CH : 09 OOPs

(#i) 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: Enmies in game
- We will have classes & object using which you can create fun() & its data
- This way of writting 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 में drone के features add कर दिये
    5) polymorphism : one thing having many forms

#12 CLASS - OBJECT

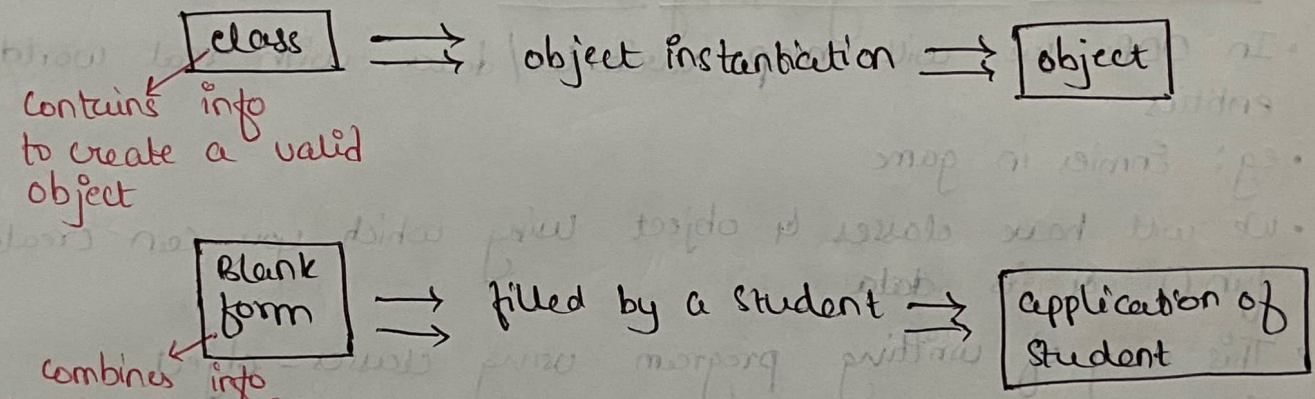
* class

- class is a blueprint or template for creating objects
- It defines the properties & method 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 classname :
    //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 change //←

```
vishu.language = "JavaScript"  // instance attribute take prefrence over class attribute during assignment & retrieval
print(vishu.language)
```

Output:
Vishwajeet , 12,000
JavaScript

---

eg:

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

class method ←

```
    def get(self):   // self means the object on which this method is called
        print("The language is",
              self.language, "of user",
              self.name)
```

object is passed and that object is self

```
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
इसे हर class method में self देना पड़ता है नहीं तो argument error आएगा।

Output:

The language is C++ of user Vishwajeet

* class Attributes

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

```
class Emp:
    Company = "Google" // class attribute

vishu = Emp() // class's object or instance of class
vishu.company = "facebook" // object can change the
                              data 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:
    Compone = "Google" // if you don't change 'Compone'
                          value then this default value
                          will be taken as "Google"
vishu = Emp()

vishu.age = 22          ] // object attributes
vishu.sal = 11,000      ]
vishu.compone = "Facebook"
print (vishu.age, vishu.compone)
```

Output:
22. Facebook

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

~~#3 Attributes/data members, Data Functions/FUNCTIONS~~

#13 ATTRIBUTES & FUNCTIONS

* Class Attributes

- Attributes: are noth but properties or data members 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 ~~ob~~ attribute copy is shared among all objects/instance
- 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 :. memory efficient becoz address is there 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. & 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 parameter inside every class
    def __init__(self, color):
        self.color = color // Instance attribute unique per object
        Car.brand = "Bmw" // class attributed 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 attributed created using method inside class

// Creating class attribute from outside
class Updater:

    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 = "Dic Breaks" // Creating new class attribute using another class function

Car.country = "Germany"

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

c2 = Car("Pink") // This class object will internally look like:
                 c2 = Car(c2, "Pink")
                 ↳ This c2 is internally pass which is noth but object & so we need self parameter inside functions & Constructor

c1.set_speed(250)

u = Updater() ——> Creating & modifying Car class attribute
u.update_brand("Tesala")

//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)                     // Dics 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 data variables 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 from another class also. using built in function setattr(obj, att, val)
- setattr(obj, attribute_name, val)
  - obj → pass the class object
  - attribute_name → pass the attribute you want to create
  - val → 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 C#

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 ~~init~~ inside another class using setattr function

u = Updater()
u.Update(c1, manfacturing_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")
```

- इस "@staticmethod" fun को mark करने से क्या होता है की fun self नहीं लेता, self का या class का कोई 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()
```

(42) 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
- मतलब जो हम पहला argument भेजेंगे तो (self) instance/obj नही होगा, तो class होगा as cls
- हम कोई भी name variable use कर सकते है self, cls या कुछ और भी
- syntax:

```
@classmethod
def fun_name(cls, p1, p2):
    #other code
```

- 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 -> it is self
        cls.company = newCompany

e1 = Emp()
e1.name = "vishu"
e1.show()
e1.change("microsoft")
e1.show()
print(Emp.company) ----> apple
print(e1.company) ----> Microsoft
```

→ you can't change class variable using self bcoz self is instance or object

↳ e1.comany is obj variable

eg:

```
class emp:
    company = "apple"
    def show(self):
        print(f"name is {self.name} comp is {self.company}")

    @classmethod
    def change(cls, newCompany):
        cls.company = newCompany
```

Emp class

```
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 noth but the object which calls that functions
- 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):          # in constructor also by default first parameter is self
        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

- These methods belong to the class itself (like we have static methods for C#)
- 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 & its 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 noth 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

- These are the methods (not particulary belong to instance or class) which don't take "self" or "cls" parameter
- These are normal functions, but these are not equal to to static functions like C++
- you can call these function using ~~name~~ 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

#4) 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
- Constructor 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 then self
   - if you don't make any Constructor then compiler gives you one default Constructor

```
eg: class Car:
        def __init__(self):
            # Empty
```

   - 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

> - you can use *args & **kwargs for constructor overloading
> - you can use class methods also to achive Constructor overloading

> - 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
   - its used to initialized 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 Over loading 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 also 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 & 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 then 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
        self.age = age
    def info(self):
        print("name is", self.first_name)
        print("age is", self.age)

a = Person("vishu", 22)
a.info()
```

// data members or attributes of class created using constructor

Output:

```
name is vishu
age is 22
```

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

#5 Destructor

* Destructor

- Destructor : its 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 implict (automatic) destructor defined unlike Constructor
- if you don't define destructor, then python GC will automatically do clean up operations
- Use Case : of to 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
```

eg:

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

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

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

```
//output:
Constructor invoked
Destructor invoked
End of program
```

gc.collect() this function will call gc & GC will call destructor, its optional to call GC.Collect()

- Destructor gets automatically called when :
  1) when refrence count becomes zero, means when no variable references that object anymore then its Refrence count become 0
  2) when object goes out of scope : means when a function finishes its Execution that time destructor comes & 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 neskesary features to the user.
- In simple words, user just knows what the function does rather then 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 achive abstraction, By making some attributes/methods as private or protected, you can hid internal details
  7) through modules & functions also you can Achive 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 becoz 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 overriden 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 to instantiation (create objects
- @abstract method 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 achive 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):          // interface2 bcoz it also have only abstract function
    @abstractmethod
    def Add (s, n, m):
        pass

class AlertSystem (abst_1, Notifier):          // class which is implementing the interface (pure abstract class)
    def log (self, msg):
        print (msg)

    def fun2 (s):
        print ("Hell 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 then

```
obj = AlertSystem()
obj.log ("lavada")
obj.fun2()
obj. Add (10, 20)
```

40 ABSTRACTION USING PROPERTIES

* Getters & Setter Properties

- **Getter** : its a method (or property of class) to read the value of a property/attribute
- **Setter** : its 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 set the age attribute which is private by convention
    self._age = val
```

- 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

    @name.setter ----> //the setter name should match fun.setter name
    def name (self, val):
        self._name = val
```

This "=" automatically calls setter

```
P1 = Person ("vishu", 22)
P1.name = "vishwajeet" ----> setter called
print (P1.name)    // output := vishwajeet
                   -> Getter called
```

- using property also Abstraction is achieved bcoz 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 एक capsule (box) के अंदर data को डाल देना & एक उसे single capsule में 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 :
  1) Defining variables & methods inside a class
  2) 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 & assum that :

modifier	syntax	meaning
public	var	can be accessed anywher
protected	_var	can be accessed, but should'nt 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 happend

* Name Mangling

- when you create attributes with "__" then, thoes 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__(s):                          → automatically python will
        s.__secarate = "hidden"                 rename this attribute

    @property
    def secrate(self):          // using getter you can access & get secarate using
        return self.__secarate      getter name

    @secrate.setter
    def secrate(self, new):     // you can set a new
        self.__secarate = new       secarate using
                                    setter name

obj = Test()

obj.__secarate = "new vale"     ]  // Gives Error, you cannot
print(obj.__secarate)           ]     access it like this

print(obj.secrate)              ]  // you can access it using
obj.secrate = "lavada lasson"   ]     getter & setter

obj._Test__secarate = "Gand mara"  ]  // you can also access
print(obj._Test__secarate)         ]     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 (inherite) into another class
- मतलब एक class के सारे properties/methods को दुसरे class मे 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 "Programer" class
- eg :

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

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

E1 = programer ()
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 programer(Employee):          // derived class
    language = "JavaScript"
    age = 21
    def get(self):
        print(self.language, self.age, self.name)

p1 = programer()
p1.name = "Vishwajeet"
p1.Emp_Id = 99          // manupulating data of Employee class
p1.language = "Python"
p1.get_Emp_Info()
p1.get()
```

output:

Google, 99

Python, 21, Vishwajeet

// fun of Employee class

// fun of programmer class

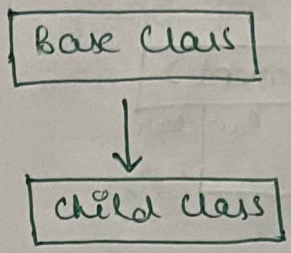
* Types of Inheritence

1) Single Inheritence
2) multiple Inheritence
3) multilevel Inheritence
4) Hybrid Inheritence

#12 SINGLE INHERITANCE

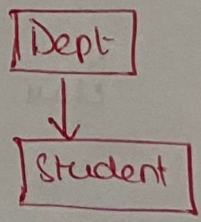
* Single inheritance

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



- eg:

```
class Dept:
    Dept_name = "Comp"
    def getInfo(self)
        print(self.dept_name)

class Student(Dept):
    Dept_name = "EntC"
    st_id = "2021010400097"
    def getStInfo(self)
        print(self.Dept_name, self.st_id)
        Print(self.name)

S1 = Student()
S1.name = "Vishwajeet"
S1.getStInfo()
S1.getInfo()
```

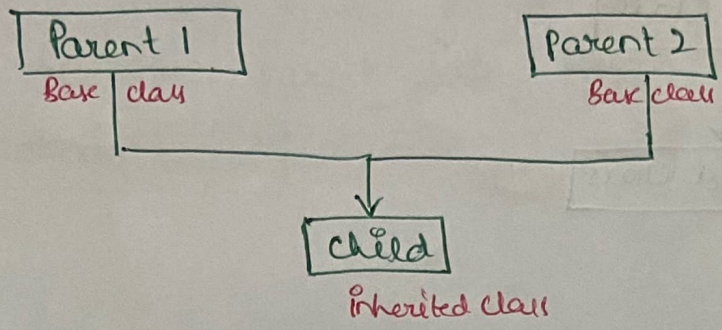


- Dept class is base class & student class has been derived through Dept class
- all properties & methods of Dept class can be accessible by student class obj

#3 MULTIPLE INHERITANCE

* MULTIPLE INHERITANCE

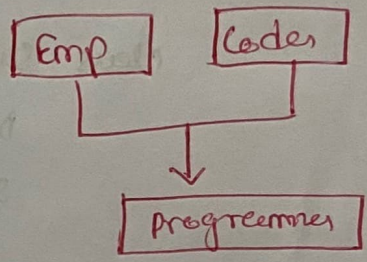
- multiple inheritance occurs when the child class inherits from 2 or more parent class



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

- eg:

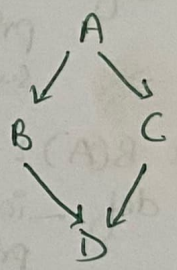
```
class Emp:
    company = "TIC"
    def show(self):
        print(self.company)
        print(self.company)

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 = "Usher"
P1.showInfo()
P1.showlang()
```



* Diamond problem

- There is no ~~dfod~~ diamond problem occuring in python, Even though multiple inheritance is allowed then also diamond problem doesn't occur in python. bcoz of MRO
- In all other languages, diamond problem occurs, so we don't have multiple inheritance in C#, Java, But Python is different
- python avoide diamond problem using a well defined Method Resolution Order (MRO) implemented by the C3 linearization algorithm.
- classes use ~~super()~~ cooperative super() meathod, becoz of which each base class's method (eg. __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.

- eg: Bad Approach, without MRO, dicreact base class calls




```
class A:
    def __init__(s):
        print("A.__init__ called")

class B(A):
    def __init__(s):
        A.__init__(s) // direct call to A Constructor
        print("B.__init__ called")

class C(A):
    def __init__(s):
        A.__init__(s). // direct call to A Constructor
        print("C.__init__ called")

class D(B, C):
    def __init__(s):
        B.__init__(s)
        C.__init__(s)
```

// direct call to B & C constructors, now each B & C Constructor will call A Constructor so there will be 2 calls to A Constructor ∴ 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 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 ordere 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 noth, 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 beoz super() follow MRO order
- eg: Correct approach using : super().__intit__(*args, **kwargs)

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

class B(A):
    def __init__(s, *args, **kwargs):
        print("B.__init__ called")
        super().__init__(*args *kwargs)

class C(A):
    def __init__(s, *args, **kwargs):
        print("C.__init__ called")
        super().__init__(* args, **kwargs)

class D(B, C):
    def __init__(s, *args, **kwargs):
        print("D.__init__ called")
        super().__init__(*args, **kwargs)

d = D()
print(D.mro())
```

As we are using super() to call parent Constructor, Beoz of which, there is only one call to A's Constructor beoz super() follows MRO order

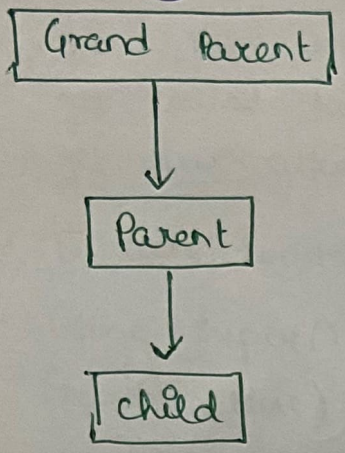
D.mro() will print a list in which it calls parent fun

```
//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

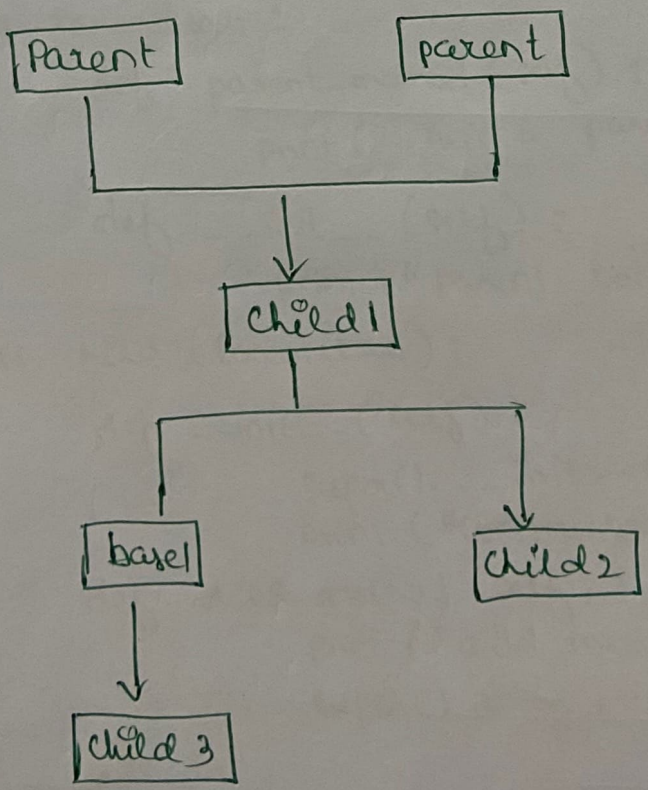


- child: child will have all properties & methods of both parent as well as grand parent class
- parent: will have all properties & methods of grand parent class

#5 HYBRID INHERITANCE

* Hybrid inheritance

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



CH : 11 MORE ON OOPs

#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__()          → This calls parent constructor
        print("Constructor of child")
    def child_method(self):
        print("child metho")
        super().parent_method()     ⟶ this calls parent_method function
obj = child()
obj.child_method()
```

CH: 04 POLYMORPHISM

#1 POLYMORPHISM

*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 achive 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 ~~dynac~~ naturally becoz of its dymaic typed language behaviour

* Types of Polymorphism

1) Compile time
- As python is dynamically typed language so it doesint 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 achive

2) Run time
- Run time polymorphism is achived through method overriding & inheritance
- child class will provide a new defination to parent class method. ∴ it will overide 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 defination of that function will only work beoz last defination overrides all previous once
- eg: // Error Code

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

obj = Calculator()
print(obj.add(10, 20, 30)) // This works beoz of 3 parameters

ans = obj.add(100, 200)   ] Gives Error beoz add with 3 no will over ride this
print(ans)                ] Gives Error
```

✗ 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 achive 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] + arg[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 fun & 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 overriden
    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

#12) DUCK TYPING

* Duck Typing

- you can Achieve Polymorphism using concept of Duck Typying
- Duck Typying means : if it looks like a duck, swim & quacks like a duck, then probably it is a duck.
- In Duck Typying, 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 typying helps to achive polymorphism without inheritance

```
eg: class Duck:
        def quack(s):
            print("Quack Quack!")

    class Person:
        def quack(s)
            print("Im quacking like Duck")

    def make_it_quack(duck_like_obj):
        duck_like_obj.quack()

    d = Duck()
    p = Person()
    make_it_quack(p)
    make_it_quack(d)
```

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

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

// output: Im quacking like Duck

// 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)
log_msg(C)
```

// output: writting in file

// output: writting on Console

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

* Dynamic Typing

- Dynamic Typing: means you don't need to decalar 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 (refrences) 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

#05 : ADVANCE OOP

#1) Decorators

* Decorators (@)

- A decorator is a function that adds extra features to another functions without changing its original code
- decorator : It a wrap up, which adds some extra features to the exsisting function
- There are many in-built decorators present & Even you can create your own Custom decorators
- Syntax :

```
def my_decorator(fun):
    def wrapper_fun():
        print("Befor function runs")
        fun()
        print("after function runs")
    return wrapper_fun
```

eg:

```
def say_hello():
    print("Good morning, hello")

def my_decorator(fn):
    def wrap_fun():
        print("lavada ah moring")
        fn()
        print("nikal lavade")
    return wrap_fun

@my_decorator // Creating function with decorator
def greet():
    print("Good night")

# my_decorator(greet)()   -> // This is same as calling it directly with decorator as greet()
greet()  -> // output : lavada ah moring
                        Good night
                        nikal lavade
say_hello()
    -> // output : Good moring, hello
```

- decorators are always applied using "@" syntax
- decorators: is a function which takes another function as an argument & returns a new function that modifies the behaviour of og function
  - decorators are noth but sugar coating
  - decorators are syntatic sugar

* function with arguments

- when function has parameters & then you want to apply decorators then :- your decorators inner wrapper() function must also take those parameters :-
- usually this wrapper() fun will take *args & **kwargs parameters, which allows the decorator to accept any no. of parameters.

```
eg: def my_decorator (fun):
        def wrap_fun (*args, **kwargs):
            print ("Before function runs")
            res = fun (*args, **kwargs) // call OG function
            print ("after function runs")
            return res
        return wrap_fun

    @my_decorator
    def add (a, b)
        return a+b

    add (10, 5)
    print (add (10,5))   ] // normal calling

        or               ↕ both will give same result

    ans = my_decorator (add)(20, 5) // calling directly
    print (ans)                        the decorator
```

* dir()

- **dir()**: This is a function, which takes a parameter & list all the function you can use on that parameter
  - like if you do dir(list_obj), then all the methods & ~~add~~ attributes of this "list_obj" will be listed

eg:
```
x = [1, 2, 3]
print (dir (x))          //output: it will list all functions & ~~add~~ variables/attributes of the list
print (x.__add__)
```
↳ output: it will tell what exactly __add__ is

- ∴ dir() function returns a list of all the attributes & methods (including dunder methods) of that object

* __dict__

- __dict__ : is an attribute, which returns a dictionary which includes all the parameters of that object
- it will return all parameters that objects takes

eg:
```
class Person:
    def __init__ (s, name, age):
        s.name = name
        s.age = age
P = Person ("vishu", 23)
print (P.__dict__)      //output: all the parameters which the object "P" takes
```

* help()

- help(): This method will give documentation for an object which helps you to use it

eg: print (help (Person)) //output: info about person class

#2 Magic / Dunder Methods

*Dunder Methods

- These are the methods with "__" & also called magic method
- These magic methods are defined under class
- The __init__() method is a dunder method, we called it as Constructor
- Some common eg of dunder methods are : __len__(), __init__(), __str__(), __repr__ etc
- you can call these methods directly ~~tt~~ without using underscrow
- for doing operator overloading also, you need to use __add__(), which is a dunder method for addition

eg:
```
class Employee
    def __init__(self, name):
        print("Constructor called")
        self.name = name

    def __len__(self):
        print("called when you do len(obj)")
        i = 0
        for c in self.name
            i++
        return i

    def __str__(self):
        return f"The name of emp is {self.name}"

    def __repr__(self):
        return f"in absence of str"

    def __call__(self):
        print("Hey you called obj like obj()")

e = Employee("vishwaject") // Constructor is automatically called

print(len(e)) // calls __len__()
print(str(e)) // calls __str__() & if its not present the calls repr
print(repr(e)) // calls __repr__()
e() // It will call __call__() method
```

#13 ITERABLE , ITERATOR , ITERATION

* ITERABLE

- ये एक ऐसा object होता है, जिसमे __iter__() या फिर __getitem__, method जो है तो define होता है
- इसका मतलब ये एक ऐसा object है जो हमे "iterator" दे सकता है, जो की iterate करने मे help करेगा।

Syntax : i = iter(object)

- iter() : this function returns an iterator, makes no as iterable
- EG OF ITERABLE :
  1) if u take a integer no, they are not iterable, so uh cannot iterate directly on the no. using for loop
  2) string : These are iterable, so u can iterate over the letters one by one, using for loop

eg: Code :

```
num = 345
for i in num
    print (i) // you will get Error
```

```
s = "vishu"
for i in s
    print (i) // you will get "vishu"
```

eg: Code using iter()

```
num = "vishwajeet"
x = 1234
a = iter(x)
print (next(a))   //output : 1
print (next(a))              2
print (next(a))              3
```

* ITRRATOR : ये एक ऐसा object है, जिसमे next(), method define होता है, मतलब हम अगले element पर जा सकते है ये next() method use करके

- you can go to next element using iterator next() funtion
- eg: iter() // This is iterator function

* ITERATION

- किसी भी element को iterate करना & उसका next element को fetch करना, उसको कहते है iteration.

* Generator function

- generators are iterators, एक asa iterator होता है जिसे हम use करके एक value generate kr सकते है & उसको traverse कर सकते है

* Comprihension

- Comprihension: means short form of writing something
- List comprihension :

Code : without Comprihension

```
L1 = [1, 5, 6, 7, 9, 12, 20]
divide_by_3 = [ ]
for item in L1:
    if item % 3 == 0:
        divide_by_3.append (item)

print (divide_by_3)
```

Code : with Comprihension

```
L1 = [1, 5, 6, 7, 9, 12, 20]
for
L2 = [a for a in L1 if a%.3 == 0]
print (L2)
```

↳ using this within a line only you can add only those element which are divided by 3 will get add in L2 List

- There is :
  1) List Comprihension
  2) Dictionary "
  3) Set "
  4) genrator "