2
Bangalore   1
Name: City, dtype: int64
```

8. Reading / Writing Data

- **Read CSV:** `pd.read_csv("file.csv")`
- **Read Excel:** `pd.read_excel("file.xlsx")`
- **Save CSV:** `df.to_csv("file.csv", index=False)`
- **Save Excel:** `df.to_excel("file.xlsx", index=False)`

9. Handling Missing Values

In pandas, **missing values** in a DataFrame can appear as **NaN** (Not a Number). You can **detect, handle, and fill or drop** them. Here's a simple guide:

A. Detect Missing Values

```
import pandas as pd
data = {
    'Name': ['Raju', 'Anita', 'Ravi'],
    'Age': [25, None, 22],
    'City': ['Mysore', 'Bangalore', None]
}
df = pd.DataFrame(data)
print(df.isnull())       # True where missing
print(df.isnull().sum()) # Count of missing values per column
```

Output:

```
   Name    Age   City
0  False  False  False
1  False   True  False
2  False  False   True


Age     1
City    1
Name    0
dtype: int64
```

B. Handle Missing Values

a) Drop Missing Values

```
df_clean = df.dropna()  # Drops rows with any missing value
print(df_clean)
```

b) Fill Missing Values

```
df['Age'].fillna(df['Age'].mean(), inplace=True)  # Fill Age with mean
df['City'].fillna('Unknown', inplace=True)        # Fill City with 'Unknown'
print(df)
```

Output after filling:

```
    Name   Age       City
0   Raju  25.0     Mysore
1  Anita  23.5  Bangalore
2   Ravi  22.0    Unknown
```

Note

- isnull() or isna() → check missing values
- dropna() → remove missing rows
- fillna() → replace missing values with a value (mean, median, mode, or custom)

10. Handling Duplicate Values

A. Detect Duplicate Rows

```
import pandas as pd

data = {
    'Name': ['Raju', 'Anita', 'Ravi', 'Raju'],
    'Age': [25, 28, 22, 25],
    'City': ['Mysore', 'Bangalore', 'Mandya', 'Mysore']
}
df = pd.DataFrame(data)
# Check for duplicate rows
print(df.duplicated())       # True for duplicates
print(df.duplicated().sum()) # Count of duplicate rows
```

Output:

```
0    False
1    False
2    False
3     True
dtype: bool

1
```

2. Remove Duplicate Rows

```
df_clean = df.drop_duplicates()
print(df_clean)
```

Output:

```
  Name  Age       City
0   Raju   25     Mysore
1  Anita   28  Bangalore
2   Ravi   22     Mandya
```

3. Detect Duplicate Columns

```
data = {
    'Name': ['Raju', 'Anita', 'Ravi'],
    'Age': [25, 28, 22],
    'Age': [25, 28, 22]  # Duplicate column name
}

df = pd.DataFrame(data)
```

```
# Check columns
print(df.columns)  # Only unique columns are kept automatically
```

Output:

```
Index(['Name', 'Age'], dtype='object')
Note: pandas automatically keeps the last column if duplicate column names
exist.
```

Data Visualisation

Data Visualization is the process of representing data in a graphical or visual format, like charts, graphs, or plots, to make it easier to understand, analyze, and interpret.

Instead of analysing raw numbers or large datasets, visualization helps in identifying patterns, trends, and relationships quickly. It plays a crucial role in data analysis, business intelligence, and decision-making by converting complex data into a format that is easy to comprehend.

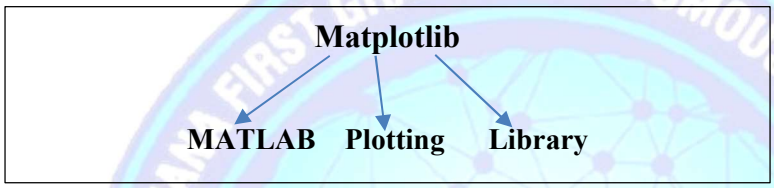
In Python, popular libraries for data visualization include **Matplotlib, Seaborn, and Plotly**, which allow creating a variety of plots like line charts, bar charts, histograms, scatter plots, pie charts, and heatmaps.

Features

- Makes complex data simple to understand.
- Helps find trends, patterns, and outliers.
- Used in business, research, and data analysis.

Matplotlib

Matplotlib is a Python external library used for creating 2D plots, charts, and graphs, similar to MATLAB, to visualize data in an easy and effective way.



MATLAB Plotting Library

- **MAT** → Refers to MATLAB, as Matplotlib was originally designed to replicate MATLAB-style plotting in Python.
- **Plot** → Refers to its main purpose: creating plots and visualizations.
- **Lib** → Stands for library, meaning it is a collection of tools for plotting.

Since Matplotlib is an external library, we need to install Matplotlib before using it.
we can install **Matplotlib** in Python using **pip**, which is Python's package manager.

Command to Install Matplotlib

```
pip install matplotlib
```

Pyplot in Matplotlib

`pyplot` is a **module in Matplotlib** that provides a collection of functions for creating and controlling plots in Python.

Different Types of Charts Using Pyplot

1. Line Chart

A line chart is a graphical representation of data points connected by straight lines, typically used to visualize trends, changes, or continuous data over a period of time or ordered categories.

Syntax:

```
plt.plot(x_values, y_values, style_options)
plt.title("Plot Title")
plt.xlabel("X-axis Label")
plt.ylabel("Y-axis Label")
plt.legend(["Label1", "Label2"])
plt.grid(True)
plt.show()
```

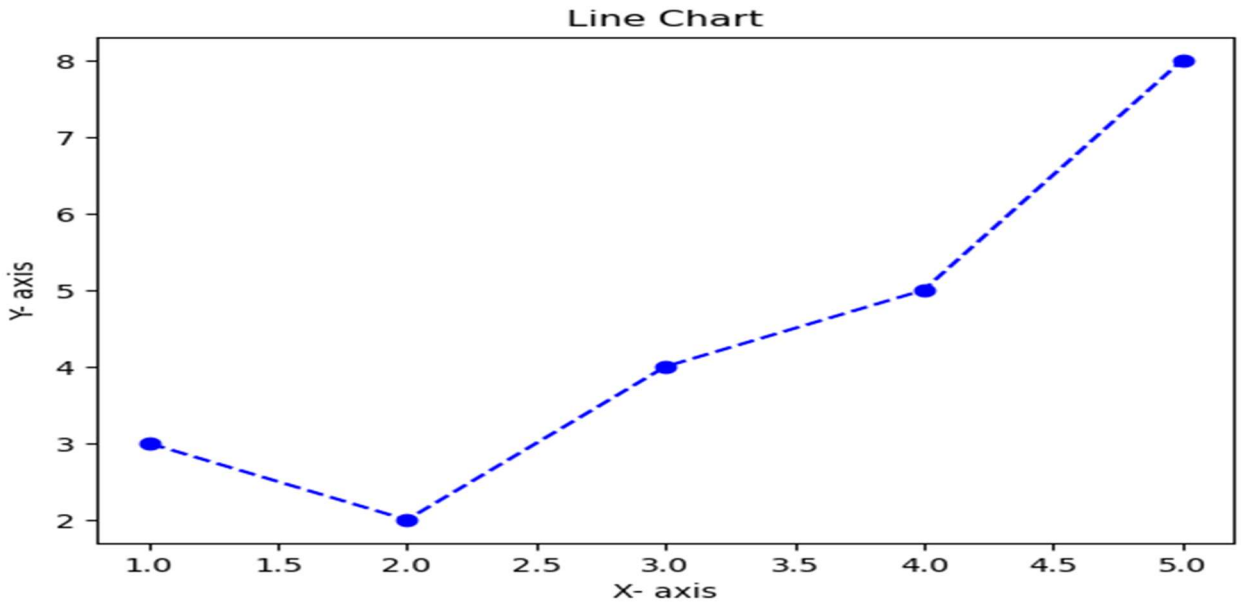
Explanation:

- **plt.plot()** → Draws a line chart with points marked by circles (marker='o') and a solid line.
- **plt.title()** → Sets the chart title.
- **plt.xlabel()** / **plt.ylabel()** → Labels the axes.
- **plt.grid(True)** → Adds grid lines.
- **plt.legend()** → Shows the legend for the line.
- **plt.show()** → Displays the plot.

Example:

```
import matplotlib.pyplot as plt
x_values = [1, 2, 3, 4, 5]
y_values = [3, 2, 4, 5, 8]

plt.plot(x_values, y_values, color='blue', marker='o', linestyle='-')
plt.title("Line Chart Example")

plt.xlabel("X-axis")
plt.ylabel("Y-axis")

plt.show()
```

Output:



2. Bar Chart

A bar chart is a graphical representation of data using rectangular bars, where the length or height of each bar is proportional to the corresponding value. It is primarily used for comparing quantities across different categories.

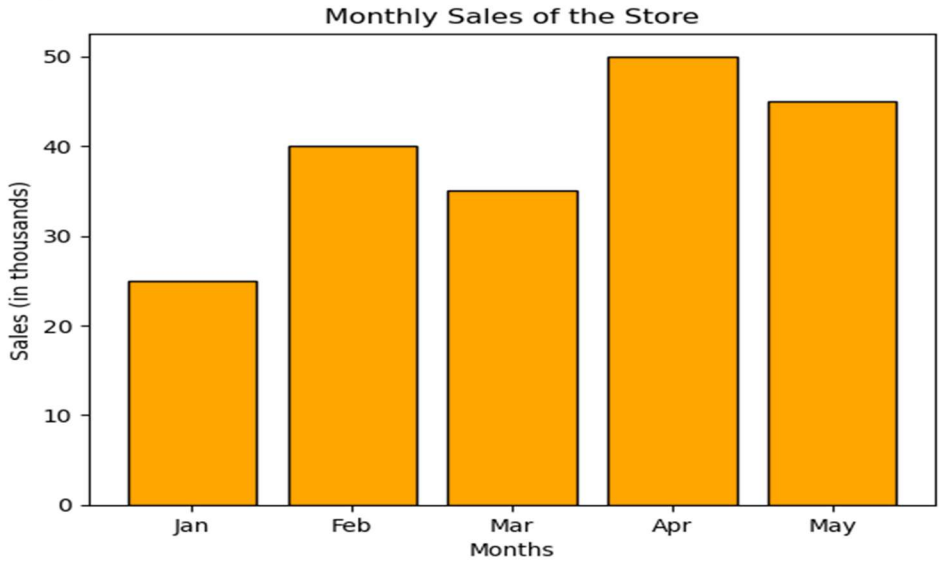
Syntax:

```
plt.bar(x_values, y_values, color='color', edgecolor='color')
plt.title("Chart Title")
plt.xlabel("X-axis Label")
plt.ylabel("Y-axis Label")
plt.legend(["Label1", "Label2"])
plt.grid(True)
plt.show()
```

Example:

```
import matplotlib.pyplot as plt

months = ['Jan', 'Feb', 'Mar', 'Apr', 'May']
sales = [25, 40, 35, 50, 45]

plt.bar(months, sales, color='orange', edgecolor='black')
plt.title("Monthly Sales of the Store")
plt.xlabel("Months")
plt.ylabel("Sales (in thousands)")

plt.grid(True)
plt.show()
```

Output:



3. Histogram

A histogram is a graphical representation of data that shows the distribution of numerical values by dividing them into intervals (bins) and counting the number of values in each bin.

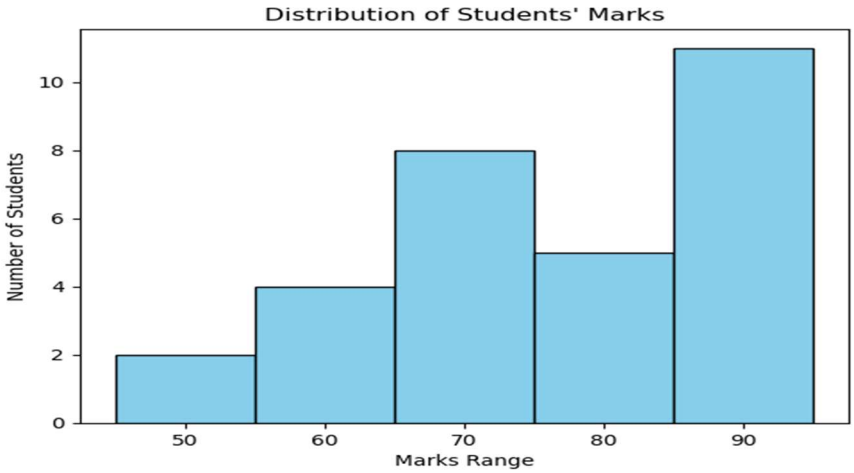
Syntax:

```
plt.hist(data, bins=number_of_bins, color='color', edgecolor='color',
alpha=transparency)
plt.title("Histogram Title")
plt.xlabel("X-axis Label")
plt.ylabel("Frequency")
plt.grid(True)
plt.show()
```

Example:

```
import matplotlib.pyplot as plt

marks = [55, 75, 90, 45, 65, 90, 90, 70, 60, 75,
         80, 85, 50, 65, 77, 89, 95, 68, 70, 85,
         60, 72, 85, 90, 55, 65, 78, 88, 92, 70]

plt.hist(marks, bins=5, color='skyblue', edgecolor='black')
plt.title("Distribution of Students' Marks")
plt.xlabel("Marks Range")
plt.ylabel("Number of Students")
plt.show()
```

Output:



Explanation of Bin

A **bin** is a **range of values** used to group continuous data in a histogram.
To calculate bins:

1. Find **minimum** and **maximum** values.
2. Compute the **range = max – min**.
3. Divide the range by the **number of bins** to get **bin width**.

Example: If data is from **45 to 95** and bins = **5**,

- Range = 50
- Bin width = 50 / 5 = 10
- Bins → 45–55, 55–65, 65–75, 75–85, 85–95

Each bin shows **how many values fall in that interval**.
Bars in a histogram **touch each other** because bins represent continuous data.

4. Pie Chart

A pie chart is a graphical representation of data in the form of a circle divided into slices, where each slice shows the proportion or percentage of a category relative to the whole.

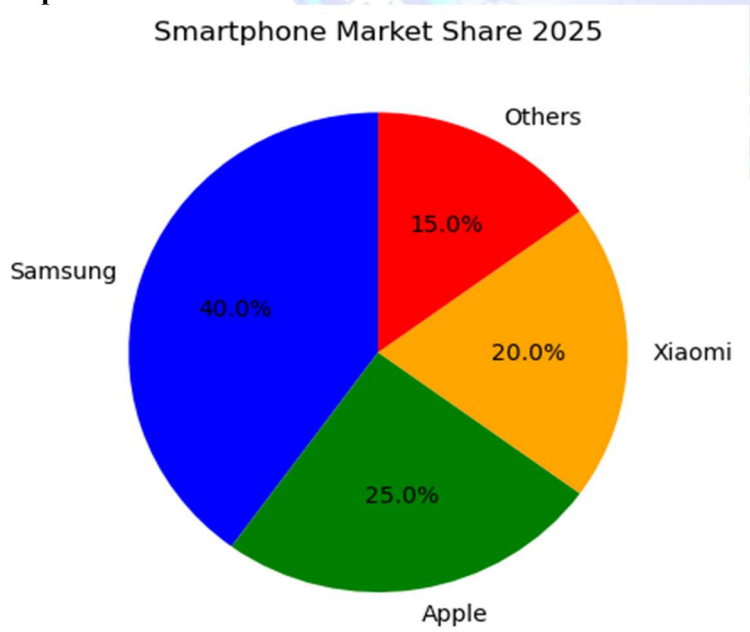
Syntax:

```
plt.pie(sizes, labels=labels, autopct='%1.1f%%', colors=color_list,
        startangle=angle)

plt.title("Pie Chart Title")
plt.show()
```

Example:

```
import matplotlib.pyplot as plt

sizes = [40, 25, 20, 15]
labels = ['Samsung', 'Apple', 'Xiaomi', 'Others']

plt.pie(sizes, labels=labels, autopct='%1.1f%%', colors=['blue', 'green',
'orange', 'red'], startangle=90)

plt.title("Smartphone Market Share 2025")

plt.show()
```

Output:



Explanation

- Matplotlib is imported to draw the pie chart.
- sizes holds the percentage values and labels contains the company names.
- plt.pie() creates the pie chart using these values.
- autopct shows the percentage on each slice.
- colors assigns different colors to each section.
- startangle=90 rotates the chart for a neat look.
- A title is added for description.
- plt.show() displays the chart.

How Percentages Are Calculated in the Pie Chart

1. The values in the sizes list represent the parts of the whole.
   Example: [40, 25, 20, 15].
2. First, all values are added to find the total.
   Total = 40 + 25 + 20 + 15 = 100.
3. Each value's percentage is calculated using:

$$\text{Percentage} = \frac{\text{Value}}{\text{Total}} \times 100$$

4. Since the total is 100, the values already represent percentages:
   - Samsung → 40%
   - Apple → 25%
   - Xiaomi → 20%
   - Others → 15%
5. The parameter autopct='%1.1f%%' automatically shows the percentage on the chart.
6. Each slice of the pie chart is drawn based on its percentage portion of 360°.
   Example:
   Samsung's angle = $40\% \times 360 = 144^\circ$