opps-python

August 14, 2023

1 OPP's [Object Programming Language]

Every thing in Python is Object and opps programming based on the concept of "Object",

The object contain both data and code.

Data in th foam of properties (Known as attributes) and code , in the foam of methods (Actions that object can perfoamm)

For example, A Bike is an object, as it has the following properties:

1. name,price,color as attributes
2. breaking, acceleration as behaviour

OPP's use the concept of DRY (Don't Repeat Yourself)

Suppose I want to make a house , so the first thing what we are doing is to go to Architect to design and told him/ her that we want a house of this design, we want 1 hall and 4 bedroom then according to the requirements he makes a blue print and then we build a house

NOTE: Suppose you want to build another house of the same design would you go to the architech again The answer is no Becuase you already have a blueprint

In Python everything is an object . A class is a blueprint for the object To create any object we required a model or plan or blueprint which is nothing but class.

For example, you are creating a vehicle according to the Vehicle blueprint (template). The plan contains all dimensions and structure. Based on these descriptions, we can construct a car, truck, bus, or any vehicle. Here, a car, truck, bus are objects of Vehicle class

2 A class contains the properties (attribute) and action (behavior) of the object. Properties represent variables, and the methods represent actions. Hence class includes both variables and methods.

3 Object is an instance of a class. The physical existence of a class is nothing but an object. In other words, the object is an entity that has a state and behavior. It may be any real-world object like the mouse, keyboard, laptop, etc.

[46]:
```
class Classname:
    '''documentation string'''
# we we create any class we use class keyword it is good in practice if we use␣
 ↪First letter capital of class name

# Here we Create a class means create a blue print where i only write the␣
 ↪'''documentation string'''
```

Now from this class means from this blueprint i want to create real world objects

[47]:
```
Obj1 = Classname()
Obj2 = Classname()
Obj3 = Classname()

# Here i create 3 objects
```

[48]:
```
Obj1.__doc__
```

[48]:
```
'documentation string'
```

[50]:
```
class Employee:

    company_name = 'ABC Company'

    def show(self,name,salary):
        self.name=name
        self.salary=salary
        print('Employee:', self.name, self.salary, self.company_name)


emp1 = Employee()
emp1.show("Harry", 12000)


emp2 = Employee()
```

```
emp2.show("Emma", 10000)
```

```
Employee: Harry 12000 ABC Company
Employee: Emma 10000 ABC Company
```

In the above code Employee is a class and company_name is attribute show is the method:

4 Constructor in Python

In object-oriented programming, A constructor is a special method used to create and initialize an object of a class. This method is defined in the class.

1. The constructor is executed automatically at the time of object creation.
2. The primary use of a constructor is to declare and initialize data member/ instance variables of a class. The constructor contains a collection of statements (i.e., instructions) that executes at the time of object creation to initialize the attributes of an object.

For example, when we execute obj = Sample(), Python gets to know that obj is an object of class Sample and calls the constructor of that class to create an object.

Note: In Python, internally, the **new** is the method that creates the object, and **del** method is called to destroy the object when the reference count for that object becomes zero.

Object Creation is divided into 2 parts first is Object Creation and 2nd is Object Intialization.

1. Internally, the **new** is the method that creates the object.
2. Using the **init** method we can implement constructor to initialize the object.

5 def init(self):

1. def: The keyword is used to define function.
2. **init**() Method: It is a reserved method. This method gets called as soon as an object of a class is instantiated.
3. self: The first argument self refers to the current object. It binds the instance to the **init**() method. It's usually named self to follow the naming convention.

[1]:
```
class Student:
    def __init__(self,name):
        self.r=name
    def show(self):
        print("My name is :",self.r)

s1 = Student("Rohit")
s1.show()
```

```
My name is : Rohit
```

Note:

1. For every object, the constructor will be executed only once. For example, if we create four objects, the constructor is called four times.

2. In Python, every class has a constructor, but it's not required to define it explicitly. Defining constructors in class is optional.
3. Python will provide a default constructor if no constructor is defined.

6 Types of Constructor:

1. Default (Not using **init** method)
2. Non-Parametrized (def **init**(self))
3. Parametrized (def **init**(self,name,sex,age))

Note: In the upcoming codes we will breifly explain all the codes.

Default Constructor

Python will provide a default constructor if no constructor is defined. Python adds a default constructor when we do not include the constructor in the class or forget to declare it. It does not perform any task but initializes the objects. It is an empty constructor without a body.

[2]:
```
class Employee:
    def display(self):
        print("In this class we are not using __init__ method")
emp1 = Employee()
emp1.display()
```

```
In this class we are not using __init__ method
```

As you can see in the example, we do not have a constructor, but we can still create an object for the class because Python added the default constructor during a program compilation.

Non-Parametrized Constructor A constructor without Arguments is called Non-Parametrized Constructor. This type of constructor is used to intialize each object with defaut value. This constructor does't accept any arguments.

[6]:
```
class Student:
    def __init__(self):
        self.name="Dummy"
        self.rollno=4
        self.Class=5
    def info(self):
        print(f"Student name is {self.name} and roll no is {self.rollno} and he␣
 ↪is studied in class {self.Class}")
s1 = Student()
```

[7]:
```
s1.info()
```

```
Student name is Dummy and roll no is 4 and he is studied in class 5
```

Parameterized Constructor: In parameterized Constructor we define arguments and we can pass different value to each object at the time of creation: The first parameter to constructor is self that is a reference to the being constructed, and the rest of the arguments are provided by the programmer

[10]:
```
class Student:
    def __init__(self , name , rollno , Class):
        self.name=name
        self.rollno=rollno
        self.Class=Class
    def info(self):
        print(f"Student name is {self.name} and roll no is {self.rollno} and he␣
 ↪is studied in class {self.Class}")
s1 = Student("Rohit",1,10)
s2 = Student("Kaushik",2,11)
```

[11]:
```
s1.info()
```

```
Student name is Rohit and roll no is 1 and he is studied in class 10
```

[12]:
```
s2.info()
```

```
Student name is Kaushik and roll no is 2 and he is studied in class 11
```

NOTE : we can also use default values in constructor Below is the code for your understanding

[14]:
```
class Student:
    # constructor with default values age and classroom
    def __init__(self, name, age=12, classroom=7):
        self.name = name
        self.age = age
        self.classroom = classroom

    # display Student
    def show(self):
        print(self.name, self.age, self.classroom)
```

[16]:
```
kelly = Student('Kelly')
kelly.show()
```

```
Kelly 12 7
```

7 Self Keyword in Python

The self Keyword which is always the first argument always refers to current object

[30]:
```
class Student:
    # constructor
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # self points to the current object
```

```
    def show(self):
        # access instance variable using self
        print(self.name, self.age)

# creating first object
emma = Student('Rohit', 12)
emma.show()
```

```
Rohit 12
```

[31]:
```
# creating Second object
kelly = Student('Kaushik', 13)
kelly.show()
```

```
Kaushik 13
```

In simple language when we create any object then the object name takes the place of self

Python Destructors Destructor is a method that is called when the object gets destroyed. we use **del**() method

[28]:
```
class Student:

    # constructor
    def __init__(self, name):
        print('Inside Constructor')
        self.name = name
        print('Object initialized')

    def show(self):
        print('Hello, my name is', self.name)

    # destructor
    def __del__(self):
        print('Inside destructor')
        print('Object destroyed')

# create object
s1 = Student('Rohit')
s1.show()
```

```
Inside Constructor
Object initialized
Inside destructor
Object destroyed
Hello, my name is Rohit
```

[32]:
```
del s1
```

```
Inside destructor
Object destroyed
```

[34]:
```
s1.show()

# Here now the s1 is delete and when we delete s1 using del s1 __del__␣
 ↪automatically called.
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[34], line 1
----> 1 s1.show()
      3 # Here now the s1 is delete and when we delete s1 using del s1 __del__␣
 ↪automatically called.

NameError: name 's1' is not defined
```

8 In Object-oriented programming, when we design a class, we use instance variables and class variables.

1. Instance variables: If the value of a variable varies from object to object, then such variables are called instance variables.
2. Class Variables: A class variable is a variable that is declared inside of class, but outside of any instance method or **init**() method.

Now we only Focus on isinstance variable : when we define emma and kelly object in above code's you notice for emma there is different value and for kelly there is different value

For every object, a separate copy of the instance variable will be created.

Instance variables are not shared by objects. Every object has its own copy of the instance attribute. This means that for each object of a class, the instance variable value is different.

We can access the instance variable using the object and dot (.) operator

Instance variables are declared inside a method using the self keyword. We use a constructor to define and initialize the instance variables

[35]:
```
class Employee:
    # constructor
    def __init__(self, name, age):
        # Instance variable
        self.name = name
        self.age = age
E1 = Employee("A", 20)
```

[37]:
```
# Lets access the instance variable of class E1
print(E1.age)
```

```
print(E1.name)
```

```
20
A
```

[39]:
```
E2 = Employee("B", 11)
print(E2.age)
print(E2.name)
```

```
11
B
```

1.When we created an object, we passed the values to the instance variables using a constructor

2. Each object contains different values because we passed different values to a constructor to initialize the object.

We can modify the value of the instance variable and assign a new value to it using the object reference.

Note: When you change the instance variable's values of one object, the changes will not be reflected in the remaining objects because every object maintains a separate copy of the instance variable.

[42]:
```
E2.name='C'
E2.name
# Here we change the object E2 name B from C
```

[42]:
```
'C'
```

[43]:
```
# We can also Use getattr instead of E2.name
print('Name:', getattr(E2, 'name'))
```

```
Name: C
```

Dynamically Add Instance Variable to a Object

object_referance.variable_name = value

[44]:
```
E2.department = "HR"
```

[46]:
```
print('Name:', E2.name, 'Age:', E2.age, 'Marks:', E2.department)
```

```
Name: C Age: 11 Marks: HR
```

1. we cannot add an instance variable to a class from outside because instance variables belong to objects.

2. Adding an instance variable to one object will not be reflected the remaining objects because every object has a separate copy of the instance variable.

[48]:
```
# If you want to delete the instance variable of any class just use del
```

```
del E2.department
```

[50]:
```
class Student:
    def __init__(self, roll_no, name):
        # Instance variable
        self.roll_no = roll_no
        self.name = name

s1 = Student(10, 'Jessa')
print('Instance variable object has')
print(s1.__dict__)
```

```
Instance variable object has
{'roll_no': 10, 'name': 'Jessa'}
```

9 Python Class Variables

Class Variables: A class variable is a variable that is declared inside of class, but outside of any instance method or **init**() method.

value of a variable is not varied from object to object (Also called static Variable) and it is also shared by all instances of class.

Class variables are declared when a class is being constructed

[53]:
```
class Employee:
    Company = "XYZ"
    def __init__(self, Department, name):
        # Instance variable
        self.Department = Department
        self.name = name
    def show(self):
        print(f"Employee name is {self.name} and his/her department is {self.
 ↪Department} and working in {Employee.Company}")

E1 = Employee("Finance", 'Rohit')
print('Instance variable object has')
E1.show()
```

```
Instance variable object has
Employee name is Rohit and his/her department is Finance and working in XYZ
```

1. Access class variable inside instance method by using either self of class name
2. Access from outside of class by using either object reference or class name.

[54]:
```
class Employee:
    Company = "XYZ"
    def __init__(self, Department, name):
        # Instance variable
```

```
        self.Department = Department
        self.name = name
    def show(self):
        print(f"Employee name is {self.name} and his/her department is {self.
 ↪Department} and working in {self.Company}")

E1 = Employee("Finance", 'Rohit')
print('Instance variable object has')
E1.show()
```

```
Instance variable object has
Employee name is Rohit and his/her department is Finance and working in XYZ
```

[57]:
```
# Modify class Variable
Employee.Company = "Quation"
E1.show()
```

```
Employee name is Rohit and his/her department is Finance and working in Quation
```

Note:

It is best practice to use a class name to change the value of a class variable. Because if we try to change the class variable's value by using an object, a new instance variable is created for that particular object, which shadows the class variables.

10 Python Methods in Opp's

If we use isinstance variable inside a method , such methods are caleed instatnce Mathod These Instance Methods perfoams a set of action on the data/value provided by a instance variable.

1. A instance method is bounded to the object of class
2. It can change or modify the object state by changing the value of a instance variable
3. When we create a class in Python, instance methods are used regularly. To work with an instance method, we use the self keyword. We use the self keyword as the first parameter to a method. The self refers to the current object.
4. Any method we create in a class will automatically be created as an instance method unless we explicitly tell Python that it is a class or static method.

[7]:
```
class Mobile:
    def __init__(self,Ram,Rom):
        self.Ram = Ram
        self.Rom = Rom
    def show(self):
        print(f"The Ram of Mobile is {self.Ram}GB and Rom is {self.Rom}GB")
obj1 = Mobile(2,16)
obj2 = Mobile(8,128)

obj1.show()
# Here the show() is Instance Method
```

```
obj1.Rom
```

```
The Ram of Mobile is 2GB and Rom is 16GB
```

[7]: 16

11 Note:

Inside any instance method, we can use self to access any data or method that reside in our class. We are unable to access it without a self parameter.

An instance method can freely access attributes and even modify the value of attributes of an object by using the self parameter.

By Using self.___class___ attribute we can access the class attributes and change the class state. Therefore instance method gives us control of changing the object as well as the class state.

12 Modify Instance Variable inside Instance Method:

[9]:
```
class Mobile:
    def __init__(self,Ram,Rom):
        self.Ram = Ram
        self.Rom = Rom
    def show(self):
        print(f"The Ram of Mobile is {self.Ram}GB and Rom is {self.Rom}GB")
    def update(self,Ram,Rom):
        self.Ram = Ram
        self.Rom = Rom
obj1 = Mobile(2,16)
obj1.show()
```

```
The Ram of Mobile is 2GB and Rom is 16GB
```

[11]:
```
obj1.update(4,64)
obj1.show()

# Let's create the instance method update() method to modify the Mobile Ram and␣
 ↪Rom /.
```

```
The Ram of Mobile is 4GB and Rom is 64GB
```

[14]:
```
class Mobile:
    def __init__(self,Ram,Rom):
        self.Ram = Ram
        self.Rom = Rom
    def show(self,camera):
        self.camera=camera
```

```
        print(f"The Ram of Mobile is {self.Ram}GB and Rom is {self.Rom}GB and␣
 ↪camera is {self.camera} Pixel")
obj1 = Mobile(2,16)
obj1.show(16)
```

```
The Ram of Mobile is 2GB and Rom is 16GB and camera is 16 Pixel
```

13 Dynamically Add Instance Method to a Object

Usually, we add methods to a class body when defining a class. However, Python is a dynamic language that allows us to add or delete instance methods at runtime. Therefore, it is helpful in the following scenarios.

1. When class is in a different file, and you don't have access to modify the class structure
2. You wanted to extend the class functionality without changing its basic structure because many systems use the same structure.

[22]:
```
import types
class Student:
    def __init__(self,name,age):
        self.name=name
        self.age=age

    def info(self,sex):
        self.sex=sex
        print("name :", self.name , "Age :",self.age , 'sex :',self.sex)
obj1=Student("Rohit",26)
obj1.info("Male")
```

```
name : Rohit Age : 26 sex : Male
```

[23]:
```
#I want to add 1 more method in this class Job profile
def job(self,profile):
    self.profile=profile
    print("name :", self.name , "Age :",self.age , 'sex :',self.sex , "Job␣
 ↪profile :",self.profile)
```

[27]:
```
obj1.job = types.MethodType(job, obj1)
obj1.info('Male')
```

```
name : Rohit Age : 26 sex : Male
```

[28]:
```
obj1.job("Data Scientist")
```

```
name : Rohit Age : 26 sex : Male Job profile : Data Scientist
```

[30]:
```
# if you want's to delete it
```

```
del obj1.job
```

14 Class Method in Python

Class methods are methods that are called on the class itself, not on a specific object instance. Therefore, it belongs to a class level, and all class instances share a class method.

1. A class method is bound to the class and not the object of the class. It can access only class variables.
2. It can modify the class state by changing the value of a class variable that would apply across all the class objects.

Class methods are used when we are dealing with factory methods. Factory methods are those methods that return a class object for different use cases. Thus, factory methods create concrete implementations of a common interface.

Any method we create in a class will automatically be created as an instance method. We must explicitly tell Python that it is a class method using the @classmethod decorator or classmethod() function.

[16]:
```
from datetime  import date
class Student:
    def __init__(self,name,age):
        self.name=name
        self.age=age

    @classmethod
    def calculate_age(cls,name,Birth_year):
        return cls(name,date.today().year - Birth_year )

    def show(self):
        print(self.name , self.age)

s1 = Student('Rohit', 26)
s1.show()

rohit = Student.calculate_age("Rohit",1996)
rohit.show()
```

```
Rohit 26
Rohit 27
```

In This above example, we created 2 objects s1 and rohit. 1. s1 is created using the constructor 2. rohit is created using the calculate_age()

The @classmethod decorator is used for converting calculate_age() method to a class method.

The calculate_age() method takes Student class (cls) as a first parameter and returns constructor by calling Student(name, date.today().year - birthYear), which is equivalent to Student(name, age).

15 Create Class Method Using classmethod() function

Apart from a decorator, the built-in function classmethod() is used to convert a normal method into a class method. The classmethod() is an inbuilt function in Python, which returns a class method for a given function.

syntex : classmethod(function)

Note: The method you want to convert as a class method must accept class (cls) as the first argument, just like an instance method receives the instance (self).

[18]:
```
class School:
    name = 'ABC School'

    def school_name(self):
        print("School name is :",self.name)
s1 = School()
s1.school_name()
```

```
School name is : ABC School
```

[19]:
```
School.school_name=classmethod(School.school_name)
```

[20]:
```
School.school_name()
```

```
School name is : ABC School
```

[22]:
```
# Access Class Variables in Class Methods
School.name
```

[22]:
```
'ABC School'
```

[24]:
```
# Modify Class Variables in Class Methods
School.name = "Sanskriti School"
School.school_name()
```

```
School name is : Sanskriti School
```

16 Dynamically Add Class Method to a Class

We need to use the classmethod() function to add a new class method to a class.

[29]:
```
class Car:
    def __init__(self,company):
        self.company=company

    def engine(self,name,model):
        self.name=name
        self.model=model
```

```
        print(f"The Car is {self.company} {self.name} and model is {self.
 ↪model}")


c1=Car('Maruti Suzuki')
c1.engine("Swift",2020)
```

```
The Car is Maruti Suzuki Swift and model is 2020
```

[35]:
```
def price(self,price):
    self.price=price
    print(f"The Car is  price is {self.price} lacks")
# Add price in Car class
Car.price=classmethod(price)
```

[36]:
```
c2=Car("Maruti Suzuki")
c2.price(8.5)
```

```
The Car is  price is 8.5 lacks
```

17 Dynamically Delete Class Methods

We can dynamically delete the class methods from the class. In Python, there are two ways to do it:

By using the del operator By using delattr() method

18 Static Methods in Python

A static method is a general method that performs a task in isolation. It is bounded by its class not the object , so we called it using the class name. A static method does't have acess to the class and instance variable because it doed not receive an implicit first arguments like self and cls.

Therefore it cannot modify the state of the object or class.

[42]:
```
class Employee:
    def __init__(self,name,salary,project_name):
        self.name=name
        self.salary = salary
        self.project_name=project_name

    @staticmethod
    def gather_requirement(project_name):
        if project_name == 'ABC Project':
            requirement =['Task1','Task2','Task3']
        else:
            requirement = ['Task1']
        return requirement
```

```
    def work(self):
        requirement=self.gather_requirement(self.project_name)
        for task in requirement:
            print('Complete',task)

emp = Employee("Kelly",12000,'ABC Project')
emp.gather_requirement('ABC Project')
```

[42]:
```
['Task1', 'Task2', 'Task3']
```

[43]:
```
emp.work()
```

```
Complete Task1
Complete Task2
Complete Task3
```

19 Python Class Method vs. Static Method vs. Instance Method

1. Instance method performs a set of actions on the data/value provided by the instance variables. If we use instance variables inside a method, such methods are called instance methods.
2. Class method is method that is called on the class itself, not on a specific object instance. Therefore, it belongs to a class level, and all class instances share a class method.
3. Static method is a general utility method that performs a task in isolation. This method doesn't have access to the instance and class variable.

20 Encapsulation, abstraction, inheritance, and polymorphism

21 1. Encapsulation in Python

Encapsulation in Python describe the concept of bundling data and methods within a single unit.

When we create a class , we implement encapsulation , we binds all the data member and methods into a single data.

[54]:
```
class Train:
    def __init__(self,coach,Seat):
        self.coach = coach
        self.Seat=Seat
#  self.coach = coach self.Seat=Seat  is Data Members

    def info(self):
        print(self.coach , self.Seat )
# This def info() is a method

obj1 = Train('B5',44)
obj1.info()
```

B5 44

22 Here The Data Members coach and Seat binds with info() method and working together as a single unit

Using encapsulation, we can hide an object's internal representation from the outside. This is called information hiding.

encapsulation allows us to restrict accessing variables and methods directly and prevent accidental data modification by creating private data members and methods within a class.

23 Access Modifiers in Python

Encapsulation can be achived by declaring the data members and methods of a class either as private or protected we can achieve this by using single underscore and double underscore

1. Public Member : Accessible anywhere from outside the class
2. Private Member : Accessible within the class, Using 2 Underscore as a prefix
3. Protected Member : Accessible within the class and it's sub-classes , Using 1 Underscore as a prefix

24 Public Member

Public data members are accessible within and outside of a class. All member variables of the class are by default public

[55]:
```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data members
        self.name = name
        self.salary = salary

    # public instance methods
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:', self.salary)

# creating object of a class
emp = Employee('Jessa', 10000)

# accessing public data members
print("Name: ", emp.name, 'Salary:', emp.salary)

# calling public method of the class
emp.show()
```

```
Name:  Jessa Salary: 10000
Name:  Jessa Salary: 10000
```

25 Private Member

To define a private variable add two underscores as a prefix at the start of a variable name.

Private members are accessible only within the class, and we can't access them directly from the class objects.

[60]:
```
class emp:
    def __init__(self,name,salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary
emp = emp('Jessa',10000)

print("Salary :",emp.__salary)

# Salary is a private variable
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[60], line 9
      6         self.__salary = salary
      7 emp = emp('Jessa',10000)
----> 9 print("Salary :",emp.__salary)

AttributeError: 'emp' object has no attribute '__salary'
```

We can access private members from outside of a class using the following two approaches

1. Create public method to access private members
2. Use name mangling

[66]:
```
# Access Private member outside of a class using an instance method

class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

    # public instance methods
    def show(self):
```

```
        # private members are accessible from a class
        print("Name: ", self.name, 'Salary:', self.__salary)

# creating object of a class
emp = Employee('Jessa', 10000)

# calling public method of the class
emp.show()
```

```
Name:  Jessa Salary: 10000
```

Name Mangling to access private members

The name mangling is created on an identifier by adding two leading underscores and one trailing underscore, like this _classname___dataMember, where classname is the current class, and data member is the private variable name.

[65]:
```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

# creating object of a class
emp = Employee('Jessa', 10000)

print('Name:', emp.name)
# direct access to private member using name mangling
print('Salary:', emp._Employee__salary)
```

```
Name: Jessa
Salary: 10000
```

26 Protected Member

Protected member are accessible within the class and also available to it's sub-classes . To define a protected member , prefix the member name with a single underscore _ .

[71]:
```
class company:
    def __init__(self):
        self._project="Opp's"
class employee(company):
    def __init__(self,name):
        self.name = name
        company.__init__(self)
```

```
    def show(self):
        print("Employee name :" , self.name)
        print("Working on project :",self._project)

c = employee("Rohit")
c.show()
```

```
Employee name : Rohit
Working on project : Opp's
```

[72]:
```
# Direct access protected data member
print('Project:', c._project)
```

```
Project: Opp's
```

27 Getter and Setter in Python

To implement the proper encapsulation in Python , we need to use setter and getter , we use getter method to access data members and the setter methods to modify the data members.

The getter and setter method are often used when: 1. When we want to avoid direct access to private variables 2. To add validation logics for setting a value

[87]:
```
class Student:
    def __init__(self, name, age):
        # private member
        self.name = name
        self.__age = age

    # getter method
    def get_age(self):
        return self.__age

    # setter method
    def set_age(self, age):
        self.__age = age

stud = Student('Jessa', 14)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())

# changing age using setter
stud.set_age(16)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())
```

```
Name: Jessa 14
Name: Jessa 16
```

28 2. Inheritance in Python

The process of inheriting the properties of the parent class into a child class is called inheritance.

The Existing class is called a base class or Parent class and the new class is called is called subclass or child class.

The main purpose of inheritance is the reusability of code because we can use the existing class to create a new class instead of creating it from scratch.

In inheritance , the child class acquires all the data members, properties and functions from the parents class.

[95]:
```
class Baseclass:
    """Body of base class"""
class Derivedclass(Baseclass):
    """Body of derived class"""
```

Types of Inheritance

Based upon the number of child and parents classes involved , there are three types of inheritance.

1. Single inheritance
2. Multiple Inheritance
3. Multilevel Inheritance

1. Single Inheritance

[97]:
```
#In single inheritance, a child class inherits from a single-parent class. Here␣
 ↪is one child class and one parent class.

# Base class
class Vehicle:
    def vehicle_info(self):
        print("Inside Vehicle class")

# Child class
class Car(Vehicle):
    def car_info(self):
        print("Inside Car Class")

car1=Car()
car1.vehicle_info()
```

```
Inside Vehicle class
```

2. Multiple Inheritance

[102]:
```
# In Multiple inheritance , one child class can inhert from multiple parent␣
 ↪class.
# Means one child class having the properties of multiple parents class.c

class Person:
    def person_info(self,name,age):
        self.name=name
        self.age=age
        print("Name :",self.name , "Age :",self.age)

class Company:
    def company_info(self,company_name,Designation):
        self.company_name = company_name
        self.Designation = Designation
        print('Name:', company_name, 'Designation:', Designation)

class Employee(Person,Company):
    def employee_info(self,salary , skills):
        self.salary=salary
        self.skills=skills
        print('Salary:', salary, 'Skill:', skills)

emp=Employee()

emp.person_info("Rohit Kaushik" , 26)
emp.company_info("Quation","Consultant")
emp.employee_info(42000,"Artificial Intelligence Enginear")
```

```
Name : Rohit Kaushik Age : 26
Name: Quation Designation: Consultant
Salary: 42000 Skill: Artificial Intelligence Enginear
```

3. Multilevel inheritance

Suppose three classes A,B,C . “A is the superclass” , “B is the child class of A” and “C is the child class of B”

Now Class C having the properties of both A and B

we can say a chain of classes is caleed multilevel inheritance.

[103]:
```
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')

# Child class
class Car(Vehicle):
    def car_info(self):
```

```
        print('Inside Car class')

# Child class
class SportsCar(Car):
    def sports_car_info(self):
        print('Inside SportsCar class')

# Create object of SportsCar
s_car = SportsCar()

# access Vehicle's and Car info using SportsCar object
s_car.Vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

```
Inside Vehicle class
Inside Car class
Inside SportsCar class
```

Python super() function

In child class, we can refer to parent class by using the super() function. The super function returns a temporary object of the parent class that allows us to call a parent class method inside a child class method.

Benefits of using the super() function.

1. We are not required to remember or specify the parent class name to access its methods.
2. We can use the super() function in both single and multiple inheritances.
3. The super() function support code reusability as there is no need to write the entire function

[109]:
```
class Company:
    def company_name(self,name):
        self.name = name

class Employee(Company):
    def info(self):
        c_name = self.name
        print("Rohit works at ",c_name)

emp = Employee()
emp.company_name("Google")
emp.info()
```

```
Rohit works at  Google
```

[110]:
```
class Parent:
    def greet(self):
```

```
        print("Hello from Rohit")
class Child(Parent):
    def greet(self):
        super().greet()
        print("Hello from child")

ch1 = Child()
ch1.greet()
```

```
Hello from Rohit
Hello from child
```

[115]:
```
class A:
    def greet(self):
        print("Hello from A")

class B(A):
    def greet(self):
        print("Hello from B")

class C(A):
    def greet(self):
        print("Hello from C")

class D(B, C):
    def greet(self):
        super(B, self).greet()  # Calling method from specific parent class
        super(C, self).greet()
        print("Hello from D")

d = D()
d.greet()
```

```
Hello from C
Hello from A
Hello from D
```

[116]:
```
class Parent:
    def __init__(self, name):
        self.name = name

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)  # Call parent class constructor
        self.age = age

child = Child("Alice", 25)
```

```
print(child.name)
print(child.age)
```

```
Alice
25
```

[117]:
```
class A:
    def greet(self):
        print("Hello from A")

class B(A):
    def greet(self):
        print("Hello from B")

class C(A):
    def greet(self):
        print("Hello from C")

class D(B, C):
    pass

d = D()
d.greet()

# Remember that the order in which you use super() can affect the behavior due␣
 ↪to the method resolution order (MRO). It's important to understand the MRO␣
 ↪and how super() interacts with it when working with multiple inheritance␣
 ↪scenarios.
```

```
Hello from B
```

29 3. Abstraction in Python

When I touch the finger print of my mobile it unlock my phone , I don't know what is running behind the system. Same is in the Abstraction

Abstraction is used to hide the internal functionality of the function from the user.

The user only intract with the basic implementation of the function , but innew working is hidden.

User is familiar with that "What function does" but they don;t know "how it does"

30 In Python, abstraction can be achieved by using abstract classes and interfaces.

Abstraction provides a programmer to hide all the irrelevant data/process of an application in order to reduce complexity and increase the efficiency of the program. Now, we can start learning how we can achieve abstraction using the Python program.

Abstract Method : A method pnly with the declaration but not the defination is called Abstract class.

A class that consists of one or more abstract method is called the abstract class.

Python provides the abc module to use the abstraction in the Python program.

Concrete Class : Class without any Abstract method.

We can not instantiate any object for Abstract class Object only be instantiated for concrete class

[111]:
```
from abc import ABC, abstractmethod
# ABC means Abstract base classes
# we will use abstractmethod as a decorator

class Car(ABC):
    @abstractmethod
    def mileage(self):
        None

class Tesla(Car):
    def mileage(self):
        print("The mileage is 100Kmph")

t1 = Tesla()
t1.mileage()
```

```
The mileage is 100Kmph
```

[114]:
```
from abc import ABCMeta, abstractmethod
# Python program to define
# abstract class

from abc import ABC

class Polygon(ABC):

   # abstract method
   def sides(self):
      pass

class Triangle(Polygon):


   def sides(self):
      print("Triangle has 3 sides")

class Pentagon(Polygon):
```

```
   def sides(self):
      print("Pentagon has 5 sides")

class Hexagon(Polygon):

   def sides(self):
      print("Hexagon has 6 sides")

class square(Polygon):

   def sides(self):
      print("I have 4 sides")

# Driver code
t = Triangle()
t.sides()

s = square()
s.sides()

p = Pentagon()
p.sides()
```

```
Triangle has 3 sides
I have 4 sides
Pentagon has 5 sides
```

[118]:
```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
```

```
        return self.width * self.height

circle = Circle(5)
rectangle = Rectangle(4, 6)

print("Circle Area:", circle.area())
print("Rectangle Area:", rectangle.area())
```

```
Circle Area: 78.5
Rectangle Area: 24
```

[119]:
```
from abc import ABC, abstractmethod

class PaymentGateway(ABC):
    @abstractmethod
    def process_payment(self, amount):
        pass

class CreditCard(PaymentGateway):
    def process_payment(self, amount):
        print("Processed credit card payment of", amount)

class PayPal(PaymentGateway):
    def process_payment(self, amount):
        print("Processed PayPal payment of", amount)

credit_card = CreditCard()
paypal = PayPal()

credit_card.process_payment(100)
paypal.process_payment(50)
```

```
Processed credit card payment of 100
Processed PayPal payment of 50
```

31 4. Polymorphism in Python

Polymorphism in Python is the ability of an object to take many forms. In simple words, polymorphism allows us to perform the same action in many different ways.

For example, Rohit acts as an employee when he is at the office. However, when he is at home, he acts like a Husband. Also, he represents himself differently in different places. Therefore, the same person takes different forms as per the situation.

In polymorphism, a method can process objects differently depending on the class type or data type

[122]:
```
List1 = ['Science', 'behind', 'Data']
string1 = 'ABC School'
dict1 = {"fName":"Rohit","lName":"Kaushik"}

# calculate count
print(len(List1))
print(len(string1))
print(len(dict1))

#The len() method treats an object as per its class type
```

```
3
10
2
```

32 Polymorphism With Inheritance

Polymorphism is mainly used with inheritance. In inheritance, child class inherits the attributes and methods of a parent class. The existing class is called a base class or parent class, and the new class is called a subclass or child class or derived class.

Using method overriding polymorphism allows us to defines methods in the child class that have the same name as the methods in the parent class. This process of re-implementing the inherited method in the child class is known as Method Overriding

[123]:
```
class Vehicle:

    def __init__(self, name, color, price):
        self.name = name
        self.color = color
        self.price = price

    def show(self):
        print('Details:', self.name, self.color, self.price)

    def max_speed(self):
        print('Vehicle max speed is 150')

    def change_gear(self):
        print('Vehicle change 6 gear')


# inherit from vehicle class
class Car(Vehicle):
    def max_speed(self):
        print('Car max speed is 240')

    def change_gear(self):
```

```
        print('Car change 7 gear')


# Car Object
car = Car('Car x1', 'Red', 20000)
car.show()
# calls methods from Car class
car.max_speed()
car.change_gear()

# Vehicle Object
vehicle = Vehicle('Truck x1', 'white', 75000)
vehicle.show()
# calls method from a Vehicle class
vehicle.max_speed()
vehicle.change_gear()
```

```
Details: Car x1 Red 20000
Car max speed is 240
Car change 7 gear
Details: Truck x1 white 75000
Vehicle max speed is 150
Vehicle change 6 gear
```

we have a vehicle class as a parent and a 'Car' and 'Truck' as its sub-class. But each vehicle can have a different seating capacity, speed, etc., so we can have the same instance method name in each class but with a different implementation. Using this code can be extended and easily maintained over time.

33 Overrride Built-in Functions

[125]:
```
class Shopping:
    def __init__(self, basket, buyer):
        self.basket = list(basket)
        self.buyer = buyer

    def __len__(self):
        print('Redefine length')
        count = len(self.basket)
        # count total items in a different way
        # pair of shoes and shir+pant
        return count * 2

shopping = Shopping(['Shoes', 'dress'], 'Jessa')
print(len(shopping))
```

```
Redefine length
```

```
4
```

34 Polymorphism In Class methods

Polymorphism with class methods is useful when we group different objects having the same method. we can add them to a list or a tuple, and we don't need to check the object type before calling their methods. Instead, Python will check object type at runtime and call the correct method. Thus, we can call the methods without being concerned about which class type each object is. We assume that these methods exist in each class.

Python allows different classes to have methods with the same name.

1. Let's design a different class in the same way by adding the same methods in two or more classes.
2. Next, create an object of each class.
3. Next, add all objects in a tuple.
4. In the end, iterate the tuple using a for loop and call methods of a object without checking its class.

[126]:
```
class Ferrari:
    def fuel_type(self):
        print("Petrol")

    def max_speed(self):
        print("Max speed 350")

class BMW:
    def fuel_type(self):
        print("Diesel")

    def max_speed(self):
        print("Max speed is 240")

ferrari = Ferrari()
bmw = BMW()

# iterate objects of same type
for car in (ferrari, bmw):
    # call methods without checking class of object
    car.fuel_type()
    car.max_speed()
```

```
Petrol
Max speed 350
Diesel
Max speed is 240
```

35 Polymorphism with Function and Objects

We can create polymorphism with a function that can take any object as a parameter and execute its method without checking its class type. Using this, we can call object actions using the same function instead of repeating method calls

[127]:
```
class Ferrari:
    def fuel_type(self):
        print("Petrol")

    def max_speed(self):
        print("Max speed 350")

class BMW:
    def fuel_type(self):
        print("Diesel")

    def max_speed(self):
        print("Max speed is 240")

# normal function
def car_details(obj):
    obj.fuel_type()
    obj.max_speed()

ferrari = Ferrari()
bmw = BMW()

car_details(ferrari)
car_details(bmw)
```

```
Petrol
Max speed 350
Diesel
Max speed is 240
```

36 Polymorphism In Built-in Methods

[128]:
```
students = ['Emma', 'Jessa', 'Kelly']
school = 'ABC School'

print('Reverse string')
for i in reversed('PYnative'):
    print(i, end=' ')

print('\nReverse list')
for i in reversed(['Emma', 'Jessa', 'Kelly']):
```

```
    print(i, end=' ')
```

```
Reverse string
e v i t a n Y P
Reverse list
Kelly Jessa Emma
```

37 Method Overloading

The process of calling the same method with different parameters is known as method overloading. Python does not support method overloading. Python considers only the latest defined method even if you overload the method. Python will raise a TypeError if you overload the method.

[130]:
```
def addition(a, b):
    c = a + b
    print(c)


def addition(a, b, c):
    d = a + b + c
    print(d)

# This line will call the second product method
addition(3, 7, 5)
```

```
15
```

[132]:
```
# User-defined polymorphic method
class Shape:
    # function with two default parameters
    def area(self, a, b=0):
        if b > 0:
            print('Area of Rectangle is:', a * b)
        else:
            print('Area of Square is:', a ** 2)

square = Shape()
square.area(5)

rectangle = Shape()
rectangle.area(5, 3)
```

```
Area of Square is: 25
Area of Rectangle is: 15
```

38 Operator Overloading in Python

Operator overloading means changing the default behavior of an operator depending on the operands (values) that we use. In other words, we can use the same operator for multiple purposes.

For example, the + operator will perform an arithmetic addition operation when used with numbers. Likewise, it will perform concatenation when used with strings.

[134]:
```
# add 2 numbers
print(100 + 200)

# concatenate two strings
print('Jess' + 'Roy')

# merger two list
print([10, 20, 30] + ['jessa', 'emma', 'kelly'])
```

```
300
JessRoy
[10, 20, 30, 'jessa', 'emma', 'kelly']
```

[135]:
```
class Book:
    def __init__(self, pages):
        self.pages = pages

# creating two objects
b1 = Book(400)
b2 = Book(300)

# add two objects
print(b1 + b2)

#We can overload + operator to work with custom objects also. Python provides␣
 ↪some special or magic function that is automatically invoked when associated␣
 ↪with that particular operator.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[135], line 10
      7 b2 = Book(300)
      9 # add two objects
---> 10 print(b1 + b2)

TypeError: unsupported operand type(s) for +: 'Book' and 'Book'
```

[136]:
```
class Book:
    def __init__(self, pages):
        self.pages = pages

    # Overloading + operator with magic method
    def __add__(self, other):
        return self.pages + other.pages

b1 = Book(400)
b2 = Book(300)
print("Total number of pages: ", b1 + b2)
```

```
Total number of pages:  700
```

39 Overloading the * Operator

The * operator is used to perform the multiplication. Let's see how to overload it to calculate the salary of an employee for a specific period. Internally * operator is implemented by using the **mul**() method.

[137]:
```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def __mul__(self, timesheet):
        print('Worked for', timesheet.days, 'days')
        # calculate salary
        return self.salary * timesheet.days


class TimeSheet:
    def __init__(self, name, days):
        self.name = name
        self.days = days


emp = Employee("Jessa", 800)
timesheet = TimeSheet("Jessa", 50)
print("salary is: ", emp * timesheet)
```

```
Worked for 50 days
salary is:  40000
```

[ ]:

[ ]:

[ ]: