What is OOP?

OOP ek programming style hai jismein hum apna program **objects** aur **classes** ki madad se banate hain.

OOP ka sabse bada faida yeh hai ke yeh aapko apna code chhote chhote hisso (objects) mein todne ka moka deta hai, jise aap asaani se samajh sakte hain, dobara use kar sakte hain, aur badal bhi sakte hain bina pura program kharab kiye. Har tukda apni zimmedari leta hai

, jisse humara code strong, flexible aur asaani se samajh aane wala ban jata hai

OOP matlab: Program ko chhoti chhoti cheezon (objects) mein divide karna. Har cheez ka apna kaam aur apne data hote hain.

Real-Life Example:

Agar aap car ka game bana rahi hain, to:

- **Car** ek **class** hogi (jaise ek *naqsha* ya *blueprint*)
- Us class se banne wali **har car** ek **object** hogi (jaise "Car 1", "Car 2", etc.)

Har car ka apna:

- **color** (attribute ya property)
- **speed**
- aur **functions** jaise accelerate() ya brake() honge

Example:
Sochiye aap gaari bana rahe ho:

- Engine, Steering, Aur Wheels sab alag alag objects hain.
- Agar engine mein koi problem hai to sirf engine object ko fix karo, baki sab ko chhedo mat.

OOP kyu use karte hain? (Why OOP is useful):

✅ **Modularity:**	Bada code chhote-chhote parts (classes) mein divide hota hai
🔁 **Reusability:**	Ek class baar-baar reuse ho sakti hai
🛠️ **Maintainability:**	Code ko maintain aur update karna asaan hota hai
⬆️ **Scalability:**	Future mein naye features asaani se add ho jate hain
🌍 **Real-world model:**	Real-life cheezon jaise (Car, Student) ka model banana easy hota hai

OOP ke 4 Asasi Principles (Pillars):

1️⃣ **Encapsulation:** Data + functions ko ek class mein band karna, aur data ko directly access se rokna

2️⃣ **Abstraction:** Sirf important cheezein dikhana, details chhupana

3️⃣ **Inheritance:** Ek class doosri class ke features inherit kare (reuse code)

4️⃣ **Polymorphism:** Ek hi function different classes mein alag kaam kare

Key Principles of OOP

🔐 1. Encapsulation — “Chhupao aur control do”

Definition: Data aur methods ko aik class ke andar band karna, aur bahar walon ko sirf zarurat ki cheez dikhana. Aur kuch cheezein private rakhna, taake koi galti se data ko seedha na badal sake aur sensitive data ko **direct access se bachana**.

👩‍🍳 Real Life:

Tumhari pencil box — us ke andar eraser, pencil, sharpener hain. But kisi ko pata nahi andar kya kya cheez hai — sirf tum use kar sakti ho.

```
class Car:
   def __init__(self, color, speed):
      self.color = color        # Public attribute (sab use kar sakte hain)
      self.__speed = speed      # Private attribute (direct access nahi milega)

   def accelerate(self):        # Method to increase speed
      self.__speed += 10        # Sirf yeh function speed ko change kar sakta hai

   def get_speed(self):         # Method to safely get speed
      return self.__speed       # Direct speed dikhata nahi, function ke zariye deta hai

#OUTPUT:

car = Car("red", 0)  # Provide values for color and speed
print("Current speed: ", car.get_speed()) # Call the get_speed method

# speed up bro
for i in range(10):
   car.accelerate()

print("Speed after acceleration: ", car.get_speed()) # Call the get_speed method

#Current speed:  0
#Speed after acceleration:  100
```

Example:

Car class mein speed ko private rakhna taake sirf functions ke zariye hi speed change ho.

Is example mein __speed private hai, usay access karne ke liye sirf get_speed() aur accelerate() methods ka use hota hai. Yeh safety aur control provide karta hai."

Function ka naam wohi ho zaroori nahi, bus function ke andar sahi private variable use ho.

2. Abstraction — “Sirf zarurat ki cheez dikhana”

faaltu details chhupa dena aur sirf zaroori cheezen dikhana". Bilkul waise hi jaise aap car chalate waqt sirf **steering**, **brake**, aur **accelerator** ka use karte ho — aapko ye nahi pata hota ke engine andar kaise kaam kar raha hai.

Definition: Detail chhupa do, sirf kaam dikhayo.

Real Life:

Tum light on karne ke liye switch dabati ho — andar wire kaise kaam kar rahi, woh nahi pata.

Example:

Car ka drive() method call karo, engine khud start ho jata hai, aapko uska detail nahi pata.

Iska faida:

- Code simple lagta hai
- Complex details chhup jati hain

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

    def start_engine(self):    # abstraction: user ke liye simple method
        self._start_engine()   # internal detail chhupi hui hai

    def _start_engine(self):   # protected method (logic chhupa hua)
        print("Generic car engine started")

class BMW(Car):
    def _start_engine(self):   # override kar diya — detail BMW wali
        print("BMW engine started")

# ✅ Use
bmw = BMW("BMW", "i5")
bmw.start_engine()
#BMW engine started
```

👪 3. Inheritance — “Warasat milna”

Inheritance ek class doosri class ke features ko use kare, bina dobara likhe.
Jaise bacha apne walid ki cheezein (ghar, naam, etc.) **inherit** karta hai, waise hi ek class doosri class ke **functions aur variables** inherit karti hai.

- Code reuse hota hai
- Code organize hota hai

```
class Vehicle:
    def __init__(self, color, speed):
        self.color = color
        self.speed = speed

    def drive(self):
        print("Driving Speed = ", self.speed)

    def accelerate(self):
        self.speed += 10

class Car(Vehicle):  # Car inherits from Vehicle
    def __init__(self, color, speed):
        super().__init__(color, speed)

class Truck(Vehicle):  # Truck inherits from Vehicle
    def __init__(self, color, speed):
        super().__init__(color, speed)

car = Car("red", 120)
truck = Truck("blue", 80)

car.drive()
truck.drive()

#Driving Speed =  120
#Driving Speed =  80
```

Q: What is the use of super()?

A:

super() ka use child class mein parent class ke constructor (__init__) ko call karne ke liye hota hai. Jaise Car aur Truck ne super().__init__() likh kar Vehicle ka constructor use kiya.

🐾 4. Polymorphism — "Ek naam, kaam alag"

Polymorphism har **method ka mukhtalif classes mein alag behavior**".

Yani aap **ek function ya method** use kar rahe ho, lekin **object kis class ka hai us hisaab se result alag hoga.**

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self) -> None:
        pass

class Dog(Animal):
    def speak(self) -> None:
        print(type(self), ":  Woof!")

class Cat(Animal):
    def speak(self) -> None:
        print(type(self), ":  Meow!")

def animal_sound(animal: Animal) -> None:
    animal.speak()

dog = Dog()
cat = Cat()
animal_sound(dog)  # Output: Woof!
animal_sound(cat)  # Output: Meow!

#<class '__main__.Dog'> :  Woof!
#<class '__main__.Cat'> :  Meow!
```

👩‍👦 Real Life:

Tumhara mobile mein "Play" button sab media apps mein hota hai — but har app mein alag kaam karta hai. Aap ne chaar key pillars seekh liye:

1. ✅ **Encapsulation** – Data + methods ko ek unit mein band karna
2. ✅ **Abstraction** – Sirf zaroori cheezein dikhana, baaki hide karna
3. ✅ **Inheritance** – Ek class doosri class ke features lena
4. ✅ **Polymorphism** – Same method name, different behavior (dynamic action)

- Code flexible hota hai
- Different objects ko ek jaisa treat kar sakte hain

2. Basics of Classes and Objects:

What is a Class?

Ek **class ek naqsha (blueprint)** hoti hai — jaise **ek design** jisse hum kai cheezen bana sakte hain.
Jaise ek "Car" ka design sabke liye same hota hai, lekin har car ki **color, speed, ya model** alag ho sakti hai.

What is an Object?

Class se banaya gaya real item **ko hum** object **kehte hain.**

Defining Attributes and Methods:

Attributes = object ke (data/properties)
Methods = object ke (actions/behaviors)

Attributes woh variables hote hain **jo object ke andar data ko represent karte hain**.

Methods woh functions hote hain **jo kisi class ke andar likhe jaate hain**, aur wo object ka behavior define karte hain.

The self Keyword:

"self ka use hum object ke current instance ko refer karne ke liye karte hain.
Agar hume object ke attribute ya method ko access karna hai class ke andar, to hum self use karte hain."jab object create karti ho, self usi object ki **andar ki cheezen (attributes/methods)** ko access karne ke liye hota hai.

1. Class kya hoti hai?

Class ek blueprint hoti hai jisme hum attributes aur methods define karte hain.Iska kaam hota hai ek structure dena jis se hum objects bana saken.

2. Object kya hota hai?

"Object ek class ka instance hota hai.
Jab hum class se koi cheez banate hain, use object kehte hain.
Har object ka apna alag data hota hai.

3. Attributes kya hote hain?

"Attributes object ke andar ke data hote hain — jaise color, speed.
Hum unko self.color ke through class ke andar access karte hain."

4. Methods kya hote hain?

"Methods class ke andar likhe hue functions hote hain.
Yeh object ke actions define karte hain.
Jaise accelerate(), brake() etc."

5.Constructor kya hota hai? (__init__)

"__init__ *Constructor ek special method hota hai jo automatically chal jata hai jab hum class ka object banate hain.*
Iske andar hum object ke initial attributes set karte hain.
Isme self hamesha first parameter hota hai.

6. Private attribute ya method kya hota hai?

"Jab kisi method ya attribute ke naam se pehle double underscore __ laga dete hain,to wo private ho jata hai.
Private ka matlab hota hai ki use sirf class ke andar hi access kiya ja sakta hai,class ke bahar se nahi."

Last Summary Line for Viva:

- **Class** ek design hai,
- **Object** us design ka product hai,
- **Attributes** data hote hain,
- **Methods** actions hote hain,
- **self** current object ko refer karta hai,
- Aur **private** cheezen sirf class ke andar hi access hoti hain."

---

2. Parameterized Constructor kya hai?

Agar hum constructor ko kuch parameters dena chahte hain, jese naam, age, brand, to usay parameterized constructor kehte hain."

```
class Student:
    def __init__(self, name, age):  # ← yeh parameterized constructor hai
        self.name = name
        self.age = age
```

3. Constructors and Destructors

1. Constructor:

Jab bhi hum class ka object banate hain, ek special method __init__() automatically chalta hai — isse **constructor** kehte hain.
Ye object ke variables ko initialize karta hai.

Why Use Constructor?

- Har object ke sath data **initialize** karna hota hai.
- Object creation ke waqt hi automatically kaam start kar deta

2. Parameterized Constructor:

Jab hum constructor mein **parameters** pass karte hain (jaise name, age), usse **parameterized constructor** kehte hain.

```
class Person:
    def __init__(self, name, age):  # constructor with parameters
        self.name = name
        self.age = age

# object banate waqt arguments bhi diye
person1 = Person("Alice", 30)

print(person1.name)  # Output: Alice
print(person1.age)   # Output: 30
```

3. Default Constructor:

__init__() method na likhein to Python khud ek **default constructor** provide karta hai jo kuch nahi karta.

```
class Student:
   pass

s1 = Student()
print("Student object created!")
#output

Student object created!
```

Lekin aap khud bhi ek aisa constructor bana sakte ho jo **kisi parameter ke bina** chalta ho.

```
class Dog:
   def __init__(self):  # default constructor
      self.name = "Unknown"
      self.age = 0

dog1 = Dog()
print(dog1.name)  # Output: Unknown
print(dog1.age)   # Output: 0
```

Question: Agar __init__() na likhein to kya hota hai?
Answer: "Python ek default constructor use karta hai jo kuch bhi initialize nahi karta."

Question: Default constructor kya hota hai?

Answer: "Woh constructor jo bina parameters ke ho, jese __init__(self)."

Agar tum parameterized constructor likhte ho:

def __init__(self, name):

To object banate waqt argument dena **zaroori** ho jata hai.

Default constructor mein argument ki **zarurat nahi** hoti.

4 .Destructor (_del_):

Destructor ek special method hota hai jo tab chalta hai jab object destroy hone wala hota hai, matlab uska kaam khatam ho jata hai.

Iska naam hota hai: __del__()
Iska kaam hota hai: **cleanup karna, message dena, ya file band karna etc.**

```
class Student:
    def __init__(self, name):
        self.name = name
        print(f"{self.name} joined the class.")

    def __del__(self):
        print(f"{self.name} left the class.")

s1 = Student("Asma")
del s1                          # Manually destroy object
  Output:
Asma joined the class.

Asma left the class.
```

Full Example (Constructor + Destructor):

```
class Car:
    def __init__(self, brand, model):              # parameterized constructor
        self.brand = brand
        self.model = model
        print(f"A {self.brand} {self.model} has been created.")

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

    def __del__(self):  # destructor
        print(f"The {self.brand} {self.model} has been destroyed.")

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

print(f"My car is a {my_car.brand} {my_car.model}.")
my_car.display()

del my_car                  # destroy object explicitly
```

Output:

```
A Toyota Corolla has been created.
My car is a Toyota Corolla.
Car: Toyota Corolla
The Toyota Corolla has been destroyed.
```

4. Class Attributes and Instance Attributes

Python mein do tarah ke attributes hote hain:

1. Class Attributes
2. Instance Attributes

Dono ka use object ke data ko store karne ke liye hota hai, lekin scope aur behavior alag hota hai."

1. Class Attributes:

Ye directly class ke andar define kiye jate hain, kisi method ke bahar."Ye sabhi objects ke liye common hote hain — yani ek baar define karne se sabhi objects usko use kar sakte hain."

```
class School:

    school = "PAF"
```

```
class Dog:

    species = "Canis familiaris"  # Class attribute
```

2. Instance Attributes:

"Instance attributes har object ke liye unique or alag alag hote hain, hote hain, aur ye hum __init__ method ke andar define karte hain."

Instance attribute wo hoti hai jo **constructor (init)** ke andar **self** ke through define ki jati hai.

```
class Dog:
        def __init__(self, name, age):
           self.name = name     # Instance attribute
           self.age = age       # Instance attribute"
```

Yahan name aur age har dog ke liye unique value lete hain."

```
class Student:
   # Class attribute: sab students ka school name same hoga
   school_name = "The Smart School"
   # Constructor: name aur age instance attributes hain
   def __init__(self, name, age):
      self.name = name      # Instance attribute

      self.age = age        # Instance attribute

   def display(self):
      print(f"Student: {self.name}, Age: {self.age}, School:
{self.school_name}")
# Students ke object banao
s1 = Student("Ali", 14)
s2 = Student("Sara", 13)
# Dono ka school name same hoga
s1.display()
s2.display()
Student: Ali, Age: 14, School: The Smart School
Student: Sara, Age: 13, School: The Smart School
```

Ab class attribute change karte hain:

```
# Ab hum school ka naam class se change karte hain

Student.school_name = "Happy Valley School"

# Check karo dono students ka updated school

s1.display()

s2.display()

Student: Ali, Age: 14, School: Happy Valley School

Student: Sara, Age: 13, School: Happy Valley School
```

Class attribute update hone se sab students pe effect aaya!

Ab agar object se change karein:

```
s1.school_name = "City Star School"  # sirf Ali ke liye change
s1.display()
s2.display()

Student: Ali, Age: 14, School: City Star School
Student: Sara, Age: 13, School: Happy Valley School
```

Ali ka school alag ho gaya!
Ye hua instance-level override (shadowing).

Accessing & Modifying Attributes:

Class attribute ko hum class name se ya object se access kar sakte hain:"

```
print(Dog.species)     # Class name se
print(dog1.species)    # Object se
```

Lekin agar hum object se modify karte hain to ek naya instance attribute ban jata hai."

🔍 __dict__ Attribute:

Python mein har object aur class ke paas ek __dict__ hoti hai jo unke saare attributes ko dictionary form mein store karti hai."

```
print("Class attributes:", Student.__dict__)
print("s1 instance attributes:", s1.__dict__)
print("s2 instance attributes:", s2.__dict__)
```

Sample Output:SS

```
Class attributes: {..., 'school_name': 'Happy Valley School', ...}
s1 instance attributes: {'name': 'Ali', 'age': 14, 'school_name': 'City Star School'}
s2 instance attributes: {'name': 'Sara', 'age': 13}
```

Key Takeaways (Viva mein line by line bolo):

1. *"Class attributes sab objects ke liye common hote hain."*
2. *"Instance attributes har object ke liye unique hote hain."*
3. *"Object se class attribute ko modify karne par ek naya instance attribute ban jata hai."*
4. *"__dict__ ka use karke hum object ya class ke attributes dekh sakte hain."*

5. Methods in Python Classes

Python mein class ke andar hum functions define karte hain jinhe methods kehte hain. Methods teen tarah ke hote hain: instance method, class method, aur static method."

1. Instance Methods

"Yeh sabse common method hota hai Isse hum object ke attributes ko access aur modify kar sakte hain." *yeh method self parameter leta hai aur specific object ka data access karta hai.*

Example:

```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks

    def show_info(self):  # Instance method
        print(f"Student: {self.name}, Marks: {self.marks}")

s1 = Student("Ali", 90)
s1.show_info()
# Output: Student: Ali, Marks: 90
```

self ka matlab: jis object ne method ko call kiya usi ka data lena.

2. Class Methods (@classmethod cls)

"Yeh method class ke upar kaam karta hai, na ki object ke. Iska pehla parameter **cls** hota hai jo class ko refer karta hai.
Isse hum class-level attributes ko modify kar sakte hain."

Example:

```
class Student:
    school_name = "ABC School"  # Class attribute

    @classmethod
    def change_school(cls, new_name):  # Class method
        cls.school_name = new_name
```

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

3. Static Methods (@staticmethod (NO Paramter…..)

"Yeh method class ya object dono se independent hota hai. Iska koi **self** ya **cls** parameter nahi hota.Yeh mostly utility functions ke liye use hota hai."

Static method woh hoti hai **jo na class ke data ko use karti hai, na object ke data ko**.Sirf ek **general/utility function** hoti hai jo class ke andar likhi hoti hai.

```
class Student:
    @staticmethod
    def welcome():
        print("Welcome to the Student Management System!")

Student.welcome()
# Output: Welcome to the Student Management System!
```

School Example - Sab Methods Ek Sath:

```
class Student:
    school_name = "City School"  # Class Attribute

    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

    def show_info(self):  # Instance Method
        print(f"{self.name} is in grade {self.grade} at {Student.school_name}")

    @classmethod
    def update_school(cls, new_name):  # Class Method
        cls.school_name = new_name

    @staticmethod
    def rules():  # Static Method
        print("School rules: Be on time. Wear uniform.")
```

```
# Create students
s1 = Student("Zara", "5th")
s2 = Student("Ahmed", "6th")

# Show info using instance method
s1.show_info()    # Zara is in grade 5th at City School
s2.show_info()    # Ahmed is in grade 6th at City School
```

```
# Update school name using class method
Student.update_school("The Bright School")

# Show info again
s1.show_info()    # Zara is in grade 5th at The Bright School
s2.show_info()   # Ahmed is in grade 6th at The Bright School

# Print rules using static method
Student.rules()    # School rules: Be on time. Wear uniform
```

6. Encapsulation:

"Data (attributes) aur us par kaam karne wale methods ko ek hi unit (class) mein band karna aur direct access ko control karna. **hai taake kisi bhi object ka internal data bina permission ke change na ho.**

Socho tumhari **Bank Account** hai. Usmein 3 cheezein hoti hain:

1. **Naam** – sabko dikh sakta hai (Public)
2. **Balance** – thoda sensitive hai (Protected)
3. **PIN** – secret hota hai (Private)

1. Public (عوامی / sab ke liye):

Jo sab ko dikh sakta hai

```
self.name = "Asma"  # Public
```

Sab jagah se access ho sakta hai Koi special nishan nahi lagta Isko har koi class ke bahar se bhi use kar sakta hai:

2. Protected (محفوظ / androni istemal)

Naam ke aagay **_ (ek underscore)** lagta hai **Bahar se access possible hai**, lekin kehna yeh hota hai: *"Ye sirf andr ka kaam hai, bahar se na chedho!"*

```
self._balance = 5000  # Protected
```

🔒 **Class aur uski subclass ke andar use hota hai**

3.Private (نجی / sirf class ke andar)

Naam ke aagay **__ (do underscore)** lagta hai
Python isko **mangling** kar deta hai — yani naam ko andar hi change kar deta hai Bahar se seedha access mushkil banata hai

Sirf class ke andar hi access hota hai

```
self.__pin = "1234"  # Private
```

Access Modifiers:

Q: Python mein access modifiers kaise kaam karte hain?

Python mein access modifiers ke liye hum naming conventions use karte hain — jaise koi variable public ho to direct likhte hain, protected ho to ek underscore _ aur private ho to double underscore __ use karte hain.

```
class BankAccount:
    def __init__(self, holder, balance):
        self.name = holder        # Public
        self._balance = balance    # Protected
        self.__pin = "1234"       # Private

    def display(self):
        print(f"Account Holder: {self.name}")
        print(f"Balance: Rs. {self._balance}")

    def get_pin(self):
        return "PIN is private, cannot show!"

    def set_balance(self, amount):
```

```
        if amount >= 0:
            self._balance = amount
            print("Balance updated.")
        else:
            print("Invalid balance.")


# object banate hain
acc = BankAccount("Asma", 5000)

# ✅ Public: direct access
print(acc.name)       # Asma

# ⚠️ Protected: possible but not recommended
print(acc._balance)    # 5000

# ❌ Private: error if try to access directly
# print(acc.__pin)     # ❌ ERROR

# ✅ Access through method
print(acc.get_pin())   # PIN is private, cannot show!

# ✅ Change balance safely
acc.set_balance(6000)  # Balance updated.
acc.display()      # Account Holder: Asma     Balance: Rs. 6000
```

Getter and Setter Methods

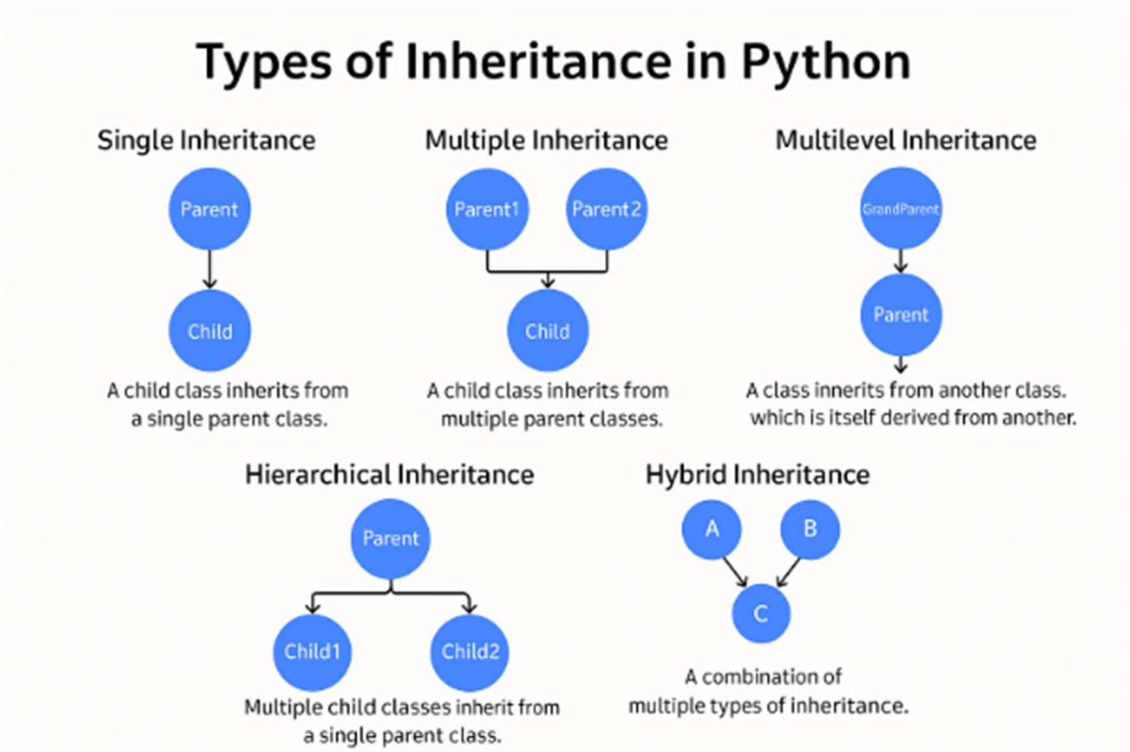
getter se hum data ko safely read karte hain, aur setter se safely **update** karte hain— *bina directly attribute ko access kiye.*

7. Inheritance

"*Aik class (child/subclass) doosri class (parent/superclass) ke attributes aur methods ko **reuse** kar sakti hai.*
Yani bar bar likhne ki zarurat nahi – reuse kar lo!

Q: Method overriding kya hota hai?

jab child class ek aisa method define karti hai jo already parent class mein defined hai, lekin uska behavior badal diya jata hai.



Types of Inheritance in Python (with Simple Explanation)

Python mein mainly **5 types of inheritance** hote hai

1. Single Inheritance (وراثت اکیلی)

☞ Ek **child class** ek hi **parent class** se inherit karti hai.

```
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return "Animal sound"

# Child class
class Dog(Animal):
    def speak(self):
        return "Woof!"

dog = Dog("Buddy")
print(f"{dog.name} says {dog.speak()}")  # Output: Buddy says Woof!
```

Dog class ne **Animal** se __init__ aur speak() inherit kiya, phir speak() ko override kar diya.

2. Multiple Inheritance وراثتیں کئی

☞ Ek **child class**, **do ya zyada parent classes** se inherit karti hai.

```
class Bird:
    def fly(self):
        return "Flying high!"

class Fish:
    def swim(self):
        return "Swimming deep!"

class FlyingFish(Bird, Fish):
    pass

flying_fish = FlyingFish()
print(flying_fish.fly())   # Flying high!
print(flying_fish.swim())  # Swimming deep!
```

3. **Multilevel Inheritance**

☞ Ek class, **ek aur child class** se inherit karti hai, jo khud ek **parent class** se inherit karti hai.

```
class GrandParent:
    pass

class Parent(GrandParent):
    pass

class Child(Parent):
    pass
```

3-level hierarchy: GrandParent → Parent → Child

4. Hierarchical Inheritance

☞ **Ek parent class**, multiple **child classes** ko inherit karata hai.

```
class Parent:
    pass

class Child1(Parent):
    pass

class Child2(Parent):
    pass
```

One parent → multiple children

5. Hybrid Inheritance

☞ Jab **multiple types of inheritance ek saath** use hoti hain (mix of single, multiple, multilevel etc.)

```
class A:
    pass

class B(A):
    pass

class C:
    pass

class D(B, C):  # Hybrid = Multilevel + Multiple
    pass
```

Method Overriding (Tariqa badalna)

Jab **child class** kisi **inherited method** ko **dobara likhti hai** (same name but different behavior), use kehte hain **Method Overriding**.

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

class Dog(Animal):
    def speak(self):  # Method overriding
        return "Woof!"
```

8. Polymorphism(Aik naam – multiple kaam)

Polymorphism ek hi method ya operator different classes mein different tareeqay se kaam kare.Yani **"ek hi interface, lekin different behavior."**
“Ek hi naam, lekin alag behavior.”
💡 **Poly = Many** + **Morph = Forms**

Yani **ek hi method ya operator** alag-alag class ke objects ke liye **alag kaam karega**.

🎯 Real-Life Example:

Agar main bolun **"Drive!"**, to:

- Car chalegi 🚗
- Truck chalega 🚚
- Bike chalegi 🏍️

Sabko **Drive()** bola, lekin sabka **style alag tha** — **yeh hi polymorphism hai**!

Types of Polymorphism in Python

1. **Method Overriding** (Wirasat mein method ko dobara likhna)

Jab subclass apne parent class ke method ko dobara define karta hai apne tareeqay se.

Example:

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

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

def animal_sound(animal):
    print(animal.speak())
animal_sound(Dog())   # Output: Woof!
animal_sound(Cat())   # Output: Meow
```

animal_sound() ek hi function hai — lekin Dog aur Cat alag response deti hain.

Yehi hota hai: **polymorphism through method overriding**.

2. **Operator Overloading** (Operators ka behavior badalna)

Jab hum normal Python operators (like +, -) ko apni class ke objects ke liye custom behavior de dete hain.

Example:

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

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

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

p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2
print(p3)  # Output: Point(4, 6)
```

Jab + lagate hain, to __add__() method chal jata hai. Python objects ka behavior aap define kar saktay ho.

3. **Duck Typing** ("Agar woh duck ki tarah chalti hai...")

Python mein **type important nahi**, behavior important hota hai. Agar koi object quack() method rakhta hai, to Python use duck maan leta hai.
Mujhe fark nahi padta ye object kis class ka hai. Agar ismein wo method ya function hai jo mujhe chahiye — to kaam ho jaayega

Example:

```
class Duck:
    def quack(self):
        return "Quack!"
class Person:
    def quack(self):
        return "I can quack like a duck!"
def make_it_quack(thing):
    print(thing.quack())
make_it_quack(Duck())    # Output: Quack!
make_it_quack(Person())  # Output: I can quack like a duck!
```

Function `make_it_quack()` ko `Duck` bhi diya aur `Person` bhi.

Dono mein `quack()` method tha — to kaam ho gaya!
Isay kehtay hain Duck Typing:

✅ Simple Jawab:

Inheritance-based polymorphism mein hum base class banate hain aur subclasses usay inherit karte hain, aur same method ko override karte hain.

Duck typing mein koi inheritance nahi hoti — agar object ke paas required method hai, to wo accept ho jata hai.

📌 **"Method ka naam same hona zaroori hota hai."**

📌 Ek Line mein farq:

Inheritance mein rishta hota hai, duck typing mein sirf method ka hona zaroori hota hai. class ka type important nahi hota.

🔑 Key Takeaways (Viva ke liye)

1. **Method Overriding** – subclass method ko override karta hai.
2. **Operator Overloading** – operators ka custom use.
3. **Duck Typing** – object ka behavior matter karta hai, uski class nahi.

9. Abstraction– Sirf "Kya", "Kaise" nahi

Abstraction Sirf important cheezon ko dikhana aur details ko chhupa dena.

complex cheezon ko simple tarike se represent karna. Hum sirf **function kya karega** ye batate hain — **kaise karega**, ye baad mein decide hota hai.

🔒 1. What is Abstraction? (Asan Alfaaz Mein)

Imagine tum remote se TV chala rahi ho — tum sirf **button dabati ho** (start()), magar **andar kya wiring hai, kaise signals ja rahe hain**, yeh sab **tumhein nahi pata hona chahiye** — **bas kaam hona chahiye**.
Yeh hi hota hai **abstraction** — *focus on **what** it does, not **how** it does.*

from abc import ABC, abstractmethod

ABC = Abstract Base Class
🏷 **@abstractmethod** = Ye method **lazmi** har child class ko banana hoga

Final Line:

Abstraction ek "farmaan" ki tarah hota hai —
"Jo bhi mera bacha ho (child class), usay yeh kaam lazmi karna hoga, lekin tareeqa us ka apna hoga!" 🎯

---

10. Python `object` Class & Unified Type System - Full Summary with Examples

🔹 1. `object` Class — Python ki Maa Class

✅ Base Class

- Har class Python mein `object` class se inherit karti hai (directly ya indirectly).
- Agar aap koi class bina kisi parent ke banate ho, Python by default `object` ko parent banata hai.

```
class MyClass:
    pass
obj = MyClass()
print(isinstance(obj, object))  # True
```

Agar aap koi class banate ho bina kisi base class ke, Python **automatically object ko parent bana deta hai**.

2. Default Methods

Iske paas kuch **methods already bana hotay hain**, jaise __init__, __str__, __repr__, etc.

Tumhari class agar yeh override nahi karti, to yeh object wale use ho jatay hain.

3. Unified Type System

Python mein **har cheez object hai**:

- Integers → int
- Strings → str
- Lists → list
- Functions, Classes, Modules... sab kuch!

```
print(type(5))       # <class 'int'>

print(type("hello"))  # <class 'str'>

print(type([1, 2]))   # <class 'list'>
```

"Python ka Unified Type System ka matlab hai ke har cheez — numbers, strings, lists, aur even functions — sab objects hote hain. Ye sab object class se inherit karte hain. Is wajah se unmein kuch common behavior hota hai jaise str(), type(), aur unke methods ko access karna dir() se. Is system se Python ka behavior consistent, flexible, aur dynamic hota hai."

✅ Benefits of the Unified Type System in Python — Simple Explanation

1. Simplicity (Asaani)

Har cheez object hai — alag-alag rules ya syntax nahi yaad rakhne padte.Aap ek hi tarike se har data type ke saath kaam kar sakte ho.

```
print(type(10))       # int

print(type("hello"))   # str
```

2. **Consistency (Ek Jaise Rules)**

- Sab types me similar behavior hota hai — same functions like type(), str(), dir() sab pe kaam karte hain.
- Chahe built-in ho ya custom class

```
print(str(42))          # "42"
print(str([1, 2, 3]))    # "[1, 2, 3]"
```

3. **Extensibility (Apni Classes Banana Aasaan)**

- Aap **apni custom classes** bana sakte ho jo built-in types ki tarah behave karein.
- Python ke special methods (like __str__, __len__, __getitem__) isme help karte hain.

```
class MyClass:
   def __str__(self):
      return "I'm custom!"

obj = MyClass()
print(str(obj))   # I'm custom
```

!

4. **Polymorphism (Ek Interface, Multiple Objects)**

Agar 2 alag types ke objects same method follow karein, to aap unhe **interchangeably** use kar sakte ho.

Yeh object-oriented concept Python mein naturally kaam karta hai

```
print(len("hello"))     # 5
print(len([1, 2, 3]))   # 3
```

✨ Conclusion:

Python ka unified type system ye ensure karta hai ke sab kuch object ho, aur sabhi same rules follow karein. Isse programming simple,

consistent, aur powerful ban jaati hai. Chahe aap kisi bhi type ke object ke saath kaam karo — aapka tarika ek jaisa hi rahega.

🔷 3. Important Magic Methods (Special Methods)

__str__ vs __repr__ → Display Karna

- __str__: For users (friendly)
- __repr__: For "Developer-view"

```
- def __str__(self): return "Nice Toy!"
- def __repr__(self): return "Toy('Nice Toy!')"
```

3. __len__ and __getitem__ → Length aur indexing

```
def __len__(self): return len(self.toys)
def __getitem__(self, index): return self.toys[index]
```

4. __eq__, __lt__, etc → Comparison Support

```
def __eq__(self, other): return self.size == other.size
```

4. Full Example — MagicToyCatalog

```
class MagicToyCatalog:
    def __init__(self):
        self.toys = {}

    def __setitem__(self, name, value):
        self.toys[name] = value

    def __getitem__(self, name):
        return self.toys.get(name)

    def __contains__(self, name):
```

```
        return name in self.toys

    def __len__(self):
        return len(self.toys)

    def __str__(self):
        return f"Catalog: {list(self.toys.keys())}"

    def __call__(self, discount):
        return f"Applied {discount}% discount"

    def __delitem__(self, name):
        del self.toys[name]

# ✅ Example Usage:
store = MagicToyCatalog()
store["RoboDog"] = (49.99, "RC robot dog")  # __setitem__
print(store["RoboDog"])                     # __getitem__
print("RoboDog" in store)                   # __contains__
print(len(store))                           # __len__
print(store)                                # __str__
print(store(10))                            # __call__
del store["RoboDog"]                        # __delitem__
```

🔷 Final Summary:

- `object` is Python ka root class.
- Har class (built-in ya custom) `object` se inherit karti hai.
- Python ka **Unified Type System** programming ko:
    - Simple
    - Consistent
    - Flexible banata hai.
- Special methods se aapki classes built-in types ki tarah behave karti hain.

💡 Viva Tip: "Python mein sab kuch object hai, isliye custom classes ko bhi powerful banana possible hota hai using magic methods."
