Object-Oriented Programming (OOP) in Python

1. Introduction to OOP in Python

Object-Oriented Programming ek programming style hai jisme hum real-world entities ko model karte hain as "objects". Python ek object-oriented language hai jisme OOP concepts fully supported hain.

Roman Urdu: OOP ek tareeqa hai jisme hum code ko objects ki form mein likhte hain, jaise car, person, etc.

2. What is OOP?

OOP stands for Object-Oriented Programming. Ismein code reusable, scalable aur organized hota hai using classes and objects.

Roman Urdu: OOP ka matlab hai ke code ko chhote chhote blocks (objects) mein divide kar ke likhna.

3. Key Principles of OOP

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

Roman Urdu: In 4 pillars se OOP mazboot hota hai. Har ek ka role important hota hai.

4. Basic Classes & Objects

Class : Python mein **class** ek **blueprint** ya **template** hoti hai jiske zariye aap **objects** bana sakte hain.

Example:

```
class Person:

 def __init__(self, name):

 self.name = name



p = Person("Ali")

print(p.name)
```

Object: **Object** ek **real-world entity** ka **programming version** hota hai. Jab aap kisi **class** ka use karke koi **instance** banate ho, usse **object** kehte hain.

```
class Car:

    def __init__(self, brand, color):

        self.brand = brand

        self.color = color



    def drive(self):

        print(f"{self.color} {self.brand} is driving")



# make Object

my_car = Car("Toyota", "Red")

my_car.drive()
```

5. Constructor and Destructor

Constructor: __init__() method jo object create hone par call hoti hai.

Destructor: __del__() method jo object delete hone par call hoti hai.

Example:

class Test:

def __init__(self): print("Constructor called")

def __del__(self): print("Destructor called")

obj = Test()

Important Points about Destructor in Python:

- Destructor ka naam **`__del__()`** hota hai.
- Destructor **automatically call** hota hai jab object destroy hota hai.
- Destructor ka use **cleanup** ke liye hota hai (e.g., file close karna, memory free karna).
- Python **decide karta hai** ke destructor kab chalega; iska time **predictable nahi hota**.
- Agar aap **`del object_name`** likho to destructor **turant call** ho sakta hai.
- **Circular references** destructor ko rok sakti hain (agar objects ek doosre ko refer karte hain).
- Destructor ke andar kuch attributes pehle se **None** ya **destroyed** ho sakte hain.¶

Roman Urdu: Constructor object banate waqt chalta hai, destructor jab object destroy hota hai.

6. Class Attributes vs Instance Attributes

- Class attribute: Sab objects ke liye same.

- Instance attribute: Har object ke liye alag.

Example:
class Car:

wheels = 4 # class attribute

```
def __init__(self, color):
self.color = color # instance attribute

c1 = Car("Red")
c2 = Car("Blue")
```

Roman Urdu: wheels sab cars ke liye same, color har car ka alag.

7. Methods in Python Classes

- Instance Method: object se call hoti hai
- Class Method: @classmethod decorator use hota hai
- Static Method: @staticmethod decorator use hota hai

1 Instance Method

- Yeh method **class ke object (instance)** se call hoti hai.
- Is method ko **object ke data ko access ya change karne** ke liye use kiya jata hai.
- Pehla parameter hamesha `self` hota hai, jo us specific object ko refer karta hai.

```
class Student:
  def __init__(self, name):
```

```
        self.name = name



    def greet(self):

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



student1 = Student("Ali")

student1.greet()  # Output: Hello, Ali!
```

2 Class Method

- Yeh method class ke upar operate karta hai, na ke kisi specific object par.
- Is method ke pehle parameter mein `cls` aata hai, jo class ko represent karta hai.
- Is method ko define karne ke liye `@classmethod` **decorator** lagaya jata hai.
- Class method se aap class ki properties ya class level data ko modify kar sakte hain.

```
class Student:
    school_name = "ABC School"

    @classmethod
    def change_school(cls, new_name):
        cls.school_name = new_name

Student.change_school("XYZ School")
print(Student.school_name)  # Output: XYZ School
```

Static Method

- Yeh method na to class ki property ko access karta hai na object ki.
- Yeh ek **general utility function** hota hai jo class ke andar rehta hai but kisi object ya class data pe depend nahi karta.

- Isko define karne ke liye `@staticmethod` **decorator** lagaya jata hai.
- Static method ko aap class ya object dono se call kar sakte hain.

```
class Math:
    @staticmethod
    def add(a, b):
        return a + b

print(Math.add(5, 7))  # Output: 12
```

8. Encapsulation

Encapsulation programming ka aik important concept hai, jo **data hiding** aur **data protection** ke liye use hota hai.

Iska matlab hai **class ke andar data (attributes) aur functions (methods) ko is tarah chhupana ke direct bahar se access na kiya ja sake**. Yeh **security aur control** dene ke liye hota hai ke koi dusra programmer ya user object ke andar ke data ko directly badal na sake, balke sirf methods ke zariye hi data ko access ya modify kar sakay.

Python mein Encapsulation kaise hota hai?

Python mein **private variables** ko banane ke liye **underscore** (`_`) ya **double underscore** (`__`) ka use hota hai:

- **Single underscore (`_variable`)**: Ye ek convention hai jo batata hai ke yeh variable private hai, matlab "iske saath ehtiyat karo, ye bahar se direct use nahi karna chahiye". Lekin technically access ho sakta hai.
- **Double underscore (`__variable`)**: Yeh variable naam ko internally modify kar deta hai (name mangling), jis se bahar se access mushkil ho jata hai. Isko **strong private** variable samajh sakte hain.

```
class BankAccount:

    def __init__(self, owner, balance):

        self.owner = owner

        self.__balance = balance   # private variable



    def deposit(self, amount):
```

```
        if amount > 0:
            self.__balance += amount
            print(f"{amount} deposit hua. New balance: {self.__balance}")
        else:
            print("Invalid amount!")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"{amount} withdraw hua. Remaining balance: {self.__balance}")
        else:
            print("Insufficient balance ya invalid amount!")

    def get_balance(self):
        return self.__balance
```

Usage:

```
account = BankAccount("Ali", 1000)

account.deposit(500)          # 500 deposit hua. New balance: 1500
account.withdraw(200)         # 200 withdraw hua. Remaining balance: 1300
print(account.get_balance())  # 1300

print(account.__balance)      # Error! Direct access nahi ho sakta
```

🔐 What are Access Modifiers in Python?

Access modifiers batate hain ke **class ke variables (attributes)** aur **methods** ko kahaan se access kiya ja sakta hai — **class ke andar se, bahar se, ya child classes se.**

Python mein 3 tarah ke access modifiers hote hain:

1. Public (default)

- **Kisi bhi variable ya method ko jab aap bina underscore ke likhte hain**, to wo **public** hota hai.
- Public members ko aap **class ke bahar se** bhi **direct** access kar sakte hain.

```
class Student:

    def __init__(self):

        self.name = "Ali"  # public


obj = Student()

print(obj.name)  # ✅ Access allowed
```

`self.name` ek **public** variable hai, isliye hum usay class ke bahar se direct access kar sakte hain.

Public members kisi bhi jagah se access ho sakte hain — class ke andar ya bahar.

2. Protected (`_single_underscore`)

- Jab aap kisi variable ya method ke naam ke aagay **ek underscore `_` lagate hain**, to wo **protected** ban jata hai.
- Conventionally, yeh **child class** ko access dene ke liye hota hai, **lekin bahar se access technically possible hota hai.**

```
class Student:

    def __init__(self):

        self._marks = 80  # protected


obj = Student()

print(obj._marks)  # ✅ Allowed but discouraged
```

- `self._marks` ek **protected** variable hai.
- Python isay private nahi banata, lekin batata hai ke "ye sirf class aur subclass ke liye hai."
- Bahar se access possible hai, lekin **best practice yeh hai ke aap use na karein.**

3. Private (`__double_underscore`)

- Jab aap variable ya method ke naam ke aagay **do underscore `__` lagate hain**, to wo **private** ban jata hai.
- Private members sirf **class ke andar** accessible hote hain, bahar se **direct access allowed nahi hota.**
- Python inka naam **mangle** kar deta hai, taake accidentally ya intentionally access na ho.

```
class Student:
    def __init__(self):
        self.__grade = "A"  # private

    def show_grade(self):
        print(self.__grade)

obj = Student()
obj.show_grade()       # ✅ Allowed
# print(obj.__grade)   # ❌ Error: AttributeError
```

`self.__grade` ek **private** variable hai.

Isay sirf class ke andar ke method (`show_grade`) se access kiya ja sakta hai.

Agar aap bahar se `obj.__grade` likhenge, to error milega.

Modifier	Syntax	Accessible from Class	Accessible from Subclass	Accessible from Outside
Public	`self.name`	✅ Yes	✅ Yes	✅ Yes
Protected	`self._name`	✅ Yes	✅ Yes	⚠️ Technically Yes (but discouraged)
Private	`self.__name`	✅ Yes	❌ No	❌ No

💡 Python Note:

Python mein private variables ko technically access kiya ja sakta hai using **name mangling**, jaisay:

print(obj._Student__grade) # Not recommended, but works

Agar zarurat ho to aap `obj._ClassName__variable` se private variables ko access kar sakte ho, lekin yeh **best practice** nahi hai.

9. Inheritance

Inheritance (Wirasat) Kya Hoti Hai?

Inheritance Object Oriented Programming (OOP) ka ek ahem concept hai. Iska matlab hai ke ek **class**, doosri class ki **properties** (attributes) aur **functions** (methods) ko **wirasat mein le leti hai**, yani use reuse karti hai.

Example:

```
class Animal:

 def speak(self): print("Animal sound")



class Dog(Animal):
 pass



d = Dog()

d.speak()
```

🔶 Types of Inheritance in Python

1. Single Inheritance

Ek child class sirf ek parent class se wirasat leti hai.

```
class Animal:
    def speak(self):
        print("Ye animal hai.")

class Dog(Animal):  # Inheritance ho rahi hai
    def bark(self):
        print("Bhow bhow!")

dog = Dog()
dog.speak()   # Parent ka method
dog.bark()    # Apna method
```

2. Multilevel Inheritance

Ek class doosri class se inherit karti hai, jo khud kisi teesri class se inherit karti hai.

```
class Animal:
    def speak(self):
        print("Animal bolta hai.")

class Dog(Animal):
    def bark(self):
        print("Dog bhonkta hai.")
```

```
class Puppy(Dog):
    def weep(self):
        print("Puppy ro raha hai.")

puppy = Puppy()
puppy.speak()
puppy.bark()
puppy.weep()
```

3. Multiple Inheritance

Ek class, do ya zyada classes se inherit karti hai.

```
class Father:
    def gardening(self):
        print("Baaghbani ka kaam karta hoon.")

class Mother:
    def cooking(self):
        print("Khana pakati hoon.")

class Child(Father, Mother):
    def sports(self):
        print("Khelta hoon.")

child = Child()
child.gardening()
child.cooking()
```

```
child.sports()
```

4. Hierarchical Inheritance

Ek parent class se multiple child classes inherit karti hain.

```
class Animal:
    def speak(self):
        print("Animal sound.")

class Dog(Animal):
    def bark(self):
        print("Dog bhonkta hai.")

class Cat(Animal):
    def meow(self):
        print("Cat meow karti hai.")

dog = Dog()
cat = Cat()
dog.speak()
dog.bark()
cat.speak()
cat.meow()
```

5. Hybrid Inheritance

Ismein multiple inheritance types mix hoti hain.

```
class A:
```

```
    def methodA(self):
        print("Class A")

class B(A):
    def methodB(self):
        print("Class B")

class C(A):
    def methodC(self):
        print("Class C")

class D(B, C):  # Hybrid: Multilevel + Multiple
    def methodD(self):
        print("Class D")

obj = D()
obj.methodA()
obj.methodB()
obj.methodC()
obj.methodD()
```

🔶 **Important Concepts in Inheritance**

✅ **`super()` Function:**

Parent class ke method ko call karne ke liye use hota hai.

```
class Parent:
    def __init__(self):
```

```
        print("Parent ka constructor")


class Child(Parent):
    def __init__(self):
        super().__init__()  # Parent ka constructor call
        print("Child ka constructor")


obj = Child()
```

✅ Points to be Noted on Inheritance.

1. Inheritance ka matlab hai ek class (child) doosri class (parent) se features lena.
2. Code reusability kaafi behtareen hoti hai – code dobara likhne ki zarurat nahi.
3. Parent class ke methods aur attributes child class mein automatically available ho jaate hain.
4. `super()` function parent class ke constructor ya method ko call karne ke liye use hota hai.
5. Child class parent class ke method ko override kar sakti hai (same method name se naya likhna).
6. Agar child class apna `__init__()` constructor banaye to parent ka constructor auto-call nahi hota – use `super()` se call karna padta hai.

7. Types of Inheritance

- Single Inheritance – Ek parent, ek child.
- Multilevel Inheritance – Grandparent → Parent → Child.
- Multiple Inheritance – Ek child, multiple parents.
- Hierarchical Inheritance – Ek parent, multiple children.
- Hybrid Inheritance – In sab ka mix.

7. **Private attributes/methods (`__name`) inherit nahi hote directly.**
8. **Python mein Method Resolution Order (MRO) define karta hai ke multiple inheritance mein kaunsa method pehle chalega.**
9. **Inheritance se code ka structure zyada maintainable aur scalable ban jaata hai.**
10. **`ChildClass(ParentClass)` syntax se inheritance hoti hai.**
11. **Ek class mein agar multiple parents ho to unka order important hota hai (MRO ke liye).**
12. **Inheritance ke zariye real-world relationships (jaise "Dog is an Animal") ko easily model kiya ja sakta hai.**
13. **Har class Python mein object class se by default inherit karti hai.**
14. **Inheritance ka misuse na karo – agar relation "is-a" na ho to use mat karo (jaise Car is a Vehicle ✔, but Car is an Engine ✘).**

10. Polymorphism

Polymorphism ka matlab hai ke ek method ya function alag-alag objects ke liye alag tarah se behave kare.

🔷 Types of Polymorphism

1. Compile-time Polymorphism (Method Overloading)

- Python mein **directly supported nahi hota** jaise Java mein hota hai.
- Lekin default parameters ya `*args` se achieve kiya ja sakta hai.

```
def add(a, b=0, c=0):
    return a + b + c

print(add(5))        # Sirf ek argument
print(add(5, 10))    # Do arguments
print(add(5, 10, 15)) # Teen arguments
```

2. Run-time Polymorphism (Method Overriding)

- Jab child class, parent class ka method **apne hisaab se overwrite karti hai**.
- **Same method name**, lekin alag behavior in child.

Example: class Animal:

```
    def speak(self):

        print("Animal bolta hai")
```

class Dog(Animal):

```
    def speak(self):

        print("Dog bhonkta hai")
```

class Cat(Animal):

```
    def speak(self):

        print("Cat meow karti hai")
```

```
# Function that uses polymorphism

def make_sound(animal):

    animal.speak()




dog = Dog()

cat = Cat()




make_sound(dog)  # Output: Dog bhonkta hai

make_sound(cat)  # Output: Cat meow karti hai
```

🔷 Duck Typing (Python Specific Concept)

Python mein agar **object ke paas required method hai**, to usko accept kar leta hai — chahe class ka naam kuch bhi ho.

```
class Duck:

    def walk(self):

        print("Duck chal rahi hai")




class Human:

    def walk(self):

        print("Insaan chal raha hai")



def lets_walk(obj):

    obj.walk()



duck = Duck()

human = Human()
```

lets_walk(duck)   # Output: Duck chal rahi hai

lets_walk(human) # Output: Insaan chal raha hai

✅ Points to Remember (Roman Urdu)

1. **Polymorphism** ka matlab hai ek hi naam ka method/members ka **different behavior** rakhna depending on object.
2. Python mein **method overloading** default form mein nahi hoti — lekin optional arguments se banayi ja sakti hai.
3. **Method overriding** parent-child relationship mein hoti hai.
4. **Function overriding** runtime pe decide hoti hai — isiliye isse **runtime polymorphism** kehte hain.
5. **Duck typing** Python ka khaas feature hai — class ka naam important nahi, method hona chahiye.

11. Abstraction

Abstraction ka matlab hai ke user ko sirf functionality dikhai jaye, lekin us functionality ke peeche ka complex logic chhupa diya jaye.

✅ `ABC` (Abstract Base Class) aur `@abstractmethod`

Ye module Python mein `abc` (abstract base class) se import hota hai.

from abc import ABC, abstractmethod

```
class Animal(ABC):  # Abstract base class
    @abstractmethod
    def speak(self):  # Abstract method
        pass

class Dog(Animal):
    def speak(self):
        print("Dog bhonkta hai")

class Cat(Animal):
    def speak(self):
        print("Cat meow karti hai")
```

a = Animal() ❌ Error: Abstract class ka object nahi ban sakta

dog = Dog()

dog.speak() # Output: Dog bhonkta hai

cat = Cat()

cat.speak() # Output: Cat meow karti hai

🔶 Important Points (Roman Urdu mein):

1. **Abstract class** wo hoti hai jisme **kam az kam ek abstract method** hota hai.
2. Abstract class ka **object nahi banaya ja sakta**.
3. **Abstract method** wo hota hai jiska **sirf naam hota hai**, implementation nahi hoti (i.e., method body `pass` hoti hai).
4. **Child class ko zaroori hai** ke wo abstract methods ko **override kare**.
5. Ye concept **interface aur framework design** mein kaafi useful hai.

from abc import ABC, abstractmethod

```
class Bank(ABC):

    @abstractmethod

    def loan(self):

        pass



    @abstractmethod

    def interest_rate(self):

        pass



class HBL(Bank):

    def loan(self):

        print("HBL: Personal loan")
```

```
    def interest_rate(self):
        print("HBL: 14% annual")


class UBL(Bank):
    def loan(self):
        print("UBL: Home loan")

    def interest_rate(self):
        print("UBL: 12% annual")


bank1 = HBL()
bank1.loan()
bank1.interest_rate()
```

```
bank2 = UBL()

bank2.loan()

bank2.interest_rate()
```

12. The Object Class

Definition: Python mein har class by default object class se inherit karti hai.

Example:

```
class MyClass:
 pass

print(isinstance(MyClass(), object)) # True
```

Roman Urdu: Har class Python mein object se bana hoti hai.

13. Dunder Methods (Magic Methods)

Definition: Special methods like __init__, __str__, __len__ jo Python automatically call karta hai.

Example:

```
class Book:
 def __init__(self, title):
 self.title = title

 def __str__(self):
 return f"Book: {self.title}"

b = Book("Python")
print(b)
```

Dunder methods special kaam ke liye use hote hain jaise printing, length, etc.

✅ Common Dunder Methods List (with Roman Urdu Explanation):

Dunder Method	Kaam (Roman Urdu)
`__init__(self, ...)`	Object banate waqt run hota hai (constructor)
`__str__(self)`	Jab `print(object)` karein to kya show ho
`__repr__(self)`	Official string representation (debugging)
`__len__(self)`	Jab `len(object)` call ho

`__add__(self, other)`	Jab `+` operator ka use ho do objects ke darmiyan
`__sub__(self, other)`	`-` operator ke liye
`__mul__(self, other)`	`*` operator ke liye
`__eq__(self, other)`	Jab `==` compare karein
`__lt__(self, other)`	Jab `<` operator ka use ho
`__getitem__(self, key)`	Jab object[index] access karein
`__setitem__(self, key, value)`	Jab object[index] = value karein
`__del__(self)`	Jab object destroy ho (destructor)
`__contains__(self, item)`	Jab `in` operator use ho