**Object-Oriented Programming (OOP)**

**What is OOP?**

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects, which represent real-world entities. These objects encapsulate **data (attributes)** and **behavior (methods)**, making the code more modular and reusable.

(or)

"OOPS" uses objects, which are created from classes, to structure data and code, enabling modular, reusable, and scalable code through concepts like encapsulation, inheritance, polymorphism, and abstraction.

**Key Concepts of OOP in Python**

1. **Class** – A blueprint for creating objects.
2. **Object** – An instance of a class.
3. **Encapsulation** – Hiding the internal details of an object and restricting direct access to some of its components.
4. **Abstraction** – Hiding complex implementation details and exposing only the necessary parts.
5. **Inheritance** – A mechanism that allows a new class to derive properties and behavior from an existing class.
6. **Polymorphism** – The ability to take multiple forms, e.g., methods with the same name but different implementations in different classes.

**OOP follows four main principles:**

1. **Encapsulation** – Hiding the internal state of an object and restricting access to it.
2. **Abstraction** – Hiding complex implementation details and exposing only the necessary parts.
3. **Inheritance** – Enabling new classes to derive properties and methods from existing ones.
4. **Polymorphism** – Allowing the same interface to be used for different underlying data types.

**Why Do We Use OOP?**

OOP is used because it provides:

✅ **Better Code Organization** – It structures code into objects, making it more readable and maintainable.  
✅ **Reusability** – Classes can be reused in different parts of a program, reducing redundancy.  
✅ **Scalability** – Easier to expand and modify code without affecting other parts of the program.  
✅ **Data Security** – Encapsulation ensures that sensitive data is not directly accessible.  
✅ **Code Maintenance** – The modular approach allows updates and debugging without affecting the entire system.

**Example Without OOP (Procedural Approach)**

# Procedural Programming

car\_brand = "Toyota"

car\_model = "Corolla"

def display\_info():

print(f"Car: {car\_brand} {car\_model}")

display\_info()

❌ The above approach lacks **encapsulation**, **reusability**, and **modularity**.

**Example With OOP (Object-Oriented Approach)**

class Car:

def \_\_init\_\_(self, brand, model):

self.brand = brand

self.model = model

def display\_info(self):

print(f"Car: {self.brand} {self.model}")

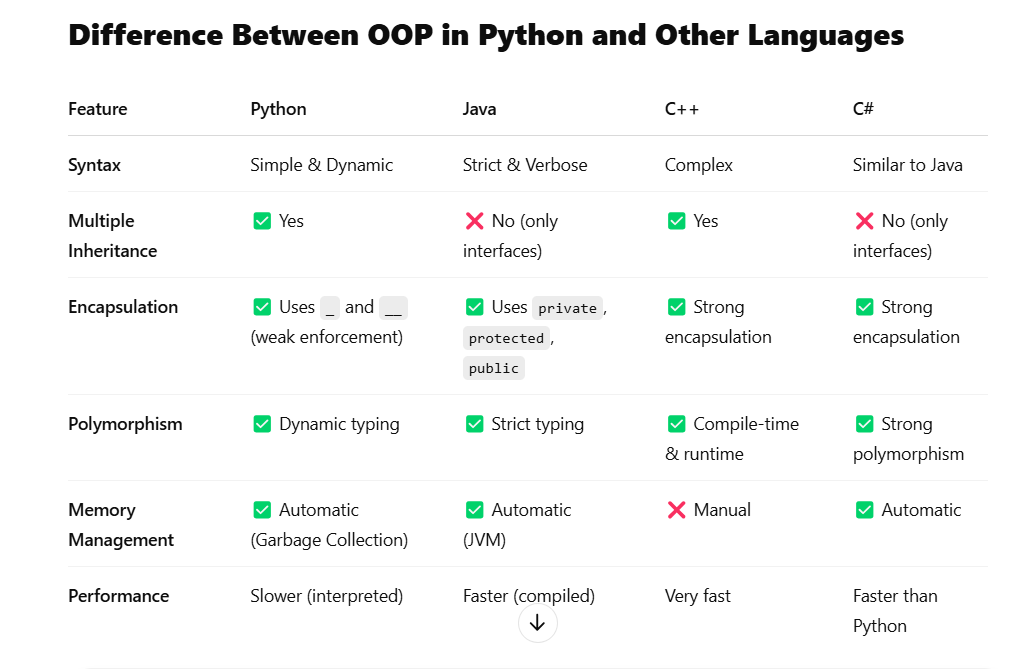
car1 = Car("Toyota", "Corolla")

car1.display\_info()

✅ **More structured, reusable, and scalable!**

**Difference Between OOP in Python and Other Languages**

|  |
| --- |



**Key Differences:**

1. Python is more **dynamic** and **flexible**, while Java, C++, and C# require **strict type declarations**.
2. Python supports **multiple inheritance**, while Java and C# use interfaces.
3. Python is **interpreted**, making it slower than compiled languages like C++ and Java.
4. Python has **automatic memory management**, whereas C++ requires manual handling.

**Benefits of OOP**

**✅ 1. Code Reusability (DRY Principle)**

* Inheritance allows reusing existing code without rewriting it.
* Example: A Car class can be reused in multiple projects.

**✅ 2. Modularity (Easier Code Organization)**

* Classes help break down large code into smaller, manageable parts.
* Example: A BankAccount class can have different methods like deposit() and withdraw().

**✅ 3. Encapsulation (Data Security)**

* Prevents direct modification of sensitive data.
* Example: A private \_\_balance attribute in a BankAccount class.

**✅ 4. Scalability & Flexibility**

* Polymorphism allows the same method to work for different types.
* Example: A make\_sound() method can work for both Dog and Cat classes.

**✅ 5. Real-World Representation**

* Objects model real-world entities, making software design intuitive.
* Example: Car, Employee, Customer, etc.

**What is a Class?**

A **class** in Python is a blueprint or template for creating objects. It defines attributes (variables) and methods (functions) that the objects will have.

**Syntax of a Class in Python**

class ClassName:

# Constructor (optional)

def \_\_init\_\_(self, param1, param2):

self.param1 = param1

self.param2 = param2

# Method

def method\_name(self):

print("Method executed")

* class keyword is used to define a class.
* \_\_init\_\_() is a special method (constructor) that initializes object attributes.
* self represents the instance of the class.

**What is an Object?**

An **object** is an instance of a class. When a class is defined, no memory is allocated until an object is created.

**Creating an Object in Python**

# Define a class

class Car:

def \_\_init\_\_(self, brand, model):

self.brand = brand

self.model = model

def display\_info(self):

print(f"Car: {self.brand} {self.model}")

# Creating an object of the class

car1 = Car("Toyota", "Corolla")

car1.display\_info()

**Explanation:**

* Car is a class that has attributes brand and model and a method display\_info().
* car1 = Car("Toyota", "Corolla") creates an object of the class.
* car1.display\_info() calls the method to display the details of the car.

**1. Class Attributes and Instance Attributes**

* **Class Attributes** are shared among all objects of the class.
* **Instance Attributes** are unique to each object.

class Car:

wheels = 4 # Class attribute (common to all instances)

def \_\_init\_\_(self, brand, model):

self.brand = brand # Instance attribute

self.model = model # Instance attribute

car1 = Car("Toyota", "Corolla")

car2 = Car("Honda", "Civic")

print(car1.wheels) # Output: 4

print(car2.wheels) # Output: 4

print(car1.brand) # Output: Toyota

print(car2.brand) # Output: Honda

**2. The \_\_init\_\_() Method (Constructor)**

* It initializes attributes when an object is created.
* It automatically runs when an object is instantiated.

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

person1 = Person("Alice", 25)

print(person1.name) # Output: Alice

print(person1.age) # Output: 25

**3. Methods in a Python Class – Detailed Explanation**

In Python, methods are functions defined inside a class that operate on class attributes. There are three types of methods:

1. Instance Methods – Work on specific objects (instances) of a class.
2. Class Methods – Work on the class itself, not on instances.
3. Static Methods – Independent functions inside a class that don’t modify class or instance attributes.

**1. Instance Methods (self)**

**Definition**

* Instance methods work with **specific instances** of a class.
* They **can modify instance attributes** and **access instance-specific data**.
* They take self as the first parameter, which represents the instance of the class.

**Example of Instance Method**

class Car:

def \_\_init\_\_(self, brand, model):

self.brand = brand # Instance variable

self.model = model # Instance variable

def display\_info(self): # Instance method

print(f"Car: {self.brand} {self.model}")

# Creating an object

car1 = Car("Toyota", "Corolla")

car1.display\_info() # Output: Car: Toyota Corolla

**Key Points**

✅ Uses self to access instance attributes.  
✅ Can **read and modify** instance variables.  
✅ Called using an **object of the class** (car1.display\_info()).

**2. Class Methods (@classmethod)**

**Definition**

* Class methods work with **the class itself**, not with instances.
* They take cls as the first parameter, which represents the class.
* They **can modify class attributes** (but not instance attributes).
* Decorated with @classmethod.

**Example of Class Method**

class Car:

wheels = 4 # Class variable

def \_\_init\_\_(self, brand, model):

self.brand = brand

self.model = model

@classmethod

def change\_wheels(cls, new\_wheels):

cls.wheels = new\_wheels # Modifying class attribute

@classmethod

def show\_wheels(cls):

print(f"All cars have {cls.wheels} wheels")

# Using class method without creating an object

Car.change\_wheels(6) # Changing class attribute

Car.show\_wheels() # Output: All cars have 6 wheels

**Key Points**

✅ Uses cls instead of self to access class attributes.  
✅ Modifies class variables, affecting all instances.  
✅ Can be called using either **a class or an instance** (Car.show\_wheels() or car1.show\_wheels()).

**3. Static Methods (@staticmethod)**

**Definition**

* Static methods **do not access instance (self) or class (cls) variables**.
* They behave like regular functions but are **inside a class** for better organization.
* Decorated with @staticmethod.

**Example of Static Method**

class MathOperations:

@staticmethod

def add(x, y):

return x + y

@staticmethod

def multiply(x, y):

return x \* y

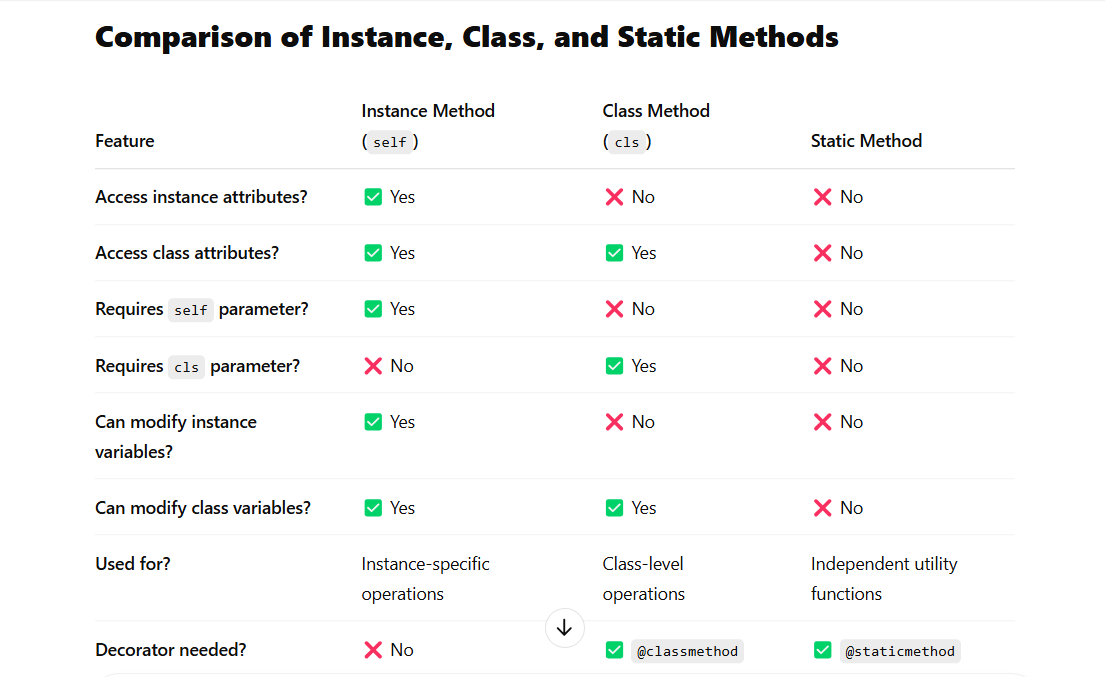
# Calling static methods without creating an object

print(MathOperations.add(5, 10)) # Output: 15

print(MathOperations.multiply(4, 6)) # Output: 24

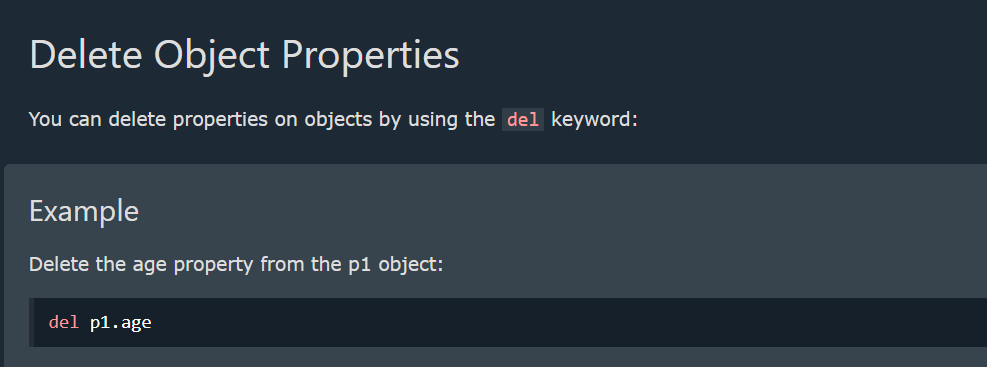
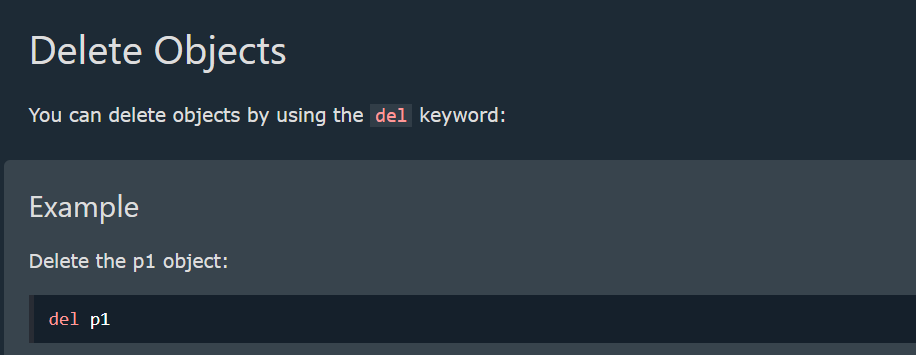
**Key Points**

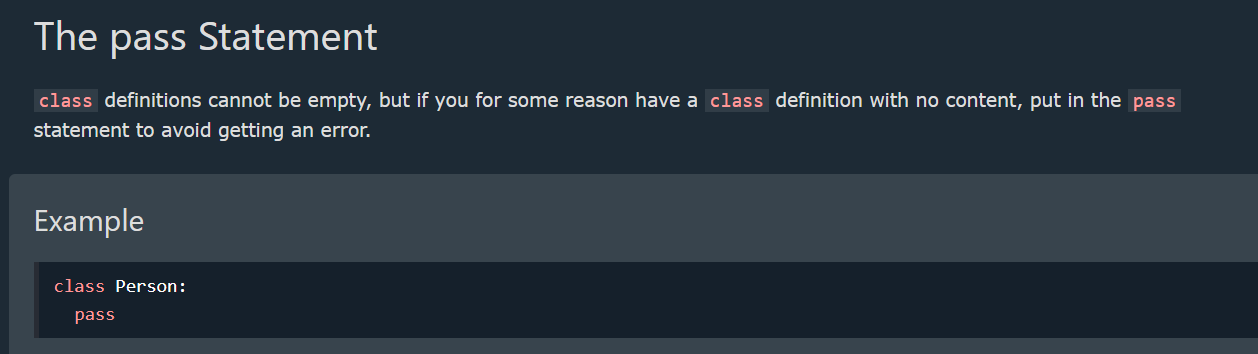
✅ **No self or cls**, meaning it cannot access instance or class attributes.  
✅ Works like a normal function but is grouped logically inside a class.  
✅ Can be called using **a class or an instance** (MathOperations.add(5, 10)).

****

**When to Use Each Method?**

| **Scenario** | **Method to Use** |
| --- | --- |
| Need to modify or access instance attributes | **Instance Method** |
| Need to modify or access class attributes | **Class Method** |
| Need a function inside a class that does not interact with instance or class variables | **Static Method** |



**Inheritance**

**What is Inheritance?**

Inheritance is a fundamental concept of Object-Oriented Programming (OOP) that allows one class (**child/subclass**) to inherit the attributes and methods of another class (**parent/base class**).

It enables **code reusability**, **hierarchical class organization**, and **extensibility** in programs.

**Types of Inheritance in Python**

Python supports five types of inheritance:

1. **Single Inheritance** – One child class inherits from one parent class.
2. **Multiple Inheritance** – A child class inherits from multiple parent classes.
3. **Multilevel Inheritance** – A child class inherits from another child class (grandparent → parent → child).
4. **Hierarchical Inheritance** – Multiple child classes inherit from a single parent class.
5. **Hybrid Inheritance** – A combination of two or more types of inheritance.

**1. Single Inheritance**

A child class inherits from a single parent class.

**Example: Single Inheritance**

# Parent class

class Animal:

def \_\_init\_\_(self, name):

self.name = name

def speak(self):

print("This animal makes a sound.")

# Child class inheriting from Animal

class Dog(Animal):

def speak(self): # Overriding parent method

print(f"{self.name} barks.")

# Creating an object of the child class

dog = Dog("Buddy")

dog.speak() # Output: Buddy barks.

✅ **Code reuse**: The Dog class inherits properties from Animal.  
✅ **Method Overriding**: The speak() method in Dog replaces the one in Animal.

**2. Multiple Inheritance**

A child class inherits from **two or more parent classes**.

**Example: Multiple Inheritance**

# Parent class 1

class Father:

def personality(self):

print("Father is disciplined.")

# Parent class 2

class Mother:

def nature(self):

print("Mother is caring.")

# Child class inheriting from both

class Child(Father, Mother):

def show\_traits(self):

self.personality() # Inherited from Father

self.nature() # Inherited from Mother

# Creating object

child = Child()

child.show\_traits()

# Output:

# Father is disciplined.

# Mother is caring.

✅ **Can inherit from multiple sources**.  
❌ **May cause conflicts** if the same method exists in multiple parent classes.

🔹 **Note**: Python follows the **Method Resolution Order (MRO)** to decide which method to call when multiple parent classes have methods with the same name.

**3. Multilevel Inheritance**

A child class inherits from another child class, forming a **chain of inheritance** (grandparent → parent → child).

**Example: Multilevel Inheritance**

# Grandparent class

class Grandparent:

def show\_grandparent(self):

print("I am the grandparent.")

# Parent class inheriting from Grandparent

class Parent(Grandparent):

def show\_parent(self):

print("I am the parent.")

# Child class inheriting from Parent

class Child(Parent):

def show\_child(self):

print("I am the child.")

# Creating an object of Child

child = Child()

child.show\_grandparent() # Output: I am the grandparent.

child.show\_parent() # Output: I am the parent.

child.show\_child() # Output: I am the child.

✅ **Creates a hierarchy of classes**.  
✅ **Allows gradual specialization of behaviors**.  
❌ **Too many inheritance levels** can make code complex and harder to manage.

**4. Hierarchical Inheritance**

Multiple child classes inherit from a **single parent class**.

**Example: Hierarchical Inheritance**

# Parent class

class Vehicle:

def general\_info(self):

print("Vehicles are used for transportation.")

# Child class 1

class Car(Vehicle):

def car\_info(self):

print("Cars have four wheels.")

# Child class 2

class Bike(Vehicle):

def bike\_info(self):

print("Bikes have two wheels.")

# Creating objects

car = Car()

car.general\_info() # Output: Vehicles are used for transportation.

car.car\_info() # Output: Cars have four wheels.

bike = Bike()

bike.general\_info() # Output: Vehicles are used for transportation.

bike.bike\_info() # Output: Bikes have two wheels.

✅ **Reduces code duplication** since multiple child classes use the same parent class.  
❌ **If changes are made in the parent class, all child classes may be affected**.

**5. Hybrid Inheritance**

A combination of multiple types of inheritance (e.g., Multiple + Multilevel).

**Example: Hybrid Inheritance**

# Parent class

class Engine:

def engine\_info(self):

print("Engine is essential for a vehicle.")

# Parent class 2

class Vehicle:

def vehicle\_info(self):

print("A vehicle is used for transportation.")

# Intermediate class (inherits from Engine and Vehicle)

class Car(Engine, Vehicle):

def car\_info(self):

print("Cars usually have 4 wheels.")

# Final subclass

class SportsCar(Car):

def sports\_car\_info(self):

print("Sports cars are fast.")

# Creating an object of SportsCar

sports\_car = SportsCar()

sports\_car.engine\_info() # Output: Engine is essential for a vehicle.

sports\_car.vehicle\_info() # Output: A vehicle is used for transportation.

sports\_car.car\_info() # Output: Cars usually have 4 wheels.

sports\_car.sports\_car\_info() # Output: Sports cars are fast.

✅ **Combines different types of inheritance for flexibility**.  
❌ **Can be complex** due to multiple parent-child relationships.

**Method Overriding in Inheritance**

A **child class can override** a method from its parent class by defining a method with the same name.

**Example: Overriding a Method**

class Parent:

def show\_message(self):

print("This is the parent class.")

class Child(Parent):

def show\_message(self): # Overriding parent method

print("This is the child class.")

child = Child()

child.show\_message() # Output: This is the child class.

✅ **Allows customization of behavior in child classes**.  
✅ **Parent class remains unchanged**.

**Using super() in Inheritance**

The super() function is used to **call a method from the parent class** inside the child class.

**Example: Using super()**

class Parent:

def show\_message(self):

print("This is the parent class.")

class Child(Parent):

def show\_message(self):

super().show\_message() # Call parent method

print("This is the child class.")

child = Child()

child.show\_message()

# Output:

# This is the parent class.

# This is the child class.

✅ **Avoids redundant code by reusing the parent class method**.

**Key Takeaways**

| **Feature** | **Description** |
| --- | --- |
| **Single Inheritance** | One child inherits from one parent. |
| **Multiple Inheritance** | One child inherits from multiple parents. |
| **Multilevel Inheritance** | Parent → Child → Grandchild. |
| **Hierarchical Inheritance** | One parent, multiple children. |
| **Hybrid Inheritance** | Combination of two or more inheritance types. |
| **Method Overriding** | Child class redefines a method from the parent. |
| **super() Function** | Calls a method from the parent class in the child class. |

**Polymorphism in Classes**

* Polymorphism allows methods in **different classes to have the same name but different behaviors**.
* Polymorphism means **"many forms."** It allows the **same function or method to be used for different types of objects**.

**Example of Polymorphism**

class Animal:

def make\_sound(self):

print("Some generic animal sound")

class Dog(Animal):

def make\_sound(self):

print("Woof!")

class Cat(Animal):

def make\_sound(self):

print("Meow!")

animals = [Dog(), Cat()]

for animal in animals:

animal.make\_sound() # Calls the respective method in each subclass

**Types of Polymorphism in Python**

1. **Method Overriding (Runtime Polymorphism)** – Child class redefines a method from the parent class.
2. **Method Overloading (Compile-time Polymorphism) – Not directly supported in Python** (handled using default arguments).
3. **Operator Overloading** – Using operators (+, \*, ==, etc.) with user-defined classes.
4. **Polymorphism with Functions and Classes** – Functions and methods working with different objects.

**1. Method Overriding (Runtime Polymorphism)**

A **child class provides its own implementation** of a method defined in the **parent class**.

**Example: Animals Making Sounds (Method Overriding)**

class Animal:

def make\_sound(self):

print("Animal makes a sound.")

class Dog(Animal):

def make\_sound(self): # Overriding parent method

print("Dog barks.")

class Cat(Animal):

def make\_sound(self): # Overriding parent method

print("Cat meows.")

# Creating objects

animals = [Dog(), Cat(), Animal()]

# Using polymorphism

for animal in animals:

animal.make\_sound()

**Output**

Dog barks.

Cat meows.

Animal makes a sound.

✅ **Method Overriding allows different classes to have different behaviors for the same method.**  
✅ **Achieves runtime polymorphism.**

**2. Method Overloading (Using Default Arguments in Python)**

Python does not **directly support** method overloading like Java or C++. However, we can **achieve it using default arguments or variable-length arguments**.

**Example: Addition Function (Method Overloading using Default Arguments)**

class MathOperations:

def add(self, a, b=0, c=0):

return a + b + c

math = MathOperations()

print(math.add(5)) # Output: 5

print(math.add(5, 10)) # Output: 15

print(math.add(5, 10, 20)) # Output: 35

✅ **Handles different numbers of arguments using default values.**  
✅ **No need to define multiple functions with different parameters.**

**3. Operator Overloading (Magic Methods in Python)**

Python allows us to **override operators** like +, -, \*, etc., by defining special methods like \_\_add\_\_(), \_\_sub\_\_(), etc.

**Example: Adding Two Custom Objects**

class Book:

def \_\_init\_\_(self, pages):

self.pages = pages

def \_\_add\_\_(self, other): # Overloading +

return Book(self.pages + other.pages)

def \_\_str\_\_(self):

return f"Total pages: {self.pages}"

book1 = Book(100)

book2 = Book(200)

book3 = book1 + book2 # Uses \_\_add\_\_()

print(book3) # Output: Total pages: 300

✅ **Allows arithmetic operations on user-defined objects.**  
✅ **Enhances the readability and usability of custom classes.**

**4. Polymorphism with Functions and Classes**

A single function can operate on **different types of objects** without modification.

**Example: A Common Function for Different Shapes**

class Circle:

def area(self, radius):

return 3.14 \* radius \* radius

class Square:

def area(self, side):

return side \* side

# Function using polymorphism

def print\_area(shape, value):

print(f"Area: {shape.area(value)}")

# Creating objects

circle = Circle()

square = Square()

# Calling function with different objects

print\_area(circle, 5) # Output: Area: 78.5

print\_area(square, 4) # Output: Area: 16

✅ **Function print\_area() works for both Circle and Square.**  
✅ **No need for separate functions for each shape.**

**Encapsulation in Python 🔒**

**Encapsulation** is one of the core principles of **Object-Oriented Programming (OOP)**. It refers to **hiding data (variables) and restricting direct access** to them. This helps in **data protection and better control** over the attributes of a class.

**🔹 1. Why Use Encapsulation?**

✅ **Protects Data** – Prevents accidental modification.  
✅ **Restricts Direct Access** – Variables are accessed through methods.  
✅ **Improves Code Maintainability** – Controlled access via getter & setter methods.

**🔹 2. Encapsulation in Python (Using Private Variables)**

In Python, we make variables **private** by using **double underscores (\_\_)** before the variable name.

class BankAccount:

def \_\_init\_\_(self, balance):

self.\_\_balance = balance # Private variable

def get\_balance(self): # Getter method to access private variable

return self.\_\_balance

def deposit(self, amount): # Public method to modify private variable

if amount > 0:

self.\_\_balance += amount

return f"Deposited {amount}. New balance: {self.\_\_balance}"

else:

return "Deposit amount must be positive"

# Creating an object

account = BankAccount(1000)

# Accessing private variable using getter method

print(account.get\_balance()) # ✅ Output: 1000

# Depositing money using public method

print(account.deposit(500)) # ✅ Output: Deposited 500. New balance: 1500

# Trying to access private variable directly (❌ Will not work)

# print(account.\_\_balance) # ❌ AttributeError: 'BankAccount' object has no attribute '\_\_balance'

**🔹 3. Accessing Private Variables (Name Mangling): ⚠️ Not recommended but possible**

Even though private variables cannot be accessed directly, Python allows access using **name mangling** (\_ClassName\_\_variable).

print(account.\_BankAccount\_\_balance) # ⚠️ Not recommended but possible

This is a **workaround**, but it **defeats the purpose of encapsulation**.

**🔹 4. Using Getter and Setter Methods**

We can use **getter and setter methods** to safely access and modify private variables.

class Student:

def \_\_init\_\_(self, name, age):

self.\_\_name = name

self.\_\_age = age

def get\_age(self): # Getter method

return self.\_\_age

def set\_age(self, new\_age): # Setter method

if new\_age > 0:

self.\_\_age = new\_age

else:

print("Age must be positive")

# Creating object

s1 = Student("Alice", 20)

print(s1.get\_age()) # ✅ Output: 20

s1.set\_age(25) # ✅ Modifying private variable safely

print(s1.get\_age()) # ✅ Output: 25

**🔹 5. Using @property Decorator (More Pythonic Way)**

Instead of defining get and set methods manually, Python provides **@property decorators**.

class Employee:

def \_\_init\_\_(self, salary):

self.\_\_salary = salary # Private variable

@property

def salary(self): # Getter method

return self.\_\_salary

@salary.setter

def salary(self, amount): # Setter method

if amount > 0:

self.\_\_salary = amount

else:

print("Salary must be positive")

# Creating object

e1 = Employee(50000)

print(e1.salary) # ✅ Output: 50000 (calls the getter method)

e1.salary = 60000 # ✅ Calls the setter method

print(e1.salary) # ✅ Output: 60000

e1.salary = -1000 # ❌ Output: "Salary must be positive"

**Abstraction in Python (OOPs) 🚀**

**Abstraction** is one of the core principles of **Object-Oriented Programming (OOP)**. It is the process of **hiding implementation details** and **showing only the necessary features** of an object.

**🔹 1. Why Use Abstraction?**

✅ **Hides unnecessary details** – Users don’t need to know how things work internally.  
✅ **Simplifies complex systems** – Makes code easier to maintain and use.  
✅ **Improves security** – Prevents direct modification of internal data.

**🔹 2. Abstraction in Python (Using Abstract Classes & Methods)**

Python provides **ABC (Abstract Base Class)** from the abc module to implement abstraction.

from abc import ABC, abstractmethod

class Animal(ABC): # Abstract Class

@abstractmethod

def make\_sound(self): # Abstract Method (No implementation)

pass

class Dog(Animal): # Concrete Class

def make\_sound(self):

return "Woof! Woof!"

class Cat(Animal): # Concrete Class

def make\_sound(self):

return "Meow! Meow!"

# Creating objects

dog = Dog()

cat = Cat()

print(dog.make\_sound()) # ✅ Output: Woof! Woof!

print(cat.make\_sound()) # ✅ Output: Meow! Meow!

# animal = Animal() ❌ Error: Cannot instantiate abstract class

**🔹 3. Explanation**

* ABC (Abstract Base Class) is used to create an **abstract class**.
* @abstractmethod forces **child classes** to implement the method.
* We **cannot create an object of an abstract class** (Animal in this case).
* Only **child classes (Dog, Cat) provide implementation** of make\_sound().

**🔹 4. Real-World Example: Payment System**

from abc import ABC, abstractmethod

class Payment(ABC): # Abstract Class

@abstractmethod

def process\_payment(self, amount):

pass

class CreditCardPayment(Payment): # Concrete Class

def process\_payment(self, amount):

return f"Processing credit card payment of ${amount}"

class PayPalPayment(Payment): # Concrete Class

def process\_payment(self, amount):

return f"Processing PayPal payment of ${amount}"

# Creating objects

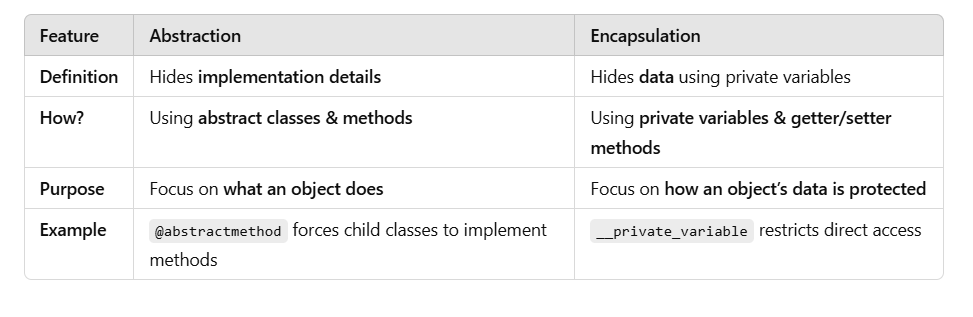
payment1 = CreditCardPayment()

payment2 = PayPalPayment()

print(payment1.process\_payment(100)) # ✅ Output: Processing credit card payment of $100

print(payment2.process\_payment(200)) # ✅ Output: Processing PayPal payment of $200

**🔹 5. Key Differences Between Abstraction & Encapsulation**

****

**🚀 Summary**

✅ **Abstraction hides implementation details** using **abstract classes & methods**.  
✅ ABC (Abstract Base Class) & @abstractmethod **enforce method implementation** in child classes.  
✅ **Cannot create objects of abstract classes**.  
✅ **Real-world applications**: Payment systems, databases, frameworks, etc.