Object Oriented programming in python "Apna College"

Introduction to oops in python

→ oops is paradigm that models real life entities

→ procedual programming focuses on functions,

oops focuses on objects

→ Benefits: modularity, reuseability, Scalability and better Code organization

→ Till now we only used functions which → redudancy ↓
→ reuseability ↑

→ To map with real world Scenarios, we started using objects in code.

→ This is called object-oriented programming

Object? → anything can be made a object
→ but before that we will make the class of the object

Class → like a blue print

Classes and Objects:-

- Class: Blueprint for Creating objects
- → object:- Instance of a class

\# Creating class

\# we can imagine this as a classroom, but without students class cannot be good!!

```
Class Student:
    name = "Himesh"
```

Class → Keyword!

\# Creating object (instance)

```
s1 = Student()
print(s1.name)  → Himesh
```

class (Blueprint) lopala vunde attributes fix ayyipothai manam use chesi prathi object ki adhe value comes

Constructor:- → This will invoke (Execute) during obj Creation

→ All classes have a function called __init__ (Self Calling function) which is always executed when the class is being initiated

\# Creating a class

```
Class Student:
    def __init__(self, fullname):
        self.name = fullname
```

\# Creating object

```
s1 = Student("Jimmy")
print(s1.name)
```

Self → obj variable (self-variable = var2)

Object ko create karte time hum log () isiliye lagate hai taki __init__ Const. khud se call hojaye!

Types of Constructors:-

① **Default:-** These are those constructors which has only single parameter, self. If we don't write this in python, it will automatically write.

```
def __init__(self):
    pass
```

→ This is the basic outlet of default constructors.

② **Parameterized Constructors.**

```
def __init__(self, a, b, etc)
```

These are those constructors which has attributes other than self.

→ Developers usually do like this:

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

Custom practice

⚠ If we have more than one constructors in the same class, then the one which matches our attributes will be executed.

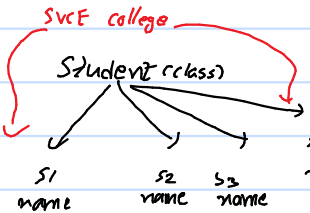
→ Generally as such we don't make many constructors we see the need and make them acc to the need

Class and instance attributes:-

Attributes are those type of variables or data in my class.

Class Attributes:- These are like college → All branches have same eg

Object Attributes:- These are like branches, each branch is different.



⇒ Class Variable → Stored only once!

This is fixed for all objects

We define the variable at after class.

name ⇒ Instance variable,

different for different obj

always defined by self.x = x

if we have same named variable in class & object attributes then:

obj attribute > class attributes.

Methods:-

Usually every class has → data (attributes)
→ methods (like func)

→ Methods are functions that belongs to objects.

whenever we use methods in class always make the first parameter should be self.

#Creating Class

```
class Student:

    def __init__(self, full name):

        self.name = full name

    def hello(self):

        print("hello", self.name)
```

#Creating object

```
S1 = Student("Kannan")

S1.hello()
```

* In lists & dictionaries or other databases we used a lot of methods

Those methods are derived this way

- i can change the value of the attributes after obj creation

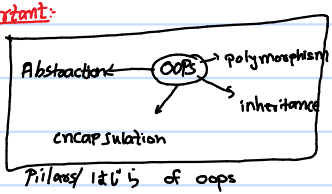
Static methods:-

Methods that don't use the self parameter (work at class level)

```
class Student:

    @staticmethod #decorator

    def college():

        Print("IIT Tokyo")
```

Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it

Important:-



Pillars/ ಆಧಾರ ಸ್ತಂಭ of oops

Abstraction:-

→ Hiding the implementation details of a class and only showing the essential features to the user.

→ eg: When using a phone, you don't need to know how it works internally.

→ Python uses abstract classes via the abc module:

```
from abc import ABC, abstractmethod

Class Shape(ABC):

    @abstractmethod
    def area(self): → Pass
```

Encapsulation:-

- Bundling data and methods that operates on that data

→ wrapping data and functions into a single unit (object)

- Access modifiers:
  - Public: accessible everywhere
  - protected: accessible within class & Subclasses (_var)
  - private: accessible only within class (__var)

Part 2

Del Keyword

- The del keyword is used to delete object properties or the entire object itself.
- objects occupy memory space due to their methods and attributes.
- This helps in removing unused objects or attributes to free up memory.
- Syntax: del object_name.Property (or) del object_name
- Code example:
  - Attempting to access a deleted attribute or object will result in error.

Private Attributes:-

→ In languages like C++ or Java, public attributes/methods are accessible outside the class, while private ones are not.

→ Python's concept of private attributes is conceptual, not strictly enforced like in other languages.

→ To make an attribute or method "private" in python, we prefix its name with two underscores. This triggers "name mangling".

→ "Private" attributes/method are intended to be used only with the class and not accessible from outside.

→ The purpose is to prevent exposing instance attributes outside the class.

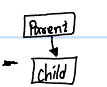
We can point — Secret inside the class but outside the class? — No.

Inheritance:-

- Definition: When one class (child/derived class) derives properties and methods from another class (parent/base class).
- Analogous to real-life inheritance, where values are passed from parents to children.
- promotes code reusability by allowing common logic to be defined once and inherited by multiple classes.

Types of inheritance:-

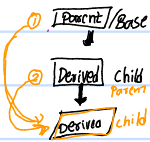
- Single Inheritance (single-level-inheritance): A single parent class and a single derived class.



Eg: Car → Toyota Car → Fortuner

Multi-level inheritance:- A base class, a derived class from it, and another class derived from the first derived class.

eg: Car → ToyotaCar → Fortuner



Multiple inheritance : A child class inherits from multiple parent classes.

→ Instead of multi-level inheritance, here the inheritance can be done in a way so that many ⟶ one.

→ Calling the parents will be like Class(□,□) ○→ child

- - - ⇒ Parents

Super Method:-

The "super" method is a special tool used in inheritance. Here's how it works:

Purpose: It let's the child classes access methods(or) properties of their parent class without directly referencing the parent class name. eg: super().__init__() ⇒ Calls the parent class Constructor.

Advantages:- Avoid's hard coding parent class names making it easier to adjust class heirarchies later.

Usage: Typical used in Constructors or overridden methods.

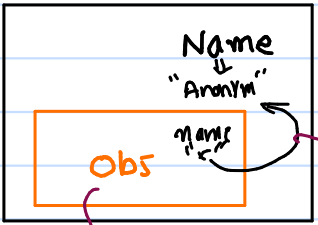
Class method:-

→ A class method is bound to the class.

→ Defined using @classmethods, they work with the class itself, not instances.

Static methods:-

→ Defined using @staticmethod, they don't access or modify class state & ideal for utility functions.



Created a new name but we want to change the name that is already there.

We can fix this in various ways

① Use `Class.att = att`

```
Class Person:
    name = "x"
  def changeName(self, name):
    Person.name = name
P = Person()
P.changeName("y")
Print(P.name) = y
Print(Person.name) = y
```

earlier this was "x"

② Inside the def function,

self.__class__.name = name

↑
This total says that i am the obj of this class having this attribute.

We can either write classname/ self.__class__ } but process is same

③ Using class methods.

→ It uses cls instead of self

Final summary:-

① Static ⇒ use when you don't want anything to change

② class (cls) ⇒ changes class attributes

③ instance (self) ⇒ changes instance attributes

Property decorator:-

Purpose: A @property decorator let's you define methods that acts like attributes, making access dynamic

```
Class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks
    @property
    def grade(self):
        return "Pass" if self.marks >= 40 else "Fail"
Student = Student("Rahul", 50), Print(St.grade) = Output: Pass
```

Usage: use @property for Computed attributes without exposing methods.

Polymorphism:

Definition:- Polymorphism allows methods in different classes to share the same name but behave completely differently.

eg A draw() method in Circle and Rectangle classes perform Shape-specific drawing

Types:-

Compile-time polymorphism: Achived via method overloading (less common in python)

Run-time polymorphism: Achived via method overriding in inheritance

Example Code:

```
class Animal:
    def speak(self):
        pass
class Dog(Animal):
    def speak(self):
        return "woof!"
class Cat(Animal):
    def speak(self):
        return "Meow"

animals = [Dog(), Cat()]
for animal in animals:
    Print(animal.speak())
```

→ output: woof!
Meow

Simplified polymorphism:

polymorphism = one interface, many forms

→ The same function name or operator work differently depending on the object.

eg `print(3+5)` # int add^n → 8

`print("a"+"b")` # string concatenation "ab"

Here '+' is polymorphic, it depends on operand types.

overloading in python:-

Unlike Java and C++, python does not support true method/ functional overloading. if we define the same function twice, the last definition replaces the previous one.

But overloading is achieved by:

- Default arguments
- variable-length arguments (*args, **kwargs)
- functools.singledispatch for function overloading by type

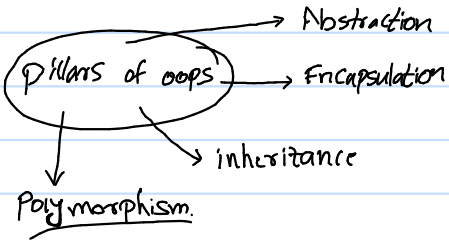
Types of overloading:-

- operator overloading: Redifining how operators work on user-defined objs

```
class Vector:
    def __init__(self,x,y):
        self.x,self.y= x,y
    def __add__(self,other):
        return Vector(self.x + other.x , self.y + other.y)

V1= Vector(1,2)
V2 = Vector(3,4)
V3 = V1+V2
print(V3.x,V3.y)
```

1+3=4, 2+4=6

Method overloading:- Not supported natively, but stimulated using default/variable arguments.



Dunder functions:-

These are double underscore functions like __len__, __add__, __str__, etc are special methods that let your objects behave like built-in types. They are the backbone of polymorphism in python.

operators & Dunder functions:-

a + b	# addition	a.__add__(b)
a - b	# subtraction	a.__sub__(b)
a * b	# multiplication	a.__mul__(b)
a / b	# division	a.__truediv__(b)
a % b	# remainder	a.__mod__(b)

Practice Questions

Q. Define a Circle class to create a circle with radius r using the Constructor. Define an Area() method of the class which calculates the area of the circle, Define a perimeter() method of the class which allows you to calculate the perimeter of the circle.

Q. Define a Employee class with attributes role, department & salary. This class also has show details method

Create an Engineering class that inherits properties from Employee & has additional attributes: name & age

Q. Create a class called Orders which stores item & it's price.

Use Dunder func __gt__() to convey that:

Order1 > Order2 if price of Order1 > price of Order2