Fundamental of Machine Learning

Unit 4

4.1 NumPy Library

NumPy (Numerical Python) is a powerful Python library used for **numerical computing**. It provides support for:

- Multidimensional arrays (like matrices)
- Mathematical functions
- Linear algebra
- Statistical operations
- Fast computations using C under the hood

Use of NumPy

- **Faster** than regular Python lists for numerical operations
- Easy to work with **large datasets**
- Built-in **vectorized operations** (no need for loops)
- Base for many other libraries (like Pandas, SciPy, TensorFlow)

Example: Creating an array using numpy library

```
import numpy as np

# Create an array

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

# performing simple math operations

b = a * 2  #Multiplying all element with 2, Ans: b= [2, 4, 6, 8]

c = np.sum(a)#Finding sum of all elements of an array, Ans: c = 10
```

Examples of some methods of NumPy library

np.array() : To create arrays

np.zeros() : To create array of zeros

np.ones() : To create array of ones

np.mean(a) Average of array

np.std(a) Standard deviation

np.dot(a, b) Dot product of arrays (used in matrix multiplication)

4.2 SciPy (Scientific Python)

SciPy (Scientific Python) is an open-source Python library built on top of NumPy. While NumPy provides basic array operations, SciPy adds more advanced scientific and technical computing tools.

SciPy is like a **toolbox** for:

- Optimization
- Signal processing
- Integration (calculus)
- Interpolation
- Linear algebra (advanced)
- Statistics
- Solving differential equations
- Fourier transforms

Key Modules in SciPy

Module	Purpose
scipy.optimize	Minimization, curve fitting
scipy.integrate	Calculus: integration, differential equations
scipy.linalg	Advanced linear algebra
scipy.fft	Fourier transforms (signal processing)
scipy.stats	Probability distributions, statistics
scipy.spatial	Distance metrics, spatial algorithms (KD-trees)
scipy.ndimage	Image processing
scipy.signal	Signal processing (filters, convolution, etc.)

4.3 Pandas

Pandas is a Python library used for **data manipulation and analysis**. It provides **easy-to-use tools** to work with **structured data**, like spreadsheets and databases.

It can be assumed like **Excel for Python**, but much more powerful.

Use of Pandas

- Makes it easy to **load, clean, explore**, and **analyze** data
- Built on top of **NumPy**
- Works great with CSVs, Excel, SQL, JSON, and more
- Handy for **dataframes** and **series** (tabular and columnar data)

Pandas library has two main data structures:

1. Series
2. DataFrame

Before using the objects of pandas library, we need to import the library.

import pandas as pd   # pd is alias for pandas library

Series:
Series is an object of pandas library that represents one dimensional data structure. It is similar to an array and there are two arrays associated with each other. The main array holds the data and the other array is used for indexing. To create a series, Series() constructor is called by passing values as an argument.

```
a=pd.Series([7,9,13,15])

print(a)
```

Output:

```
0    7
1    9
2    13
3    15
dtype: int64
```

```
# Indexing

a=pd.Series([7,9,13,15],index=['i','ii','iii','iv'])
```

```
print(a)
```

Output:

```
i      7
ii     9
iii   13
iv    15
dtype: int64
```

An element of the series can be selected by considering it as numpy array or providing the index.

```
a[2]
```

Output:

```
13
```

```
A['iii']
```

Output:

```
13
```

Filtering values: Values in a series can be filtered according to a particular criteria.

```
a[a>9]
```

Output:

```
iii    13
iv     15
dtype: int64
```

Operations and Mathematical functions: We can apply mathematical operation (+, - , *, /) to a series.

```
a/2
```

Output:

```
i       3.5
ii      4.5
iii     6.5
iv      7.5
dtype: float64
```

unique() function

This functions returns the unique values in a series excluding duplicate.

```
a=pd.Series([10,10,30, 10, 30, 20, 20, 40, 50])
a.unique()
```

Output:

```
array([10, 30, 20, 40, 50], dtype=int64)
```

value_counts() function

This function returns the counts of unique values.

```
a=pd.Series([10,10,30, 10, 30, 20, 20, 40, 50])
a.value_counts()
```

Output:

```
10    3
20    2
30    2
50    1
40    1
dtype: int64
```

isin( ) function

This function evaluates the membership an element or a series of elements in a data structure. It returns boolean value 'Ture' if the element is contained in the data structure and 'False' if the element is not contained in the data structure.

```
a=pd.Series([10,10,30, 10, 30, 20, 20, 40, 50])
a.isin([10])
```

Output:

```
0     True
1     True
2    False
3     True
4    False
5    False
6    False
7    False
8    False
```

```
dtype: bool
```

Reading files using pandas

import pandas as pd

```
# uploading CSV file "data.csv"
df = pd.read_csv("data.csv")
# uploading Excel file "data.xlsx"
df = pd.read_excel("data.xlsx")
# uploading JSON file "data.json"
df = pd.read_json("data.json")
```

Basic indexing and slicing in **DataFrame**

Indexing and slicing in a DataFrame using pandas can be done using different ways:

Let us create a DataFrame having name 'df' using dictionary

```
import numpy as np
import pandas as pd
dict= {"Name":['Rakesh', 'Druv', 'Aina', 'Ekta', 'Eva'], "Age":
[24,   27,   22,   32,   29],   "City":['Dehradun',   'Lucknow',
'Delhi','Chandigarh','Kolkata']}
df=pd.DataFrame(dict)
df
```

Output:

	Name	Age	City
0	Rakesh	24	Dehradun
1	Druv	27	Lucknow
2	Aina	22	Delhi
3	Ekta	32	Chandigarh
4	Eva	29	Kolkata

1. Selecting columns

```
print(df['Name'])         # returns a Series
```

Output:

```
0   Rakesh
1     Druv
2     Aina
```

```
3     Ekta
4     Eva
```

```
print(df[['Name', 'City']])  # returns a DataFrame with two columns
```

Output:

```
     Name     City
0    Rakesh   Dehradun
1    Druv     Lucknow
2    Aina     Delhi
3    Ekta     Chandigarh
4    Eva      Kolkata
```

2. Selecting rows by index using .iloc method

Access a single row:

```
print(df.iloc[0])          # First row
```

Output:

```
Name     Rakesh
Age      24
City     Dehradun
```

```
print(df.iloc[-1])          # Last row
```

Output:

```
Name      Eva
Age       29
City      Kolkata
```

Access multiple rows:

```
print(df.iloc[2:4])       # Row 2 to 3 (excludes 4)
```

Output:

```
     Name    Age     City
2    Aina    22      Delhi
3    Ekta    32      Chandigarh
```

Access specific row and column:

```
print(df.iloc[0, 1])  # Row 0, Column 1
```

Output:

```
24
```

Access a block (rows and columns):

```
print(df.iloc[1:4, 0:2])
```

Output:

```
   Name  Age
1  Druv   27
2  Aina   22
3  Ekta   32
```

3. Selecting rows by index using .loc

Access a single row by label:

```
print(df.loc[0])
```

Output:

```
Name      Rakesh
Age           24
City    Dehradun
```

Access multiple rows and columns by label

```
print(df.loc[1:4, ['Name', 'City']])
```

Output:

```
    Name        City
1  Druv     Lucknow
2  Aina       Delhi
3  Ekta  Chandigarh
4   Eva     Kolkata
```

4. Re-indexing

We can reindex a single row or multiple rows by using reindex() method. Default values in the new index that are not present in the dataframe are assigned NaN.

```
new_index = [0,1,2,3,4]
df_new = df.reindex(new_index)
df_new
```

Output:

```
        Name       Age   City
0       Rakesh     24    Dehradun
1       Druv       27    Lucknow
2       Aina       22    Delhi
3       Ekta       32    Chandigarh
4       Eva        29    Kolkata
```

5. Sorting in DataFrame:

Sorting in a DataFrame in Python (using pandas) allows you to rearrange rows or columns based on values or labels. There are two main types:

Sorting by Values

Let the initial data has indexing not in ascending order.

```
import numpy as np
import pandas as pd
dict= {"Name":['Rakesh', 'Druv', 'Aina', 'Ekta', 'Eva'],
       "Age": [24, 27, 22, 32, 29],
   "City":['Dehradun','Lucknow','Delhi','Chandigarh','Kolkata'],
      }
df=pd.DataFrame(dict, index= [2, 0, 3, 1, 4])
df
```

Output:

```
         Name    Age    City
 2       Rakesh  24     Dehradun
 0       Druv    27     Lucknow
 3       Aina    22     Delhi
 1       Ekta    32     Chandigarh
 4       Eva     29     Kolkata
```

Use sort_index() to sort by rows

```
df_sorted = df.sort_index()  # for sorting by rows
```

```
df_sorted
```

Output:

	Name	**Age**	**City**
0	Druv	27	Lucknow
1	Ekta	32	Chandigarh
2	Rakesh	24	Dehradun
3	Aina	22	Delhi
4	Eva	29	Kolkata

Use sort_index(axis =1) to sort by column labels

```
df_sorted = df.sort_index(axis =1)
df_sorted
```

	Age	**City**	**Name**
2	24	Dehradun	Rakesh
0	27	Lucknow	Druv
3	22	Delhi	Aina
1	32	Chandigarh	Ekta
4	29	Kolkata	Eva

Use sort_values() to sort by column values.

```
df_sorted = df.sort_values(by='Age')
df_sorted
```

Output:

	Name	**Age**	**City**
3	Aina	22	Delhi
2	Rakesh	24	Dehradun
0	Druv	27	Lucknow
4	Eva	29	Kolkata
1	Ekta	32	Chandigarh

In place of single column, we can provide a list of columns in which order we wish to sort.

7. Filtering in DataFrame

Filtering in DataFrame can be done by defining conditions.
To filter the data of students having age>=25

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

Output:

```
        Name   Age   City
0       Druv   27     Lucknow
1       Ekta   32     Chandigarh
4       Eva    29     Kolkata
```

to filter the data of students having age>=25 and City =Lucknow

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

Output:

```
        Name Age   City
0       Druv  27   Lucknow
```

8. Ranking in DataFrame

Ranking assigns ranks to values in a Series or DataFrame, typically used to determine relative positions (like 1st, 2nd, etc.) of numeric values.

Basic syntax is: Series.rank(axis=0, method='average', ascending=True)

```
df['Rank'] = df['Age'].rank(ascending=False)
print(df)
```

Output:

```
        Name        Age        City          Rank
2      Rakesh       24         Dehradun      4.0
0      Druv         27         Lucknow       3.0
3      Aina         22         Delhi         5.0
1      Ekta         32         Chandigarh    1.0
4      Eva          29         Kolkata       2.0
```

By default ascending order is True.

```
df['Rank'] = df['Age'].rank()
print(df)
```

Output:

```
        Name      Age      City         Rank
2       Rakesh    24       Dehradun     2.0
0       Druv      27       Lucknow      3.0
3       Aina      22       Delhi        1.0
1       Ekta      32       Chandigarh   5.0
4       Eva       29       Kolkata      4.0
```

4.4 Normalization

Normalization is a technique used to scale data so that it fits within a specific range. Different normalizing methods are discussed below. For normalization, we use the following sample data 'df'

X1	X2
26	36
35	37
110	100
89	65
98	89
68	110
84	256

1. **Min-Max Normalization**

$$X_{nor} = \frac{X - X_{min}}{X_{Max} - X_{min}}$$

where,
X_{max} , X_{min} are the maximum , minimum values of the attribute X.

$$X_{1,min} = 26, X_{1,max} = 110$$

$$X_{2,min} = 36, X_{2,max} = 256$$

It converts a data into [0, 1] range.
Normalized data:

X1	X2	X1_nor	X2_nor
26	36	0.00	0.00
35	37	0.11	0.00
110	100	1.00	0.29
89	65	0.75	0.13
98	89	0.86	0.24
68	110	0.50	0.34
84	256	0.69	1.00

Python programming

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
normalized_data = scaler.fit_transform(df)
```

2. Z-score Normalization

$$X_{nor} = \frac{X - \mu}{\sigma}$$

where, μ and σ are the mean and standard deviation values of the attribute X.

$$\mu_1 = 72.86, \sigma_1 = 29.40$$

$$\mu_2 = 99, \sigma_2 = 69.52$$

It converts a data into standard Normal distribution with in [-3, 3] range with mean = 0 and standard deviation =1.

Normalized data:

X1	**X2**	**X1_nor**	**X2_nor**
26	36	-1.59	-0.91
35	37	-1.29	-0.89
110	100	1.26	0.01
89	65	0.55	-0.49
98	89	0.86	-0.14
68	110	-0.17	0.16
84	256	0.38	2.26

Python programming

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
standardized_data = scaler.fit_transform(data)
```

3. Max Absolute Scaling

$$X_{nor} = \frac{X}{Max\ (|X|)}$$

It converts a data into [-1, 1] range.

$$X_{1,max} = 110$$
$$X_{2\ max} = 256$$

Normalized data:

X1	**X2**	**X1_nor**	**X2_nor**
26	36	0.24	0.14
35	37	0.32	0.14
110	100	1.00	0.39
89	65	0.81	0.25
98	89	0.89	0.35
68	110	0.62	0.43
84	256	0.76	1.00

```
from sklearn.preprocessing import MaxAbsScaler
scaler = MaxAbsScaler()
normalized_data = scaler.fit_transform(df)
```

4.5 Data Alignment

Alignment refers to how Pandas automatically aligns data by labels (index or column names) when performing operations like addition, subtraction, or merging. For example, let we have two series

```
S1:
a   1
b   2
c   3

S2:
b   4
c   5
d   6

S1+ S2

a   NaN
b   6.0
c   8.0
d   NaN
```

4.6 Aggregation

Aggregation means reducing a dataset to summary statistics like mean, sum, count, min, max, etc. In python programming agg() method is used for the purpose. Let us consider the DataFrame used in normalization, suppose we wish to find 'minium', 'maximum', 'mean', 'median', 'variance', 'standard deviation'.

```
df.agg(['min', 'max', 'mean', 'median', 'var', 'std' ])
```

Output:

```
        X1            X2
min     26.000000     36.000000
max     110.000000    256.000000
mean    72.857143     99.000000
median  84.000000     89.000000
var     1008.142857   5640.000000
std     31.751265     75.099933
```

If some of the attribute is categorical and we wish to find the aggregate according to different labels, then groupby() method is used. Let us consider the DataFrame

```
data = pd.DataFrame({
    'Department': ['HR', 'HR', 'IT', 'IT', 'Sale', 'IT', 'Sale'],
    'Salary': [40000, 42000, 50000, 52000, 60000, 55000, 62000]
     })
data
```

Output:

```
    Department Salary
0   HR     40000
1   HR     42000
2   IT     50000
3   IT     52000
4   Sale   60000
5   IT     55000
6   Sale   62000
```

To find aggregation 'minium', 'maximum', and 'mean' department-wise:

```
data.groupby("Department").agg(['min', 'max', 'mean'])
```

Output:

```
                              Salary
               min            max            mean
Department
HR             40000          42000          41000.000000
IT             50000 5        5000           52333.333333
Sale           60000          62000          61000.000000
```

4.7 Summarization

Summarization gives a quick overview or statistical snapshot of your data. describe() method is used for the same.

```
data.describe()
```

Output:

```
          Salary
count     7.000000
mean      51571.428571
std       8363.753999
min       40000.000000
25%       46000.000000
50%       52000.000000
75%       57500.000000
max       62000.000000
```

4.8 Date and Time Series Analysis

Time series data consists of sequential data points recorded over time which is used in industries like finance, pharmaceuticals, social media and research. Analyzing and visualizing this data helps us to find trends and seasonal patterns for forecasting and decision-making. In this article, we will see more about Time Series Analysis and Visualization in depth.

Time series data analysis involves studying data points collected in chronological time order to identify current trends, patterns and other behaviors. This helps extract actionable insights and supports accurate forecasting and decision-making.

Key Concepts in Time Series Analysis

- **Trend:** It represents the general direction in which a time series is moving over an extended period. It checks whether the values are increasing, decreasing or staying relatively constant.
- **Seasonality:** Seasonality refers to repetitive patterns or cycles that occur at regular intervals within a time series corresponding to specific time units like days, weeks, months or seasons.
- **Moving average:** It is used to smooth out short-term fluctuations and highlight longer-term trends or patterns in the data.
- **Noise:** It represents the irregular and unpredictable components in a time series that do not follow a pattern.
- **Differencing:** It is used to make the difference in values of a specified interval. By default it's 1 but we can specify different values for plots.
- **Stationarity:** A stationary time series is statistical properties such as mean, variance and autocorrelation remain constant over time.
- **Order:** The order of differencing refers to the number of times the time series data needs to be differenced to achieve stationarity.
- **Autocorrelation**: Autocorrelation is a statistical method used in time series analysis to quantify the degree of similarity between a time series and a lagged version of itself.
- **Resampling**: Resampling is a technique in time series analysis that is used for changing the frequency of the data observations.

Types of Time Series Data

Time series data can be classified into two sections:

1. **Continuous Time Series**: Data recorded at regular intervals with a continuous range of values like temperature, stock prices, Sensor Data, etc.
2. **Discrete Time Series**: Data with distinct values or categories recorded at specific time points like counts of events, categorical statuses, etc**.**

Date and Time Series Analysis is a powerful feature in Pandas used for analyzing and working with time-stamped data, such as stock prices, weather data, and sensor readings.

Pandas makes it easy to:

- Parse and manipulate dates
- Create time-indexed data
- Resample time series (e.g., daily to monthly)
- Perform time-based filtering and shifting
- Handle missing time periods

Creating Date Time Data and converting a column to datetime:

```
import pandas as pd
df = pd.DataFrame({
    'date':  ['2024-02-01','2024-01-15'  ,'2024-02-03',  '2024-02-03','2024-03-02','2024-03-10' ],
    'Sale': [1000, 500, 1500, 920, 1200, 1050]
    })
df['date'] = pd.to_datetime(df['date'])
df
```

Output:

	date	**Sale**
0	2024-02-01	1000
1	2024-01-15	500
2	2024-02-03	1500
3	2024-02-03	920
4	2024-03-02	1200
5	2024-03-10	1050

Set Datetime as Index

```
df.set_index('date', inplace=True)
```

```
df
```

Output:

date	**Sale**
2024-02-01	1000
2024-01-15	500
2024-02-03	1500
2024-02-03	920
2024-03-02	1200
2024-03-10	1050

Resampling: To better understand the trend of the data we use the **resampling method** which provides a clearer view of trends and patterns when we are dealing with daily data.

df_resampled = df.resample('M').mean(numeric_only=True):
It resamples data to monthly frequency and calculates the mean of all numeric columns for each month. 'ME' is used in new versions of Python in place of M

```
monthly = df.resample('ME').mean() #
monthly
```

Output:

Date	**Sale**
2024-01-31	500.0
2024-02-29	1140.0
2024-03-31	1125.0

Moving Average

A moving average in time series analysis is a calculation that smooths out fluctuations in data by averaging values over a fixed-width sliding window. This process helps identify underlying trends and seasonal variations within the data.

Purpose of Moving Averages:

- **Smoothing:** Moving averages reduce short-term noise and fluctuations, making it easier to see long-term trends.
- **Trend Identification:** By averaging data points over a period, moving averages can help reveal the overall direction (upward or downward) of the time series.
- **Seasonal Variation:** Moving averages can be used to identify and account for seasonal patterns within the data.
- **Forecasting:** While not a perfect forecasting tool, moving averages can be used to predict future values based on past data.

How it works:

1. **Define a window:** A fixed number of data points (the "window") are selected.
2. **Calculate the average:** The average of the values within the window is computed.
3. **Slide the window:** The window is shifted forward one data point, and a new average is calculated.
4. **Repeat:** This process is repeated until the window has moved through the entire series.

```
df['rolling_mean'] = df['Sale'].rolling(window=2).mean()
```

Output:

```
date            Sale        rolling_mean
2024-02-01      1000        NaN
2024-01-15      500         750.0       #  (mean of 1000, 500)
2024-02-03      1500        1000.0      #  (mean of 500, 1500)
2024-02-03      920         1210.0      #  (mean of 1500, 920)
2024-03-02      1200        1060.0      #  (mean of 920, 1200)
2024-03-10      1050        1125.0      #  (mean of 1200, 1050)
```

4.9 Matplotlib

Matplotlib is a powerful Python library used for **creating static, interactive, and animated visualizations**. It can be viewed as the **"drawing tool"** of Python — perfect for plotting graphs, charts, and figures.

Plot Types

Plot Type	Function	Use
Line Plot	plt.plot()	Trends over time
Bar Chart	plt.bar()	Comparing categories
Histogram	plt.hist()	Distribution of data
Scatter Plot	plt.scatter()	Correlation between two variables
Pie Chart	plt.pie()	Parts of a whole
Box Plot	plt.boxplot()	Summary statistics (median, IQR)
Subplots	plt.subplot()	Multiple plots in one figure

Line Plot

A **Line Plot** (or **Line Chart**) is a graph that connects individual data points with a straight line. It's used to show **trends over time** or the **relationship between two continuous variables**.

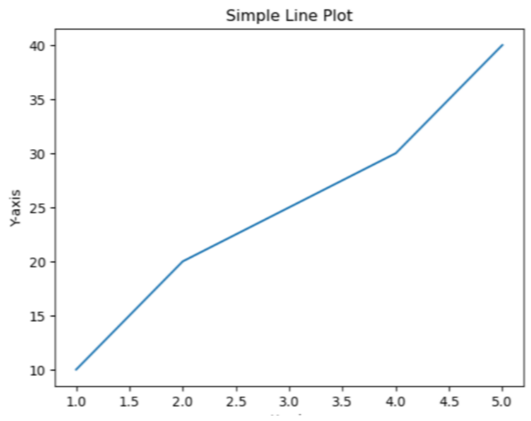
Example Use Cases:

- Tracking **stock prices** over time
- Monitoring **temperature** across days
- Showing **sales growth** month by month

Line Plot Python Example:

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]
plt.plot(x, y)
plt.title('Simple Line Plot')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.show()
```

Output:



Bar Graph

A Bar Graph is a chart that uses rectangular bars to represent and compare quantitative data across different categories.

- Each bar's height (or length) shows the value of the category.
- Bars can be vertical or horizontal.

Use of a bar graph:

- Compare values across categories (e.g., sales by product)
- Visualize frequencies or counts
- Highlight differences or trends

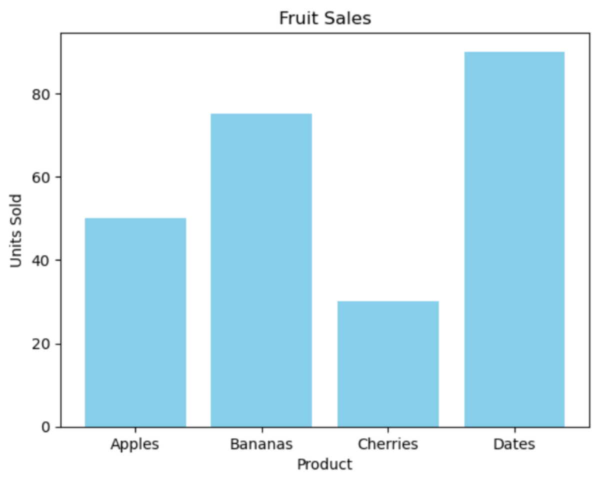
Key Features

- X-axis: Categories (e.g., fruits, cities, students)
- Y-axis: Values (e.g., sales, marks, population)
- Bars do not touch, it is a one-dimensional graph. Measurement is only on Y axis. that's what distinguishes it from a histogram.

```
import matplotlib.pyplot as plt
# Data
products = ['Apples', 'Bananas', 'Cherries', 'Dates']
sales = [50, 75, 30, 90]
# Create bar chart
plt.bar(products, sales, color='skyblue')
```

```
 #Add titles and labels
plt.title('Fruit Sales')
plt.xlabel('Product')
plt.ylabel('Units Sold')
# Show the chart
plt.show()
```

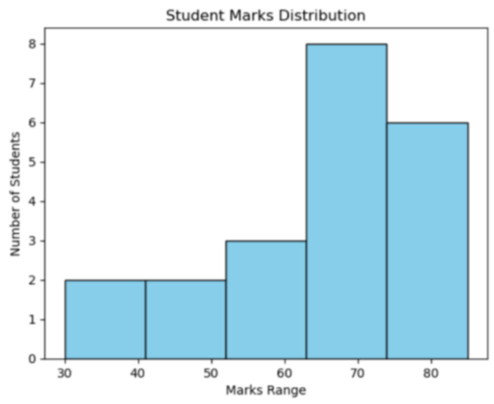
Output:



Histogram Example

```
import matplotlib.pyplot as plt
# Sample data: marks of 20 students
marks = [30, 35, 42, 73, 72, 85, 55, 77, 70, 75, 80, 68, 65, 55,
60, 78, 63, 75, 50, 66, 63]
# Create histogram
plt.hist(marks, bins=5, color='skyblue', edgecolor='black')
# Add labels and title
plt.title('Student Marks Distribution')
plt.xlabel('Marks Range')
plt.ylabel('Number of Students')
# Show plot
plt.show()
```

Output:



Scatter Plot

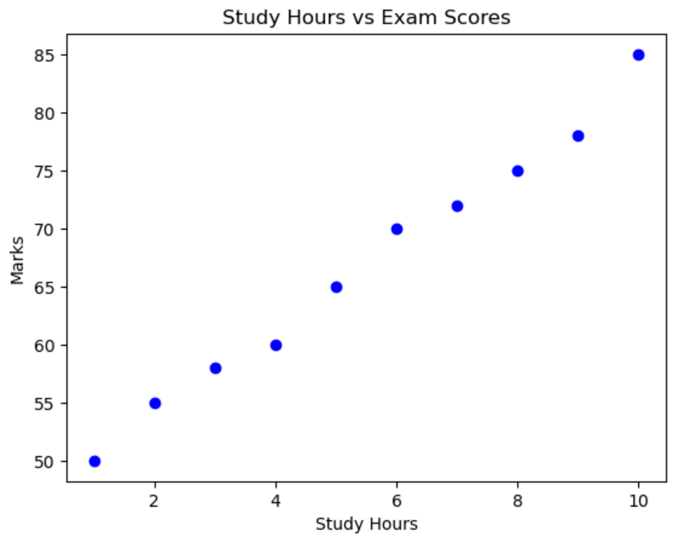
A Scatter Plot is a type of graph used to visualize the relationship between two variables. It displays points (dots) for each observation in the dataset — one variable on the x-axis, and the other on the y-axis.

It's great for:

- Checking for **correlation** (positive, negative, none)
- Identifying **clusters** or **outliers**

```
import matplotlib.pyplot as plt
# Sample data: Study hours vs Exam scores
Study_hours = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Marks = [50, 55, 58, 60, 65, 70, 72, 75, 78, 85]
# Create scatter plot
plt.scatter(Study_hours, Marks, color='blue', marker='o')
# Add labels and title
plt.title('Study Hours vs Exam Scores')
plt.xlabel('Study Hours')
plt.ylabel('Marks')
# Show the plot
plt.show()
```

Output:



Pie chart

A pie chart is a circular graph that uses slices to represent the proportions of different categories within a whole. Each slice's area is proportional to the value it represents, visually illustrating how different parts contribute to the total.

Key aspects of a pie chart:

Circular Representation: The chart is a circle divided into slices, with each slice representing a different category or group.

Proportional Slices: The size of each slice is directly related to the percentage or proportion it represents of the total.

Visualizing Proportions: Pie charts are effective for showing how different components contribute to a whole, especially when comparing parts to each other and to the total.

Limitations:

Pie charts are best used for small datasets and are not ideal for comparing values across different categories.

Use a pie chart:

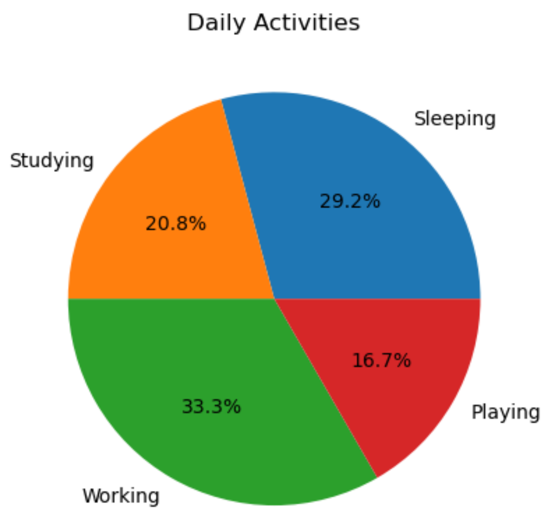
Showing Composition: To illustrate the breakdown of a whole into its components.

- **Comparing Parts to the Whole:** To visually represent how each part contributes to the total.

- **Representing Percentages:** To show how different categories make up a percentage of a whole.

```
import matplotlib.pyplot as plt
# Sample data
activities = ['Sleeping', 'Studying', 'Working', 'Playing']
hours = [7, 5, 8, 4]
# Create pie chart
plt.pie(hours,        labels=activities,        autopct='%1.1f%%',
startangle=0)
# Add title
plt.title('Daily Activities')
# Show chart
plt.show()
```

Output:


Daily Activities
Sleeping
Studying
Playing
Working