CODEBOOSTERS TECH
THINK LEARN GRAB

Codeboosters Tech
team_codeboosters
www.codeboosters.in

# 🧩 Modules and Packages in Python

## 1. Introduction & Theory

### What is a Module?

A **module** is a file containing Python code—functions, classes, or variables—that can be reused in other Python programs. Think of it as a toolbox: instead of rewriting code, you can import and use existing tools.

### What is a Package?

A **package** is a collection of modules organized in directories. It allows for a hierarchical structuring of the module namespace using dot notation. Imagine a package as a toolbox containing multiple smaller toolkits (modules).

### Why Use Modules and Packages?

- **Reusability**: Write once, use multiple times.
- **Organization**: Keeps code organized and manageable.
- **Namespace Management**: Avoids naming conflicts.

### Real-World Analogy

Consider a library (package) containing books (modules). Each book covers specific topics (functions/classes). Instead of writing your own book, you borrow one from the library.

## 2. Code Example with Line-by-Line Explanation

### Creating and Using a Module

**Step 1**: Create a module named `greetings.py`.

```python
# greetings.py

def say_hello(name):
    return f"Hello, {name}!"

def say_goodbye(name):
    return f"Goodbye, {name}!"
```

**Step 2**: Use the module in another script.

```python
# main.py

import greetings

print(greetings.say_hello("Alice"))
print(greetings.say_goodbye("Bob"))
```

**Explanation**:

- `import greetings`: Imports the `greetings` module.
- `greetings.say_hello("Alice")`: Calls the `say_hello` function from the `greetings` module with "Alice" as an argument.
- `greetings.say_goodbye("Bob")`: Calls the `say_goodbye` function from the `greetings` module with "Bob" as an argument.

**Alternative Import Methods**:

- `from greetings import say_hello`: Imports only the `say_hello` function.
- `from greetings import *`: Imports all functions from the module (not recommended for large modules due to potential naming conflicts).

### Creating and Using a Package

**Step 1**: Create a directory structure:

```markdown
my_package/
├── __init__.py
├── math_operations.py
└── string_operations.py
```

**math_operations.py**:

```python
def add(a, b):
    return a + b
```

**string_operations.py**:

```python
def to_uppercase(s):
    return s.upper()
```

**init.py**:

This file can be empty or used to initialize the package.

**Step 2**: Use the package in another script.

```python
# main.py

from my_package import math_operations, string_operations

print(math_operations.add(5, 3))
print(string_operations.to_uppercase("hello"))
```

**Explanation**:

- `from my_package import math_operations, string_operations`: Imports the modules from the package.
- `math_operations.add(5, 3)`: Calls the `add` function from the `math_operations` module.
- `string_operations.to_uppercase("hello")`: Calls the `to_uppercase` function from the `string_operations` module.

## 3. Mini Tasks / Coding Exercises

### Task 1: Create a Module for Basic Math Operations

**Description**: Create a module named `basic_math.py` with functions for addition, subtraction, multiplication, and division.

**Expected Output**:

```makefile
Addition: 15
Subtraction: 5
Multiplication: 50
Division: 2.0
```

**Solution**:

```python
# basic_math.py

def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    return a / b
```

```python
# main.py

import basic_math

print("Addition:", basic_math.add(10, 5))
print("Subtraction:", basic_math.subtract(10, 5))
print("Multiplication:", basic_math.multiply(10, 5))
print("Division:", basic_math.divide(10, 5))
```

**Explanation**:

- Defines four basic arithmetic functions in `basic_math.py`.
- Imports and uses these functions in `main.py`.

### Task 2: Create a Package for Text Utilities

**Description**: Create a package named `text_utils` with a module `formatter.py` containing a function `capitalize_words` that capitalizes each word in a sentence.

**Expected Output**:

```
makefile

Original: hello world
Capitalized: Hello World
```

**Solution**:

```
python

# text_utils/formatter.py

def capitalize_words(sentence):
    return sentence.title()
```

```
python

# main.py

from text_utils import formatter

sentence = "hello world"
print("Original:", sentence)
print("Capitalized:", formatter.capitalize_words(sentence))
```

**Explanation**:

- The `capitalize_words` function uses the `title()` method to capitalize each word.
- The package structure allows for organized code management.

### Task 3: Use Aliases for Modules

**Description**: Import the `math` module using an alias and calculate the square root of 16.

**Expected Output**:

```csharp
Square root of 16 is 4.0
```

**Solution**:

```python
import math as m

print("Square root of 16 is", m.sqrt(16))
```

**Explanation**:

- `import math as m`: Imports the `math` module with an alias `m` for convenience.
- `m.sqrt(16)`: Calls the `sqrt` function from the `math` module.

### Task 4: Import Specific Functions

**Description**: Import only the `randint` function from the `random` module and generate a random integer between 1 and 10.

**Expected Output**:

```yaml
Random number: 7
```

**Solution**:

```python
from random import randint

print("Random number:", randint(1, 10))
```

**Explanation**:

- `from random import randint`: Imports only the `randint` function.
- `randint(1, 10)`: Generates a random integer between 1 and 10.

### Task 5: Create a Nested Package

**Description**: Create a package `utilities` with a subpackage `math_tools` containing a module `operations.py` with a function `square` that returns the square of a number.

**Expected Output**:

```csharp
Square of 5 is 25
```

**Solution**:

```markdown
utilities/
├── __init__.py
└── math_tools/
    ├── __init__.py
    └── operations.py
```

**operations.py**:

```python
def square(n):
    return n * n
```

**main.py**:

```python
from utilities.math_tools import operations

print("Square of 5 is", operations.square(5))
```

**Explanation**:

- Demonstrates how to structure and import from nested packages.

## 4. Summary

- **Modules**: Single Python files containing reusable code.
- **Packages**: Directories containing multiple modules, organized with `__init__.py`.
- **Importing**: Use `import` statements to access functions and classes from modules and packages.
- **Aliases**: Simplify module names using `as`.
- **Selective Import**: Import specific functions or classes using `from module import name`.

## 5. Oral Questions for Students

1. **What is the difference between a module and a package in Python?**

   *Expected Answer*: A module is a single Python file with reusable code, while a package is a directory containing multiple modules and an `__init__.py` file.

2. **How do you import a specific function from a module?**

   *Expected Answer*: Use the syntax `from module_name import function_name`.

3. **What is the purpose of the `__init__.py` file in a package?**

   *Expected Answer*: It indicates that the directory is a Python package and can be used to initialize the package or set up the `__all__` list.

4. **How can you avoid naming conflicts when importing modules?**

   *Expected Answer*: Use aliases with the `as` keyword, e.g., `import module_name as alias`.

5. **Can you import a module from a subpackage? How?**

   *Expected Answer*: Yes, by using dot notation, e.g., `from package.subpackage import module`.

# SECTION 2 - NUMPY

## 📘 NumPy Library

### 1. Introduction & Theory

#### What is NumPy?

NumPy, short for **Numerical Python**, is an open-source library that facilitates efficient numerical computations in Python. It introduces the `ndarray` object, a powerful N-dimensional array, and provides a suite of functions for performing operations on these arrays.

#### Why Use NumPy?

- **Performance**: NumPy arrays are more efficient than Python lists for numerical operations.
- **Functionality**: Offers a vast collection of mathematical functions.
- **Convenience**: Simplifies tasks like linear algebra, statistical operations, and more.

#### Real-World Analogy

Think of NumPy arrays as Excel spreadsheets. Each cell holds a number, and you can perform operations across rows and columns efficiently.

### 2. Code Examples with Line-by-Line Explanation

#### Creating a NumPy Array

```python
import numpy as np  # Importing the NumPy library

# Creating a 1D array
```

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

**Explanation**:

- `import numpy as np`: Imports the NumPy library and assigns it the alias `np` for convenience.
- `np.array([1, 2, 3, 4, 5])`: Creates a NumPy array from a Python list.
- `print(arr)`: Displays the array.

### Array Indexing and Slicing

```
python

import numpy as np

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

print(arr[0])     # Accessing the first element
print(arr[1:4])   # Slicing elements from index 1 to 3
```

**Explanation**:

- `arr[0]`: Accesses the first element of the array.
- `arr[1:4]`: Slices the array from index 1 up to, but not including, index 4.

### Shape Manipulation

```
python

import numpy as np

arr = np.array([[1, 2], [3, 4], [5, 6]])
print(arr.shape)  # Outputs the shape of the array

reshaped_arr = arr.reshape(2, 3)
print(reshaped_arr)
```

**Explanation**:

- `arr.shape`: Returns a tuple representing the dimensions of the array.
- `arr.reshape(2, 3)`: Reshapes the array to have 2 rows and 3 columns.

### Array Iteration

```python
import numpy as np

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

for row in arr:
    print("Row:", row)
    for element in row:
        print("Element:", element)
```

**Explanation**:

- `for row in arr`: Iterates over each row in the 2D array.
- `for element in row`: Iterates over each element in the current row.

### Joining and Splitting Arrays

```python
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

joined = np.concatenate((arr1, arr2))
print("Joined:", joined)

split = np.array_split(joined, 3)
print("Split:", split)
```

**Explanation**:

- `np.concatenate((arr1, arr2))`: Joins two arrays into one.
- `np.array_split(joined, 3)`: Splits the array into 3 sub-arrays.

### Searching, Sorting, and Filtering

```python
import numpy as np
```

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

# Searching
index = np.where(arr == 30)
print("Index of 30:", index)

# Sorting
sorted_arr = np.sort(arr)
print("Sorted:", sorted_arr)

# Filtering
filtered = arr[arr > 25]
print("Filtered:", filtered)
```

**Explanation**:

- `np.where(arr == 30)`: Returns the indices where the condition is true.
- `np.sort(arr)`: Returns a sorted copy of the array.
- `arr[arr > 25]`: Filters elements greater than 25.

---

## 3. Mini Tasks / Coding Exercises

### Task 1: Create a 2D Array and Access Elements

**Description**: Create a 2D NumPy array and access specific elements.

**Expected Output**:

```
java

Element at (0,1): 2
Element at (1,0): 3
```

**Solution**:

```
python

import numpy as np

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

```
print("Element at (0,1):", arr[0, 1])
print("Element at (1,0):", arr[1, 0])
```

**Explanation**:

- `arr[0, 1]`: Accesses the element in the first row, second column.
- `arr[1, 0]`: Accesses the element in the second row, first column.

### Task 2: Reshape a 1D Array to 2D

**Description**: Convert a 1D array of 6 elements into a 2D array with 2 rows and 3 columns.

**Expected Output**:

lua

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

**Solution**:

python

```python
import numpy as np

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

**Explanation**:

- `arr.reshape(2, 3)`: Reshapes the array to 2 rows and 3 columns.

### Task 3: Iterate Over a 2D Array

**Description**: Print each element of a 2D array.

**Expected Output**:

```
1
2
3
4
```

**Solution**:

```python
import numpy as np

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

for row in arr:
    for element in row:
        print(element)
```

**Explanation**:

- Nested loops are used to access each element in the 2D array.

### Task 4: Join Two Arrays Horizontally

**Description**: Join two 2D arrays horizontally.

**Expected Output**:

```lua
[[1 2 5 6]
 [3 4 7 8]]
```

**Solution**:

```python
import numpy as np

arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])

joined = np.hstack((arr1, arr2))
print(joined)
```

**Explanation**:

- `np.hstack((arr1, arr2))`: Stacks arrays horizontally (column-wise).

### Task 5: Filter Elements Greater Than a Value

**Description**: From a given array, filter elements greater than 3.

**Expected Output:**

```csharp
[4 5]
```

**Solution:**

```python
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
filtered = arr[arr > 3]
print(filtered)
```

**Explanation:**

- `arr[arr > 3]`: Applies a boolean mask to filter elements greater than 3.

## 4. Summary

- **NumPy**: A powerful library for numerical computations in Python.
- **Arrays**: Use `np.array()` to create arrays.
- **Indexing/Slicing**: Access elements using indices and slices.
- **Shape Manipulation**: Use `reshape()` to change the shape of arrays.
- **Iteration**: Loop through arrays using standard Python loops.
- **Joining/Splitting**: Combine or divide arrays using functions like `concatenate()` and `array_split()`.
- **Searching/Sorting/Filtering**: Use functions like `where()`, `sort()`, and boolean indexing for data analysis.

## 5. Oral Questions for Students

1. **What is NumPy and why is it used?**

*Expected Answer*: NumPy is a Python library used for numerical computations, providing efficient array operations and mathematical functions.

2. **How do you create a NumPy array?**

   *Expected Answer*: By using the `np.array()` function and passing a list or tuple.

3. **Explain the difference between Python lists and NumPy arrays.**

   *Expected Answer*: NumPy arrays are more efficient for numerical operations and support multi-dimensional data, whereas Python lists are general-purpose containers.

4. **How can you reshape a NumPy array?**

   *Expected Answer*: By using the `reshape()` method, specifying the new dimensions.

---

# 🐼 Pandas Library Module

---

## 1. Introduction & Theory

### 📌 What is Pandas?

Pandas is a **Python library** used for **data manipulation and analysis**. It provides powerful, easy-to-use structures for handling **tabular data**, much like spreadsheets or SQL tables.

### 🔍 Why Use Pandas?

- Handles **large data sets** efficiently.
- Provides **data cleaning**, **transformation**, and **analysis** tools.
- Ideal for working with **CSV files**, **Excel**, and **database outputs**.

### 📊 Real-World Analogy

Imagine you have a **spreadsheet** with rows (records) and columns (fields). Pandas acts like a **super-spreadsheet tool**, allowing you to slice, filter, group, merge, and analyze that data using Python.

---

## 2. Code Examples with Line-by-Line Explanation

### 🔹 Creating a Series

```python
import pandas as pd

# Creating a Series
data = pd.Series([10, 20, 30, 40])
print(data)
```

**Explanation**:

- `import pandas as pd`: Imports the Pandas library and gives it an alias `pd`.
- `pd.Series([...])`: Creates a one-dimensional labeled array called a **Series**.
- A Series is like a column in Excel with an index.

---

### 🔹 Creating a DataFrame

```python
import pandas as pd

# Creating a DataFrame
data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35]
}
```

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

**Explanation**:

- `pd.DataFrame(data)` : Creates a 2D table (like an Excel sheet).
- `data` is a dictionary where keys are column names and values are lists of data.

---

## 🔹 Filtering Rows Based on Condition

python

```python
import pandas as pd

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

# Filter people older than 28
filtered = df[df["Age"] > 28]
print(filtered)
```

**Explanation**:

- `df["Age"] > 28` : Creates a boolean Series based on condition.
- `df[...]` : Filters the rows where the condition is True.

---

## 🔹 Grouping Data

python

```python
import pandas as pd

data = {
    "Department": ["HR", "HR", "IT", "IT"],
    "Salary": [3000, 3500, 4000, 4200]
}

df = pd.DataFrame(data)

grouped = df.groupby("Department").mean()
print(grouped)
```

**Explanation**:

- `groupby("Department")`: Groups rows based on the "Department" column.
- `.mean()`: Calculates the average salary per department.

---

## 🔹 Merging DataFrames

```python
import pandas as pd

df1 = pd.DataFrame({
    "ID": [1, 2],
    "Name": ["Alice", "Bob"]
})

df2 = pd.DataFrame({
    "ID": [1, 2],
    "Salary": [3000, 4000]
})

merged = pd.merge(df1, df2, on="ID")
print(merged)
```

**Explanation**:

- `pd.merge(...)`: Combines two DataFrames using a common column (`ID`).
- `on="ID"`: Specifies the column to merge on.

---

## 🔹 List Comprehension with DataFrames

```python
import pandas as pd

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

# Add 5 years to each age
df["Age_plus_5"] = [age + 5 for age in df["Age"]]
print(df)
```

**Explanation:**

- `[age + 5 for age in df["Age"]]`: List comprehension to modify a column.
- `df["new_col"] = ...`: Creates a new column with computed values.

---

## 🔹 Concatenating DataFrames

```python
import pandas as pd

df1 = pd.DataFrame({"Name": ["Alice"], "Age": [25]})
df2 = pd.DataFrame({"Name": ["Bob"], "Age": [30]})

combined = pd.concat([df1, df2])
print(combined)
```

**Explanation:**

- `pd.concat([...])`: Combines multiple DataFrames vertically (adds rows).

---

## 🔹 Transforming Data (Apply Function)

```python
```

```
import pandas as pd

df = pd.DataFrame({
    "Name": ["Alice", "Bob"],
    "Salary": [3000, 4000]
})

# Increase salary by 10%
df["Increased"] = df["Salary"].apply(lambda x: x * 1.10)
print(df)
```

**Explanation**:

- `.apply(lambda x: ...)`: Applies a function to each element in the column.

---

## 3. 5 Mini Tasks / Coding Exercises

### 🧩 Task 1: Create a Series and Print the Second Element

**Description**: Create a Series of numbers `[5, 10, 15]` and print the second item.
**Expected Output**: `10`

python

```
import pandas as pd

series = pd.Series([5, 10, 15])
print(series[1])
```

**Concept**: Indexing a Series

---

### 🧩 Task 2: Create a DataFrame and Add a New Column

**Description**: Create a DataFrame with columns "Name" and "Age", then add a "City" column.
**Expected Output**:

markdown

```
    Name  Age   City
0  Alice   24  Delhi
1    Bob   30  Delhi
```

python

```python
import pandas as pd

df = pd.DataFrame({
    "Name": ["Alice", "Bob"],
    "Age": [24, 30]
})

df["City"] = "Delhi"
print(df)
```

**Concept**: Column assignment

---

## 🧩 Task 3: Filter Data Where Age > 25

python

```python
import pandas as pd

df = pd.DataFrame({
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [22, 28, 35]
})

print(df[df["Age"] > 25])
```

**Expected Output**:

markdown

```
      Name  Age
1      Bob   28
2  Charlie   35
```

**Concept**: Boolean filtering

## 🧩 Task 4: Group by Department and Sum Salaries

```python
import pandas as pd

df = pd.DataFrame({
    "Department": ["HR", "IT", "HR"],
    "Salary": [3000, 4000, 3500]
})

print(df.groupby("Department").sum())
```

**Expected Output**:

```yaml
            Salary
Department
HR             6500
IT             4000
```

**Concept**: Grouping and aggregation

## 🧩 Task 5: Merge Two DataFrames on Common Key

```python
import pandas as pd

df1 = pd.DataFrame({"ID": [1, 2], "Name": ["Alice", "Bob"]})
df2 = pd.DataFrame({"ID": [1, 2], "Age": [24, 30]})

merged = pd.merge(df1, df2, on="ID")
print(merged)
```

**Expected Output:**

```nginx
   ID   Name  Age
0   1  Alice   24
1   2    Bob   30
```

**Concept**: Merging

## 4. Summary

- **Series**: 1D data (like a single column).
- **DataFrame**: 2D data structure (like Excel).
- **Filtering**: Use boolean conditions on columns.
- **Grouping**: Summarize or analyze data based on a key.
- **Merging & Concatenating**: Combine datasets.
- **List Comprehension & Apply**: Perform custom transformations.

## 5. Oral Questions for Students

1. **What are the two main data structures in Pandas?**
   *Expected Answer*: Series (1D) and DataFrame (2D)
2. **How do you filter rows where a column meets a condition?**
   *Expected Answer*: Use boolean indexing like `df[df["Age"] > 25]`
3. **How does groupby() work?**
   *Expected Answer*: It groups rows by the values in a column and allows aggregate functions (like sum, mean) to be applied.
4. **What's the difference between merge() and concat()?**
   *Expected Answer*: `merge()` joins DataFrames on common keys; `concat()` stacks them either row-wise or column-wise.

5. **How would you increase each value in a column by 10%?**
   *Expected Answer*: Use `.apply(lambda x: x * 1.10)` or list comprehension.

# 📈 Matplotlib Library Module

This module introduces students to **data visualization** using Matplotlib, a popular library for plotting data in Python.

## 1. Introduction & Theory

### 🧠 What is Matplotlib?

Matplotlib is a Python library used for **creating static, interactive, and animated visualizations**. It's widely used for plotting charts from data stored in lists, arrays, or Pandas DataFrames.

### 🎯 Why Use It?

- Makes data easier to understand visually.
- Helps spot trends, patterns, and outliers.
- Can create various charts like **line plots**, **scatter plots**, **bar charts**, and **histograms**.

### 📊 Real-World Analogy

Imagine you're analyzing student scores. Reading a table of numbers is hard—but plotting a graph instantly shows who's ahead or behind. Matplotlib is like a **drawing tool** for your data.

# 2. Code Examples with Line-by-Line Explanation

### 🔹 Line Plot

python

```
import matplotlib.pyplot as plt

# Data
x = [1, 2, 3, 4]
y = [10, 20, 25, 30]

# Create a line plot
plt.plot(x, y)
plt.title("Line Plot Example")
plt.xlabel("X Axis")
plt.ylabel("Y Axis")
plt.show()
```

**Explanation**:

- `import matplotlib.pyplot as plt` : Imports the plotting module.
- `plt.plot(x, y)` : Plots a line graph with x and y values.
- `title` , `xlabel` , `ylabel` : Adds labels and titles.
- `plt.show()` : Displays the plot.

---

### 🔹 Scatter Plot

python

```
x = [5, 7, 8, 10]
y = [12, 14, 15, 18]

plt.scatter(x, y, color='red')
plt.title("Scatter Plot")
plt.xlabel("X Axis")
plt.ylabel("Y Axis")
plt.show()
```

**Explanation**:

- `plt.scatter()` : Plots individual data points (dots).
- `color='red'` : Changes dot color.

---

## 🔹 Histogram

```python
ages = [22, 55, 62, 45, 21, 22, 34, 42, 42, 29, 30]

plt.hist(ages, bins=5, color='green')
plt.title("Age Distribution")
plt.xlabel("Age")
plt.ylabel("Frequency")
plt.show()
```

**Explanation**:

- `plt.hist()` : Groups data into bins.
- `bins=5` : Divides ages into 5 ranges.

---

## 🔹 Bar Plot

```python
languages = ['Python', 'Java', 'C++']
popularity = [100, 80, 60]

plt.bar(languages, popularity)
plt.title("Programming Language Popularity")
plt.ylabel("Popularity")
plt.show()
```

**Explanation**:

- `plt.bar()` : Creates vertical bars.

### 🔹 Subplots

```python
import matplotlib.pyplot as plt

x = [1, 2, 3]
y1 = [2, 4, 6]
y2 = [1, 2, 3]

plt.subplot(1, 2, 1)  # 1 row, 2 cols, 1st plot
plt.plot(x, y1)
plt.title("First")

plt.subplot(1, 2, 2)  # 1 row, 2 cols, 2nd plot
plt.plot(x, y2)
plt.title("Second")

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

**Explanation**:

- `plt.subplot(rows, cols, index)`: Creates multiple plots in one figure.
- `tight_layout()`: Adjusts spacing to prevent overlapping.

---

## 3. 5 Mini Tasks / Coding Exercises

### 🧩 Task 1: Line Plot of Square Numbers

```python
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [i**2 for i in x]

plt.plot(x, y)
plt.title("Squares")
plt.xlabel("Number")
plt.ylabel("Square")
plt.show()
```

**Expected Output:** A line plot showing squares of numbers 1–5.

## 🧩 Task 2: Scatter Plot of Height vs Weight

```python
height = [150, 160, 170, 180]
weight = [50, 60, 70, 80]

plt.scatter(height, weight)
plt.title("Height vs Weight")
plt.xlabel("Height (cm)")
plt.ylabel("Weight (kg)")
plt.show()
```

**Expected Output:** Scatter plot showing increasing height vs weight.

## 🧩 Task 3: Create a Histogram of Marks

```python
marks = [35, 45, 55, 60, 70, 75, 80, 85, 90, 95]

plt.hist(marks, bins=5)
plt.title("Marks Distribution")
plt.xlabel("Marks")
plt.ylabel("Frequency")
plt.show()
```

**Expected Output:** Histogram with marks divided into 5 bins.

## 🧩 Task 4: Bar Chart of Fruit Sales

```python
fruits = ['Apple', 'Banana', 'Orange']
sales = [50, 75, 30]

plt.bar(fruits, sales)
plt.title("Fruit Sales")
plt.ylabel("Sales Count")
plt.show()
```

**Expected Output**: Bar chart comparing fruit sales.

## 🧩 Task 5: Subplots of Two Functions

```python
x = [1, 2, 3, 4]
y1 = [10, 20, 30, 40]
y2 = [5, 10, 15, 20]

plt.subplot(2, 1, 1)
plt.plot(x, y1)
plt.title("Y1 Plot")

plt.subplot(2, 1, 2)
plt.plot(x, y2)
plt.title("Y2 Plot")

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

**Expected Output**: Two vertically stacked plots with titles.

# 4. Summary

- **Line plot**: Shows trends (e.g. time series).
- **Scatter plot**: Shows relationship between two variables.
- **Histogram**: Shows distribution of data.
- **Bar plot**: Compares categories.
- **Subplots**: Allows multiple plots in one figure.

## 5. Oral Questions for Students

1. **What does `plt.plot()` do?**
   *Expected*: Draws a line graph using x and y values.
2. **When would you use a scatter plot instead of a line plot?**
   *Expected*: When plotting raw points without connecting them—like height vs weight.
3. **What is the purpose of `plt.show()`?**
   *Expected*: It displays the plot window.
4. **How do you create multiple plots in one figure?**
   *Expected*: Use `plt.subplot()`.
5. **What are bins in a histogram?**
   *Expected*: Bins divide the range of data into intervals for grouping.

# 🖼️ Tkinter: Basics

**Topic**: GUI (Graphical User Interface) programming using Tkinter for beginners

# 1. Introduction & Theory

## 🎨 What is Tkinter?

**Tkinter** is the **standard GUI library for Python**. It lets you create windows, buttons, labels, text boxes, and other GUI components in a simple way.

## 🧠 Why Use Tkinter?

- Helps create **interactive applications**.
- Turns command-line programs into **user-friendly apps**.
- Built-in with Python—no installation needed.

## 🪟 Real-World Analogy

Think of a Tkinter window as a **virtual canvas or form**—you can place buttons like "Submit", "Exit", or "Clear", just like forms on a website or software screen.

---

# 2. Code Examples with Line-by-Line Explanation

## 🔹 Create a Basic Window

```python
import tkinter as tk

window = tk.Tk()
window.title("My First GUI")
window.geometry("300x200")

window.mainloop()
```

**Explanation**:

- `import tkinter as tk`: Imports Tkinter with the alias `tk`.
- `tk.Tk()`: Creates the main window.
- `.title(...)`: Sets the window title.
- `.geometry(...)`: Sets the window size (`width x height`).

- `.mainloop()` : Starts the GUI event loop (keeps window open).

## Adding a Label and a Button

```python
import tkinter as tk

def say_hello():
    print("Hello, World!")

window = tk.Tk()
window.title("Label and Button")

label = tk.Label(window, text="Welcome to GUI!")
label.pack()

button = tk.Button(window, text="Click Me", command=say_hello)
button.pack()

window.mainloop()
```

**Explanation**:

- `Label(...)` : Displays text on the window.
- `Button(...)` : Creates a clickable button.
- `command=say_hello` : Calls a function when the button is clicked.
- `.pack()` : Automatically places the widget in the window (layout manager).

## Entry Box (Text Input)

```python
import tkinter as tk

def show_input():
    user_input = entry.get()
    print("You typed:", user_input)

window = tk.Tk()
window.title("Input Example")

entry = tk.Entry(window)
entry.pack()

button = tk.Button(window, text="Submit", command=show_input)
button.pack()

window.mainloop()
```

**Explanation**:

- `Entry(...)` : Creates a text input field.
- `.get()` : Retrieves the current text in the input box.

---

## 🔹 Change Label Text Dynamically

python

```
import tkinter as tk

def change_text():
    label.config(text="You clicked the button!")

window = tk.Tk()
window.title("Dynamic Label")

label = tk.Label(window, text="Original Text")
label.pack()

button = tk.Button(window, text="Change Text", command=change_text)
button.pack()

window.mainloop()
```

**Explanation**:

- `label.config(...)`: Updates label text dynamically on an event.

---

### 🔹 GUI Layout Using `grid()`

```python
import tkinter as tk

window = tk.Tk()
window.title("Grid Layout")

tk.Label(window, text="Username").grid(row=0, column=0)
tk.Entry(window).grid(row=0, column=1)

tk.Label(window, text="Password").grid(row=1, column=0)
tk.Entry(window).grid(row=1, column=1)

window.mainloop()
```

**Explanation**:

- `.grid(row=..., column=...)`: Places widgets in a **table-like layout**.

---

## 3. 5 Mini Tasks / Coding Exercises

### 🧩 Task 1: Create a Window with Title and Size

```python
import tkinter as tk

window = tk.Tk()
window.title("Simple Window")
window.geometry("250x150")
window.mainloop()
```

**Expected Output**: A blank window with the title "Simple Window" and size 250x150.

## 🧩 Task 2: Add a Button that Prints "Hi"

```python
import tkinter as tk

def say_hi():
    print("Hi!")

window = tk.Tk()
button = tk.Button(window, text="Say Hi", command=say_hi)
button.pack()
window.mainloop()
```

**Expected Output**: Clicking the button prints "Hi!" in the console.

## 🧩 Task 3: Input Field That Shows Typed Text on Button Click

```python
import tkinter as tk

def show_text():
    print("Input:", entry.get())

window = tk.Tk()
entry = tk.Entry(window)
entry.pack()

button = tk.Button(window, text="Show Text", command=show_text)
button.pack()

window.mainloop()
```

**Expected Output**: Whatever you type in the box appears in the console when you click the button.

## 🧩 Task 4: Button That Changes Label Text

```python
import tkinter as tk

def update_label():
    label.config(text="Updated!")

window = tk.Tk()
label = tk.Label(window, text="Old Text")
label.pack()

button = tk.Button(window, text="Update", command=update_label)
button.pack()
window.mainloop()
```

**Expected Output**: Label changes from "Old Text" to "Updated!" when clicked.

---

## 🧩 Task 5: Create a Login Form Layout Using Grid

```python
import tkinter as tk

window = tk.Tk()
window.title("Login Form")

tk.Label(window, text="Username").grid(row=0, column=0)
tk.Entry(window).grid(row=0, column=1)

tk.Label(window, text="Password").grid(row=1, column=0)
tk.Entry(window, show="*").grid(row=1, column=1)

tk.Button(window, text="Login").grid(row=2, column=1)
window.mainloop()
```

**Expected Output**: A simple login form layout using grid layout.

---

## 4. Summary

- **Tk()**: Starts the GUI app.
- **Label, Button, Entry**: Basic widgets for text, buttons, and input.
- **command=...**: Connects a button to a function.
- **pack() / grid()**: Layout managers to position widgets.
- **mainloop()**: Keeps the window running and responding to events.

---

## 5. Oral Questions for Students

1. **What does `mainloop()` do in a Tkinter application?**
   *Expected*: It keeps the GUI window open and listens for user actions.
2. **How do you make a button perform an action?**
   *Expected*: Use `command=function_name` in the button.
3. **What is the difference between `pack()` and `grid()`?**
   *Expected*: `pack()` places widgets vertically or horizontally; `grid()` places them in a table-like layout.
4. **How can you change the text of a label after a button is clicked?**
   *Expected*: Use `label.config(text="new text")`.
5. **What does `Entry(window)` do?**
   *Expected*: It creates a text box for user input.

---