**Class:**

Everything in python is a class. Class is a blueprint of an object. We can create an object using the class. For example if we talk about list, integer, string, tuple than we can say that these are a datatype of class list, class int, class str, class tuple and the functionality(list has append, pop) are method. Using that list class we create an object.

L = [1,2,3,4] # this is object literal

L.append(3)

k = list() # this is object representation

in above example L is an object of list class and append is a method(functionality) of list class.

Class

Data or Property or Attribute

Method or Behaviour

Class SchoolStudetn(): #Pascal case representation

def \_\_init\_\_(self,name,section):

Attribute(Privet, public)

self.name = name

selef.section = section

def show\_data(self):

Method / Public

print(self.name,self.section)

s1 = SchoolStudent()

s1.show\_data(“avi”,”a”)

Object

wagonar = Car()

Dog = Animal()

Cricket = Game()

**Note :** Everything are public in python. Nothing Private.

**Constructor:**

Class constructor is a special method which is always called whenever object is created. We don’t have to call the class constructor. In java we are created a class constructor by the class name but in python there is a special ***\_\_init\_\_()*** which is used as class constructor. The attribute of the class always define in class constructor i.e inside **\_\_*init\_\_()*** method.

It is also known as special or magic or dunder method. Every method start and end with double underscore. \_\_abs\_\_, \_\_add\_\_, \_\_ ceil\_\_.

**Default Constructor:**

class Student:

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

self.name = “Avijeet”

self.age = 24

def print\_name(self):

print(self.name, self.age)

s1 = Student()

s1.print\_name()

**Parameterized Constructor:**

class Student:

def \_\_init\_\_(self,name,age):

self.name = name

self.age = age

def print\_name(self):

print(self.name, self.age)

s1 = Student()

s1.print\_name(“avijeet”,24)

**Where are we use class constructor?**

In bigger project some functionality is given to the user end but some functionality is not access by user. Those functionalities which are not access by user or developer do not want to give access to user that functionalities is written inside the class constructor and these functionality automatically called when the application open.

For example if we talk about uber than the user detail(source to destination location) enter by user but the uber is automatically connected with the internet so the automatically connected with internet functionality or assigned to the cab driver is written inside class constructor method.

Another Example, if we talk about nature the god is programmer and human beings are class. The eating, the sunbath, the breathing is in our control but the death is not in our control so the death is class constructor which is defined by programmer(the god) but not in user’s(human being) control.

**Self:**

Self represents the instance of the class means current object. By using the self we can access the attribute and methods of the class in python.It always point to the current objects I.e the object in which we are working.

class Student:

def \_\_init\_\_(self,name,age):

self.name = name

self.age = age

self.laptop()

print(id(self)

def laptop(self):

print(“laptop is …..”)

def print\_name(self):

print(self.name, self.age)

s1 = Student()

s1.print\_name(“avijeet”,24)

print(id(s1))

s2 = Student()

s2.print\_name(“bib”,23)

In above example the s1 is an object and the self points to the s1 object when we use s1.print\_name(“avijeet”,24) but for s2.print\_name(“bib”,23) the self is pointing s2 object.

The question is **why we use self?** The access of method and attribute is only by objects. Inside the class we cannot access method to access another method or use the functionalities of another method so using self we can access functionalities of another method. In above example inside constructor we use self.laptop().

Making own data type using **Magic or dunder method:**

class Fraction:

def \_\_init\_\_(self,numerator,denominator):

self.num = numerator

self.deno = denominator

def \_\_add\_\_(self,other):

temp\_num = self.num \* other.deno + self.deno \* other.num

temp\_deno = self.deno \* other.deno

return f”{temp\_num}/{temp\_deno}”

frac1 = Fraction(2,4)

frac2 = Fraction(3,5)

print(frac1+frac2)

**Instance variable:**

The variable in which the value of the variable is different for different objects.

**Make variable private:**

Some time we want to hide our instance variable because if the instance variable is not private at that time the variable is easily access from outside of the class and programmer change by mistake and the program through an error so to resolve that issue we make those variable private.

For that we use double underscore before the instance variable.

self.\_\_name = name

we can also hide or make private method to add double underscore before name.

**Example:**

if attribute is not private: **self.name = “Avijeet”**

**s1.name = “kumar”**

Change the value

if attribute is private : **self.\_\_name = “avijeet”**

**s1.\_\_name = “kumar”**

Value is not changed

Because the **\_\_name** is

another instance variable and change by

**\_Student\_\_name\_**

variable

**Note:**  we can not make attributes and methods private because python is made for adults not for child.

**Encapsulation:**

1. Need of encapsulation: Sometime we want to make private or hide our data to make sure that someone do not fetch or change data. For that we need encapsulation. Using double underscore we hide that data and using getter and setter method we have to access and change that data as well.

Example : Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when due to some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”. Using encapsulation also hides the data. In this example, the data of the sections like sales, finance, or accounts are hidden from any other section.

1. using \_\_ (double underscore) we make data private.

2. using getter and setter we can fetch and change data.

3. Class diagram:

Class(Student)

Attribute

-name(private)

+age(public)

Methods

print\_name()

Memory Management

Memory

**Reference Variable:**

Stack

Private/ Heap

s1 = Student()

Variable referencing to

The address of the

Student object

We can say the s1 is a

Variable not an object

It is reference to the

Student class.

Memory

Reference

Data

Private/ Heap

Stack

Student is actually an object

which is not inside memory

but it has address so the s1

points to that address of the Student() object.

Reference

Data

**Call by reference:**

class Student:

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

self.name = name

def greet(student):

print(id(student))

student.name =”bib”

print(student.name)

print(id(student))

st = Student()

print(id(st))

greet(st)

print(st.name)

All Id are same that indicates that in python objects are also mutable like list, sets and dictionary.

def change(L):

print(id(L))

L.append(5)

print(id(L))

L1 = [1,2,3,4]

print(id(L1))

print(L1)

change(L1)

# change(L1[:])

print(L1)

**Static Method:**

**Public**

class student:

counter = 1

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

self.name = name

self.sr\_no = student.counter

student.counter = student.counter+1

def show\_name(self):

print(self.name)

s1 = student("avi")

print(s1.sr\_no)

s2 = student("bib")

print(s2.sr\_no)

print(s1.counter)

print(s2.counter)

print(student.counter)

**Private:**

class student:

\_\_counter = 1

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

self.\_\_name = name

self.\_\_sr\_no = student.\_\_counter

student.\_\_counter = student.\_\_counter+1

def show\_name(self):

print(self.\_\_name)

@staticmethod

def get\_counter():

print(student.\_\_counter)

@staticmethod

def set\_counter(new):

if type(new) == int:

student.\_\_counter = new

else:

print(“Enter only integer value”)

s1 = student("avi")

Access by s1.\_student\_\_sr\_no

print(s1.\_\_sr\_no)

s2 = student("bib")

**Relationship between Classes:**

when we are working on a big project than we are tackle with a lots of classes so to reduce our efforts or specially reduce the code we get some functionalities from different classes. To get functionalities from different classes we make a relationship between these classes.

**Relationship**

**Aggregation**

**(Has-a)**

**Inheritance**

**(Is-a)**

Example: If we are working on a social media website than we have customer class and we have customer address class, we have product class.

**Customer**

**Address**

Customer has a address so if customer want to edit address and customer want to insert address so the properties that edit or insert address are coming from address class not all method coming from address only those which we want.

class Customer:

def \_\_init\_\_(self,name,email,address):

self.name = name

self.email = email

self.address = address

def change\_address(self,new\_city, new\_pincode, new\_state):

self.address.new\_address(new\_city, new\_pincode, new\_state)

class Address:

def \_\_init\_\_(self, city, pincode, state):

self.city = city

self.pincode = pincode

self.state = state

def new\_address(self, new\_city, new\_pincode, new\_state):

self.city = new\_city

self.pincode = new\_pincode

self.state = new\_state

add = Address(“patna”,202223,”bihar”)

cust = Customer(“avi”,”123@gmail.com”,add)

print(cust.address.city) **#patna**

cust.change\_address(“delhi”,303322,”delhi”)

print(cust.address.city) **#delhi**

**Inheritance:**

Inheritance allows you to create a hierarchy of classes that means you can share a set of properties and methods by deriving a class from another class. It has bottom up approach which means we can inherit properties from parent to child class not child to parent class. It makes your code reusable.

**Learning Website**

(Udemy)

**User**

**class User:**

Login

Registration

**Student**

**Instructor**

Login,reg are class

Inherit from

**User** class.

**class Instructor:**

Create video

Answer comm

**class Student:**

Enroll

Review

**What are we inherit?**

**Child**

1. Data Members (variables)

2. Members functionalities (Methods)

3. Constructor

Note: Private members are not inherited

**Parent**

################## Inheritance ################

class User:

def login(self):

print("Login")

def register(self):

print("Register")

class Student(User):

def enrol(self):

print("Enrol")

def review(self):

print("Review")

class Instructor(User):

def create(self):

print("Create video")

def comment(self):

print("Commented")

stu1 = Student()

stu1.login()

in1 = Instructor()

in1.login()

################# Inheriting Constructor #############

class User:

def \_\_init\_\_(self,name,email):

self.name = "kumar"

self.email = email

class Student(User):

# def \_\_init\_\_(self,name,email):

# self.name = name

# self.email = email

def register(self):

print("Registerd")

s = Student("avi","123")

print(s.name)

################# Inheriting Private data ##############

class User:

def \_\_init\_\_(self,name,email):

self.name = name

self.\_\_email = email

class Student(User):

def register(self):

print("Registerd")

s = Student("avi","123")

print(s.name)

# print(s.\_\_email)

# print(s.\_Student\_\_email)

# print(s.\_User\_\_email)

####################### Question on privet class ####################

class User:

def \_\_init\_\_(self,name,email):

self.name = name

self.\_\_email = email

def get\_email(self):

print(self.\_\_email)

class Student(User):

def register(self):

print("Registerd")

s = Student("avi","123")

print(s.name)

s.get\_email()

####################### Polymorphism Mehod overriding ##############

class User:

def \_\_init\_\_(self,name,email):

self.name = name

self.\_\_email = email

def register(self):

print("We are in User register")

class Student(User):

def register(self):

print("We are in Student register")

s = Student("avi","123")

print(s.name)

print(s.register())

####################### Super Keyword for overriding ################

class User:

def \_\_init\_\_(self,name,email):

self.name = name

self.\_\_email = email

def register(self):

print("We are in User register")

class Student(User):

def register(self):

print("We are in Student register")

super().register()

s = Student("avi","123")

print(s.name)

s.register()

####################### Super Keyword with constructor ###############

class User:

def \_\_init\_\_(self,name,email):

self.name = name

self.email = email

def register(self):

print("We are in User register")

class Student(User):

def \_\_init\_\_(self,name,email,section,college):

super().\_\_init\_\_(name,email)

self.section = section

self.college = college

def register(self):

print("We are in Student register")

super().register()

s = Student("avi","123","a","pce")

print(s.name)

s.section

####################### super method on privet class ################

class User:

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

self.\_\_name = name

def get\_name(self):

print(self.\_\_name)

class Student(User):

def \_\_init\_\_(self,name,email):

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

self.\_\_email = email

def get\_email(self):

print(self.\_\_email)

s = Student("avi","123")

s.get\_name()

s.get\_email()

**Types of Inheritance:**

1. Single Level Inheritance

Parent

2. Multi Level Inheritance

Grand Father

Parent

Child

3. Hierarchical Level Inheritance

Parent

Child2

Child1

4. Multiple Inheritance

Father

Mother

Child

5. Hybrid Inheritance

Grand Father

Father

Mother

Child

Grand Child

Method Resolution order → MRO

**Polymorphism :**

The word Polymorphism means having many Form. The Function or the method or the class behaves different for different inputs.

**Built In Polymorphic function:**

**print(**len(“Avi”))

**print(**len([1,2,3]))

**User defined polymorphic function:**

def addition(a,b,c=0):

print(a+b+c)

addition(1,2,3)

addition(1,2)

**1. Method overriding:**

When we inherit classA to classB. A method name is same in both classes in this situation method overloading occurs.

class School:

def school\_name(self):

print(“we are in same school”)

def student\_info(def):

print(“Student are in same school but different class”)

class Student1(School):

def student\_info(self):

print(“student1 information”)

class Student2(School):

def student\_info(self):

print(“student2 information)

s1 = Student1()

s1.school\_name()

s1.student\_info()

s2 = Student2()

s2.school\_name()

s2.student\_info()

**Method Overloading:**

class Geometry:

def area(self,r):

print(“radius of circle is : “, 3.14 \* r \* r)

def area(self,l,b):

print(“radius of rectangle is : “, l \* b)

obj = Geometry()

obj.area(4)

The above example throw an error because python is not support method overloading but we can perform this task using input variable.

class Geometry:

def area(self,a,b=0):

if b == 0:

print(“radius of circle is : “, 3.14 \* r \* r)

else:

print(“radius of rectangle is : “, l \* b)

obj = Geometry()

obj.area(4)

**Operator Overloading:**

Operator Overloading is a concept that we behaves an operator using our own functionalities

That “+” behave like addition but our own custom type.

class Fraction:

def \_\_init\_\_(self, numenator, denomenator):

self.num = numenator

self.den = denomenator

def \_\_str\_\_(self):

return f'{self.num}/{self.den}'

def \_\_add\_\_(self,other):

temp\_num = self.num \* other.den + self.den \* other.num

temp\_den = self.den \* other.den

return f'{temp\_num}/{temp\_den}'

def \_\_sub\_\_(self,other):

temp\_num = self.num \* other.den - self.den \* other.num

temp\_den = self.den \* other.den

return f'{temp\_num}/{temp\_den}'

def \_\_mul\_\_(self,other):

temp\_num = self.num \* other.num

temp\_den = self.den \* other.den

return f'{temp\_num}/{temp\_den}'

def \_\_truediv\_\_(self,other):

temp\_num = self.num \* other.den

temp\_den = self.den \* other.num

return f'{temp\_num}/{temp\_den}'

fraction1 = Fraction(3,4)

fraction2 = Fraction(4,5)

print(fraction1+fraction2)

print(fraction1-fraction2)

print(fraction1\*fraction2)

print(fraction1/fraction2)

In above example “+” add to fraction but in real the addition does not add two fraction.