**Github link:**

**https://github.com/panaversity/learn-modern-ai-python/blob/main/00\_python\_colab/13\_traditional\_oop\_part\_2/Agentic\_AI\_Traditional\_Python\_Lesson\_13\_OOP\_Objects\_%26\_Classes\_Part\_2.ipynb**

**Dunder methods ya magic methods wo special methods hain jinka naam \_\_like\_this\_\_ hota hai, aur Python khud unko call karta hai specific operations ke liye.**

**11. Class and Static Variables:**

🔹 **Class aur Static Variables:**

Class aur static variables wo data store karte hain jo **class ke sath related hota hai**, na ke class ke objects (instances) ke sath.  
Ye variables sabhi objects (instances) ke liye **common** hote hain.

### **Class aur Static Variables ka Difference:**

Python main log aksar "class variable" aur "static variable" ko ek hi maante hain, lekin asal main **class variable** term zyada use hoti hai.

* **Class Variables :**
  + Ye class ke sath attach hote hain.
  + Class ke ander define hote hain **lekin kisi method ke ander nahi**.
  + Ye sabhi objects ke liye **same value** share karte hain.
* **Instance Variables :**
  + Ye har ek object ke liye **alag hoti hain**.
  + Inko hum \_\_init\_\_ constructor ke ander define karte hain.
  + Har object apni alag value rakhta hai.

👉 **Short samajh lo:**

* **Class Variable = sabhi objects ke liye common data.**
* **Instance Variable = har object ka apna data.**

### 🔹 **Accessing and Modifying Class Variables:**

Class variables ko hum **class name** ya phir **instance (object)** dono ke zariye access aur modify kar sakte hain.

* Directly class name se: ClassName.variable\_name
* Instance (object) se: instance\_name.variable\_name

### **Modifying (badalna)**

* Agar aap **instance ke zariye** class variable ko change karte ho → to asal main aap ek **naya instance variable bana rahe hote ho**, jo us object ke liye class variable ko shadow (chhupa) deta hai.
* Agar aap **class variable asli change karna** chahte ho → to aapko **class name se** hi us variable ko change karna hoga.

**Agar aap chahte ho ke real class variable hi change ho, to hamesha class name se change karo,**

|  |
| --- |
| **class Bakery:**  **type = "cake" *# Class variable***  **def \_\_init\_\_(self, flavor, price):**  **self.flavor = flavor *# Instance variable***  **self.price = price *# Instance variable***  **def update\_cake\_count(cls, count):**  **cls.cake\_count = count**  ***#Accessing***  **print(Bakery.type) output cake**  **cake1 = Bakery("Chocolate", 25.00)**  **cake2 = Bakery("Vanilla", 22.00)**  **print(cake1.flavor) output: chocolate**  **print(cake2.price) output: 22.0**  ***#Modifying***  **Bakery.type = "pastry" #** Directly class name se: ClassName.variable\_name  **Yahan humne class name se type ko change kiya, isliye sab cakes ke liye type = "pastry" ho gaya.**  **print(cake1.type) output: pastry**  **cake1.type = "cookie"**  **print(cake1.type) output: cookie #** Instance (object) se: instance\_name.variable\_name  **print(cake2.type) output: pastry** |

 Class Variable = "school ka common rule" (sab students ke liye same).

 Instance Variable = "student ka apna roll number / name" (har student ka alag).

# ****12. Composition and Aggregation:****

Dono **"has-a relationship"**— matlab **ek class ke andar doosri class ka object hona**.

Lekin **difference hota hai unka "relationship kitna strong ya weak hai."**

Composition mein object class ke andar banate hain, jabke Aggregation mein object pehle se bana hota hai aur sirf link hota hai.

Composition mein object ke andar doosra object hota hai aur dono ka lifecycle connected hota hai.

## **1. Composition (Strong Relationship) – "Ek part doosre ka hissa hai"**

Ye ek **"has-a" relationship** hai. Matlab ek object ke ander doosra object **contain** hota hai. Jo object contain hua hai uska **lifecycle depend** karta hai container object par.Agar container destroy ho jaye contained object bhi destroy ho jata hai. Ye **strong relationship** hai.

👉 Example:

* **Car aur Engine**
  + Car ke bina Engine ka koi matlab nahi.
  + Engine car ka hissa hai aur car ke sath hi chalti/band hoti hai.

**Simple Words:** Composition = "Agar parent destroy ho jaye, child bhi destroy ho jata hai."

|  |
| --- |
| class Engine:  def start(self):  print("Engine started")  class Car:  def \_\_init\_\_(self):  self.engine = Engine() # Composition  def drive(self):  self.engine.start()  print("Car is moving") |

 Engine() = Engine class ka ek naya object banaya.

 self.engine = Engine() = Is object ko Car ke andar store kiya.

 Yeh method nahi hai, yeh object creation hai.

 Yeh line Composition ko dikhati hai.

## **Agar hum Engine class ko delete kar dein (khatam kar dein) to?**

* **Composition mein, Engine Car ka essential part hota hai**.
* Matlab, **Car engine ke bina kaam nahi karega**.
* Agar hum Engine class ko hata dein, ya Engine object na banayein, to Car ka self.engine attribute hi nahi banega.
* Isliye, Car ka proper function bhi ruk jayega.

**Yeh hi "Strong Relationship" hota hai jo Composition mein hota hai.**

## **Yeh weak relationship nahi hai**

* Weak relationship (Aggregation) mein, contained object independent hota hai.
* Agar Engine class ya object delete ho jaye, Car fir bhi exist kar sakta hai (lekin uske paas Engine nahi hoga).
* Lekin Composition mein, Car aur Engine dono saath connected hain — ek ke bina doosra adhoora hai.

### **🔹 Aggregation (Weak Relationship):**

**Aggregation ek weak relationship hoti hai.  
Contained object independent hota hai, matlab agar main object destroy ho jaye, to doosra object zinda rehta hai.  
Jaise University aur Departments — university band ho jaye, to department phir bhi exist kar sakte hain.**

### **🔹 👉 1 Line Main Difference:**

* **Composition:** Strong "has-a", contained object dependent on container.
* **Aggregation:** Weak "has-a", contained object independent of container.

### **💡 Extra Tip (Agar time ho to bolo):**

**Composition mein object andar hi create hota hai.**  
**Aggregation mein object pehle se bana hota hai, aur class usko bas use karti hai.**

✅ **Aggregation (Weak Relationship) – "Ek object doosre se juda hua hai, lekin alag bhi reh sakta hai"**

* **Aggregation mein** ek object doosre object se sirf **connected hota hai**, lekin **uska hissa nahi hota.**
* Agar parent object delete ho jaye, to **child object zinda rehta hai.**
* **Independent hota hai.**
* Jaise: **University aur Departments**  
  Department University ka part hai, lekin agar University band ho jaye to Department kisi aur University se attach ho sakta hai.

|  |
| --- |
| **class Department:**  **def \_\_init\_\_(self, name):**  **self.name = name**  **class University:**  **def \_\_init\_\_(self, departments):**  **self.departments = departments # Aggregation**  **math\_dept = Department("Math")**  **cs\_dept = Department("CS")**  **uni = University([math\_dept, cs\_dept])** |

## 🔹 **Difference Between Composition and Inheritance:**

### **1. Relationship :**

* **Composition:** "has-a" relationship hota hai.(Ek object ke ander doosra object hota hai).
* **Inheritance:** "is-a" relationship hota hai.(Child class ek type hoti hai parent class ka)

**2.Flexibility :**

* **Composition:** Zyada flexible hai → aap easily components ko replace/change kar sakte ho bina poore system ko todhe.
* **Inheritance:** Kam flexible hai → ek parent change karne se child classes pe bhi asar padta hai.

### **3. Coupling (dependence)**

* **Composition:** Loose coupling (kam dependence) deta hai, code maintain karna asaan hota hai.
* **Inheritance:** Tight coupling create karta hai, jisse code samajhna aur maintain karna mushkil ho jata hai.

### **4. Code Reuse**

* **Composition:** Code reuse objects ko combine karke karta hai.
* **Inheritance:** Code reuse parent class ke methods/attributes ko inherit karke karta hai.

# ****13. Method Resolution Order (MRO):****

### **🔹 What is MRO (Method Resolution Order)?**

### **MRO ka matlab hota hai "method resolution order".**

Jab aapka class structure complex hota hai (jaise ek class 2 ya zyada classes se inherit karti hai), to Python decide karta hai ke method ya attribute pehle kis class se liya jaye — is order ko **MRO** kehte hain.

Iska pata lagane ke liye hum ClassName.mro() method use karte hain.  
Jo order return hota hai, usi order mein Python method ya attributes ko dhoondhta hai.

Python mein decorators ek aise functions hote hain jo kisi function ya method ke behavior ko bina uska code directly change kiye modify kar dete hain. Agar hum ek function ko decorator se decorate karte hain, to wo function pehle decorator ke through jaata hai, phir execute hota hai.

### **🔹 Python ka C3 Linearization Algorithm**

Python ek special algorithm use karta hai jiska naam hai **C3 Linearization**.

 **Subclass** hamesha apne **parent class se pehle** aati hai.

 Inheritance order preserve rehta hai (jo sequence aap likhte ho wahi follow hota hai).

 Koi class ek hi bar visit hoti hai (duplicate nahi hoti).

|  |
| --- |
| **class A:**  **def greet(self):**  **return "Hello from A"**  **class B(A):**  **def greet(self):**  **return "Hello from B"**  **class C(A):**  **def greet(self):**  **return "Hello from C"**  **class D(B, C):**  **pass**  ***# Create an instance of D***  **d = D()**  ***# Check the MRO of class D***  **print(D.mro()) *# Output: [, , , , ]***  ***# Call the greet method***  **print(d.greet()) *# Output: Hello from B*** |

* Python pehle D class mein greet() method dhoondhta hai — nahi mila.
* Fir B mein gaya — mil gaya — to wahi chalaya.
* Baaki classes (C, A, object) check hi nahi ki gayi.
* Ye sequence MRO ke rules follow karta hai.

|  |
| --- |
| * **[<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>]** * **Hello from B** |

### **📌 ✅ Is tarah tum viva/exam main confidently bol sakti ho:**

### "Sir, MRO Python ka wo rule hai jo decide karta hai ke multiple inheritance ke case main method/attribute kis order main search hoga. Hum *ClassName.mro()* use karke order dekh sakte hain, jo C3 Linearization algorithm follow karta hai."

# ****14. Decorators in Classes:****

## 🔹 **Definition**

Decorators Python ka ek powerful feature hai jo kisi function ya method ke behavior ko **modify (badal) ya extend (barha)** karne ke liye use hota hai.

Jab inhe **classes** par apply karte hain, to ye class ya uske methods ke behavior ko enhance kar dete hain.

Python main kuch special **property decorators** diye gaye hain (@property, @setter, @deleter) jo attribute access ko controlled way main handle karte hain.

1. **Function/Class Decorators – poore function ya class ka behavior change karte hain**
2. **Property Decorators – class ke attributes ko control karte hain (getter, setter, deleter)**

## **Class Decorators– (Class ko modify karna):**

* Ek class decorator asal main ek function hota hai jo **class ko input leta hai aur modified/extended class return karta hai**.
* Zyada tar use hota hai:
  + extra functionality add karne ke liye
  + rules/constraints enforce karne ke liye
  + class behavior ko modify karne ke liye

### **✅ Example:**

|  |
| --- |
| **class CountCalls:**  **def \_\_init\_\_(self, func):**  **self.func = func**  **self.call\_count = 0**  **def \_\_call\_\_(self, \*args, \*\*kwargs):**  **self.call\_count += 1**  **print(f"Decorator: Call {self.call\_count} of {self.func.\_\_name\_\_}")**  **return self.func(\*args, \*\*kwargs)**  **@CountCalls**  **def say\_hello(name):**  **print(f"Hello, {name}!")**  **say\_hello("Alice") Decorator: Call 1 of say\_hello Hello, Alice!**  **say\_hello("Bob") Decorator: Call 2 of say\_hello Hello, Bob!** |

👉 In this example, CountCalls ek class decorator hai jo count karta hai kitni dafa ek function call hua. @CountCalls syntax say\_hello function ke upar apply karta hai decorator. Har dafa jab say\_hello call hota hai, CountCalls ka \_\_call\_\_ method execute hota hai, jo call\_count increment karta hai aur ek message print karta hai.

|  |
| --- |
| *Define a class decorator*  def add\_greeting(cls):  def greet(self):  return f"Decorator: Hello from {self.\_\_class\_\_.\_\_name\_\_} | ", self.name  cls.greet = greet  return cls  *# Apply the decorator to a class*  @add\_greeting  class Person:  def \_\_init\_\_(self, name):  self.name = name  *# Create an instance of the decorated class*  person = Person("Alice")  print(person.greet()) *# Output: Hello from Person*    ('Decorator: Hello from Person | ', 'Alice') |

**Property Decorators– (Attributes ko method jesa control karna):**

@property decorator allow karta hai control karne ka kaise ek attribute access, set, ya delete hota hai class ke ander. Ye allow karta hai logic add karna (jaise validation ya calculations) jabke still ye ek simple attribute ki tarah lagta h

 @property decorator ek method ko **getter** banata hai.

 Iski wajah se aap class ke attribute ko **property ki tarah access** kar sakte ho, method call karne ki zaroorat nahi.

|  |  |
| --- | --- |
| class Student:  def \_\_init\_\_(self, marks):  self.\_marks = marks  @property  def marks(self):  return self.\_marks  @marks.setter  def marks(self, value):  if value < 0:  raise ValueError("Marks cannot be negative")  self.\_marks = value  s = Student(90)  print(s.marks) # Output: 90  s.marks = 100 # Setter called  print(s.marks) # Output: 100  # s.marks = -5 ❌ Raises error: Marks cannot be negative | **class Person:** **def \_\_init\_\_(self, name):** **self.\_name = name**  **# private variable convention (\_)** **@property** **def name(self):** **"""Getter for name"""** **return self.\_name****p = Person("Alice")** **print(p.name)**  **output:**  **Alice** |

### **1️⃣.. Basic Getter (Read-Only Property) :**

Ek method ko "getter" bana deta hai (jaise attribute read karte ho).  
Parentheses ki zaroorat nahi!

➡️ name attribute ki tarah behave karta hai, lekin asal main ye ek method hai!

👉 Viva line: “@property ek method ko attribute ki tarah access karne deta hai. Isse hum getter, setter, aur deleter bana kar controlled access karte hain.” p.name method hai lekin bina () ke use hota hai — jaise normal attribute.

**2️⃣ Setter (Change a Value with Validation) :**

|  |
| --- |
| class Person:  def \_\_init\_\_(self, name):  self.\_name = name  @property  def name(self):  return self.\_name  @name.setter  def name(self, new\_name):  if not isinstance(new\_name, str):  raise ValueError("Name must be a string!")  self.\_name = new\_name  *# Usage*  p = Person("Bob")  p.name = "Charlie" *# Works*  print(p.name) *# Output: Charlie*  *#p.name = 123 # ❌ Error! (ValueError: Name must be a string!) # uncomment to see error* |

**decorator value set karne ke liye use hota hai. Isme hum validation bhi add kar sakte hain. Jaise agar name string na ho to error throw karte hain."**

**3️⃣ Deleter (Remove an Attribute) :**

|  |
| --- |
| **class Person:**  **def \_\_init\_\_(self, name):**  **self.\_name = name**  **@property**  **def name(self):**  **return self.\_name**  **@name.deleter**  **def name(self):**  **print("Deleting name!")**  **del self.\_name**  ***# Usage***  **p = Person("Dave")**  **print(p.name) *# Output: Dave***  **del p.name *# Runs deleter***  ***#print(p.name) # ❌ Error! (AttributeError: 'Person' has no attribute '\_name') # uncomment to see error***    **Dave**  **Deleting name!** |

## 

**@deleter attribute delete karne ke liye use hota hai. Jab hum del obj.name likhte hain, to ye method call hota hai."**

### **4️⃣ Computed Property (Dynamic Value)**

Value ko fly pe calculate karna.

**Example:** BMI calculate karna height & weight se

|  |
| --- |
| **class Person:**  **def \_\_init\_\_(self, weight\_kg, height\_m):**  **self.weight = weight\_kg**  **self.height = height\_m**  **@property**  **def bmi(self):**  **"""Body Mass Index (weight / height²)"""**  **return self.weight / (self.height \*\* 2)**  ***# Usage***  **p = Person(70, 1.75) *# 70kg, 1.75m***  **print(p.bmi) *# Output: 22.857...***    **22.857142857142858** |

➡️ bmi.setter define nahi hai → is liye bmi directly change nahi kar sakte!

**" @property ka use karke BMI compute kar rahe hain. Ye ek dynamic value hai jo weight aur height se calculate hoti hai. Isko directly set nahi kar sakte."**

### **🎯 Key Takeaway:**

@property tumhara code cleaner & safer banata hai attribute access ko control karke, bina users ke interact karne ka tareeqa badle. 🚀

**✅ Decorators kya hain?  
→ Ye aise functions hote hain jo doosre function ya method ko wrap karte hain.  
→ Iska faida: aap us function ke behavior mein kuch extra add kar sakte hain bina usay change kiye.**

**✅ @property ka use kya hai?  
→ Jab aap chahte hain ke method ek variable ki tarah behave kare, to @property lagate hain.  
→ Iska faida: method call ki tarah nahi lagta, simple variable ki tarah lagta hai.**

**✅ @setter ka kaam kya hai?  
→ Jab aap chahte hain ke kisi value ko set karte waqt check ya condition lagayein (jaise: "sirf string allow ho").  
→ To setter use hota hai, jo property ko safely update karta hai.**

**✅ @deleter ka kaam kya hai?  
→ Jab aap kisi attribute ko delete karna chahein, aur us waqt kuch special kaam bhi karna ho (jaise: print karna, memory clear karna).  
→ To deleter lagaya jata hai.**

### **📌 Callable :**

**Jo cheez function ki tarah call ho sakti hai (yaani () laga ke chalayi ja sakti ho), usse callable kehte hain.**

* **Agar aap kisi object ko () ke saath likhein, aur wo chal jaye, to wo callable hai.**

**📌 Jaise:**

|  |
| --- |
| **print("Hello") # ✅ print() is callable** |

### 🔹 **Callable kaise banta hai? :**

Agar kisi object ke andar **\_\_call\_\_ method** ho to wo object **callable** hota hai.

|  |
| --- |
| class MyClass:  def \_\_call\_\_(self):  print("Main callable hoon!")  obj = MyClass()  obj() # Output: Main callable hoon! |

### 🔹 **Callable ke Examples**

1. **Functions** → Normal functions jo hum def keyword se banate hain.
2. **Lambda functions** → Choti anonymous (naam k bina) functions.
3. **Classes** → Class ko call karne se uska **instance** banta hai.
4. **Methods** → Class ke andar jo functions hote hain.
5. **Objects jin mein \_\_call\_\_ method ho** → Agar koi class \_\_call\_\_ method define kare to uska object bhi callable ban jata hai.

|  |  |
| --- | --- |
| **def my\_function():**  **pass**  **print(callable(my\_function)) # True**  **class MyClass:**  **def \_\_call\_\_(self):**  **pass**  **obj = MyClass()**  **print(callable(obj)) # True**  **print(callable("hello")) # False** | ✅ Function → callable ✅ Object with \_\_call\_\_ → callable ❌ String → callable nahi |

👉 Simple lafzon mai:

**✅ Jo cheez ko () laga ke chala sakte ho, wo callable hai.  
✅ Jo object \_\_call\_\_() method rakhta hai, wo bhi callable ban jata hai.**

# ****15. Working with Modules and Packages in OOP****

👉 **Module**  **= Ek Python file (.py)** (jisme functions, classes, variables likhe jaate hain).

👉 **Package** = ek folder hota hai jisme multiple modules hote hain **aur ek \_\_init\_\_.py file** hoti hai.

### **✅ Package:**

* **Ek folder jisme kai modules hoon + ek \_\_init\_\_.py file bhi ho.**
* **Ye Python ko batata hai ke ye folder ek package hai.**

**Socho package = ek folder jisme modules (files) hoti hain**

### **🔹 Q2: Package kya hota hai?**

* **Package = Ek folder (directory) jo multiple modules rakhta hai.**
* **Usme ek special file hoti hai \_\_init\_\_.py — jise Python use karta hai us folder ko package samajhne ke liye.**
* **Example: animals/, vehicles/**

**📦 Project Ka Structure (asani se yaad rakhne ka tarika)**

|  |
| --- |
| **my\_project/**  **│**  **├── animals/ ← 📁 Package (folder)**  **│ ├── mammals.py ← 🐶 Module (Dog, Cat)**  **│ ├── birds.py ← 🐦 Module (Parrot, Sparrow)**  **│ └── \_\_init\_\_.py ← Python ko batata hai: "ye package hai"**  **│**  **├── vehicles/**  **│ ├── cars.py ← 🚗 Module (Car)**  **│ ├── bikes.py ← 🏍️ Module (Bike)**  **│ └── \_\_init\_\_.py**  **│**  **└── main.py ← 🧠 Main file jo sabko use karti hai** |

## **📥 Classes ko Import Karna (3 tarike)**

### **🟢 1. Import poora module:**

|  |
| --- |
| **import animals.mammals**  **dog = animals.mammals.Dog()** |

|  |
| --- |
| **from animals.mammals import Dog, Cat**  **dog = Dog()** |

**✅ Ye method best hai!**

### **🔴 3. from module import \* (sab kuch import) ❌**

|  |
| --- |
| **from animals.mammals import \*** |

**⚠️ Yeh chhoti scripts ke liye theek hai, large projects mein avoid karo (confusion ho sakta hai).**

## **📘 init.py ka Role (Important Exam Point)**

**\_\_init\_\_.py hota hai har package folder mein:**

|  |
| --- |
| **# animals/\_\_init\_\_.py**  **from .mammals import Dog, Cat**  **from .birds import Parrot, Sparrow** |

|  |
| --- |
| **from animals import Dog, Cat** |

**🧪 main.py — Saari Classes ka Use**

|  |
| --- |
| **from animals import Dog, Cat, Parrot, Sparrow**  **from vehicles import Car, Bike**  **dog = Dog()**  **print(dog.speak()) # Woof!**  **car = Car("Toyota Corolla")**  **print(car.display()) # Car: Toyota Corolla** |

**Yeh aapki main controller file hoti hai — yahan aap sab classes ko import karte ho aur use karte ho.**

## **📌 Why is this useful in OOP?**

1. **🔄 Reusability → Bar bar likhne ki zarurat nahi**
2. **🔒 Encapsulation → Har file ka apna scope**
3. **🧩 Maintainability → Changes easily track ho jate hain**
4. **🚀 Scalability → Large codebase mein ye zaruri hota hai**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# ****16. Advanced OOP Concepts****

### **🔷 1. Metaclass**

**Python mein har cheez ek object hai—even classes bhi.  
Ek metaclass woh class hoti hai jo dusri classes ko banati hai.**

* **Normal class → objects banati hai.**
* **Metaclass → classes banati hai.**

|  |
| --- |
| * ***# Custom metaclass*** * **class Meta(type):** * **def \_\_new\_\_(cls, name, bases, dct):** * **print(f"Creating class: {name}")** * **return super().\_\_new\_\_(cls, name, bases, dct)** * ***# Class using the custom metaclass*** * **class MyClass(metaclass=Meta):** * **pass** * ***# Output: Creating class: MyClass*** * **Creating class: MyClass** |

#### **🔍 Urdu Explanation:**

* **Meta ek metaclass hai jo har class banate waqt message print karega.**
* **Jab MyClass banayi gayi, Meta ka \_\_new\_\_ method chala.**
* **Output: Creating class: MyClass**

#### **🎯 Use Kab Karein?**

* **Jab aapko har class ke creation ko customize karna ho (e.g., validation, tracking, dynamic features).**

### **🔷 2. Singleton Pattern (Ek hi object har dafa mile)**

#### **📌 Basic Idea:**

**Aap chahte hain ke koi class ka sirf ek hi object ho — baar baar class() karne se naya object na mile.**

#### **✅ Code:**

|  |
| --- |
| **class Singleton:**  **\_instance = None**  **def \_\_new\_\_(cls, \*args, \*\*kwargs):**  **if cls.\_instance is None:**  **cls.\_instance = super().\_\_new\_\_(cls)**  **return cls.\_instance**  ***# Create instances of the Singleton class***  **singleton1 = Singleton()**  **singleton2 = Singleton()**  ***# Check if both instances are the same***  **print(singleton1 is singleton2) *# Output: True***  **print(id(singleton1) == id(singleton2)) *# Output: True***    **True**  **True** |

#### **🔍 Urdu Explanation:**

* **Pehli dafa Singleton() call karne par ek object banta hai.**
* **Agli dafa wahi object wapas milta hai.**
* **is ka matlab: kya dono object memory mein same hain? Haan!**

#### **🎯 Use Kab Karein?**

* **Jab aapko ek hi global cheez chahiye ho (e.g., database connection, settings).**

### **🔷 3. Factory Pattern (Object banana — but smart tareeqe se)**

#### **📌 Basic Idea:**

**Factory pattern se aap input ke base par alag alag objects bana sakte ho — bina class ka naam likhe.**

|  |
| --- |
| ***# Product interface***  **class Animal:**  **def speak(self):**  **pass**  ***# Concrete products***  **class Dog(Animal):**  **def speak(self):**  **return "Woof!"**  **class Cat(Animal):**  **def speak(self):**  **return "Meow!"**  ***# Factory class***  **class AnimalFactory:**  **@staticmethod**  **def create\_animal(animal\_type):**  **if animal\_type == "dog":**  **return Dog()**  **elif animal\_type == "cat":**  **return Cat()**  **else:**  **raise ValueError("Invalid animal type")**  ***# Use the factory to create animals***  **dog = AnimalFactory.create\_animal("dog")**  **cat = AnimalFactory.create\_animal("cat")**  ***# Call the speak method***  **print(dog.speak()) *# Output: Woof!***  **print(cat.speak()) *# Output: Meow!***    **Woof!**  **Meow!** |

#### **🔍 Urdu Explanation:**

* **Aap user se input lete hain (e.g., "dog" ya "cat").**
* **Factory class decide karti hai ke kaunsi class banani hai.**
* **Aap ko directly Dog() ya Cat() likhne ki zarurat nahi hoti.**

#### **🎯 Use Kab Karein?**

* **Jab aapko future mein naye types add karne hoon (extensibility).**
* **Jab object creation ko encapsulate karna ho (e.g., forms, API clients, GUIs).**

# ****17. Error Handling in OOP****

**Jab aap Python ka program likhti ho aur koi galti hoti hai (jaise kisi cheez ka wrong input), to Python aapko error dikhata hai.  
Error handling ka matlab hota hai:  
✅ “Jab kuch galat ho, to hum panic na karein — us galti ko pyar se sambhalein.”**

## **Raising Exceptions in Methods:**

## **Raise Keyword — Jab Khud Se Error Dikhana Ho**

**Agar aap kisi method ke andar dekh rahi ho ke user ne kuch galat kiya hai (jaise – negative paise deposit karna), to aap khud keh sakti ho:**

**“Hey, yeh galat hai!”**

**Aur aap likhogi:**

|  |
| --- |
| **raise ValueError("Deposit amount must be positive.")** |

**Yeh kya karega?  
Python ko bolega ke “Stop karo! Ye kaam theek nahi hai!”**

## **Custom Exceptions — Apni Galti Banani**

**Jaise toys alag alag design ke hote hain, Python ke errors bhi alag hote hain.**

**Aap khud bhi ek naya toy (error) bana sakti ho:**

|  |
| --- |
| **class InsufficientFundsError(Exception):**  **...** |

**ska matlab hai:**

**“Main ek naya type ka error banayi hoon — jab balance kam ho, to usey dikhana!”**

## **Real Life Example: Bank Account**

**Zara sochiye — aapki ek piggy bank hai. Usmein paise daal bhi sakti ho (deposit), nikaal bhi sakti ho (withdraw), aur dekh bhi sakti ho ke kitna paisa hai (display).**

**Agar aap kuch galat karne ki koshish karo — jaise:**

**🚫 -100 rupees daalna  
🚫 2000 rupees nikalna jab 1000 hi hain**

**To Python kya karega?  
“Nahi!” kahega, aur error dega.**

**💡 Code Samajhna Step by Step**

|  |
| --- |
| **class BankAccount:**  **def deposit(self, amount):**  **if amount <= 0:**  **raise ValueError("Deposit amount must be positive.")** |

**👧🏻 Aap deposit mein negative paisa nahi daal sakti ho. Isi liye:  
✅ Raise ValueError**

|  |
| --- |
| **def withdraw(self, amount):**  **if amount <= 0:**  **raise ValueError("Withdrawal amount must be positive.")**  **if self.balance < amount:**  **raise InsufficientFundsError(self.balance, amount)** |

**👧🏻 Agar aapke paas ₹1000 hain, aur aap ₹2000 nikaalna chaho:  
❌ Toh custom error ayega: InsufficientFundsError**

## **try & except — Pyar Se Galti Sambhalna**

|  |
| --- |
| **try:**  **account.withdraw(2000)**  **except InsufficientFundsError as e:**  **print(e)** |

**try:  
➡️ Matlab: “Koshish karo…”**

**except:  
➡️ Matlab: “Agar error aaye, toh panic nahi — pyar se sambhalo!”**

## **🎓 Exam/Viva ke liye Tips:**

* **💬 Agar poocha jaaye: “Custom exception kya hoti hai?”  
  ➤ “Jab humein koi khaas type ki galti banana ho, toh hum apna naya error banate hain using class MyError(Exception).”**
* **💬 “Withdraw ke time kya check karte hain?”  
  ➤ “Amount negative na ho aur balance kaafi ho.”**
* **💬 “Kya karein jab error aaye?”  
  ➤ “try-except block use karke program ko crash hone se bachaate hain.”**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# ****18. Testing OOP Code****

**Testing ka matlab hota hai:  
Hum check karte hain ke jo program (ya class/method) likha gaya hai wo sahi kaam kar raha hai ya nahi.**

**📌 Testing se humein pata chalta hai:**

* **Kya result sahi aa raha hai?**
* **Kya koi error toh nahi aa raha?**
* **Kya methods expected tareeqay se behave kar rahe hain?**

## **🔍 2. Types of Testing (Jo hum seekh rahe hain)**

1. **Unit Testing  
   👉 Har choti cheez (method/class) ko alag alag test karna  
   📌 Example: add() method sahi kaam kar raha hai ya nahi?**

**Frameworks  
Hum do use karte hain:**

* + **unittest (Python ka built-in)**
  + **pytest (easy aur zyada popular)**

## **🧮 3. Calculator Class (jo hum test karenge)**

|  |
| --- |
| **class Calculator:**  **def add(self, a, b):**  **return a + b**  **def subtract(self, a, b):**  **return a - b**  **def multiply(self, a, b):**  **return a \* b**  **def divide(self, a, b):**  **if b == 0:**  **raise ValueError("Cannot divide by zero.")**  **return a / b** |

### **🔍 Is code mein kya ho raha hai?**

* **add: 2 numbers jodta hai.**
* **subtract: minus karta hai.**
* **multiply: multiply karta hai.**
* **divide: divide karta hai. Agar 0 se divide kiya, toh error deta hai (ValueError).**

## **✅ 4. Testing with unittest (Asaan explanation)**

|  |
| --- |
| **import unittest**  **class TestCalculator(unittest.TestCase):**  **def setUp(self):**  **self.calc = Calculator()** |

### **🔍 Explanation:**

* **unittest.TestCase = Special class for testing**
* **setUp() = har test se pehle yeh Calculator banata hai**

### **🎯 Tests:**

|  |
| --- |
| **def test\_add(self):**  **self.assertEqual(self.calc.add(2, 3), 5)**  **def test\_subtract(self):**  **self.assertEqual(self.calc.subtract(5, 3), 2)**  **def test\_multiply(self):**  **self.assertEqual(self.calc.multiply(2, 3), 6)**  **def test\_divide(self):**  **self.assertEqual(self.calc.divide(6, 3), 2)**  **def test\_divide\_by\_zero(self):**  **with self.assertRaises(ValueError):**  **self.calc.divide(6, 0)** |

### **📌 Har test kya check kar raha hai?**

* **test\_add: 2+3 = 5 check karta hai**
* **test\_subtract: 5−3 = 2**
* **test\_multiply: 2×3 = 6**
* **test\_divide: 6÷3 = 2**
* **test\_divide\_by\_zero: Error raise hona chahiye**

### **▶️ Run kaise karte hain?**

|  |
| --- |
| **if \_\_name\_\_ == '\_\_main\_\_':**  **unittest.main(argv=['first-arg-is-ignored'], exit=False)** |

**🔹 Isse test run ho jaate hain, result milta hai: OK, FAIL, etc.**

## **🔥 Viva Questions for unittest:**

1. **Q: Unit testing kya hoti hai?  
   A: Ek individual method ya class ko test karna unit testing kehlata hai.**
2. **Q: setUp method ka kya role hota hai?  
   A: Har test se pehle run hota hai, Calculator object banata hai.**
3. **Q: assertEqual kya karta hai?  
   A: Yeh check karta hai ke result expected ke equal hai ya nahi.**
4. **Q: ValueError kyun raise karte hain?  
   A: Jab user 0 se divide karta hai, toh error generate karte hain.**

**🧪 5. Testing with pytest**

|  |
| --- |
| **import pytest**  **@pytest.fixture**  **def calc():**  **return Calculator()** |

### **🔍 @pytest.fixture ka matlab:**

* **Yeh ek function hai jo object return karta hai — yahan Calculator.**
* **Har test mein yeh calc provide karta hai.**

|  |
| --- |
| * **def test\_add(calc):** * **assert calc.add(2, 3) == 5** * **def test\_divide\_by\_zero(calc):** * **with pytest.raises(ValueError):** * **calc.divide(6, 0)** |

### **🔹 assert:**

* **Yeh direct check karta hai ke value sahi hai ya nahi.**

### **▶️ Run kaise karein?**

**pytest –v**

**Agar koi test galat hua toh pytest woh test fail dikhaata hai.**

## **🔥 Viva Questions for pytest:**

1. **Q: pytest aur unittest mein kya fark hai?  
   A: pytest zyada readable aur flexible hota hai. unittest built-in hota hai.**
2. **Q: assert aur assertEqual mein kya farq hai?  
   A: assert pytest mein use hota hai, assertEqual unittest mein hota hai.**
3. **Q: pytest.raises ka kya kaam hai?  
   A: Exception test karta hai (jaise ValueError raise hua ya nahi).**

**✅ Output Example:**

|  |
| --- |
| **.....**  **----------------------------------------------------------------------**  **Ran 5 tests in 0.001s**  **OK** |

# ****19. Best Practices in OOP****

# ✅ ****SOLID Principles :–****

**"SOLID" ek shortcut (mnemonic) hai 5 golden rules ke liye jo Object-Oriented Programming (OOP) ko strong banate hain — code ko asaan, saf, aur future mein change karne laayak banate hain.**

## **🔴 1. SRP – Single Responsibility Principle**

**Ek class sirf ek kaam kare.**

### **🧒 Simple Story:**

**Ek class sirf ek kaam kare.  
Agar class ko sirf ek wajah se change karna ho — toh woh SRP follow kar rahi hai.+**

**Socho ek bacha homework bhi kar raha hai, jhooti bhi polish kar raha hai aur khana bhi paka raha hai — kya woh sab kuch ache se karega? Nahi na?**

**Isi tarah, ek class sirf ek zimmedari uthaye — ya report banaye, ya report save kare.**

### **❌ Ghalat Code (Bacha har kaam akela kar raha hai)**

|  |
| --- |
| **class Report:**  **def generate\_report(self, data):**  **# Report banata hai**  **pass**  **def save\_report(self, file\_path):**  **# Report ko save bhi karta hai**  **pass** |

**🧠 Yeh class report bhi banati hai aur file bhi save karti hai — 2 kaam = 2 responsibilities = violates SRP**

**✅ Sahi Code (Har kaam ke liye alag bacha 😄)**

|  |
| --- |
| **class ReportGenerator:**  **def generate\_report(self, data):**  **# Sirf report banata hai**  **pass**  **class ReportSaver:**  **def save\_report(self, report, file\_path):**  **# Sirf report save karta hai**  **pass** |

**💡 Ek class = ek zimmedaari  
🎯 Viva Question:  
Q: SRP kya hai?  
A: Har class ka sirf ek kaam hona chahiye. Agar ek class do alag kaam kare, toh woh SRP tod rahi hai.**

## **🟡 2. OCP – Open/Closed Principle**

**Code ko extend karo, lekin usay modify mat karo.**

### **🧒 Simple Story:**

**Agar aap game bana rahe ho aur har naye level ke liye purana code todna pad raha ho — to woh thik nahi.**

**Is principle ka matlab: naya feature add karo bina purana code tod ke.**

### **❌ Ghalat Code (Har shape ke liye “if” likhna pad raha hai)**

|  |
| --- |
| **class AreaCalculator:**  **def calculate\_area(self, shape):**  **if shape.type == "circle":**  **return 3.14 \* shape.radius \*\* 2**  **elif shape.type == "rectangle":**  **return shape.length \* shape.width** |

**👎 Har baar naya shape (triangle, square, etc.) add karna ho toh code change karna padega.**

**✅ Sahi Code (Naye shapes add karo easily)**

|  |
| --- |
| **from abc import ABC, abstractmethod**  **class Shape(ABC):**  **@abstractmethod**  **def area(self):**  **pass**  **class Circle(Shape):**  **def \_\_init\_\_(self, radius):**  **self.radius = radius**  **def area(self):**  **return 3.14 \* self.radius \*\* 2**  **class Rectangle(Shape):**  **def \_\_init\_\_(self, length, width):**  **self.length = length**  **self.width = width**  **def area(self):**  **return self.length \* self.width**  **class AreaCalculator:**  **def calculate\_area(self, shape):**  **return shape.area()** |

**✨ Ab agar aap triangle banana chaho, toh bas ek Triangle class add karo. Purana code bilkul bhi nahi chedna!**

**🎯 Viva Question:  
Q: Open/Closed Principle kya hai?  
A: Code aisa likho ke naye features add ho jayein bina purana code badle.**

**📌 Ab naye shape (Triangle) add karna ho to bas naya class banao — purana code untouched.**

## **🟢 3. LSP – Liskov Substitution Principle**

### **🔤 Simple Urdu:**

**Jahan parent class ka object chale, wahan child class ka object bhi bina dikkat chale.**

**Child class parent ki jagah le sakti ho bina koi masla kiye.**

### **🧒 Simple Story:**

**Agar Mama keh kar kisi ko bulao aur Papa aa jayein — to confusion ho jaye ga 😆**

**Isi tarah: agar koi class “Bird” hai, to uska child “Ostrich” bhi usi tarah behave kare — bina kuch tod-phod kiye.**

**❌ Ghalat Code (Ostrich udaya nahi ja sakta!)**

|  |
| --- |
| **class Bird:**  **def fly(self):**  **pass**  **class Ostrich(Bird):**  **def fly(self):**  **raise NotImplementedError("Ostriches can't fly")** |

**👎 Ostrich ne fly method ko bigaad diya — yeh LSP todta hai!**

**✅ Sahi Code (Sab birds move karte hain — kisi ka fly, kisi ka run)**

|  |
| --- |
| **class Bird:**  **def move(self):**  **pass**  **class Sparrow(Bird):**  **def move(self):**  **print("Flying")**  **class Ostrich(Bird):**  **def move(self):**  **print("Running")** |

**💡 Ab koi bhi Bird ho, woh move() karegi — bina error ke. LSP follow ho gaya.**

**🎯 Viva Question:  
Q: LSP kya hai?  
A: Child class parent class ke jagah use ho sakti hai bina program today.**

## **🧱 4. I – Interface Segregation Principle (ISP)**

### **🔤 Simple Urdu:**

**Kisi class ko woh methods na do jo woh use hi nahi karti.  
Har device ko sirf uske kaam ka interface do.**

### **❌ Ghalat: Sab functionality ek hi class mein**

|  |
| --- |
| **class Printer:**  **def print\_document(self):**  **pass**  **def scan\_document(self):**  **pass**  **def fax\_document(self):**  **pass** |

**✅ Sahi: Har kaam ka apna interface (class)**

|  |
| --- |
| **class Printer:**  **def print\_document(self):**  **pass**  **class Scanner:**  **def scan\_document(self):**  **pass**  **class FaxMachine:**  **def fax\_document(self):**  **pass** |

**🧠 Har class sirf woh kaam karti hai jo uske liye zaroori hai.**

**🎯 Viva Question:  
Q: ISP kya hai?  
A: Kisi bhi class ko unnecessary methods nahi dene chahiye. Interface chhoti aur specific honi chahiye.**

## **🧱 5. D – Dependency Inversion Principle (DIP**

**Big classes (high-level) aur choti classes (low-level) dono abstract ideas (interfaces) par depend karein — ek dosre par nahi.**

### **❌ Ghalat: Switch ne LightBulb ko directly banaya**

|  |
| --- |
| **class LightBulb:**  **def turn\_on(self):**  **pass**  **def turn\_off(self):**  **pass**  **class Switch:**  **def \_\_init\_\_(self):**  **self.bulb = LightBulb() # tightly coupled**  **def operate(self):**  **if condition:**  **self.bulb.turn\_on()**  **else:**  **self.bulb.turn\_off()** |

**👎 Ab switch sirf LightBulb ke saath hi kaam karega.**

### **✅ Sahi: Abstraction ke through depend karna**

|  |
| --- |
| **from abc import ABC, abstractmethod**  **class Switchable(ABC):**  **@abstractmethod**  **def turn\_on(self):**  **pass**  **@abstractmethod**  **def turn\_off(self):**  **pass**  **class LightBulb(Switchable):**  **def turn\_on(self):**  **print("Light ON")**  **def turn\_off(self):**  **print("Light OFF")**  **class Switch:**  **def \_\_init\_\_(self, device: Switchable):**  **self.device = device**  **def operate(self):**  **if condition:**  **self.device.turn\_on()**  **else:**  **self.device.turn\_off()** |

**✨ Ab switch kisi bhi Switchable device (LightBulb, Fan, etc.) ko chala sakta hai — zyada flexible aur scalable!**

**🎯 Viva Question:  
Q: DIP kya hai?  
A: High-level aur low-level code dono abstractions par depend karein, ek dosre par directly nahi.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# ****20. Iterable:-****

**Iterable woh cheez hoti hai jisko for loop mein use kar sakte hain.  
Jaise list, string, tuple — in sab pe hum for loop chala sakte hain.**

|  |
| --- |
| **for item in something:**  **print(item)** |

**Agar something ke andar aap loop chala sakte ho, to woh Iterable hai.**

## **📦 Examples of Built-in Iterables**

**Yeh sab cheezein Iterable hain:**

* **✅ list → [1, 2, 3]**
* **✅ tuple → (1, 2, 3)**
* **✅ string → "hello"**
* **✅ dict → {"a": 1, "b": 2}**
* **✅ set → {1, 2, 3}**
* **✅ range → range(5)**
* **✅ generator → (x for x in range(3))**

**Aap in sab pe for loop chala sakte ho.**

### **💡 Ek object Iterable kab kehlata hai?**

**Agar us object ke andar \_\_iter\_\_() method hota hai (ya indirectly iter() function kaam karta hai), toh woh Iterable kehlata hai.**

## **🔎 Kya Iterable ek parent class hai?**

**Nahi! ❌**

* **Iterable parent class nahi hai.**
* **Iterable ek interface (protocol) hai — abstract base class (ABC) ke form mein defined hoti hai collections.abc module ke andar.**

**👉 Matlab:  
Aap koi bhi apni class bana lo — agar usmein \_\_iter\_\_() method likh do, toh woh iterable ban jaayegi!**

**✅ Check karne ka tareeqa: isinstance()**

**🧠 Code ka simple Urdu explanation:**

|  |
| --- |
| **from collections.abc import Iterable # Iterable ko import kiya**  **print(isinstance([1, 2, 3], Iterable))**  **# Kya list iterable hai? → Haan! → True**  **print(isinstance("hello", Iterable))**  **# Kya string iterable hai? → Haan! → True**  **print(isinstance(123, Iterable))**  **# Kya integer iterable hai? → Nahi! → False** |

## **🔧 Apni class ko iterable kaise banayein?**

**Aap \_\_iter\_\_() method define karo:**

|  |
| --- |
| **class MyList:**  **def \_\_init\_\_(self, data):**  **self.data = data**  **def \_\_iter\_\_(self):**  **return iter(self.data) # iterable bana diya**  **obj = MyList([10, 20, 30])**  **for i in obj:**  **print(i)** |

|  |
| --- |
| **10**  **20**  **30** |

## **🎓 Viva/Exam Questions + Answers:**

### **❓ Q1: Iterable kya hota hai?**

**🗣 A: Iterable aisa object hota hai jisme for loop chala sakte hain. Woh \_\_iter\_\_() method implement karta hai.**

### **❓ Q2: Kya list aur string iterable hain?**

**🗣 A: Haan, list aur string iterable hain. Hum inke elements pe loop laga sakte hain.**

### **❓ Q3: Kaise check karein ke koi object iterable hai ya nahi?**

**🗣 A: isinstance(obj, Iterable) function se check karte hain.**

### **❓ Q4: Kya integer iterable hota hai?**

**🗣 A: Nahi, int iterable nahi hota. Uspar for loop nahi chalta**

### **❓ Q2: Built-in Iterable objects kon se hain?**

**🗣 A: List, Tuple, String, Set, Dict, Range, Generator.**

### **❓ Q3: Kya custom class iterable ho sakti hai?**

**🗣 A: Haan, agar usmein \_\_iter\_\_() method ho aur woh ek iterator return kare.**

### **❓ Q4: Iterator aur Iterable mein farq?**

**🗣 A:**

* **Iterable: jisme loop chalta hai (\_\_iter\_\_() hota hai)**
* **Iterator: jo loop ko ek ek item deta hai (\_\_next\_\_() hota hai)**

|  |
| --- |
| **from collections.abc import Iterable, Iterator**  **class MyIterable(Iterable):**  **def \_\_init\_\_(self, data):**  **self.data = data**  **def \_\_iter\_\_(self):**  **return MyIterator(self.data)**  **class MyIterator(Iterator):**  **def \_\_init\_\_(self, data):**  **self.data = data**  **self.index = 0**  **def \_\_next\_\_(self):**  **if self.index >= len(self.data):**  **raise StopIteration**  **value = self.data[self.index]**  **self.index += 1**  **print("Called: MyIterator.\_\_next\_\_")**  **return value**  ***# Usage***  **my\_iterable = MyIterable([1, 2, 3])**  **for item in my\_iterable:**  **print("item : ",item) *# Output: 1, 2, 3***    **Called: MyIterator.\_\_next\_\_**  **item : 1**  **Called: MyIterator.\_\_next\_\_**  **item : 2**  **Called: MyIterator.\_\_next\_\_**  **item : 3** |

### **❓ Q5: Kya int Iterable hai?**

**🗣 A: Nahi. int pe loop nahi chalta. isinstance(123, Iterable) → False**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# ****Object-Based Language vs. Object-Oriented Language****

### **🎯 Object-Based Language kya hoti hai?**

**Yeh aisi programming language hoti hai jo:**

* **Objects banane ki facility deti hai (data + methods)**
* **Encapsulation (data hiding) support karti hai  
  Lekin:**
* **Inheritance, Polymorphism, aur Abstraction properly support nahi karti**

#### **🔧 Example:**

|  |
| --- |
| **// JavaScript (object-based example)**  **var car = {**  **brand: "Toyota",**  **drive: function() {**  **console.log("Driving...");**  **}**  **};**  **car.drive(); // Works fine** |

**Par: class Car extends Vehicle jaisa inheritance classical OOP jaisa nahi hota.**

### **🧱 Object-Oriented Language kya hoti hai?**

**Yeh woh language hoti hai jo OOP ke 4 pillars ko properly support karti hai:**

#### **✅ 1. Encapsulation – Data ko methods ke zariye hide karna**

#### **✅ 2. Inheritance – Ek class dusri class ki properties use kare**

#### **✅ 3. Polymorphism – Ek method ka alag-alag behavior**

#### **✅ 4. Abstraction – Sirf necessary information dena, details chhupana**

#### **🔧 Example in Python:**

|  |
| --- |
| **class Animal:**  **def speak(self):**  **print("Animal sound")**  **class Dog(Animal): # Inheritance**  **def speak(self): # Polymorphism (overriding)**  **print("Bark")**  **d = Dog()**  **d.speak() # Output: Bark** |

## **🐍 Python: Object-Oriented ya Object-Based?**

**✅ Python is a fully Object-Oriented Language**

### **🤯 Sab kuch object hai Python mein? Haan!**

**Har cheez — numbers, strings, functions, classes, modules — sab objects hain.**

### **🔎 Proof with type():**

|  |
| --- |
| **print(type(5)) # <class 'int'>**  **print(type("hello")) # <class 'str'>**  **print(type([1, 2, 3])) # <class 'list'>**  **print(type(None)) # <class 'NoneType'>**  **def hello(): pass**  **print(type(hello)) # <class 'function'>**  **class A: pass**  **a = A()**  **print(type(A)) # <class 'type'> (Classes bhi object hain!)**  **print(type(a)) # <class '\_\_main\_\_.A'>** |

## **📘 Viva/Interview ke liye Summary**

### **🔸 Q1: Object-Based Language kya hoti hai?**

**🗣️ A: Object-Based language wo hoti hai jo objects aur encapsulation support karti hai, lekin inheritance, polymorphism, aur abstraction jese OOP concepts ko nahi support karti.**

### **🔸 Q2: Object-Oriented Language kya hoti hai?**

**🗣️ A: Ye wo language hoti hai jo OOP ke 4 principles (encapsulation, inheritance, polymorphism, abstraction) ko fully support karti hai. Software design ko reusable aur organized banati hai.**

### **🔸 Q3: Kya Python fully object-oriented language hai?**

**🗣️ A: Haan! Python mein sab kuch object hota hai — numbers, strings, functions, even classes. Is wajah se Python consistent, flexible, aur powerful language hai.**

## **💬 Recap in 1 Line:**

**Object-based languages = partial OOP (only objects), Object-oriented languages = full OOP (objects + 4 pillars), Python = everything is an object.**

**<<<<<<<<<End>>>>>>>>>>**