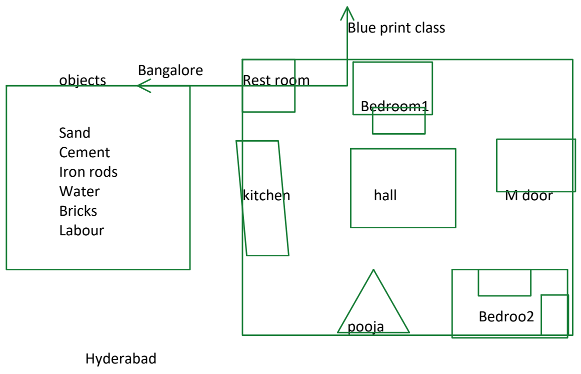
Object Oriented Programing:

19 January 2026 17:13

Procedural -->
a=10
b=20
Sum=a+b
Print(sum)

Functional
Code reusability
Recursive

Object orients



Class : a class is a blue print/template to create abject ..
Syntax: class ClassName :
#attributes(variables)
#methods(functions)
Object : instance of a class (data--> variables)
Behaviours --> methods

Syntax: --> objectname=Classname()

How can access attribute and methods?

Objectname.attribute name
Objectname.methodaname()

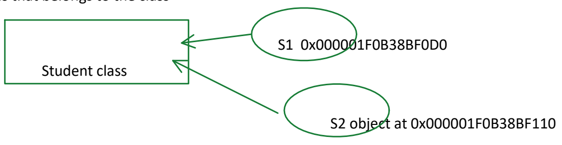
__init__ function:

Constructor :
All classes have a function called __init__(), which is always executes with the object is being initiated

```
Class Student :
    Def __init __(self,fullnames):
        Self.name=fullname

S1=Student("karan")
Print(s1.name)
```

Self parameter is a reference to the current instance of a class(object), and is used to access variables that belongs to the class



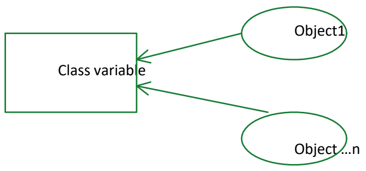
Types of variables:

1. Instance variables
2. Class variable
3. Static variables

Instance variable :- instance variables are object-specific variables created for each object
Each object can get its own separate copy

```
class Student:
    def __init__(self,name,marks):
        self.name=name #instance variables
        self.marks=marks #instance varibale
s1=Student("Murali",90)
s2=Student("ramu",85)
print(s1.name,s2.name)
```

Class variable :
Class variables are variables that are shared by all objects of the class
It is only one copy ,stored at class level, not in each object
We can able to change class variable at anytime , that can consider latest value , and that can be reflected with all object



```
    comapny="xyz" #class variabel
    def __init__(self,emp_name,emp_id,address,salary,position):
        self.emp_name = emp_name
        self.emp_id = emp_id
        self.address = address
        self.salary = salary
        self.position = position
s1 = employees("Ritz",101,"Salem,India",45000,"Developer") #xyz
s2 = employees("Raki",102,"chennai,India",25000,"Tech support")
s3 = employees("sandy",103,"Hydrabad,India",65000,"cyber security")
s4 = employees("Asma",104,"Delhi,India",35000,"Data Analyst")
s5 = employees("shri",105,"kerala,India",15000,"UI Designer")
employees.comapny="wipro technologies"
```

```
print(s1.emp_name,s1.emp_id,s1.address,s1.salary,s1.position,s1.comapny)
print(s2.emp_name,s2.emp_id,s2.address,s2.salary,s2.position,s2.comapny)
```

Static variable:-in python , static variables are simple class variable

Static variable is variables whose value remains the same for all objects and belongs to the class, not objects

In python we don't have static keyword as java,

Varibale name=static variable value

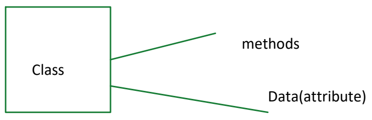
Pi=always same r*r
Tax%

Type of methods:
1.instance methods
2.class methods
3.static methods

Instance Methods: A method that work with instance(object) data.
It takes self as the first parameter

Def methodname(self)

Read/update instance variables
Most commonly used method type
Used to write business logic inside it



```
class Student:
    def __init__(self,name,marks):
        self.name=name
        self.marks=marks
    def get_avg(self):
        sum=0
        for m in self.marks:
           # sum=sum+m
            sum += m
        print(f"heloo {self.name}, your average score is: {sum/3}")
s1=Student("Kiran kumar",[99,98,97])
s1.name="iron man" # name is an attriobute we can manupulate attribute value
s1.get_avg()
```

Static methods :Method that doesnot use self parameter (works at class level)

It doesnot depending on object or class data
It is like an utility /helper function inside a class
Decorator : @staticmethod
@staticmethod
Def method_name():

Used for caluculation
Utility fuctions
Validation
Logical operations:
Ex:
@staticmethod:
Def add(a,b):
Return a+b

Class method:
A method that works with class variables .
Takes class as the first parameter
Used modify class level data.
@classmethod

Change the class variable,
Create a factory methods (alternative constructor)

```
class Student:
    college_name="University "
    def __init__(self,name,marks):
        self.name=name
        self.marks=marks
    @classmethod
    def change_cname(cls,new_name):
        cls.college_name=new_name
        print(cls.college_name)

    def welcome(self):#instance method
        print("welcome students ",self.name)
    def get_marks(self):#instance method
        return self.marks

s1=Student("vithika",97)
s1.welcome() # call the methods with respect to the object.methodname
```

```
#Student.welcome(s1)internally python can call liket his
print(s1.get_marks())
Student.change_cname("bangalore_unver")
```

```
class Employee:
    company = "TCS"      # class/static variable
    def __init__(self, name, salary):
        self.name = name        # instance variable
        self.salary = salary
    def show(self):             # instance method
        print(self.name, self.salary, Employee.company)
    @classmethod
    def change_company(cls, new):
        cls.company = new       # class method modifying class variable
    @staticmethod
    def is_eligible(salary):
        return salary > 20000   # static method (utility function)
e1 = Employee("Murali", 30000)
e2 = Employee("Kiran", 15000)
# calling instance method
e1.show()
e2.show()
# calling class method
Employee.change_company("Infosys")
# after company change
e1.show()
e2.show()
# calling static method
print(Employee.is_eligible(30000))
print(Employee.is_eligible(15000))
```

Magic/Dunder methods?

Magic methods (also called as Dunder="double undescore___)
are special methods that python calls automatically behind the scenes based on certain action
Why used:
Customize object behav
Operator overloading
Printing objects as user type
Control object creation and deletion
Improving usability of a clasees

__init__()
To initialize object variables
To execute automatically when object is created

We can assign object specific data
Avoid manual initialization

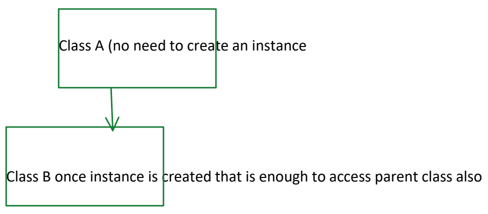
__str__() -- human readable string

Control how an object prints when we use:

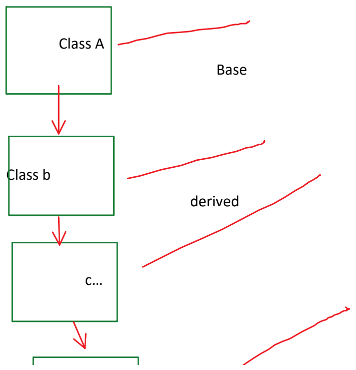
INHERITANCE :-

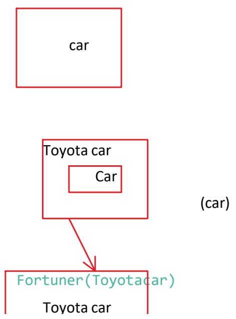
Acquiring the properties from parent class to child class
Properties:
Variables(attribute)
Methods
To avoid code duplication
Reuse parent code
Add new features in child class

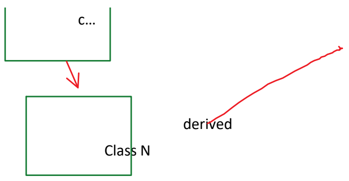
Single inheritance

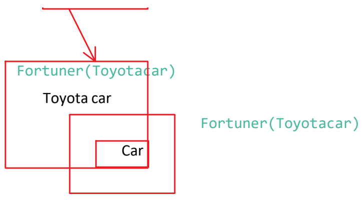

Class A (no need to create an instance
Class B once instance is created that is enough to access parent class also

Multilevel inheritance :





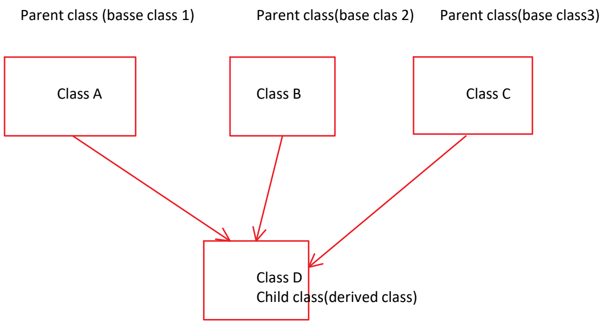




```
class Car:   #base class
    @staticmethod
    def start():
        print("the car is started")
    @staticmethod
    def stop():
        print("car is stopped")
class ToyotaCar(Car): #derived class itself and car
    def __init__(self,brand):
        self.name=brand

class Fortuner(ToyotaCar): # cotains
    def __init__(self,type):
        self.type=type

car1=Fortuner("disel")
print(car1.type)
car1.start()
car1.stop()
```


Parent class (basse class 1)
Parent class(base clas 2)
Parent class(base class3)
Class A
Class B
Class C
Class D
Child class(derived class)

Multiple Inheritance :
One class (child) is going to access properties of ay more than one class

```
class A:
    VarA="welcome to class A"

class B:
    varB="welcome to class B"
class C(A,B):
    varC="welcome class C"
C1= C()
print(C1.varC,C1.varB,C1.VarA)
```

Super() it is a method is used to access methods of the parent class

Super() = parent

```
class Car:
    def __init__(self,type):
        self.type=type
    @staticmethod
    def start():
        print("the car is started")
    @staticmethod
    def stop():
        print("car is stopped")
class ToyotaCar(Car):
    def __init__(self,name,type):
        super().__init__(type)
        self.name=name
        super().start() #Car.starte()

car1=ToyotaCar("fortuner","electic")
print(car1.type)
```

Polymorphism: many forms:
A single function/method/opeerator behaves differently depending on the object or datatype

You click "Start" on different device
Start --> car --> engine run
Start--> laptop--> Os boot run
Start -->Ac-->compressor run
Same action --> different result
This is polymorphism ..

Type of polymorphism:--

1. Duck typing
2. Method overriding (runtime polymorphism)'
3. Operator Overloading
4. Method overloading(directly not supported)

Duck typing:
If an object behaves like a duck(has required methods) , python treats it as aDuck
Python does not care about the object`s class it only cares about the method is available or not

```
class Car:
    def start(self):
        print("car is starting ...")
class Laptop:
    def start(self):
        print("Laptop is booting........")
def start_device(device):
    device.start()
```

```
car=Car()
lap=Laptop()
start_device(car)
start_device(lap)
```

Method overriding:

Child class redefines a method of [parent class with same name :
Name
Parameter
Function

```
# payment gateway:
class Payment:
    def pay(self):
        pass
class UPI(Payment):
    def pay(self):
        print("paid using UPI")
class Card(Payment):
    def pay(self):
        print("paid using credit/Debit card ")
class NetBanking(Payment):
    def pay(self):
        print("paid using Netbanking")
for method in (UPI() , Card(), NetBanking()):
    method.pay()

# n=NetBanking.pay("netbanking")
# u=UPI.pay("pay")
```

Operator overloading:
Using operators +,-,*,<,>,==

Python automatically converts operator in to a magic/dunder methods

a+b = internally a.__add__(b)

Add salaries
Comapre two students marks
Compare two products price

```
class Student:
    def __init__(self,name,marks):
        self.name=name
        self.marks=marks
    def __get__(self,other):
        return self.marks > other.marks
s1=Student("murali",97)
s2=Student("kiraana",80)
print(s1.marks > s2.marks) # print(s1 > s2)
```

Method overloading(is not directly supported)

java
Add(int a)
Add(int a, int b)

But in python does not support this format

```
#mimic method overloading bu using default arguments
class Test:
    def add(self,a,b=0,c=0):
        print(a + b + c)
t=Test()
# t.add(10)
t.add(10,20,30)
t.add(10,20,30)
```

Create a Car Rental application that manages:
- Cars
- Customers
- Rental calculations
- Vehicle types (Inheritance)
- Polymorphism (different billing rules)
- Operator Overloading (merge rental durations)
- Static methods (utilities)
- Class variables (company name, GST%)

This is a **complete OOP implementation**,