

# CH : 09      OOPs

## #1) INTRODUCTION

### \* Procedural Programming

- Procedural Programming means program that uses a step-by-step approach to solve problems
- Programs written (without OOPs concept) by using fun., for loops, dictionary etc is called procedural Programming
- programs written till now, without mapping real world Entity & without using class-object concept are called procedural programming

### \* Object Oriented Programming (OOPS)

- In OOPS we will map variables/data with real world entities
- eg: Enemies in game
- We will have classes & object using which you can create fun() & its data
- This way of writing program using classes-object (Inheritance, Abstraction, Encapsulation, polymorphism ) is called OOPS
- FEATURES OF OOPS :

- 1) class - objects : blue print having data & methods
- 2) Abstraction : data is private
- 3) Encapsulation : implementation details are hidden
- 4) inheritance : phone & drone in features add the feature
- 5) polymorphism : one thing having many forms

## #2) CLASS - OBJECT

### \* Class

- class is a blueprint or template for creating objects
- It defines the properties & methods that an object of that class will have
- Properties are the data or state of an object & methods are the actions or behaviour that an object can perform
- Syntax:

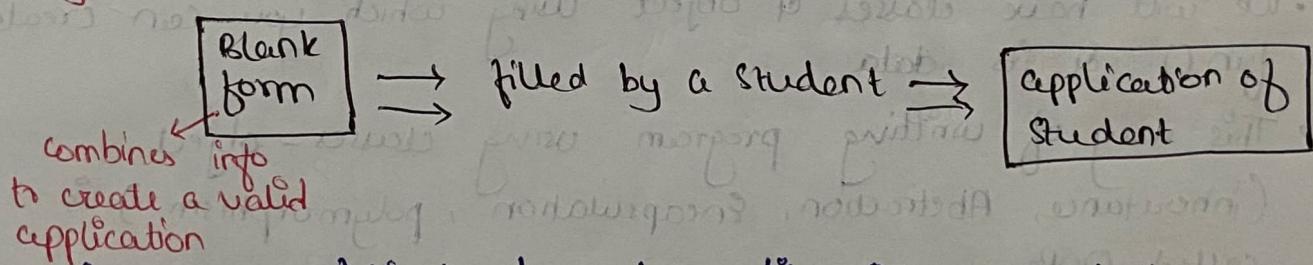
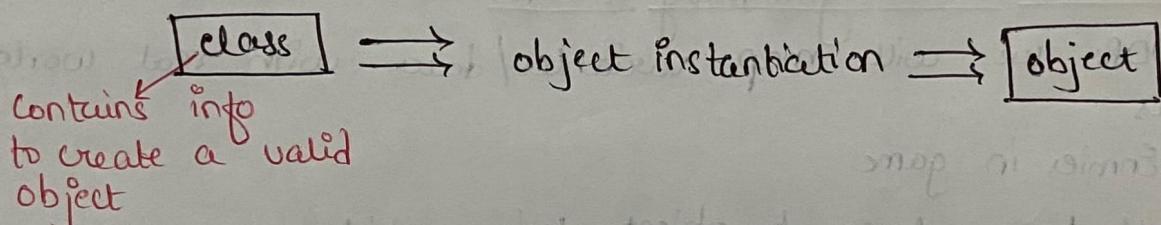
class class\_name :

    // class methods

    // class variable

class Name is written in PascalCase  
first letter in every word is Capital

- "class" is a keyword



- class is a user defined data type like int, float, tuple etc

### \* Object

- object is an instance (instantiation) of a class
- object is entity of class
- object is a variable of class as its data type
- object of class can make/invoke the methods available to it without revealing the implementation details to the user → // This is called Encapsulation & Abstraction

- Syntax :

obj\_variable\_name = class\_name()

- you can access data & methods of class using (.) dot operator

eg:

```
class Employee:  
    language = "Py" ] // class attributes  
    sal = 12,000
```

class object → // Vishu = Employee()

vishu.name = "Vishwajeet" // This is object/instance attribute

print(vishu.name, vishu.sal)

default values is // vishu.language = "JavaScript" // instance attribute take preference over class attribute during assignment & retrieval  
change  
print(vishu.language)

Output:

Vishwajeet, 12,000  
JavaScript

eg:

```
class Employee:
```

language = "Py" → class attributes

sal = 12,000

// self means the object on which this method is called

class method ← def get(self):

object is passed and that object is self

print("The language is",  
 self.language, "of user",  
 self.name)

vishu = Employee()

vishu.name = "Vishwajeet" → object attribute

vishu.language = "C++"

vishu.get() // same as Employee.get(vishu)

OR

Employee.get(vishu) // that's why we use self, self means the object on which we are calling class method

#### NOTE

→ In this class method  
→ self can use & self is argument or self is variable

Output:

The language is C++ of user Vishwajeet

## \* Class Attribute

- An attribute (data members) to the class
- eg:

class Emp :

    company = "Google" // class attribute

vishu = Emp() // class object or instance of class

vishu.company = "facebook" // object can change the value of class attribute

vishu.age = 22 // object attribute  
                  adding age attribute

## \* Instance/Object attribute

- An attribute (data member) that has been made using object

- eg:

class Emp :

    company = "Google" // if you don't change 'company' value then this default value will be taken as "Google"

vishu = Emp()

vishu.age = 22      ] // object attribute

vishu.sal = 11,000    ]

vishu.company = "Facebook"

print(vishu.age, vishu.company)

Output:

22. Facebook

- Instance/Object attribute takes preference over class attribute during assignment & retrieval

### #3 Attributes / Data Members, Data Functions / Functions

#### #3 ATTRIBUTES & FUNCTIONS

##### \* Class Attributes

- Attributes: are nothing but properties or data members <sup>of</sup> in a class
- Class Attributes: There are the attributes (data members/variables) defined directly inside class.
- These attributes belong to class & are shared by all obj objects. The memory address or attribute copy is shared among all objects/instances
- These class attributes are same as static data members in C# or Java
- Class attributes are defined in class but outside any method & changing the value of class attribute affects all instances & reflects the change everywhere
- You can access class attribute using "class\_name.attribute" or "instance.attribute" both class & object
- Saves memory <sup>as</sup> memory efficient bcoz address is shared among all the instances
- You can define class attribute (static data members in C# or Java) & instance attributes with same name just like local & global variable
- You can also add class attributes from:
  - 1) directly inside the class using directly attribute names
  - 2) directly inside the functions or constructor using class name dot attribute name of same class
  - 3) you can add class attribute from another class functions/constructor also using class name & attribute name
  - 4) directly from outside the class, from anywhere
- `className.__dict__`: all class attributes you create are stored in this "`__dict__`"
- `className.__dir__()`: all methods of that class you can see by print. bcoz they are stored in "`__dir__()`" method

eg: which show class attribute behaviour

class Car:

wheels = 4 // class attribute defined normally & gets share by  
status = "available" instance

// Constructor → every fun takes self as default first  
def \_\_init\_\_(self, color): parameter inside every class

    self.color = color // instance attribute unique per object

    Car.brand = "BMW" // class attribute added from constructor

    self.status = "sold" // instance attribute added, with  
                        same name as class attribute

def set\_speed(self, speed):

    Car.max\_speed = speed // class attribute created  
// creating class attribute using method inside class

class Updater: from outside

def \_\_init\_\_(self):

    Car.type = "Sedan" → Adding class attribute from  
                        another class constructor  
                        & function

def update\_brand(self, new\_brand):

    Car.brand = new\_brand

    Car.Breaks = "Disc Breaks" // creating new class

    Car.country = "Germany"

attribute using another  
class function

c1 = Car("Red") // class object created

~~del~~

c2 = Car("Pink") // This class object will internally look like:

c2 = Car(c1, "Pink")

c1.set\_speed(250)

→ This c2 is internally  
    pcar which is nothing  
    but object & so we  
    need self parameter  
    inside functions & constructor

U = Updater() → Creating & modifying  
    Car class attribute

// Output:

print("class attribute car.wheels:", Car.wheels) // 4

print("Access through object:", c1.wheels) // 4

print("Car Brand:", Car.brand, c1.brand) // Tesla, Tesla

print("Car status using class:", Car.status) // available

print("Car status using obj:", c1.status) // sold

print("Car Break", Car.Breaks) // Disc Breaks

## \* Instance Attribute

- Instance attributes : These are the data members (instance attributes) which belong to individual objects, not to the class itself.
- These are basically instance ~~variable~~ data variable or instance properties.
- Each instance has its separate copy of its attributes.
- If instance & class has same name attributes then if you access the attribute by class name then class attribute will get accessed & when you use instance name then instance attribute is accessed.
- You can create instance attribute from following ways:
  - 1) Defined using "self.attribute\_name" inside constructor of class
  - 2) Defined in functions of class using "self"
  - 3) You can also add/create instance attribute (instance properties) from outside the class, by directly using the object of class "obj.attribute\_name"
  - 4) You can create/add attributes ~~to~~ from another class also. using built-in function `setattr(obj, attr, val)`
- `setattr(obj, attribute-name, val)`
  - ↳ pass the class object
  - ↳ pass the attribute you want to create
  - ↳ Value of attribute
- `setattr(obj, attr, val)`
  - ↳ This fun is equivalent to = `obj.attr = val`
- "del" : Using del keyword you can delete created attributes  
eg: `del cl.color` // delete color attribute
- `obj.__dict__` : all the instance (object) attributes which you make are stored in this `"__dict__"`
- Shadowing of Attribute : If you create instance & class attribute of same name then instance attribute will shadow (hide) that class attribute. like method hiding in (#

eg:

class Car:

Brand = "BMW" → class attribute

def \_\_init\_\_(self, color):

self.color = color

self.speed = 0

] creating instance attribute inside  
constructor

def set\_owner(self, name):

self.name = name → creating instance attribute in function

def remove\_owner(self, name):

del self.name // deleting instance attribute

c1 = car("Red")

c1.model = 2025 // Creating instance attribute outside the class

directly using instance name & attribute name

setattr(c1, "milage", 15) → creating instance attribute using  
"setattr" function dynamically

class Updater:

def updateCar(self, obj, attr, val):

setattr(obj, attr, val) → // creating instance  
attribute ~~inside~~ inside

u = Updater()

u.update(c1, manufacturing-year, 2000)

del c1.model // deleting attribute

print("c1.color", c1.color)

c1.Brand = "apple" // creating instance attribute to shadow class  
attribute

print("shadowing eg", car.Brand)

print("using obj", c1.Brand)

] // shows attribute  
or instance attribute  
shadowing / hiding

## \* Self parameter

- self refers to the current object/instance of the class  
↳ is used to access variables that belongs to the class
- it is automatically passed with a member fun() call from an object of class
- self means current object
- self = object which is calling the fun.

• eg:

class Emp :

company = "Google"

age = 22

def getInfo (self): // member fun

print("age is", self.age)

→ if you don't pass self  
then you can't use  
self.age & you will get error

→ self is a object, vishu

→ vishu.age

Object //

vishu = Emp()

vishu.getInfo ()

OR

Emp.getInfo (vishu)

→ // both are same

→ by default the object is passed  
as parameter by python :- to  
catch that obj inside the member  
fun we have self

- for every fun you create in class, you need to pass 'self' parameter, you take any variable name in place of self, as 's' also
- to avoid self, you need to mention that fun as @staticmethod
- by marking a fun @staticmethod you are telling that it does not take any object as parameter

• eg:

@staticmethod

def getInfo ():

// body of fun

// same as above class method

// you can't use any class attribute inside this fun

## \* static method

- if you don't pass self in member fun of class then you will get error
- you can use "static method" by which you can define fun without self
- sometimes we need a fun that doesn't use the self parameter ∴ we can define a static method
- Syntax :

@staticmethod → // it is a decorator

```
def getInfo():  
    print("Hello")
```

- यह "@staticmethod" fun को mark करने से वहा ऐसे की fun self नहीं लेता, self का या class का को data member access नहीं करता

- eg:

```
class Emp:
```

```
    Company = "Facebook"
```

```
    age = 21
```

```
@staticmethod
```

```
def greet():
```

```
    print("Hello World")
```

```
vishu = Emp()
```

```
vishu.greet()
```

Output:

Hello World

- you can also call greet() fun using class name as

```
Emp.greet()
```

## ⑪ class methods

### \* Class methods

- A class method is a method which is bound to the class & not the object of class
- @classmethod decorator is used to create a class method
- It takes an additional argument `cls` (self) instance/obj नहीं होता, वह क्लास होता है जैसे `cls`
- इस कोई भी name variable नहीं लिया जा सकता & `self, cls` जैसे नाम नहीं लिया जा सकता
- syntax:  
`@classmethod  
def fun-name(cls, p1, p2):`
- you can't change actual class variables using `self` or object ∵ for that you need to pass class, which can be done using class method
- eg:

class Emp:

    company = "Apple"

    def show(self):

        print (self.name "and" self.company)

    def change (cls, newCompany):

        cls.company = newCompany

e1 = Emp()

e1.name = "Vishal"

e1.show()

e1.change ("microsoft")

e1.show()

print (e1.company) → apple

print (e1.company) → microsoft

↳ e1.company is obj variable

you can't change class variable using `self` becoz `self` is instance or object

eg:

class emp:

    company = "apple"

    def show(self):

        print(f" name is {self.name} comp is {self.company}")

    @classmethod

→ Emp class

    def change(cls, newCompany):

        cls.company = newCompany

e1 = emp()

e1.name = "Vishu"

e1.show()

e1.change("Tesla")

e1.show()

print(e1.company)

print(Emp.company)

Output :

Tesla

now the cls is  
the class Emp  
∴ It is Emp.company

now the class  
variable will get  
change

## \* Data Functions / Methods

• you can define 3 types of Methods inside a class

- 1) Instance Methods (default functions)
- 2) class Methods (@classmethod decorator)
- 3) static Methods (@staticmethod decorator)

### 1) Instance Methods

- these are the default function which you create without using any decorator
- these functions by default will always take self parameter which is nothing but the object which calls that function
- these instance methods can be accessed using both the object(instance) name & class name also.  
but if you use

1) using instance/object : Then directly automatically self parameter is passed

eg: c1 = Car("Red")

c1.show\_color() // automatically "c1" obj  
is passed as "self" to show\_color() function

2) using class name : Then the self parameter must be sent manually to the function while calling it

eg: c1 = Car("Red")

Car.show\_color(c1) // manually "c1" obj  
is passed as "self" parameter to function

• "self" will always be the first parameter by default

eg: class Car:

wheels = 4

def \_\_init\_\_(self, color):

self.color = color

def show\_color(self):

print(self.color)

→ Constructor called

car\_1 = Car("Red")

car\_1.show\_color() // using instance calling

Car.show\_color(c1) // calling method using class name

## 2) Class Methods

- There methods belong to the class itself (like we have static methods for (#))
- you need to use "@classmethod" decorator to define a class method
- you can access these method using both class name as well as instance of that class.
- "cls": automatically first parameter passed is the class itself & it's automatically passed while calling the function, no matter you call the function using class name or instance name first parameter is always class & no need to manually pass it at time of calling

eg: class Car:

wheels = 4

def \_\_init\_\_(self, color):

    self.color = color

@classmethod

def change\_wheels(cls, n)

    cls.wheels = n

    print("wheels changed to:", n)

this is called  
decorator

this parameter is not  
but class "Car"

Car.change\_wheels(10) // directly calling using class name  
& there is no need to pass  
"Car" as self parameter

c1 = Car("Red")

c1.change\_wheels(20)

→ // calling using instance/object  
at this time also no need to  
pass "Car" as self parameter  
while calling, automatically  
it does.

- you can call these function using both class & instance

### 3) static Methods

- There are the methods (not particularly belong to instance or class) which don't take "self" or "cls" parameter
- These are normal functions, but these are not equal to static functions like C#
- You can call these function using ~~the~~ class or instance name
- You need to use `@staticmethod` as decorator, which tells that this function doesn't take "self" or "cls" parameter
- Static method are not same as static methods of C# or C++ or Java.
- Static methods in python are not equal to class methods or doesn't belong to class or instance property only

• Eg: Class Car:

wheels = 4

def \_\_init\_\_(self, color):

self.color = color

`@staticmethod`

def change\_wheels():

print("no cls & self parameter taken")

Car.wheels = 10

c1 = Car("Red")

c1.change\_wheels

Car.change\_wheels ] you can call using both class as well as instance

## #A) Constructor & Destructors

### \* Constructor

- if there was no constructor (not even implicit default constructor) then you cannot make object same as C++, Java
- There is always a default constructor present inside the class if you don't make any explicit constructor
- Constructors are used to initialize the data attributes of class

### \* Types :

1) Default Constructor : this constructor is provided by compiler by default & it doesn't take any parameter other than self

- if you don't make any constructor then compiler gives you one default constructor

eg: class Car:  
    def \_\_init\_\_(self):  
        //Empty

you can use \*args  
& \*\*kwargs for  
constructor overloading

you can use class  
methods also to achieve  
constructor overloading

- But if you make any of the constructor like parameterized or a explicit parameterless (default) constructor then the implicit (automatic) default constructor is not present

• There is no concept of constructor overloading in python, only one constructor

2) Parameterized : This parametric constructor is explicitly defined by programmer & it will always take self as first parameter & other parameter you can pass to it

- It's used to initialize values to attributes
- In Python there is no concept of constructor overloading.
- You can make only one constructor in Python
- You can either create parameterless or only one parameterized constructor, but you can't do constructor overloading in Python

\* There is always only one constructor in Python at max, you can't do constructor overloading

## \* Constructor

- Constructor are fun which will get automatically when you make an object of class
- you can <sup>also</sup> pass parameters/arguments to constructor
- you can make default constructor & parameter constructor
- `__init__()` : is a special method which is first run as soon as the object is created by this `__init__()` method is also called constructor
- it takes self argument & can also take more arguments to set values of attributes/data of class
- Syntax :

```
def __init__(self, other-parameter):  
    // body of constructor
```

- Constructor cannot return any value other than none
- you can also make attributes / data members in constructor like we make object attributes
- eg:

class Person:

```
def __init__(self, name, age): // parametrized constructor  
    print("I am constructor")  
    self.first-name = name ] // data members  
    self.age = age ] or attributes  
                      of your created  
                      using constructor
```

```
def info(self):
```

```
    print("name is", self.first-name)  
    print("age is", self.age)
```

```
a = Person("Vishu", 22)
```

```
a.info()
```

Output:

name is Vishu  
age is 22

- when you don't pass any parameter (other than self) then it is called default constructor

## #5 Destructor

### \* Destructor

- **Destructor**: it's a method that runs when an object is about to be destroyed to free up resources / space
- The calling of the Destructor depends on the GC & is unpredictable
- There can only be one destructor, which only takes "self" (object) as the parameter & does not return anything
- If you don't define destructor, then there is no implicit (automatic) destructor defined unlike constructor
- If you don't define destructor, then Python GC will automatically do clean up operations
- Use Case: of the destructor is that, when you want to do extra clean up operations like closing a file if opened, closing data base connection, releasing lock etc

### • Syntax:

```
def __del__(self):
    // destructor body
    // Code to close file resources
```

e.g. class Demo:

```
def __init__(self):
    print("Constructor invoked")
```

```
def __del__(self):
    print("Destructor invoked")
```

### // Output:

Constructor invoked  
Destructor invoked  
End of program

```
obj = Demo()
gc.collect() // manually deleting obj
del obj
print("End of program")
```

→ gc.collect(): this function will call gc & GC will call destructor, it's optional to call (gc.collect())

### • Destructor gets automatically called when:

- 1) when reference count becomes zero, means when no variable references that object anymore then its reference count become 0
- 2) when object goes out of scope: means when a function finishes its execution that time destructor comes to do clean up
- 3) when Garbage Collector cleans up cycle: means destructor will be called when GC collects the objects

# CH : 01 ABSTRACTION

## #1 ABSTRACTION INTRO

### \* Abstraction :

- Abstraction means hiding the internal implementation & showing only the necessary features to the user.
- In simple words, user just knows what the function does rather than how that function works
- We hide the working & just show the output or feature of functions
- Real life eg: TV remote = we don't know the internal circuit but we know what output is when we press any button
- In python Abstraction can be Achieved using:
  - 1) ABC module (Abstract base classes)
  - 2) Using interfaces through ABC
  - 3) Using Duck Typing (Dynamic Abstraction)
  - 4) using @abstractmethod Decorator
  - 5) Abstraction using properties (@property)
  - 6) Using Encapsulation (Data hiding) also you can achieve abstraction, by making some attributes/methods as private or protected, you can hid internal details
  - 7) through modules & functions also you can Achieve Abstraction. for this create 2 .py files & you can import one file into other & directly use its classes & functions like libraries in C# or java.

## \* Abstract class

- Abstract class is a class which cannot do instantiation
- meaning you cannot create object/instance of a Abstract class
- You can create abstract as well as normal function inside abstract class, so bcz you can create concrete (normal) function, abstract class is not pure form of Abstraction
- if you create a abstract method in a abstract class then that method must be implemented (overridden) in its child class
- syntax: 

```
import ABC  
class class-name(ABC):  
    // class body  
    @abstractmethod  
    def fun-name-(s):  
        // abstract fun body
```

Abstract Base class, you need to inherit abc class to make your class as abstract

decorator which marks fun as abstract
- you need to import "ABC" module & abstractmethod from abc module as: `from abc import ABC, abstractmethod`

## \* Abstract Method

- abstract methods can only be created in Abstract class
- child class must provide implementation for all abstract methods
- Abstract methods can have default implementation, but then also these methods must be overridden in child class
- you can call the parent abstract method (if you provide default implementation) using super keyword
- if child class does not override all abstract methods then child class will itself become abstract & even that child class cannot be instantiated (create objects)
- `@abstractmethod` can be combined with `@classmethod`, `@staticmethod` or `@property`
- syntax: 

```
class abs(ABC):  
    @abstractmethod  
    def fun-1(s):  
        // abstract fun body
```

Code eg is saved in notepad:  
Abstract class python---

## \* Interface Abstraction in python (via ABC)

- python doesn't have in built interfaces like other language c++ or java
- you can achieve some what behaviour like interface, by creating a abstract class , & this abstract class should have only abstract methods with no implementation.
- So, create abstract class , which only includes abstract methods
- Interface : An interface defines what methods a class must have but does not provide any default implementation for those methods , & the subclass must implement those methods. 100% abstraction
- eg: import abc, @abstractmethod

```
class abst_1(ABC):
```

```
    @abstractmethod
```

```
    def log(self, msg):  
        pass
```

```
    @abstractmethod
```

```
    def fun2(s):  
        pass
```

// This class has only 2 fun which are abstract, so you can say it will behave like interface

// you can also called it as "pure abstract class"

```
class Notifier(ABC): // Interface bcz it also have only  
    @abstractmethod  
    def Add(s, n, m):  
        pass
```

```
class AlertsSystem(abst_1, Notifier): // class which is implementing  
    def log(self, msg):  
        print(msg)  
    def fun2(s):  
        print("Hello from fun-2")
```

```
    def Add(s, n, m)
```

```
        print("Addition is ", n+m)
```

// we are providing implementation to the abstract methods (even you can we are overriding them)

```
obj = AlertsSystem()
```

```
obj.log("Lavada")
```

```
obj.fun2()
```

```
obj.Add(10, 20)
```

## (12) ABSTRACTION USING PROPERTIES

### \* Getters & Setter Properties

- Getter: It's a method (or property of class) to read the value of a property/attribute
- Setter: It's a method used to modify the values of property or private attributes
- In python you can create getters & setters using @property decorator & @attribute-name.setter decorator
- Getter & Setter are used to show Encapsulation
- usually in Setter we write some logic to set the new value by taking the new value as parameter in setter function
- syntax :  

```
@property  
def attribute_fun(self): // This is getter  
    return attribute
```

Syntax :

```
@age.setter  
def age(self, val): // This is setter, it will  
    self._age = val set the age attribute  
which is private by  
convention
```

- you can also create normal fun like get\_name or set\_name to get & set attribute values like the old way

- eg: class Person :

```
def __init__(self, name, age)  
    self._name = name → //attribute
```

@property → //getter, which returns \_name attribute  
def name(self):  
 return self.\_name

This " " automatically calls setter  
@name.setter → //the setter name should match fun.setter name  
def name(self, val):  
 self.\_name = val

P1 = Person("vishu", 22)

P1.name = "vishwajeet" → Setter called

print(P1.name) // output = vishwajeet  
L→ Getter called

• using property also Abstraction is achieved bcz they don't know how they access attribute using getter & setter

## CH : 02 ENCAPSULATION

### #1 Encapsulation

#### \* Encapsulation

- Encapsulation means wrapping (packing up) data (variables) & methods (functions) into a single unit called classes
- मतलब en-capsul, means उक्त capsule (box) को डिटेक्यूट देना व उक्त उसी single capsule को convert करना
- Encapsulation = data hiding
- Encapsulation helps to protect data from being directly accessed or modified from outside the class.
- We use Getter & Setter property to access private data, private data should not be accessed directly
- Access Modifiers : Unlike C++, C#, Python doesn't have any strict access modifiers
  - you can access the attributes from anywhere
  - all attributes & methods are public & can be accessed publically
- Encapsulation can be achieved using :
  - Defining variables & methods inside a class
  - providing getters & setters, but still you can access the variables directly using object

#### \* Access Modifiers

- There are no Access Modifiers in Python. Everything is public, all attributes & function can be accessed everywhere
- We follow a naming convention & assume that :
 

modifier	syntax	meaning
public	var	can be accessed anywhere
protected	_var	can be accessed, but shouldn't be accessed outside the class
private	--var	cannot be accessed directly but you can access it

\* Name Mangling : Python automatically renames the attribute with " \_\_ " so that no accidental access is happened

## \* Name Mangling

- when you create attributes with " \_\_ " then, those attributes are internally re-named.
- The re-named name will have the class name added with attribute name
- So you cannot directly access that attribute using its old name. Either you can access it using getter/setter or by new re-named name

• Syntax: when you define "var" attribute like:

self.\_\_var = 10

python will automatically detect " \_\_ " & rename it to:

self.\_ClassName\_\_var

• To access this now you can do:

"obj.\_ClassName\_\_var" → // access using new name

• If you do obj.\_\_var → // gives error

Eg: class Test:

def \_\_init\_\_(self):  
 self.\_\_separate = "hidden" → automatically python will  
 rename this attribute

@property  
def separate(self): // using getter you can access &  
 return self.\_\_separate get separate using  
 getter name

@separate.setter

def separate(self, new): // you can set a new  
 self.\_\_separate = new separate using  
 setter name

obj = Test()

obj.\_\_separate = "new value" ] // gives error, you cannot  
print(obj.\_\_separate) access it like this

print(obj.separate) ] // you can access it using  
obj.separate = "lambda lesson" getter & setter

obj.\_Test\_\_separate = "Grand mara" ] // you can also access  
print(obj.\_Test\_\_separate) it using its new  
name

# CH : 10 INHERITANCE

## #1 INHERITANCE

### \* INHERITANCE

- Inheritance is a way of creating a new class from an existing class
- Using inheritance you can access all methods & data members (inherited) into another class
- हराकर एक class के सारे properties/methods को दूसरे class में inherit करना
- syntax :

```
class Base : // parent class or Base class
    // code
    class child (Base) : // child class or Derived class
        // code
```

name of class whose methods/attributes are  
to be inherited

- Child class is made and Base class is inherited into Child class
- We can use the method and attribute of "Employee" class in "Programmer" class

• eg :

```
class Employee :
    id = 251
    name = "Vishu"

class programmer (Employee) :
    age = 21
    def getInfo(self):
        print("name:", self.name, self.id, self.age)
```

E1 = programmer()

E1.getInfo() // E1 object can access data of Employee class also

Output:

name: Vishu , 251 , 21

eg:

```
class Employee:           // Base class  
    company = "Google"  
    Emp_Id = 251  
  
    def get_Emp_Info(self):  
        print (self.company, self.Emp_Id)  
  
class programmer(Employee): // derived class  
    language = "JavaScript"  
    age = 21  
  
    def get(self):  
        print (self.language, self.age, self.name)
```

p1 = programmer()

p1.name = "Vishwajeet"

p1.Emp\_Id = 99 // manipulating data of Employee class

p1.language = "Python"

Output:

p1.get\_Emp\_Info()

Google, 99

p1.get()

Python, 21, Vishwajeet

↓  
"fun of Employee  
class"

↓  
"fun of programmer class"

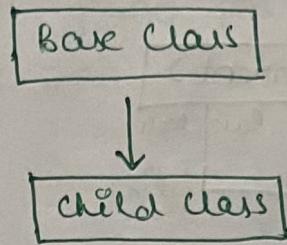
## \* Types of Inheritance

- 1) Single Inheritance
- 2) Multiple Inheritance
- 3) Multilevel Inheritance
- 4) Hybrid Inheritance

## #2 SINGLE INHERITANCE

### \* Single Inheritance

- Single Inheritance occurs when child class inherits only a single parent class



• eg:

class Dept :

    Dept\_name = "Comp"

    def getInfo (self)  
        print (Dept\_name)

class Student (Dept) :

    Dept\_name = "Ent C"

    st\_id = "202101040097"

    def getStInfo (self)

        print (self.Dept\_name, self.st\_id)

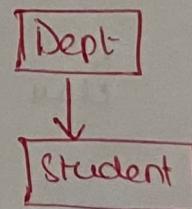
        print (self.name)

s1 = Student ()

s1.name = "Vishwajeet"

s1.getStInfo ()

s1.getInfo ()

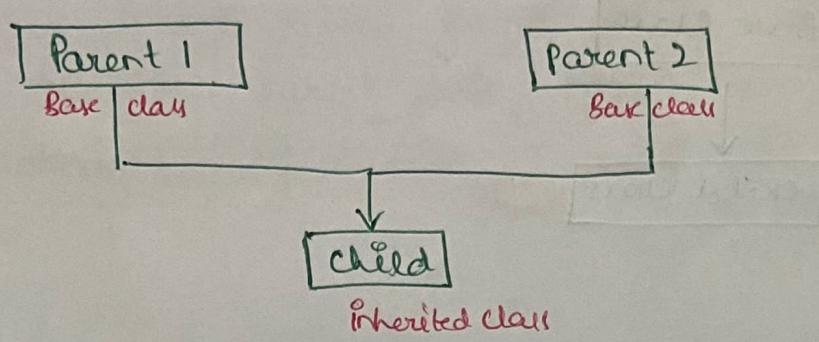


- Dept. class is base class & student class has been derived through Dept class
- all properties & methods of Dept class can be accessible by student class obj

## #2 MULTIPLE INHERITANCE

### \* MULTIPLE INHERITANCE

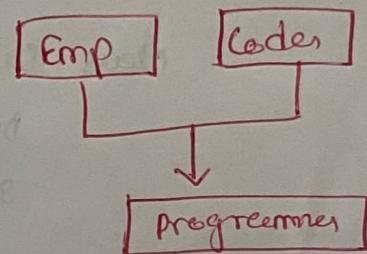
- multiple inheritance occurs when the child class inherits from 2 or more parent class



- all properties of parent1 & parent2 class will be available in child class

- eg:

```
class Emp:  
    company = "TIC"  
    def show(self):  
        print (self.company)  
        print (self.company)
```



```
class Coder:  
    language = "python"  
    def showlang(self):  
        print (self.lang)
```

```
class programmer (Emp, Coder):
```

```
    Id = "123"  
    def showInfo (self):  
        print (self.Id, self.name)  
        self.language = "C++"  
        print (self.lang, self.company)
```

```
p1 = programmer()
```

```
p1.name = "Vikas"
```

```
p1.showInfo()
```

```
p1.showlang()
```

## \* Diamond problem

- There is no ~~default~~ diamond problem occurring in Python, even though multiple inheritance is allowed. Then also diamond problem doesn't occur in Python, because of MRO.
- In all other languages, diamond problem occurs, so we don't have multiple inheritance in C++, Java, But Python is different.
- Python avoids diamond problem using a well defined Method Resolution Order (MRO) implemented by the C3 linearization algorithm.
- Classes use ~~super()~~ cooperative super() method, because of which each base class's method (e.g. \_\_init\_\_) is called only once in a deterministic order, so you don't end up with duplicates copies of data from the same common ancestor.

e.g.: Bad Approach, without MRO, direct base class calls

class A:

```
def __init__(self):  
    print("A.__init__ called")
```

class B(A):

```
def __init__(self):  
    A.__init__(self) // direct call to A constructor  
    print("B.__init__ called")
```

class C(A):

```
def __init__(self):  
    A.__init__(self) // direct call to A constructor  
    print("C.__init__ called")
```

class D(B, C):

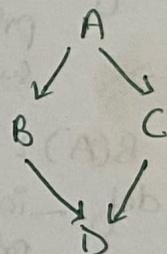
```
def __init__(self):  
    B.__init__(self)  
    C.__init__(self) ] // direct call to B & C constructors, now  
                      // each B & C constructor will call  
                      // A constructor so there will be 2  
                      // calls to A constructor so duplicate data
```

d = D() // This will call  
 // D class constructor

// Output :

```
A.__init__ called  
B.__init__ called  
A.__init__ called  
C.__init__ called
```

] // you can see that A constructor  
 // runs twice ∴ it's bad practice  
 // so always use super() keyword  
 // to call parent class method



## \* Solution for diamond Problem

- Python follows MRO to avoid diamond problem
- MRO : Method resolution Order is a order in which it should call parent class methods (or attributes), in a multiple inheritance situation
- MRO gives order that the compiler follows to call parent class methods to avoid diamond problem
- MRO is not, but it tells the order in which the inheritance has been done, & in what order you need to call the parents method.
- you can print the MRO of a class using : `className.mro()`  
& in that order it calls the parent class methods
- you need to make use of super() keyword to call parent class methods bcoz super() follow MRO order
- eg: ~~Correct~~ Correct approach using : `super().__init__(*args, **kwargs)`

class A :

```
def __init__(self, *args, **kwargs):  
    print("A.__init__ called")  
    super().__init__(*args, **kwargs)
```

class B(A) :

```
def __init__(self, *args, **kwargs):  
    print("B.__init__ called")  
    super().__init__(*args, **kwargs)
```

class C(A) :

```
def __init__(self, *args, **kwargs):  
    print("C.__init__ called")  
    super().__init__(*args, **kwargs)
```

class D(B, C) :

```
def __init__(self, *args, **kwargs):  
    print("D.__init__ called")  
    super().__init__(*args, **kwargs)
```

as we are using  
super() to call parent  
constructor, bcoz  
of which, there  
is only one call  
to A's constructor  
bcoz super() follows  
MRO order

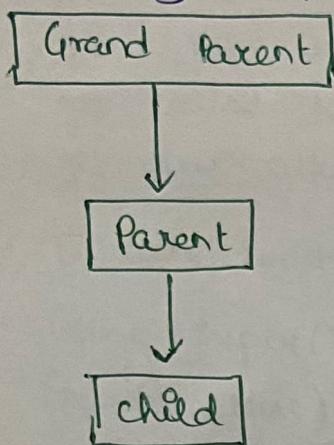
d = D()  
print(D.mro()) → D.mro() will print a list in which it calls parent fun

/output : D.\_\_init\_\_ called  
B.\_\_init\_\_ called  
C.\_\_init\_\_ called  
A.\_\_init\_\_ called

## #4 MULTI-LEVEL INHERITANCE

### \* multilevel inheritance

- When a child class becomes a parent for another child class
- It means when grand parent class makes parent class & again parent class makes child class

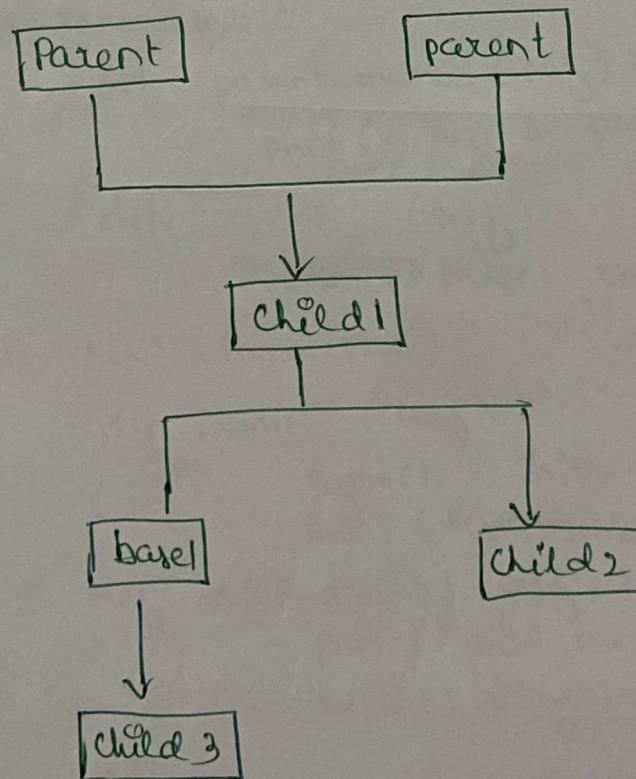


- child: child will have all properties & methods of both parent as well as grand parent class
- parent: will have all properties & methods of grand parent class

## #5 HYBRID INHERITANCE

### \* Hybrid inheritance

- Hybrid inheritance is a mixture of any two type or more type of inheritance



## CH : || MORE ON OOP,

## #1 Super () method

\*Super ()

- The super() keyword in python is used to refer to the parent class
  - We want that when we make a obj of child class then the constructor of child class will automatically get called but constructor of parent class will not get automatically called
  - To call constructor of parent class , we make use of super()
  - Using super(), you can access the methods of super class (parent class) in the derived class
  - You can use super keyword to call methods of parent class into child class
  - Syntax :

`Super().__init__()` → // for constructor

`super().parentmethod_name()` // for any other method

• eg :

```
class ParentClass :
```

```
def parent_method(self):  
    print("This is parent method")
```

```
def __init__(self):  
    print("parent constructor")
```

class child (ParentClass) :

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

super().\_\_init\_\_()

print ("Constructor of child")

def child\_method(self):

```
print ("child metho'")
```

`super().parent_method()`

his calls parent  
constructor

Obj = child()

obj. child-method ()

this calls  
parent-method  
function

## CH: 04 POLYMORPHISM

### #1 Polymorphism

- \* Polymorphism means many forms.
- \* One thing which has many forms
- \* polymorphism allows same function name to behave differently depending on the parameters & data type .
- \* ∵ polymorphism means one name , multiple behaviours depending upon the situation
- \* you can achieve polymorphism using
  - 1) method overriding : child class overrides parent method, (no abstract class)
  - 2) method overloading : giving same name to methods & accepting diff types & diff no. of parameter using \*args & \*\*kwargs
  - 3) operator Overloading : Same operator like + , you can use it to add fractions etc
- \* Python supports polymorphism ~~as~~ naturally bcoz of its dynamic typed language behaviour

### \* Types of Polymorphism

- 1) Compile time
  - \* As python is dynamically typed language so it doesn't support true compile time polymorphism like C++ & Java.
  - \* This can be Achieved using function overloading (sending diff parameters to same name functions) using \*args & \*\*kwargs
  - \* Using operator overloading also you can achieve
- 2) Run time
  - \* Run time polymorphism is achieved through method overriding by inheritance
  - \* child class will provide a new definition to parent class method . ∵ it will override parent class method

## \* method overloading

- Python doesn't support method overloading like C++ & Java.
- In python you cannot define multiple functions with the same name in the same scope, even with diff type & no. of parameters
- if you try to do so, then last definition of that function will only work bcz last definition overrides all previous ones
- eg:// Error code

```
class calculator:
```

```
    def add(self, a, b):  
        return a+b
```

```
    def add(self, a, b, c) → // only this add function  
        return a+b+c  
        will work.
```

```
obj = calculator()
```

```
print(obj.add(10, 20, 30)) // This works bcz of 3 parameters
```

X error  
ans = obj.add(100, 200)  
print(ans)

Gives Error bcz add with  
3 no will over ride this

Gives  
Error

- Python is dynamically typed language & interpreted at runtime so when you define second function with same name, it replaces the old function object in memory with new one

- \*args : you can use tuple to achieve same behaviour

```
eg: class calculator:
```

```
    def add(self, *args):
```

```
        if len(args) == 2:
```

```
            return args[0] + args[1]
```

```
        elif len(args) == 3:
```

```
            return args[0] + args[1] + args[2]
```

```
    else: pass
```

```
obj = calculator()
```

```
obj.add(*[1, 2])
```

```
obj.add(*[1, 2, 3])
```

method overloading using \*args

## \* Operator Overloading

- In python operators like "+", "-", "\*" are implemented using special methods like : `__add__`, `__sub__`, `__mul__` etc
- So, you can overload these operators using these methods

Eg: class Book

```
def __init__(self, pages):  
    self.pages = pages  
  
def __add__(self, b2):  
    return self.pages + b2.pages  
  
b1 = Book(100)  
b2 = Book(200)  
total_page = b1 + b2 // automatically "+" calls this & does the addition  
print(total_page)
```

+ operator will add books class

## \* Method Overriding

- When child class has a method with same name as parent class, then child class methods overrides parent class methods at runtime

Eg: class Animal:

```
def sound(s):  
    print("Animal makes sound")
```

class Dog(Animal): → // Sound fun is being overridden

```
def sound(s):  
    print("Dog is barking")
```

a = Animal()

d = Dog()

a.sound() → // output: Animal makes sound

d.sound() → // → Dog is barking

• Calling using :

1) Parent object : Then parent method is called

2) Child object : Then child method is called

## (#2) DUCK TYPING

### \* Duck Typing

- you can achieve polymorphism using concept of Duck Typing
- Duck Typing means: if it looks like a duck, swim & quacks like a duck, then probably it is a duck.
- In Duck Typing, type of an object is determined by its behaviour (methods & properties), not by its actual class type
- Python focus on what the object can do, rather what type of class the object is
- Duck Typing helps to achieve polymorphism without inheritance

Eg: class Duck:

```
def quack(s):  
    print("Quack Quack!")
```

class Person:

```
def quack(s):  
    print("I'm quacking like Duck")
```

```
def make_it_quack(duck_like_obj):  
    duck_like_obj.quack()
```

→ it will take any class obj which has quack() method defined

d = Duck()  
p = Person()

→ in this function calling, Type is checked at runtime & runtime polymorphism is achieved

make\_it\_quack(p) // output: I'm quacking like Duck  
make\_it\_quack(d) // output: quack quack

Eg: real life eg using file logger or msg logger

class FileLogger:

```
def write(s, msg):  
    print(f"writing in file: {msg}")
```

class ConsoleLogger:

```
def write(s, msg):  
    print(f"writing on console: {msg}")
```

def log\_msg(logger):

```
logger.write("This is a msg")
```

f = FileLogger

c = ConsoleLogger

log\_msg(f) // output: writing in file

log\_msg(c) // output: writing on console

// This works same as interface like we send any child class in interface function & that particular function gets executed like in C++ in Java

## \* Dynamic Typing

- Dynamic Typing: means you don't need to declare variables types explicitly, the type is decided at runtime based on the value assigned to it
- As Python is dynamically typed language, you can change the variables type at any time, during run time also eg:  
`x = 10 // int`  
`x = "Hello" // string`  
`x = 3.14 // float`

• same variable x can hold diff data types at diff times

- In python variables are just names (references) pointing to objects memory location
- So, the object has the type, not the variable
- When you reassign a variable, Python simply make it point to a new object of diff type