❖The Ultimate Python Guide

1. Basics of Python

- ☑ Introduction to Python & Installation
- ☑ Variables and Data Types
- ☑ Operators (Arithmetic, Logical, Comparison, Assignment, Bitwise)
- ☑ Input and Output (print, input)
- ☑ Comments and Docstrings

2. Control Flow

- ☑ Conditional Statements (if, elif, else)
- ☑ Loops (for, while)
- ☑ Loop Control (break, continue, pass)

3. Data Structures

- ☑ Lists (Methods, Slicing, List Comprehensions)
- ☑ Tuples (Immutable Sequences)
- ☑ Sets (Unique Collections)
- ☑ Dictionaries (Key-Value Pairs)
- ☑ Strings (Operations & Formatting)

4. Functions and Modules

- ☑ Defining Functions (def, return)
- ☑ Arguments (*args, **kwargs, Default Arguments)
- ☑ Lambda (Anonymous) Functions
- ☑ Importing & Using Modules (math, random, datetime, etc.)
- ☑ Creating Custom Modules

5. Object-Oriented Programming (OOP)

- ☑ Classes and Objects
- ☑ Constructors (__init__)
- ☑ Inheritance & Polymorphism
- ☑ Encapsulation & Abstraction
- ☑ Magic Methods (__str__, __repr__, etc.)

6. Exception Handling

- ☑ Try, Except, Finally Blocks
- ☑ Custom Exceptions

7. File Handling

- ☑ Reading & Writing Files (open, read, write)
- ☑ Working with CSV & JSON

8. Advanced Python Concepts

- ☑ Iterators & Generators (yield)
- ☑ Decorators & Closures
- ☑ Context Managers (with statement)

9. Libraries & Frameworks

- ☑ NumPy (Arrays & Math Operations)
- ☑ Pandas (Data Analysis)
- ☑ Matplotlib & Seaborn (Data Visualization)
- ☑ Flask/Django (Web Development)
- ☑ Tkinter (GUI Development)
- ☑ Requests (API & Web Scraping)

10. Automation & Scripting

- ☑ Regular Expressions (re module)
- ☑ Web Scraping (BeautifulSoup, Selenium)
- ☑ Automation with Python (os, shutil, subprocess)

11. Data Science & Machine Learning

- ☑ Scikit-Learn (ML Models)
- ☑ TensorFlow & PyTorch (Deep Learning)
- ☑ Natural Language Processing (NLP)
- ☑ OpenCV (Computer Vision)

12. Testing & Debugging

- ☑ Unit Testing (unittest, pytest)
- ☑ Debugging Tools (pdb)

13. Database Handling

- ☑ SQLite & PostgreSQL with Python (sqlite3, SQLAlchemy)
- ☑ CRUD Operations

14. Multi-threading & Concurrency

- ☑ threading Module
- ☑ asyncio for Asynchronous Programming

15. Deployment & DevOps

- ☑ Virtual Environments & Package Management (pip, venv)
- ☑ Docker & Kubernetes with Python
- ☑ Cloud Deployment (AWS, GCP, Azure)

➢Data Structures

1. Lists

Definition:
A list is an ordered, mutable collection of elements. It allows duplicate values and supports multiple data types.

Syntax:
my_list = [element1, element2, element3, ...]

Example 1: List Comprehension for Filtering
```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = [num for num in numbers if num % 2 == 0]
print(even_numbers)
```

Output:
[2, 4, 6, 8, 10]

Example 2: Sorting a List of Tuples by the Second Element
```
students = [("Alice", 85), ("Bob", 92), ("Charlie", 78)]
sorted_students = sorted(students, key=lambda x: x[1], reverse=True)
print(sorted_students)
```

Output:
[('Bob', 92), ('Alice', 85), ('Charlie', 78)]

---

2. Tuples

Definition:
A tuple is an ordered, immutable collection of elements. It allows duplicate values and can store multiple data types.

Syntax:
my_tuple = (element1, element2, element3, ...)

Example 1: Tuple Unpacking
```
point = (3, 5)
x, y = point
print(f"x: {x}, y: {y}")
```

Output:
x: 3, y: 5

Example 2: Returning Multiple Values from a Function

```
def divide(a, b):
    quotient = a // b
    remainder = a % b
    return quotient, remainder  # Returning a tuple

q, r = divide(10, 3)
print(f"Quotient: {q}, Remainder: {r}")
```

Output:

Quotient: 3, Remainder: 1

---

3. Sets

Definition:

A set is an unordered collection of unique elements. It does not allow duplicates and supports l set operations.

Syntax:

```
my_set = {element1, element2, element3, ...}
```

Example 1: Finding Unique Elements in a List

```
numbers = [1, 2, 3, 4, 2, 3, 1, 5]
unique_numbers = set(numbers)
print(unique_numbers)
```

Output:

{1, 2, 3, 4, 5}

Example 2: Set Operations (Union, Intersection, Difference)

```
set_a = {1, 2, 3, 4}
set_b = {3, 4, 5, 6}

print("Union:", set_a | set_b)
print("Intersection:", set_a & set_b)
print("Difference (A - B):", set_a - set_b)
```

Output:

Union: {1, 2, 3, 4, 5, 6}
Intersection: {3, 4}
Difference (A - B): {1, 2}

---

4. Dictionaries 🕮

Definition:

A dictionary is an unordered collection of key-value pairs. Keys are unique, and values can be of any data type.

Syntax:

```
my_dict = {key1: value1, key2: value2, ...}
```

Example 1: Iterating Over a Dictionary

```
student = {"name": "Alice", "age": 21, "grade": "A"}

for key, value in student.items():
    print(f"{key}: {value}")
```

Output:

```
name: Alice
age: 21
grade: A
```

Example 2: Sorting a Dictionary by Value

```
scores = {"Alice": 90, "Bob": 80, "Charlie": 95}
sorted_scores = dict(sorted(scores.items(), key=lambda item: item[1], reverse=True))
print(sorted_scores)
```

Output:

```
{'Charlie': 95, 'Alice': 90, 'Bob': 80}
```

---

5. Stacks (LIFO using Lists) 

Definition:

A stack follows the **Last-In, First-Out (LIFO)** principle. Elements are pushed and popped from the top.

Example 1: Implementing a Stack using a List

```
stack = []

# Push elements
stack.append(1)
stack.append(2)
stack.append(3)

# Pop element
print("Popped:", stack.pop())
print("Stack after popping:", stack)
```

Output:
Popped: 3
Stack after popping: [1, 2]

---

6. Queues (FIFO using collections.deque) 🚦

Definition:
A queue follows the **First-In, First-Out (FIFO)** principle. Elements are enqueued at the back and dequeued from the front.

Example 1: Implementing a Queue
```
from collections import deque

queue = deque()

# Enqueue elements
queue.append(1)
queue.append(2)
queue.append(3)

# Dequeue element
print("Dequeued:", queue.popleft())
print("Queue after dequeuing:", queue)
```

Output:
Dequeued: 1
Queue after dequeuing: deque([2, 3])

---

7. Heaps (Priority Queue using heapq) ⏳
Definition:
A heap is a complete binary tree where the smallest element is always at the root (min-heap).

Example 1: Using heapq for Priority Queue
```
import heapq

heap = []

# Push elements
heapq.heappush(heap, 3)
heapq.heappush(heap, 1)
heapq.heappush(heap, 2)
print("Heap:", heap)
```

```
# Pop the smallest element
print("Popped:", heapq.heappop(heap))
```

Output:
Heap: [1, 3, 2]
Popped: 1

Conclusion

Data Structure	Features
List	Ordered, mutable, allows duplicates
Tuple	Ordered, immutable, allows duplicates
Set	Unordered, unique elements, fast operations
Dictionary	Key-value pairs, unordered, unique keys
Stack	LIFO (Last-In, First-Out)
Queue	FIFO (First-In, First-Out)
Heap	Priority queue, min-heap by default

---

1. Lists
Lists are ordered, mutable collections of items.

Methods and Operations

1. **append()**: Adds an element to the end of the list.

```
fruits = ["apple", "banana"]
fruits.append("cherry")
print(fruits)  # Output: ['apple', 'banana', 'cherry']
```

2. **extend()**: Adds multiple elements (from another list) to the end.

```
fruits = ["apple", "banana"]
fruits.extend(["cherry", "date"])
print(fruits)  # Output: ['apple', 'banana', 'cherry', 'date']
```

3. **insert()**: Inserts an element at a specific index.

```
fruits = ["apple", "banana"]
fruits.insert(1, "cherry")
print(fruits)  # Output: ['apple', 'cherry', 'banana']
```

4. **remove()**: Removes the first occurrence of a value.

```
fruits = ["apple", "banana", "cherry"]
fruits.remove("banana")
print(fruits)  # Output: ['apple', 'cherry']
```

5. **pop()**: Removes and returns the element at a specific index (default is the last element).

```
fruits = ["apple", "banana", "cherry"]
print(fruits.pop(1))  # Output: 'banana'
print(fruits)  # Output: ['apple', 'cherry']
```

6. **index()**: Returns the index of the first occurrence of a value.

```
fruits = ["apple", "banana", "cherry"]
print(fruits.index("banana"))  # Output: 1
```

7. **count()**: Counts the number of occurrences of a value.

```
fruits = ["apple", "banana", "cherry", "banana"]
print(fruits.count("banana"))  # Output: 2
```

8. **sort()**: Sorts the list in place.

```
fruits = ["banana", "apple", "cherry"]
fruits.sort()
print(fruits)  # Output: ['apple', 'banana', 'cherry']
```

9. **reverse()**: Reverses the list in place.

```
fruits = ["apple", "banana", "cherry"]
fruits.reverse()
print(fruits)  # Output: ['cherry', 'banana', 'apple']
```

10. **clear()**: Removes all elements from the list.

```
fruits = ["apple", "banana", "cherry"]
fruits.clear()
print(fruits)  # Output: []
```

---

2. Tuples
Tuples are ordered, immutable collections of items.

Methods and Operations

1. **count()**: Counts the number of occurrences of a value.

```
fruits = ("apple", "banana", "cherry", "banana")
print(fruits.count("banana"))  # Output: 2
```

2. **index()**: Returns the index of the first occurrence of a value.

```
fruits = ("apple", "banana", "cherry")
print(fruits.index("banana"))  # Output: 1
```

---

3. Dictionaries
Dictionaries are unordered collections of key-value pairs.

Methods and Operations

1. **keys()**: Returns a list of all keys.

```
person = {"name": "Alice", "age": 25}
print(person.keys())  # Output: dict_keys(['name', 'age'])
```

2. **values()**: Returns a list of all values.

```
person = {"name": "Alice", "age": 25}
print(person.values())  # Output: dict_values(['Alice', 25])
```

3. **items()**: Returns a list of key-value pairs as tuples.

```
person = {"name": "Alice", "age": 25}
print(person.items())  # Output: dict_items([('name', 'Alice'), ('age', 25)])
```

4. **get()**: Returns the value for a key (or a default value if the key doesn't exist).

```
person = {"name": "Alice", "age": 25}
print(person.get("name", "Unknown"))  # Output: 'Alice'
print(person.get("address", "Unknown"))  # Output: 'Unknown'
```

5. **update()**: Updates the dictionary with key-value pairs from another dictionary.

```
person = {"name": "Alice", "age": 25}
person.update({"age": 26, "city": "New York"})
print(person)  # Output: {'name': 'Alice', 'age': 26, 'city': 'New York'}
```

6. **pop()**: Removes and returns the value for a key.

```
person = {"name": "Alice", "age": 25}
print(person.pop("age"))  # Output: 25
print(person)  # Output: {'name': 'Alice'}
```

7. **clear()**: Removes all key-value pairs.

```
person = {"name": "Alice", "age": 25}
person.clear()
print(person)  # Output: {}
```

---

4. Sets

Sets are unordered collections of unique elements.

Methods and Operations

1. **add()**: Adds an element to the set.

```
fruits = {"apple", "banana"}
fruits.add("cherry")
print(fruits)  # Output: {'apple', 'banana', 'cherry'}
```

2. **remove()**: Removes an element from the set (raises an error if the element doesn't exist).

```
fruits = {"apple", "banana", "cherry"}
fruits.remove("banana")
print(fruits)  # Output: {'apple', 'cherry'}
```

3. **discard()**: Removes an element from the set (does nothing if the element doesn't exist).

```
fruits = {"apple", "banana", "cherry"}
fruits.discard("banana")
print(fruits)  # Output: {'apple', 'cherry'}
```

4. **pop()**: Removes and returns an arbitrary element.

```
fruits = {"apple", "banana", "cherry"}
print(fruits.pop())  # Output: 'apple' (or any other element)
```

5. **clear()**: Removes all elements.

```
fruits = {"apple", "banana", "cherry"}
fruits.clear()
print(fruits)  # Output: set()
```

6. **Set Operations**
    - Union (|): Combines two sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1 | set2)  # Output: {1, 2, 3, 4, 5}
```

    - Intersection (&): Finds common elements.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1 & set2)  # Output: {3}
```

    - Difference (-): Finds elements in the first set but not in the second.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1 - set2)  # Output: {1, 2}
```

➢Control Flow

1. Advanced Conditional Statements (if, elif, else)

Example 1: Nested Conditions – Login System
A simple login system that checks both username and password.

```
username = "admin"
password = "secure123"

user_input = input("Enter username: ")
pass_input = input("Enter password: ")

if user_input == username:
    if pass_input == password:
        print("Login successful!")
    else:
        print("Incorrect password!")
else:
    print("Username not found!")
```

Output (if correct credentials are entered):
Enter username: admin
Enter password: secure123
Login successful!

Output (if wrong password is entered):
Enter username: admin
Enter password: wrongpass
Incorrect password!

Example 2: Decision Making with Multiple Conditions
Check if a student has passed or failed based on multiple criteria.

```
math = 85
science = 75
english = 40

if math >= 50 and science >= 50 and english >= 50:
    print("Congratulations! You passed all subjects.")
elif (math < 50 and science >= 50 and english >= 50) or (math >= 50 and science < 50 and
english >= 50) or (math >= 50 and science >= 50 and english < 50):
    print("You need to reappear in one subject.")
else:
    print("You failed in multiple subjects.")
```

Output:
You need to reappear in one subject.

---

2. Advanced Loops (for, while)

Example 3: Prime Number Checker (for loop + break)
Check if a given number is prime.

```
num = 29
is_prime = True

if num > 1:
    for i in range(2, int(num ** 0.5) + 1):  # Loop from 2 to sqrt(num)
        if num % i == 0:
            is_prime = False
            break

    if is_prime:
        print(num, "is a prime number")
    else:
        print(num, "is not a prime number")
else:
    print(num, "is not a prime number")
```

Output:
29 is a prime number

Example 4: Fibonacci Sequence (while loop)
Generate Fibonacci numbers up to n terms.

```
n = 10
a, b = 0, 1
count = 0

while count < n:
    print(a, end=" ")
    temp = a + b
    a = b
    b = temp
    count += 1
```

Output:
0 1 1 2 3 5 8 13 21 34

Example 5: Nested Loops – Printing a Pattern
Print a pyramid pattern using nested loops.

```
rows = 5

for i in range(1, rows + 1):
    for j in range(rows - i):
        print(" ", end="")  # Print spaces
    for k in range(2 * i - 1):
        print("*", end="")  # Print stars
    print()
```

Output:

```
    *
   ***
  *****
 *******
*********
```

---

3. Advanced Loop Control Statements (break, continue, pass)

Example 6: Skipping Even Numbers Using continue
Print only odd numbers from a given list.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for num in numbers:
    if num % 2 == 0:
        continue  # Skip even numbers
    print(num, end=" ")
```

Output:
1 3 5 7 9

Example 7: Finding First Repeating Character Using break
Find the first repeating character in a string.

```
text = "programming"
seen_chars = set()

for char in text:
    if char in seen_chars:
        print("First repeating character:", char)
        break
    seen_chars.add(char)
```

Output:
First repeating character: r

Example 8: Using pass in Function Placeholder
Use pass when a function is yet to be implemented.

```
def future_function():
    pass  # Placeholder for future implementation

print("This function does nothing right now.")
```

Output:
This function does nothing right now.

➢ Functions and Modules

provides powerful **functions** and **modules** that enhance code reusability, readability, and modularity. Below, we'll cover:

1. **Functions**
    - User-defined functions
    - Function arguments (positional, keyword, default, arbitrary)
    - Lambda functions
    - Recursive functions
2. **Modules**
    - Creating and importing modules
    - Built-in modules
    - Using __name__ == "__main__"

Each section includes **definitions, syntax, and advanced examples**.

---

1. Functions

Definition:
A function is a reusable block of code that performs a specific task. Functions help modularize code and improve maintainability.
Syntax:

```
def function_name(parameters):
    """Docstring describing function"""
    # Code block
    return value  # Optional
```

1.1 User-Defined Functions

Example 1: Function with Default and Keyword Arguments

```
def greet(name, message="Hello"):
    return f"{message}, {name}!"

print(greet("Alice"))  # Uses default message
print(greet("Bob", "Good morning"))  # Custom message
```

Output:
Hello, Alice!
Good morning, Bob!

1.2 Function Arguments

- Use ***args** to accept any number of positional arguments.
- Use ****kwargs** to accept any number of keyword arguments.

Example 2: Positional, Keyword, and Arbitrary Arguments (*args, **kwargs)

```
def details(name, age, *hobbies, **info):
    print(f"Name: {name}, Age: {age}")
    print("Hobbies:", ", ".join(hobbies))
    for key, value in info.items():
        print(f"{key}: {value}")

details("Alice", 25, "Reading", "Cycling", city="New York", profession="Engineer")
```

Output:
Name: Alice, Age: 25
Hobbies: Reading, Cycling
city: New York
profession: Engineer

1.3 Lambda (Anonymous) Functions

Example 3: Lambda Function for Sorting a List of Tuples

```
students = [("Alice", 85), ("Bob", 92), ("Charlie", 78)]
sorted_students = sorted(students, key=lambda x: x[1], reverse=True)
print(sorted_students)
```

Output:
[('Bob', 92), ('Alice', 85), ('Charlie', 78)]

Example 4: Lambda with map() and filter()

```
numbers = [1, 2, 3, 4, 5]

squared = list(map(lambda x: x ** 2, numbers))
evens = list(filter(lambda x: x % 2 == 0, numbers))

print("Squared:", squared)
print("Even Numbers:", evens)
```

Output:
Squared: [1, 4, 9, 16, 25]
Even Numbers: [2, 4]

1.4 Recursive Functions

Example 5: Factorial Using Recursion

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

print(factorial(5))
```

Output:
120

Example 6: Fibonacci Sequence Using Recursion

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

for i in range(6):
    print(fibonacci(i), end=" ")
```

Output:
0 1 1 2 3 5

---

2. Modules 📦

2.1 Creating and Importing Modules

A module is a Python file (.py) that contains functions, classes, or variables, which can be imported and reused.

Example 7: Creating and Using a Custom Module

Step 1: Create math_utils.py

```
# math_utils.py
def add(a, b):
    return a + b

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

Step 2: Import and Use in Another File

```
import math_utils

print(math_utils.add(5, 3))  # Output: 8
print(math_utils.subtract(10, 4))  # Output: 6
```

2.2 Importing Specific Functions from a Module

Example 8: Using from ... import

```
from math_utils import add

print(add(10, 7))  # Output: 17
```

2.3 Built-in Modules

Python comes with built-in modules like math, random, datetime, etc.

Example 9: Using math Module

```
import math

print(math.sqrt(16))  # Square root
print(math.factorial(5))  # Factorial
print(math.pi)  # Value of π
```

Output:

```
4.0
120
3.141592653589793
```

2.4 Using __name__ == "__main__" in Modules

- Use if __name__ == "__main__": to run code only when the module is executed directly (not when imported).

Example 10: Understanding __main__ in Modules
Step 1: Create greet.py

```
# greet.py
def say_hello():
   print("Hello from greet module!")

if __name__ == "__main__":
   say_hello()  # This will only run when executed directly
```

Step 2: Import greet.py into Another Script

```
import greet
```

- If **executed directly**, say_hello() runs.
- If **imported**, say_hello() does **not** run.

Summary

Concept	Key Features
Functions	Code reusability, modularity, def, return values
Arguments	Default, keyword, *args, **kwargs
Lambda Functions	Short, anonymous, used in map(), filter()
Recursion	Function calls itself, used in factorial & Fibonacci
Modules	Python files for reusable code (import, from ... import)
Built-in Modules	math, random, datetime, etc.
__name__ == "__main__"	Runs only if script is executed directly

➢ Object-Oriented Programming (OOP)

Python is an **object-oriented programming (OOP)** language, which allows structuring code using **classes and objects**. OOP helps with **modularity, reusability, and maintainability**.

Key OOP Concepts:

1. **Classes and Objects**
2. **Encapsulation** (Using private, protected, public attributes)
3. **Inheritance** (Single, Multiple, Multilevel)
4. **Polymorphism** (Method Overriding and Operator Overloading)
5. **Abstraction** (Using ABC module)

---

1. Classes and Objects

Definition:

A **class** is a blueprint for creating objects, and an **object** is an instance of a class.

Syntax:

```
class ClassName:
    def __init__(self, param1, param2):
        self.param1 = param1
        self.param2 = param2

    def method_name(self):
        return f"Using {self.param1} and {self.param2}"

# Creating an object
obj = ClassName("Value1", "Value2")
print(obj.method_name())
```

Example 1: Creating a Simple Class and Object

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def show_details(self):
        return f"Car: {self.brand} {self.model}"

# Object creation
car1 = Car("Toyota", "Corolla")
print(car1.show_details())
```

Output: Car: Toyota Corolla

2. Encapsulation 🔒

Definition:
Encapsulation is the **hiding of internal details** of a class and restricting access using **private (__) and protected (_) attributes**.

Example 2: Using Public, Protected, and Private Variables

```
class BankAccount:
    def __init__(self, account_holder, balance):
        self.account_holder = account_holder  # Public attribute
        self._account_type = "Savings"  # Protected attribute
        self.__balance = balance  # Private attribute

    def get_balance(self):
        return self.__balance  # Accessing private attribute inside the class

# Object creation
account = BankAccount("Alice", 5000)
print(account.account_holder)  # Public - Allowed
print(account._account_type)  # Protected - Allowed, but not recommended
# print(account.__balance)  # Private - Error

print(account.get_balance())  # Accessing private attribute using a method
```

Output:
Alice
Savings
5000

---

3. Inheritance 👨‍👦

Definition:
Inheritance allows a child class to inherit methods and attributes from a parent class.

Types of Inheritance in Python:

- **Single Inheritance** (One parent → One child)
- **Multiple Inheritance** (One child → Multiple parents)
- **Multilevel Inheritance** (Child → Parent → Grandparent)

Example 3: Single Inheritance

```
class Animal:
    def speak(self):
        return "Animal speaks"

class Dog(Animal):  # Inheriting from Animal class
    def speak(self):
        return "Bark!"

dog = Dog()
print(dog.speak())  # Overridden method
```

Output:
Bark!

Example 4: Multiple Inheritance

```
class Engine:
    def start(self):
        return "Engine started"

class Wheels:
    def roll(self):
        return "Wheels rolling"

class Car(Engine, Wheels):  # Inheriting from two classes
    def drive(self):
        return "Car is driving"

car = Car()
print(car.start())  # From Engine class
print(car.roll())   # From Wheels class
print(car.drive())  # From Car class
```

Output:
Engine started
Wheels rolling
Car is driving

---

4. Polymorphism 🎭

Definition:
Polymorphism allows objects of different classes to be treated as the same type through **method overriding** and **operator overloading**.

Example 5: Method Overriding

```
class Bird:
    def sound(self):
        return "Some bird sound"

class Sparrow(Bird):
    def sound(self):
        return "Chirp Chirp"

class Crow(Bird):
    def sound(self):
        return "Caw Caw"

birds = [Sparrow(), Crow()]
for bird in birds:
    print(bird.sound())
```

Output:
Chirp Chirp
Caw Caw

Example 6: Operator Overloading (+ for Custom Objects)

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(3, 4)
v2 = Vector(2, 6)
print(v1 + v2)  # Overloaded `+` operator
```

Output: Vector(5, 10)

5. Abstraction 🎭

Definition:
Abstraction hides implementation details and exposes only the necessary features. Python achieves this using the ABC module.

Example 7: Abstract Class with Abstract Method

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass  # Abstract method

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2  # Implementing abstract method

circle = Circle(5)
print(circle.area())
```

Output:
78.5

---

6. super() Keyword 🦸

Example 8: Using super() to Call Parent Class Method

```
class Parent:
    def show(self):
        return "Parent method"

class Child(Parent):
    def show(self):
        return super().show() + " & Child method"

child = Child()
print(child.show())
```

Output: Parent method & Child method

7. Class Methods and Static Methods

Definition:

- **Class Method (@classmethod)** works on class-level attributes.
- **Static Method (@staticmethod)** doesn't modify class or instance attributes.

Example 9: Using Class and Static Methods

```
class Employee:
    company = "TechCorp"  # Class variable

    @classmethod
    def change_company(cls, new_name):
        cls.company = new_name  # Changing class-level attribute

    @staticmethod
    def info():
        return "This is an Employee class"

print(Employee.company)  # Before change
Employee.change_company("NewTech")
print(Employee.company)  # After change

print(Employee.info())  # Calling static method
```

Output:

```
TechCorp
NewTech
This is an Employee class
```

Summary of OOP Concepts in Python

Concept	Description
Class & Object	Blueprint (class) and instance (object)
Encapsulation	Restricts access using _protected and __private attributes
Inheritance	Parent-child relationship (Single, Multiple, Multilevel)
Polymorphism	Different behaviors for the same method (Method Overriding, Operator Overloading)
Abstraction	Hides details, uses ABC (abstract base class)
super()	Calls parent class methods inside child classes
Class & Static Methods	@classmethod modifies class attributes, @staticmethod doesn't

Exception Handling in Python ⚠

Python provides a robust mechanism to handle runtime errors using **exception handling**. Exception handling prevents program crashes and ensures smooth execution.

---

1. What is an Exception?

An **exception** is an error that occurs during the execution of a program, disrupting its normal flow. Examples include:

- **ZeroDivisionError** (Dividing by zero)
- **TypeError** (Invalid type operations)
- **FileNotFoundError** (File not found)
- **IndexError** (Invalid index access)

---

2. Exception Handling Using try-except

Syntax:

```
try:
    # Code that may raise an exception
    risky_operation()
except ExceptionType:
    # Handle the exception
    handle_error()
```

Example 1: Handling Division by Zero

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
```

Output: Error: Cannot divide by zero!

---

3. Handling Multiple Exceptions

We can handle multiple exceptions by specifying multiple except blocks.

Example 2: Handling Multiple Errors

```
try:
    num = int(input("Enter a number: "))
    print(10 / num)
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
except ValueError:
    print("Error: Invalid input! Enter a number.")
```

Input 1: 0
Output 1: Error: Cannot divide by zero!

Input 2: hello
Output 2: typescript

Error: Invalid input! Enter a number.

---

4. Using else with try-except
The else block runs **only if no exception occurs**.

Example 3: else Block Usage

```
try:
    num = int(input("Enter a number: "))
    print(f"Success! You entered: {num}")
except ValueError:
    print("Error: Invalid number!")
else:
    print("No exceptions occurred.")
```

Input:
42

Output:
Success! You entered: 42
No exceptions occurred.

---

5. Using finally for Cleanup
The finally block **always executes**, whether an exception occurs or not.

Example 4: finally Block Usage

```
try:
    f = open("data.txt", "r")
    content = f.read()
except FileNotFoundError:
    print("Error: File not found!")
finally:
    print("Execution completed.")  # Always runs
```

Output:
Error: File not found!
Execution completed.

6. Raising Custom Exceptions (raise)
We can manually raise exceptions using raise.

Example 5: Raising an Exception

```
def check_age(age):
    if age < 18:
        raise ValueError("Age must be 18 or above!")
    return "Access granted!"

try:
    print(check_age(16))
except ValueError as e:

    print(f"Error: {e}")
```

Output:
Error: Age must be 18 or above!

---

7. Custom Exceptions (User-Defined Exceptions)
We can create our own exception classes by inheriting from Exception.

Example 6: Custom Exception Class

```
class NegativeNumberError(Exception):
    def __init__(self, message="Negative numbers are not allowed!"):
        self.message = message
        super().__init__(self.message)

def process_number(num):
    if num < 0:
        raise NegativeNumberError()
    return f"Processed number: {num}"

try:
    print(process_number(-5))
except NegativeNumberError as e:
    print(f"Error: {e}")
```

Output:
Error: Negative numbers are not allowed!

Summary of Exception Handling

Concept	Description
try-except	Handles exceptions gracefully
Multiple except	Catches multiple error types
else Block	Runs if no exception occurs
finally Block	Executes always, even if an error occurs
raise	Manually raises exceptions
Custom Exceptions	User-defined error handling

File Handling in Python

File handling in Python allows us to **read, write, update, and delete files**. Python provides built-in functions for handling files using the open() function.

---

1. Opening a File in Python
The open() function is used to open a file.

Syntax:
file = open("filename.txt", "mode")

Mode	Description
'r'	Read (default), error if file does not exist
'w'	Write, creates a new file if not exists, overwrites existing content
'a'	Append, adds data to an existing file
'x'	Create, creates a file, fails if file exists
'b'	Binary mode (rb, wb, ab) for images, PDFs, etc.

---

2. Reading a File (r Mode)
Reads content from an existing file.

Example 1: Reading a File

```
file = open("example.txt", "r")
content = file.read()  # Reads the entire file
print(content)
file.close()
```

Output (if example.txt contains Hello, Python!):
Hello, Python!

Reading Line by Line (readline() & readlines())

```
file = open("example.txt", "r")

print(file.readline())  # Reads one line
print(file.readlines())  # Reads all lines as a list

file.close()
```

---

3. Writing to a File (w Mode)

- If the file **exists**, it **overwrites** the content.
- If the file **does not exist**, it **creates** a new one.

Example 2: Writing to a File

```
file = open("example.txt", "w")
file.write("Hello, World!\nWelcome to Python File Handling.")
file.close()

# Reading the file to verify
file = open("example.txt", "r")
print(file.read())
file.close()
```

Output (example.txt content):
Hello, World!
Welcome to Python File Handling.

---

4. Appending Data to a File (a Mode)

- Adds data to the **end of the file** without overwriting existing content.

Example 3: Appending to a File

```
file = open("example.txt", "a")
file.write("\nThis is an appended line.")
file.close()

# Reading the file to verify
file = open("example.txt", "r")
print(file.read())
file.close()
```

Output (example.txt content):
Hello, World!
Welcome to Python File Handling.
This is an appended line.

---

5. Creating a New File (x Mode)

- x mode **creates a new file** but throws an error if the file already exists.

Example 4: Creating a File

```
try:
   file = open("newfile.txt", "x")
   print("File created successfully!")
   file.close()
except FileExistsError:
   print("File already exists!")
```

Output (if file exists):
File already exists!

---

6. Working with Binary Files (rb, wb, ab)
Used for handling **images, PDFs, audio, etc.**

Example 5: Copying an Image

```
with open("image.jpg", "rb") as img_file:
   content = img_file.read()

with open("copy.jpg", "wb") as new_file:
   new_file.write(content)

print("Image copied successfully!")
```

This will copy image.jpg to copy.jpg.

---

7. Using with Statement (Best Practice)
Using with open() as **automatically closes the file** after execution.

Example 6: Using with for Safer File Handling

```
with open("example.txt", "r") as file:
   content = file.read()
   print(content)  # No need to call file.close()
```

This ensures the file closes automatically after reading.

8. Deleting a File
We can use the os module to delete a file.

Example 7: Deleting a File

```
import os

if os.path.exists("example.txt"):
   os.remove("example.txt")
   print("File deleted successfully!")
else:
   print("File not found!")
```

Output:
File deleted successfully!

---

9. File Handling with Exception Handling

Example 8: Handling File Not Found Error

```
try:
   with open("nonexistent.txt", "r") as file:
      content = file.read()
      print(content)
except FileNotFoundError:
   print("Error: File not found!")
```

Output:
Error: File not found!

- **Summary of File Handling in Python** 📜

Operation	Mode	Description
Read	'r'	Reads a file (error if file not found)
Write	'w'	Creates file if not exists, overwrites content
Append	'a'	Adds content to an existing file
Create	'x'	Creates file (error if file exists)
Binary Read	'rb'	Reads binary files (images, PDFs)
Binary Write	'wb'	Writes binary files
Delete	os.remove()	Deletes a file

Advanced Python Concepts

In addition to core Python functionality, there are **advanced concepts** that allow for more powerful and efficient coding. These concepts cover a wide range of features, from **decorators** to **context managers** and **multithreading**.

---

1. Decorators
Decorators are functions that modify the behavior of other functions or methods. They provide an elegant way to enhance or alter functionality without changing the core logic.

Definition:
A **decorator** is a higher-order function that takes another function as input and extends or alters its behavior.

Syntax:

```
def decorator(func):
    def wrapper():
        print("Something before the function.")
        func()
        print("Something after the function.")
    return wrapper

@decorator
def say_hello():
    print("Hello!")

say_hello()
```

Example 1: Simple Decorator

```
def decorator(func):
    def wrapper():
        print("Before function execution")
        func()
        print("After function execution")
    return wrapper

@decorator
def greet():
    print("Hello!")

greet()
```

Output:
Before function execution
Hello!
After function execution

---

2. Generators 🔄
A **generator** is a special type of iterator in Python that allows you to **iterate over a sequence** without storing the entire sequence in memory. It uses the yield keyword to produce values.

Definition:
Generators allow you to **generate values one at a time** using the yield statement instead of returning them all at once. They are efficient for working with large datasets.

Syntax:
```
def generator_function():
    yield value
```

Example 2: Generator

```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1

counter = count_up_to(5)
for num in counter:
    print(num)
```

Output:
1
2
3
4
5

---

3. Context Managers

Context managers are used to manage resources such as **files, network connections, or locks**. The most common use is the with statement, which handles setup and teardown actions automatically.

Definition:
A context manager defines methods __enter__ and __exit__ to allocate and release resources efficiently.

Syntax:

```
class MyContextManager:
    def __enter__(self):
        # Setup code
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        # Cleanup code
        pass

with MyContextManager() as cm:
    # Code inside the context
    Pass
```

Example 3: Context Manager

```
class FileOpener:
    def __enter__(self):
        self.file = open('example.txt', 'w')
        return self.file

    def __exit__(self, exc_type, exc_value, traceback):
        self.file.close()

with FileOpener() as file:
    file.write("Hello, Context Manager!")
```

---

4. Lambda Functions

A **lambda function** is a small anonymous function that is defined using the lambda keyword. It's used for short, simple operations that don't need a full function definition.

Definition:
Lambda functions are defined in a single line and can take any number of arguments but have only one expression.

Syntax:

```
lambda arguments: expression
```

Example 4: Lambda Function

```
add = lambda x, y: x + y
print(add(5, 10))  # Output: 15
```

---

5. List Comprehensions 📜

List comprehensions provide a concise way to create lists by specifying an expression and an optional condition.

Definition:

A list comprehension combines **loops** and **conditionals** into a single line.

Syntax:

```
[expression for item in iterable if condition]
```

Example 5: List Comprehension

```
numbers = [1, 2, 3, 4, 5]
squared = [x**2 for x in numbers if x % 2 == 0]
print(squared)                                  #Output: [4, 16]
```

---

6. Metaclasses 🤯

A **metaclass** is a "class of a class" that defines the behavior of class creation. It allows customization of class instantiation and modification.

Definition:

Metaclasses define how classes themselves behave and can be used to control class construction, validation, etc.

Syntax:

```
class MyMeta(type):
    def __new__(cls, name, bases, attrs):
        # Custom class behavior
        return super().__new__(cls, name, bases, attrs)

class MyClass(metaclass=MyMeta):
    pass
```

Example 6: Using Metaclasses

```
class Meta(type):
    def __new__(cls, name, bases, attrs):
        attrs['class_name'] = name
        return super().__new__(cls, name, bases, attrs)
class MyClass(metaclass=Meta):
    pass

print(MyClass.class_name)  # Output: MyClass
```

7. Abstract Base Classes (ABC) 🏛️
An **abstract base class (ABC)** defines a common interface for a group of related classes but cannot be instantiated. It forces subclasses to implement specific methods.

Definition:
ABCs allow the definition of **abstract methods** (methods that must be overridden in subclasses).

Syntax:
```
from abc import ABC, abstractmethod

class MyAbstractClass(ABC):
    @abstractmethod
    def abstract_method(self):
        pass
```

Example 7: Abstract Base Class
```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return "Woof"

dog = Dog()
print(dog.sound())  # Output: Woof
```

---

8. Multithreading & Multiprocessing 🧠💻

Multithreading and **multiprocessing** are techniques used for executing multiple tasks simultaneously. **Multithreading** is better for I/O-bound tasks, while **multiprocessing** is better for CPU-bound tasks.

Definition:
- **Multithreading**: Running multiple threads (lightweight tasks) within a single process.
- **Multiprocessing**: Running multiple processes on multiple CPUs to parallelize CPU-bound tasks.

Example 8: Multithreading

```
import threading

def print_numbers():
    for i in range(5):
        print(i)

thread = threading.Thread(target=print_numbers)
thread.start()
thread.join()  # Wait for thread to finish
```

Example 9: Multiprocessing

```
import multiprocessing

def square_number(n):
    return n * n

if __name__ == "__main__":
    with multiprocessing.Pool(4) as pool:
        results = pool.map(square_number, [1, 2, 3, 4, 5])
    print(results)
```

---

9. Coroutine and Asyncio ⏳

Coroutines and the **asyncio** module are used for **asynchronous programming**. They allow efficient handling of I/O-bound tasks without blocking the main thread.

Definition:
A **coroutine** is a function that can yield control back to the event loop, allowing other tasks to run concurrently.

Syntax:

```
import asyncio

async def my_coroutine():
    await asyncio.sleep(1)
    print("Done")

asyncio.run(my_coroutine())
```

---

10. Type Hinting

introduced **type hinting** to improve code readability and assist with debugging. It doesn't enforce types but provides hints about expected data types.

Definition:
Type hinting is a way of specifying the expected types of variables, function arguments, and return values.

Syntax:

```
def add(a: int, b: int) -> int:
   return a + b
```

Example 10: Type Hinting

```
def greet(name: str) -> str:
   return f"Hello, {name}"

print(greet("Alice"))  # Output: Hello, Alice
```

Summary of Advanced Python Concepts

Concept	Description
Decorators	Modify or enhance functions/methods
Generators	Produce values lazily using yield
Context Managers	Efficient resource management with with
Lambda Functions	Small anonymous functions
List Comprehensions	Concise way to create lists
Metaclasses	Customize class creation behavior
Abstract Base Classes	Enforce method implementation in subclasses
Multithreading & Multiprocessing	Parallel execution of tasks
Asyncio & Coroutines	Asynchronous programming for I/O-bound tasks
Type Hinting	Specify data types to improve code clarity

➢Automation & Scripting in Python

Python is widely used for **automation** and **scripting tasks** because of its simplicity, powerful libraries, and ease of integration with other systems. Whether it's automating repetitive tasks, processing data, interacting with web services, or managing files, Python has tools to make it all easier.

---

1. Automating File and Directory Management

Python's os, shutil, and pathlib modules help automate tasks like **creating, deleting, moving**, and **renaming files and directories**.

Example 1: File Operations with os

```
import os

# Create a new directory
os.mkdir('my_directory')

# Rename a file or directory
os.rename('my_directory', 'new_directory')

# Remove a directory
os.rmdir('new_directory')

# Check if a file exists
if os.path.exists('file.txt'):
   print("File exists!")
else:
   print("File not found!")
```

---

3. Web Scraping Automation

Python libraries like **requests**, **BeautifulSoup**, and **Selenium** can be used to extract and process data from websites. This is ideal for gathering information or automating tasks that require data from the web.

Web Scraping with BeautifulSoup

```
import requests
from bs4 import BeautifulSoup
```

---

4. Automating GUI Interactions

Using **pyautogui**, Python can automate interactions with the graphical user interface (GUI), like mouse movements, clicks, keyboard presses, and more.

---

5. Automating System Tasks
You can automate system tasks, like running commands or launching programs, using the subprocess module.

6. Automating Web Interaction with Selenium
Selenium allows automation of web browsers. You can control a browser to automate tasks like logging into a website, filling forms, clicking buttons, and scraping dynamic content.

---

7. Automating Data Entry into Forms
You can automate data entry into web forms using Selenium. This is especially useful for testing or submitting forms automatically.

---

8. Automating Database Operations
You can automate database operations using **sqlite3**, **SQLAlchemy**, or other database connectors to query or manipulate database records.

---

9. Automating Social Media Posts
Using the **tweepy** library for Twitter or the **facebook-sdk** for Facebook, you can automate the posting of updates, responses, or interactions.

---

10. Automating PDF Operations
Python libraries like **PyPDF2** or **pdfminer** help automate tasks like extracting text from PDFs, merging or splitting PDFs, or adding watermarks.

Summary of Automation & Scripting Tasks

Task	Libraries/Tools	Description
File Management	os, shutil, pathlib	Create, delete, move, and rename files and directories
Email Automation	smtplib, email	Automate sending emails, with or without attachments
Web Scraping	requests, BeautifulSoup,	

Testing & Debugging in Python

Testing and debugging are crucial for maintaining the quality of code. Python provides multiple tools for identifying and fixing issues in your programs. These tools help ensure that your code is working correctly, is free from bugs, and is behaving as expected.

---

1. Debugging in Python

1.1. Using pdb (Python Debugger)

The **pdb** module is Python's built-in debugger. It allows you to step through your code, inspect variables, and modify execution flow.

How to Use pdb:

1. Import the pdb module.
2. Set a breakpoint using pdb.set_trace() in your code.
3. Use debugging commands like n (next), s (step), c (continue), q (quit) during debugging.

Example 1: Debugging with pdb

```
import pdb

def calculate(a, b):
    result = a + b
    pdb.set_trace()  # Set a breakpoint
    return result

x = 5
y = 10
print(calculate(x, y))
```

Debugger Commands:

- n (next): Go to the next line of code.
- s (step): Step into functions.
- c (continue): Continue execution until the next breakpoint.
- q (quit): Quit debugging.

---

2. Unit Testing

2.1. Introduction to Unit Testing

Unit testing involves testing individual units or functions of your program to ensure they are working as expected. Python provides the unittest framework for this purpose.

How to Use unittest:

1. Create a test class that inherits from unittest.TestCase.
2. Define test methods using assert statements to check expected results.
3. Run the tests with unittest.main().

Example 2: Basic Unit Testing with unittest

```
import unittest

# Function to test
def add(a, b):
    return a + b

# Test class
class TestMathOperations(unittest.TestCase):

    def test_add(self):
        self.assertEqual(add(2, 3), 5)  # Check if add(2, 3) returns 5

    def test_add_negative(self):
        self.assertEqual(add(-2, -3), -5)  # Check if add(-2, -3) returns -5

if __name__ == '__main__':
    unittest.main()
```

Output (if tests pass):

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

---

3. Test Coverage 🏆

Test coverage measures how much of your code is exercised by your test suite. Tools like **coverage.py** can help you check the effectiveness of your tests.

How to Use coverage.py:

1. Install the package: pip install coverage.
2. Run your test suite using coverage run -m unittest.
3. Check the coverage report using coverage report or generate an HTML report with coverage html.

Example 3: Measuring Test Coverage

```
pip install coverage
coverage run -m unittest test_module.py
coverage report
coverage html
```

---

4. Assertions in Python 🛠️

Assertions are used to test if a condition is **True**. If the condition is **False**, an AssertionError is raised. They are useful for debugging or validating assumptions during development.

How to Use Assertions:

```
def divide(a, b):
    assert b != 0, "Division by zero is not allowed!"
    return a / b
```

Example 4: Using Assertions

```
def divide(a, b):
    assert b != 0, "Cannot divide by zero"
    return a / b

result = divide(10, 2)  # This will work
print(result)

result = divide(10, 0)  # This will raise an AssertionError
print(result)
```

---

5. Logging 📝

Instead of using print() statements for debugging, it's better to use Python's **logging** module for tracking your program's behavior. Logging provides different log levels (DEBUG, INFO, WARNING, ERROR, CRITICAL) to categorize messages.

How to Use Logging:

1. Import the logging module.
2. Set the log level using logging.basicConfig().
3. Use logging methods to track events.

Example 5: Using logging for Debugging

```
import logging

# Set up basic configuration for logging
logging.basicConfig(level=logging.DEBUG)

def add(a, b):
    logging.debug(f'Adding {a} and {b}')
    return a + b

result = add(5, 10)
logging.info(f'Result: {result}')
```

Output:
DEBUG:root:Adding 5 and 10
INFO:root:Result: 15

---

6. Mocking
Mocking is used to replace parts of your system under test with **mock objects** that simulate real objects. This is useful for testing in isolation or when external systems are unavailable (like databases or APIs). Python's unittest.mock module allows you to mock objects and functions.

How to Use unittest.mock:
1. Use mock to replace dependencies.
2. Verify calls to the mock using assert_called_with().

Example 6: Mocking with unittest.mock
```
from unittest import mock
import requests

def fetch_data(url):
    response = requests.get(url)
    return response.json()

# Mocking the `requests.get` method
with mock.patch('requests.get') as mock_get:
    mock_get.return_value.json.return_value = {'key': 'value'}
    data = fetch_data('http://example.com')
    print(data)

# Verify if the mock was called with the expected arguments
mock_get.assert_called_with('http://example.com')
```

Output: {'key': 'value'}

---

7. Behavior-Driven Testing (BDD)
Behavior-driven testing (BDD) encourages collaboration between developers, testers, and non-technical team members to define expected behavior. **pytest-bdd** is an extension to the popular pytest framework, designed for writing tests in a natural language format.

How to Use BDD with pytest-bdd:
1. Install the pytest-bdd package: pip install pytest-bdd.
2. Write test cases using syntax.
3. Define step functions to execute the steps in the scenarios.

Example 7: Writing BDD Tests with pytest-bdd

```
Feature: Addition

  Scenario: Adding two numbers
    Given I have two numbers 3 and 5
    When I add them
    Then the result should be 8


from pytest_bdd import given, when, then

@given('I have two numbers 3 and 5')
def numbers():
    return 3, 5

@when('I add them')
def add(numbers):
    return sum(numbers)

@then('the result should be 8')
def check_result(add):
    assert add == 8
```

---

8. Continuous Integration and Testing 🚀

Continuous Integration (CI) tools like **Jenkins**, **Travis CI**, and **GitHub Actions** can automate testing and ensure code quality. These tools automatically run tests whenever changes are pushed to the codebase.

Example 8: Setting Up Tests with GitHub Actions

1. Create a .github/workflows/test.yml file.
2. Configure the workflow to run pytest on push events.

```
name: Python CI
on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
```

```
  - uses: actions/checkout@v2
  - name: Set up Python
    uses: actions/setup-python@v2
    with:
      python-version: '3.x'
  - name: Install dependencies
    run: |
      python -m pip install --upgrade pip
      pip install -r requirements.txt
  - name: Run tests
    run: |
      pytest
```

---

9. Pytest Framework 🧪

pytest is a powerful testing framework for writing simple to complex tests. It is widely used for unit testing, functional testing, and integration testing in Python.

How to Use pytest:

1. Write test functions with assert statements.
2. Use pytest to automatically discover and run tests.

Example 9: Writing Tests with pytest

```
# test_math.py
def add(a, b):
    return a + b

def test_add():
    assert add(3, 5) == 8
    assert add(-1, 1) == 0
```

Run the tests with:

bash

```
pytest test_math.py
```

Output: diff

```
================================== test session starts
================================
collected 2 items

test_math.py ..                                    [100%]

================================ 2 passed in 0.01 seconds
================================
```

Summary of Testing & Debugging Techniques

Tool/Technique	Description
pdb	Python Debugger for stepping through code
unittest	Built-in module for unit testing
coverage.py	Measures test coverage
Assertions	Ensures conditions are met, raising AssertionError
logging	Logs application activity for debugging
unittest.mock	Mocks external dependencies during tests
pytest	Testing framework with rich features and plugins
pytest-bdd	Behavior-Driven Testing with syntax
CI/CD Tools (e.g., GitHub Actions)	Automates testing during code integration

Database Handling in Python

Python provides several libraries and frameworks for connecting to databases, managing data, and performing various database operations. Whether you're working with SQL databases like **SQLite**, **MySQL**, **PostgreSQL**, or NoSQL databases like **MongoDB**, Python has libraries to interact with them effectively.

---

1. Working with SQLite
SQLite is a lightweight, file-based SQL database, and Python has built-in support for it through the **sqlite3** module.

How to Use SQLite in Python:

1. **Connecting** to an SQLite database.
2. **Creating** tables and performing CRUD operations (Create, Read, Update, Delete).
3. **Committing** changes and closing the connection.

---

2. Working with MySQL
To interact with MySQL databases, you can use the **mysql-connector** or **PyMySQL** library.
How to Use MySQL in Python:

1. **Install** the mysql-connector-python package: pip install mysql-connector-python.
2. **Connect** to the MySQL database.
3. Perform **CRUD operations** using cursor.execute().

---

3. Working with PostgreSQL
PostgreSQL is a powerful, open-source relational database. To interact with PostgreSQL databases, you can use the **psycopg2** library.

How to Use PostgreSQL in Python:

1. **Install** the psycopg2 package: pip install psycopg2.
2. **Connect** to PostgreSQL.
3. Perform **CRUD operations** using SQL queries.

---

4. Working with NoSQL Databases (MongoDB)
MongoDB is a popular NoSQL database. You can use the **pymongo** library to interact with MongoDB in Python.

How to Use MongoDB in Python:

1. **Install** the pymongo package: pip install pymongo.
2. **Connect** to MongoDB.
3. Perform **CRUD operations** with the insert_one, find, update_one, and delete_one methods.