**OOPS in Python**

**1. Constructor (\_\_init\_\_ method)**

* A **constructor** is a special method in a class that is **called automatically** when an object is created.
* In Python → constructor is defined using \_\_init\_\_.
* Purpose → initialize the **object’s attributes**.

**Example : 1**

class Student:

    def \_\_init\_\_(self, name, roll):   # Constructor

        self.name = name

        self.roll = roll

        print("Constructor called - Object Created")

    def display(self):

        print(f"Name: {self.name}, Roll: {self.roll}")

s1 = Student("Kiran", 101)   # Constructor runs automatically

s1.display()

 When Student("Kiran", 101) is executed → Python calls \_\_init\_\_() automatically.

 Attributes name and roll get initialized.

**Example 2:**

Default Constructor (no arguments)

class Demo:

    def \_\_init\_\_(self):   # Default constructor

        print("Object created")

d = Demo()

**Example 3:**

Constructor with Default Parameters

class Student:

    def \_\_init\_\_(self, name="Unknown", roll=0):

        self.name = name

        self.roll = roll

s1 = Student()                # uses default values

s2 = Student("Kavya", 102)    # custom values

print(s1.name, s1.roll)   # Unknown 0

print(s2.name, s2.roll)   # Kavya 102

**Example 4:**

Destructor in action

class Demo:

    def \_\_init\_\_(self):

        print("Constructor: Object created")

    def \_\_del\_\_(self):

        print("Destructor: Object destroyed")

d = Demo()

del d   # manually delete object

**Example 5:**

Destructor called automatically at program end

class Test:

    def \_\_init\_\_(self, name):

        self.name = name

        print(f"Object {self.name} created")

    def \_\_del\_\_(self):

        print(f"Object {self.name} destroyed")

t1 = Test("A")

t2 = Test("B")

print("Program running...")

**---------------------------------------------------------------------------------------------------------------------------------**

**Constructor (\_\_init\_\_)**

* Runs automatically when object created.
* Used to initialize attributes.

**2.Destructor (\_\_del\_\_)**

* Runs automatically when object deleted or program ends.
* Used for cleanup.
*  A **destructor** is a special method named \_\_del\_\_.
*  It is **called automatically** when an object is destroyed (when it goes out of scope or program ends).
*  Used to release resources (like closing a file, database connection, etc.).

class Student:

    def \_\_init\_\_(self, name):

        self.name = name

        print(f"Constructor called: {self.name} is created.")

    def \_\_del\_\_(self):

        print(f"Destructor called: {self.name} is deleted.")

# Object creation

s1 = Student("Kiran")

# Deleting object manually

del s1

print("End of program")

**Abstraction, Encapsulation, Inheritance, Polymorphism four pillars of OOPS**

**Abstraction**

Hiding the **implementation details** and only exposing the **necessary functions**.  
We use **abstract classes** with abc module.

from abc import ABC, abstractmethod

class Shape(ABC):   # Abstract class

    @abstractmethod

    def area(self):

        pass   # Child must implement

class Square(Shape):

    def \_\_init\_\_(self, side):

        self.side = side

    def area(self):

        return self.side \* self.side   # Implementation

class Circle(Shape):

    def \_\_init\_\_(self, radius):

        self.radius = radius

    def area(self):

        return 3.14 \* self.radius \* self.radius

# Create objects of child classes

s = Square(4)

c = Circle(5)

print("Square area:", s.area())   # 16

print("Circle area:", c.area())   # 78.5

**Encapsulation**

Protecting data inside a class.  
We use **private (\_\_var)** and **public methods**.

class BankAccount:

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

        self.owner = owner

        self.\_\_balance = balance   # Private variable

    def deposit(self, amount):

        self.\_\_balance += amount

        return f"Deposited {amount}, new balance = {self.\_\_balance}"

    def get\_balance(self):

        return self.\_\_balance   # Controlled access

# Create an instance of the class , means object creation

acc = BankAccount("Kiran", 1000)

print(acc.deposit(500))      # Deposited 500, new balance = 1500

print(acc.get\_balance())     # 1500

# print(acc.\_\_balance)  # ❌ ERROR: Can't access private variable

**Inheritance**

A class can **inherit properties and methods** from another class.

class Animal:   # Parent

    def speak(self):

        return "I make some sound"

class Dog(Animal):   # Child

    def speak(self):

        return "Woof!"

class Cat(Animal):   # Child

    def speak(self):

        return "Meow!"

# Create instances of the classes

dog = Dog()

cat = Cat()

print(dog.speak())  # Woof!

print(cat.speak())  # Meow!

**Types of Inheritance**

* Single
* Mutliple
* Multilevel
* Hierarchical

# -----------------------------

# 1. Single Inheritance

# -----------------------------

class Animal:   # Parent

    def speak(self):

        return "I make some sound"

class Dog(Animal):   # Child (Single Inheritance)

    def speak(self):

        return "Woof!"

# -----------------------------

# 2. Multilevel Inheritance

# -----------------------------

class Puppy(Dog):   # Child of Dog (Multilevel Inheritance)

    def speak(self):

        return "I am a small puppy, woof woof!"

# -----------------------------

# 3. Hierarchical Inheritance

# -----------------------------

class Cat(Animal):   # Another Child of Animal

    def speak(self):

        return "Meow!"

class Cow(Animal):   # Another Child of Animal

    def speak(self):

        return "Moo!"

# -----------------------------

# 4. Multiple Inheritance

# -----------------------------

class Pet:

    def pet\_info(self):

        return "I am a pet animal"

class Parrot(Animal, Pet):   # Multiple Inheritance

    def speak(self):

        return "Chirp! I can talk too."

# -----------------------------

# Create objects to demonstrate

# -----------------------------

dog = Dog()

puppy = Puppy()

cat = Cat()

cow = Cow()

parrot = Parrot()

print("Single Inheritance (Dog):", dog.speak())

print("Multilevel Inheritance (Puppy):", puppy.speak())

print("Hierarchical Inheritance (Cat):", cat.speak())

print("Hierarchical Inheritance (Cow):", cow.speak())

print("Multiple Inheritance (Parrot):", parrot.speak(), "|", parrot.pet\_info())

**Polymorphism**

**Same function name, different behavior** depending on the object.

# Method overLoading in polymorphism

class Bird:

    def fly(self):

        return "Some birds can fly"

class Sparrow(Bird):

    def fly(self):

        return "Sparrow flies high!"

class Ostrich(Bird):

    def fly(self):

        return "Ostrich can't fly!"

# Creating instances of different classes

birds = [Sparrow(), Ostrich()]  # Different objects

for bird in birds:

    print(bird.fly())

# Operator Overloading in Polymorphism

class Book:

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

        self.pages = pages

        self.new = new

    def \_\_add\_\_(self, other):

        return Book(self.pages + other.pages, self.new + other.new)

    def \_\_str\_\_(self):

        return f"({self.pages}, {self.new})"

b1 = Book(20, 20)

b2 = Book(30, 30)

b3 = Book(40, 100)

b4 = Book(90, 10)

print(b1 + b2 + b3 + b4)

**All concepts in single Example**

from abc import ABC, abstractmethod

# Abstraction

class Vehicle(ABC):

    @abstractmethod

    def fuel\_type(self):

        pass

# Encapsulation

class Car(Vehicle):

    def \_\_init\_\_(self, brand, fuel, price):

        self.brand = brand

        self.\_\_price = price   # Private

        self.fuel = fuel

    def fuel\_type(self):   # Abstraction implemented

        return self.fuel

    def get\_price(self):   # Encapsulation

        return self.\_\_price

# Inheritance

class ElectricCar(Car):

    def fuel\_type(self):

        return "Electric"

# Polymorphism

vehicles = [Car("Toyota", "Petrol", 10000), ElectricCar("Tesla", "Electric", 50000)]

for v in vehicles:

    print(v.brand, "runs on", v.fuel\_type(), "and costs", v.get\_price())

## What is a Decorator?

A **decorator** in Python is a function that takes another function as input, adds some functionality to it, and returns the modified function—**without changing the original code**.

1. **Basic Function Decorator**

def my\_decorator(func):

    def wrapper():

        print("Before function execution")

        func()

        print("After function execution")

    return wrapper

@my\_decorator   # using decorator

def say\_hello():

    print("Hello, World!")

say\_hello()

**Output :**

**Before function execution**

**Hello, World!**

**After function execution**

1. **Decorator with Arguments**

def repeat\_decorator(times):

    def decorator(func):

        def wrapper(\*args, \*\*kwargs):

            for \_ in range(times):

                func(\*args, \*\*kwargs)

        return wrapper

    return decorator

@repeat\_decorator(3)

def greet(name):

    print(f"Hello {name}")

greet("Kiran")

**Output :**

**Hello Kiran**

**Hello Kiran**

**Hello Kiran**

1. **Decorator for Functions with Arguments**

def smart\_divide(func):

    def wrapper(a, b):

        if b == 0:

            print("Error: Division by zero!")

            return

        return func(a, b)

    return wrapper

@smart\_divide

def divide(a, b):

    print(a / b)

divide(10, 2)

divide(5, 0)

**Output :**

**5.0**

**Error: Division by zero!**

1. **Chaining Multiple Decorators**

def uppercase(func):

    def wrapper():

        return func().upper()

    return wrapper

def split\_words(func):

    def wrapper():

        return func().split()

    return wrapper

@split\_words

@uppercase

def message():

    return "hello python decorators"

print(message())

**Output :**

**['HELLO', 'PYTHON', 'DECORATORS']**

1. **Class-based Decorator**

class DecoratorClass:

    def \_\_init\_\_(self, func):

        self.func = func

    def \_\_call\_\_(self, \*args, \*\*kwargs):

        print("Before execution")

        result = self.func(\*args, \*\*kwargs)

        print("After execution")

        return result

@DecoratorClass

def say\_hi(name):

    print(f"Hi {name}")

say\_hi("Kiran")

**Output :**

**Before function execution**

**Hi Kiran**

**After function execution**

1. Built-in Decorators (@staticmethod, @classmethod, @property)

class Student:

    def \_\_init\_\_(self, name):

        self.\_name = name

    @staticmethod

    def school\_name():

        return "ABC School"

    @classmethod

    def create(cls, name):

        return cls(name)

    @property

    def name(self):

        return self.\_name

# Usage

s = Student.create("Kiran")

print(s.name)           # property

print(Student.school\_name())  # staticmethod