**Python – Collections, functions and Modules:-**

**Accessing List:-**

**• Understanding how to create and access elements in a list.**

In Python, a list is a collection of items that is ordered, mutable, and allows duplicate elements. Here's a simple guide to help you understand how to create and access elements in a list.

**1. Creating a List**

You can create a list using square brackets [] and separate elements with commas:

Ex:-

fruits = ["apple", "banana", "cherry"]

numbers = [10, 20, 30, 40]

mixed = [1, "hello", 3.14, True]

empty\_list = []

**2. Accessing Elements in a List**

You access elements by their **index**, starting from **0**:

Ex:-

fruits = ["apple", "banana", "cherry"]

print(fruits[0]) # Output: apple

print(fruits[1]) # Output: banana

print(fruits[2]) # Output: cherry

**3. Negative Indexing**

You can also access items from the **end** using **negative indexes**:

Ex:-

print(fruits[-1]) # Output: cherry

print(fruits[-2]) # Output: banana

**4. Accessing a Range (Slicing)**

You can access multiple elements using **slicing**:

Ex:-

print(fruits[0:2]) # Output: ['apple', 'banana']

print(fruits[:2]) # Output: ['apple', 'banana']

print(fruits[1:]) # Output: ['banana', 'cherry']

print(fruits[:]) # Output: ['apple', 'banana', 'cherry']

**5. Nested Lists**

Lists can contain other lists:

Ex:;-

nested = [1, 2, [3, 4], 5]

print(nested[2]) # Output: [3, 4]

print(nested[2][1]) # Output: 4

* **Indexing in lists (positive and negative indexing).**

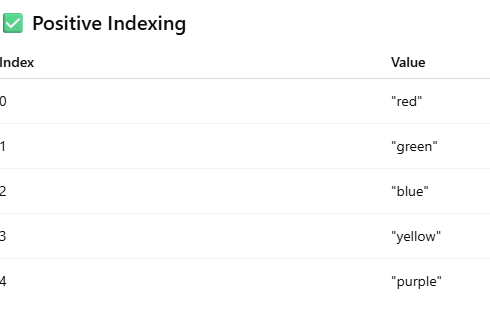
**Indexing?**

**Indexing** is a way to access individual elements in a list using their position.

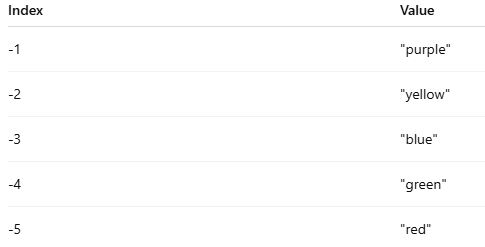
* **Positive Indexing** starts from **0** and goes from **left to right**.
* **Negative Indexing** starts from **-1** and goes from **right to left**.

Example List:--

colors = ["red", "green", "blue", "yellow", "purple"]



Negative Indexing



* **Slicing a list: accessing a range of elements.**

**Slicing** lets you extract a part (sublist) from a list using the syntax:

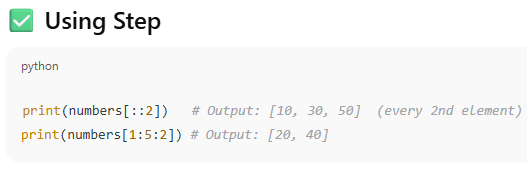
list[start : end : step]

 start → index to begin (inclusive)

 end → index to stop (exclusive)

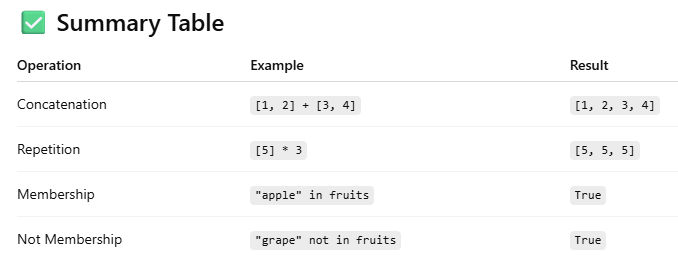
 step → how many steps to skip (optional)





**2. List Operations**

Common list operations: concatenation, repetition, membership.



**1. List Concatenation (+)**

You can **combine** two or more lists using the + operator.

list1 = [1, 2, 3]

list2 = [4, 5]

result = list1 + list2

print(result) # Output: [1, 2, 3, 4, 5]

**2. List Repetition (\*)**

You can **repeat** a list multiple times using the \* operator.

nums = [0, 1]

result = nums \* 3

print(result) # Output: [0, 1, 0, 1, 0, 1]

**3. Membership Test (in, not in)**

You can check if an item **exists** in a list using the in or not in operators.

fruits = ["apple", "banana", "cherry"]

print("banana" in fruits) # Output: True

print("mango" not in fruits) # Output: True

* **Understanding list methods like append(), insert(), remove(), pop()**



1. append() – Add an item at the **end**

fruits.append("orange")

print(fruits) # Output: ['apple', 'banana', 'cherry', 'orange']

1. insert() – Add an item at a **specific index**

fruits.insert(1, "mango")

print(fruits) # Output: ['apple', 'mango', 'banana', 'cherry']

1. remove() – Delete the **first occurrence** of a value

fruits.remove("banana")

print(fruits) # Output: ['apple', 'cherry']

1. **pop() – Remove and return an item**

* Without index → removes the **last item**
* With index → removes item at **that position**

**3. Working with Lists**

Iterating over a list using loops in Python is a common and essential practice. There are two main ways to iterate over a list: using a for loop and a while loo

1. **Using a for loop (Recommended and most common)**

fruits = ["apple", "banana", "cherry", "date"]

for fruit in fruits:

print(fruit)

1. **Using a for loop with index**

fruits = ["apple", "banana", "cherry", "date"]

for i in range(len(fruits)):

print(fruits[i])

1. **Using a while loop**

fruits = ["apple", "banana", "cherry", "date"]

i = 0

while i < len(fruits):

print(fruits[i])

i += 1

1. **Using enumerate() to get index and value**

fruits = ["apple", "banana", "cherry", "date"]

for index, fruit in enumerate(fruits):

print(index, fruit)

**Sorting and reversing a list using sort(), sorted(), and reverse().**

1. **sort() Method**

Modifies the original list in place (doesn't return a new list).

numbers = [5, 2, 9, 1]

numbers.sort()

print(numbers) # Output: [1, 2, 5, 9]

1. **sorted() Function**

**Returns a new sorted list** without modifying the original list.

numbers = [5, 2, 9, 1]

sorted\_numbers = sorted(numbers)

print(sorted\_numbers) # Output: [1, 2, 5, 9]

print(numbers) # Original list unchanged

1. **reverse() Method**

 **Only reverses the order** of elements in the list (does not sort).

 **Modifies the list in place**.

numbers = [1, 3, 5, 2]

numbers.reverse()

print(numbers) # Output: [2, 5, 3, 1]

**Basic list manipulations: addition, deletion, updating, and slicing.**

**1. Addition (Adding Elements)**

**➤ Using append() – adds to the end:**

fruits = ["apple", "banana"]

fruits.append("cherry")

print(fruits) # Output: ['apple', 'banana', 'cherry']

**2. Deletion (Removing Elements)**

**➤ Using remove() – removes by value:**

fruits.remove("banana")

print(fruits) # Output: ['apple', 'orange', 'cherry', 'grape', 'melon']

➤ Using pop() – removes by index (default is last):

fruits.pop(2)

print(fruits) # Output: ['apple', 'orange', 'grape', 'melon']

**3. Updating (Modifying Elements)**

**➤ Replace an element by index:**

fruits[0] = "kiwi"

print(fruits) # Output: ['kiwi', 'grape', 'melon']

**4. Slicing (Accessing Parts of a List)**

**➤ Syntax: list[start:stop]**

(returns elements from start index up to stop - 1)

numbers = [10, 20, 30, 40, 50]

print(numbers[1:4]) # Output: [20, 30, 40]

1. **Tuple**

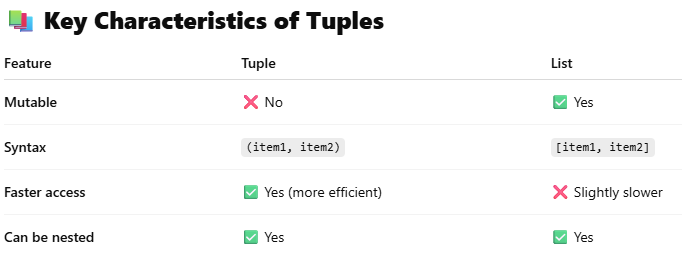
**Introduction to tuples, immutability**

A **tuple** is a built-in data type in Python that is very similar to a **list**, but with one key difference:

Syntax of a Tuple

my\_tuple = (10, 20, 30)

print(my\_tuple) # Output: (10, 20, 30)



**Creating and accessing elements in a tuple.**

**1. Creating a Tuple**

A tuple is a collection of ordered and **immutable** elements.

# Creating a tuple with different data types

my\_tuple = (10, 20, 30)

mixed\_tuple = (1, "apple", 3.14)

single\_element\_tuple = (5,) # Note the comma

empty\_tuple = ()

**2. Accessing Elements in a Tuple**

You can access tuple elements using **indexing** and **slicing**.

**Indexing:**

my\_tuple = (10, 20, 30, 40)

print(my\_tuple[0]) # Output: 10

print(my\_tuple[2]) # Output: 30

print(my\_tuple[-1]) # Output: 40 (last element)

**3. Tuple Immutability**

You **cannot change** the values of a tuple once created.

my\_tuple[1] = 50 # ❌ Error: 'tuple' object does not support item assignment

* **Basic operations with tuples: concatenation, repetition, membership.**

**Basic Operations with Tuples in Python**

Tuples are immutable, but you can perform several **basic operations** like **concatenation**, **repetition**, and **membership testing**.

**🔹 1. Concatenation (+)**

You can join two or more tuples using the + operator.

t1 = (1, 2, 3)

t2 = (4, 5)

result = t1 + t2

print(result) # Output: (1, 2, 3, 4, 5)

**2. Repetition (\*)**

You can repeat the elements in a tuple using the \* operator.

t = (7, 8)

result = t \* 3

print(result) # Output: (7, 8, 7, 8, 7, 8)

**3. Membership (in, not in)**

You can check if an element exists in a tuple using in or not in.

t = ('apple', 'banana', 'cherry')

print('banana' in t) # Output: True

print('grape' not in t) # Output: True

1. **Accessing Tuples**

**Accessing tuple elements using positive and negative indexing.**

In Python, you can access individual elements of a tuple using **indexing**—either **positive** or **negative**.

1 **. Positive Indexing**

* Starts from 0 for the first element.
* Moves left to right.

my\_tuple = ('a', 'b', 'c', 'd', 'e')

print(my\_tuple[0]) # Output: 'a'

print(my\_tuple[2]) # Output: 'c'

print(my\_tuple[4]) # Output: 'e'

**2. Negative Indexing**

* Starts from -1 for the last element.
* Moves right to left.

print(my\_tuple[-1]) # Output: 'e'

print(my\_tuple[-2]) # Output: 'd'

print(my\_tuple[-5]) # Output: 'a'

**Slicing a tuple to access ranges of elements.**

**Slicing** is used to access a **range of elements** from a tuple, creating a new tuple with selected elements.

Syntax of Slicing:

tuple[start : stop : step]

 start → index to begin (inclusive)

 stop → index to end (exclusive)

 step → how many elements to skip (optional)

Example Tuple:

my\_tuple = ('a', 'b', 'c', 'd', 'e', 'f')

1. Basic Slicing

print(my\_tuple[1:4]) # Output: ('b', 'c', 'd')

print(my\_tuple[:3]) # Output: ('a', 'b', 'c')

print(my\_tuple[3:]) # Output: ('d', 'e', 'f')

1. Slicing with Step

print(my\_tuple[::2]) # Output: ('a', 'c', 'e') — every 2nd element

print(my\_tuple[1::2]) # Output: ('b', 'd', 'f')

1. Negative Slicing

print(my\_tuple[-4:-1]) # Output: ('c', 'd', 'e')

print(my\_tuple[::-1]) # Output: ('f', 'e', 'd', 'c', 'b', 'a') — reversed

V isual Guide:

Index: 0 1 2 3 4 5

Tuple: ('a','b','c','d','e','f')

Neg Index: -6 -5 -4 -3 -2 -1

1. **Dictionaries**

**Introduction to dictionaries: key-value pairs.**

In Python, a dictionary is a collection of key-value pairs. It is used to store data in a structured way, where each value is associated with a unique key.

Syntax of a Dictionary

my\_dict = {

"name": "Alice",

"age": 25,

"city": "Mumbai"

}

**Ex:-**

my\_dict = {

"name": "Alice",

"age": 25,

"city": "Mumbai"

}

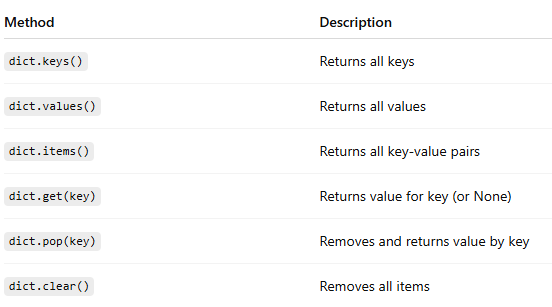
 "name", "age", and "city" are **keys**.

 "Alice", 25, and "Mumbai" are the **values**.

 Each key must be **unique** and **immutable** (e.g., strings, numbers, tuples).

 Values can be of any type.

**Common Dictionary Methods**



**Accessing, adding, updating, and deleting dictionary elements.**

Dictionary Example

student = {

"name": "John",

"age": 20,

"grade": "A"

}

**1.Accessing Elements**

You can **access** values using their keys:

print(student["name"]) # Output: John

print(student.get("age")) # Output: 20

**2.Adding Elements**

You can **add** a new key-value pair like this:

student["city"] = "Mumbai"

print(student)

# Output: {'name': 'John', 'age': 20, 'grade': 'A', 'city': 'Mumbai'}

**Final Example**

student = {"name": "John", "age": 20, "grade": "A"}

# Access

print("Name:", student["name"])

# Add

student["city"] = "Mumbai"

# Update

student["grade"] = "A+"

# Delete

del student["age"]

print(student)

* **Dictionary methods like keys(), values(), and items()**

Great! Let's explore the dictionary methods: keys(), values(), and items() in Python. These are super helpful when working with dictionaries.

**Sample Dictionary:-**

person = {

"name": "Alice",

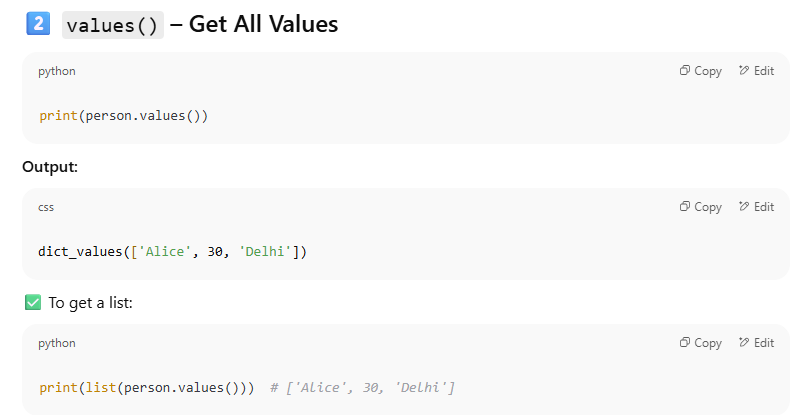
"age": 30,

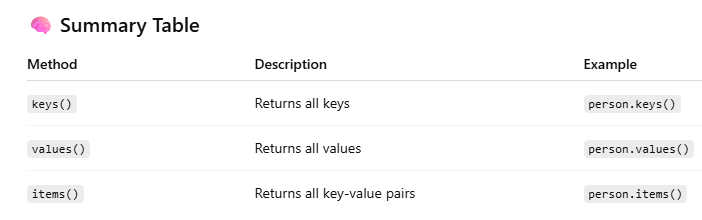
"city": "Delhi"

}

1. **keys() – Get All Keys**

print(person.keys())





**7. Working with Dictionaries**

Iterating over a dictionary using loops.

Absolutely! Iterating over a dictionary using loops is very common in Python. Let’s go over **how to loop through keys, values, and key-value pairs**.

🗂 Sample Dictionary

student = {

"name": "Rahul",

"age": 21,

"course": "Computer Science"

}

1. Loop Through **Keys**

for key in student:

print(key)

1. Loop Through **Values**

for value in student.values():

print(value)

1. Loop Through **Key-Value Pairs**

for key, value in student.items():

print(key, ":", value)

Real-world Example: Displaying Product Prices

prices = {

"apple": 100,

"banana": 40,

"mango": 60

}

for fruit, price in prices.items():

print(f"The price of {fruit} is ₹{price}")

**Merging two lists into a dictionary using loops or zip().**

ChatGPT said:

Great! Let's learn how to merge two lists into a dictionary using:

* A for loop
* The built-in zip() function

**Method 1: Using zip()**

merged\_dict = dict(zip(keys, values))

print(merged\_dict)

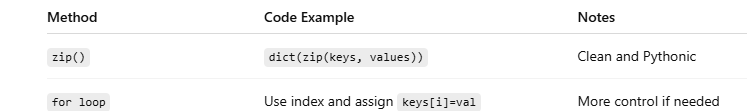
Method 2: Using a for Loop

merged\_dict = {}

for i in range(len(keys)):

merged\_dict[keys[i]] = values[i]

print(merged\_dict)



1. **Functions**

Defining functions in Python:-

In Python, **functions** are blocks of reusable code that perform a specific task. Functions help make your code modular, organized, and easier to debug.

**Basic Syntax**

def function\_name(parameters):

# Function body

# Code to execute

return result # optional

Example 1: A Simple Function:

def greet():

print("Hello, welcome to Python!")

Default Parameter Value:

def greet(name="Guest"):

print(f"Hello, {name}!")

greet() # Output: Hello, Guest!

greet("John") # Output: Hello, John!

**• Different types of functions: with/without parameters, with/without return values.**

In Python, functions can be categorized based on whether they have:

* **Parameters (input values)**
* **Return values (output values)**
* **1. Function without Parameters and without Return Value**
* This type of function does **not take any input** and does **not return anything**. It just performs a task.

def say\_hello():

print("Hello!")

Call the function:

say\_hello()

**2. Function with Parameters and without Return Value**

def greet\_user(name):

print(f"Hello, {name}!")

**3. Function without Parameters and with Return Value**

This function does **not take any input**, but it **returns a value**.

def get\_welcome\_message():

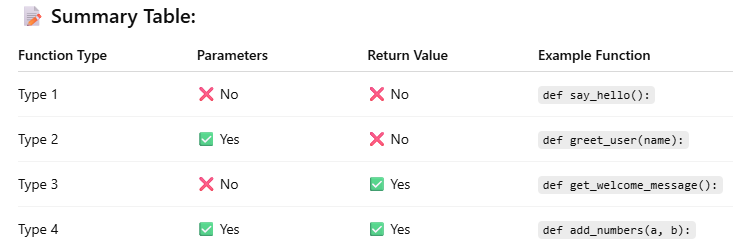
return "Welcome to Python Programming!"

**4. Function with Parameters and with Return Value**

This function takes **input** and **returns a result**.

def add\_numbers(a, b):

return a + b



**Anonymous functions (lambda functions).**

In Python, anonymous functions are functions without a name, created using the lambda keyword. These are also called lambda functions.

**When to Use Lambda Functions?**

* When the function is **short and simple**
* When used **temporarily as an argument** to functions like map(), filter(), or sorted()
* When a **named function is not needed**

**Example 1: Lambda to Add Two Numbers:-**

add = lambda x, y: x + y

print(add(5, 3)) # Output: 8

**Example 2: Lambda with No Arguments**

greet = lambda: "Hello!"

print(greet()) # Output: Hello!

**Example 3: Lambda with One Argument**

square = lambda x: x \* x

print(square(4)) # Output: 16

**Example 4: Lambda in Sorting**

names = ['Alice', 'Bob', 'Charlie', 'David']

names.sort(key=lambda name: len(name))

print(names)

**Example 5: Lambda with map()**

numbers = [1, 2, 3, 4]

squares = list(map(lambda x: x \*\* 2, numbers))

print(squares) # Output: [1, 4, 9, 16]

**9. Modules**

**Introduction to Python modules and importing modules.**

A **module** in Python is a file that contains **Python definitions, functions, classes, or variables**. It helps you organize code into **manageable, reusable** components.

* Any .py file is a **module**.
* Python has many **built-in modules** (like math, random, os), and you can also create your **own modules**.

**Benefits of Using Modules**

* Organizes code better
* Increases reusability
* Reduces redundancy
* Makes maintenance easier

**Importing Modules**

There are multiple ways to import modules:

* 1. **Import Entire Module**

import math

print(math.sqrt(16)) # Output: 4.0

* 1. **Import Specific Function or Class**

from math import sqrt

print(sqrt(25)) # Output: 5.0

* 1. **Import with Alias (Renaming)**

import math as m

print(m.pi) # Output: 3.141592653589793

* 1. **Import Multiple Functions**

from math import ceil, floor

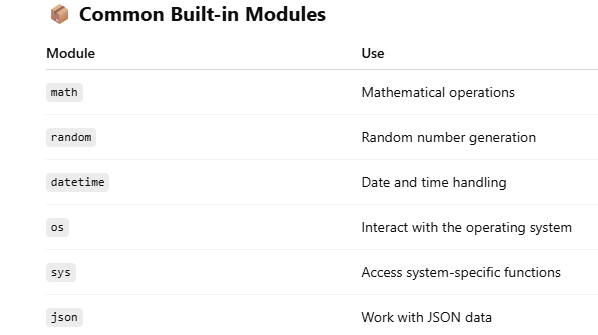
print(ceil(4.2)) # Output: 5

print(floor(4.2)) # Output: 4

* 1. **Import All Functions (Not Recommended)**

from math import \*

print(sqrt(36)) # Output: 6.0



**Standard library modules: math, random:-**

**Standard Library Modules in Python: math and random**

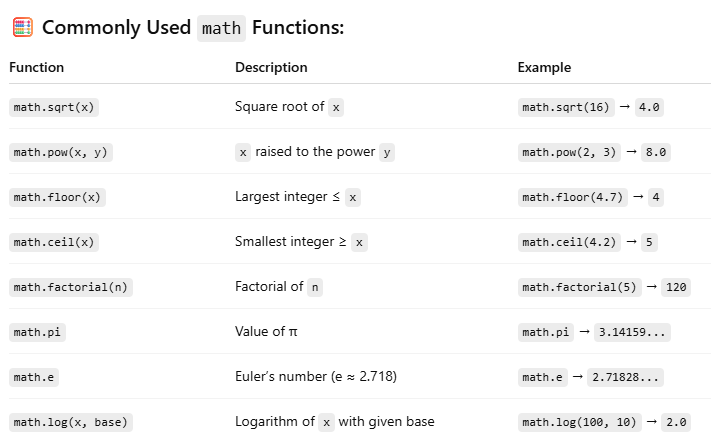
Python comes with a rich **standard library**, which includes powerful modules like math and random. These modules help perform **mathematical** and **randomization** operations efficiently.

**1. math Module**

The math module provides mathematical functions such as square roots, trigonometry, logarithms, constants, etc.

**✅ Importing:**

import math



**Example:**

import math

radius = 5

area = math.pi \* math.pow(radius, 2)

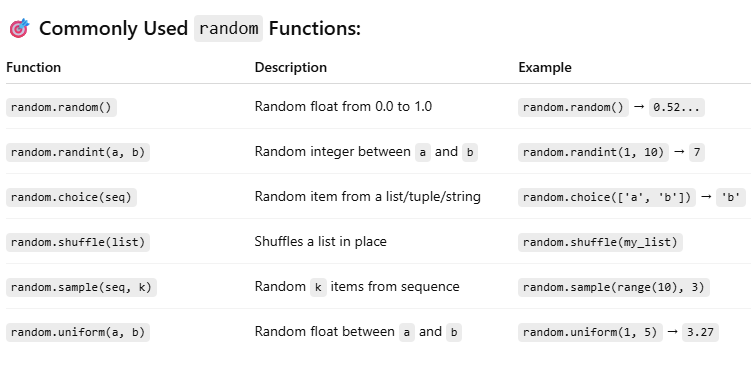
print("Area of Circle:", area)

**2. random Module**

**The random module is used to generate random numbers and perform random operations like shuffling, sampling, and selection.**

**✅ Importing:**

**import random**

****

**Example:-**

**import random**

**lucky\_number = random.randint(1, 100)**

**print("Your lucky number is:", lucky\_number)**

**Creating custom modules:-**

In Python, you can create your own modules to organize code into reusable components. A custom module is simply a .py file containing functions, classes, or variables that you can import and use in other programs.

Steps to Create and Use a Custom Module:-

**Step 1: Create a Module File**

Create a Python file named my\_module.py.

# my\_module.py

def greet(name):

return f"Hello, {name}!"

def add(a, b):

return a + b

pi = 3.14159

**Step 2: Use the Module in Another File**

Create another Python file in the same directory, for example main.py, and import the module.

# main.py

import my\_module

print(my\_module.greet("Alice"))

print("Sum:", my\_module.add(5, 3))

print("Value of Pi:", my\_module.pi)

**Best Practices**

* Name the file using lowercase letters and underscores, e.g., math\_utils.py
* Avoid using built-in module names (like math.py, random.py)
* Group related functions in one module
* Add docstrings to your functions for clarity