# UNIT-4 PART- 2

Object Oriented Programming: Concept of class, object and instances, Constructor, class attributes and destructors, Real time use of class in live projects, Inheritance, overlapping and overloading operators, Adding and retrieving dynamic attributes of classes, Programming using Oops support

Design with Classes: Objects and Classes, Data ModelingExamples, Case Study An ATM, Structuring Classes with Inheritance and Polymorphism

# Introduction

We have two programming techniques namely

1. Procedural-oriented programming technique
2. Object-oriented programming technique

Till now we have using the Procedural-oriented programming technique, in which our program is written using functions and block of statements which manipulate data. However a better style of programming is Object-oriented programming technique in which data and functions are combined to form a class. Object Oriented programming (OOP) is a programming paradigm that relies on the concept of classes and objects. It is used to structure a software program into simple, reusable pieces of code blueprints (usually called classes), which are usedto create individual instances of objects. There are many object-oriented programming languages including JavaScript, C++, Java, and Python.

Classes and objects are the main aspects of object oriented programming.

# Overview of OOP Terminology

* **Class** − A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
* **Class variable** − A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
* **Data member** − A class variable or instance variable that holds data associated with a class and its objects.
* **Function overloading** − The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
* **Instance variable** − A variable that is defined inside a method and belongs only to the

current instance of a class.

* + **Inheritance** − The transfer of the characteristics of a class to other classes that are derived from it.
  + **Instance** − An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
  + **Instantiation** − The creation of an instance of a class.
  + **Method** − A special kind of function that is defined in a class definition.
    - **Object** − A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
  + **Operator overloading** − The assignment of more than one function to a particular operator.

# Benefits of OOP

* OOP models complex things as reproducible, simple structures
* Reusable, OOP objects can be used across programs
* Allows for class-specific behavior through polymorphism
* Easier to debug, classes often contain all applicable information to them
* Secure, protects information through encapsulation

# Classes:

* 1. Class is a basic building block in python
  2. Class is a blue print or template of a object
  3. A class creates a new data type
  4. And object is instance(variable) of the class
  5. In python everything is an object or instance of some class Example :

All integer variables that we define in our program are instances of class int. >>> a=10

>>> type(a)

<class 'int'>

* 1. The python standard library based on the concept of classes and objects

# Defining a class:

Python has a very simple syntax of defining a class.

# Syntax :

Class class-name: Statement1 Statement2 Statement3

-

-

-

Statement

From the syntax, Class definition starts with the keyword class followed by class-name and a colon(:). The statements inside a class are any of these following

* + 1. Sequential instructions
    2. Variable definitions
    3. Decision control statements
    4. Loop statements
    5. Function definitions

Note : the class members are accessed through class object

Note : class methods have access to all data contained in the instance of the object **Creating objects: ( creating an object of a class is known as class instantiation)**

* + Once a class is defined, the next job is to create a object of that class. • The object can then access class variables and class methods using dot operator

# Syntax of object creation:

Object-name=class-name()

* + Syntax for accessing class members through the class object is Object-name.class-member-name

# Example :

class ABC:

a=10

obj=ABC() print(obj.a)

# self variable and class methods:

* + Self refers to the object itself ( Self is a pointer to the class instance )
  + Whenever we define a member function in a class always use a self as a first argument and give rest of the arguments
  + Even if it doesn’t take any parameter or argument you must pass self to a member function
  + We do not give a value for this parameter, when call the method, python will provide it.
  + The self in python is equivalent to the this pointer in c++

# Example 1 :

class Person:

pc=0 # Class varibles

def setFullName(self,fName,lName): self.fName=fName # instance variables self.lName=lName # instance variables

def printFullName(self):

print(self.fName," ",self.lName)

print("Person number : ",self.pc) #access Classvariable

PName=Person() #Object PName created PName.setFullName("vamsi","kurama")

PName.pc=7 #Attribute pc of PName modified PName.printFullName()

P=Person() #Object P created P.setFullName("Surya","Vinti")

P.pc=23 #Attribute pc of P modified P.printFullName()

**Output:**

>>>

vamsi kurama Person number : 7 Surya Vinti Person number : 23

# Constructor method:

A constructor is a special type of method (function) that is called when it instantiates an object of a class. The constructors are normally used to initialize (assign values) to the instance variables.

# Creating a constructor: (The name of the constructor is always the \_ \_init\_ \_().)

The constructor is always written as a function called init (). It must always take as its first argument a reference to the instance being constructed.

While creating an object, a constructor can accept arguments if necessary. When you create a class without a constructor, Python automatically creates a default constructor that doesn't do anything.

Every class must have a constructor, even if it simply relies on the default constructor. Example:

class Person:

pc=0 # Class varibles def init (self):

print("Constructor initialised ") self.fName="XXXX" self.lName="YYYY"

def setFullName(self,fName,lName): self.fName=fName # instance variables self.lName=lName # instance variables

def printFullName(self):

print(self.fName," ",self.lName)

print("Person number : ",self.pc) #access Classvariable

PName=Person() PName.printFullName() PName.setFullName("vamsi","kurama")

PName.pc=7

print("After setting Name:") PName.printFullName()

# Output:

>>>

Constructor initialised XXXX YYYY

Person number : 0 After setting Name:

vamsi kurama Person number : 7

# Destructor:

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much needed in C++ because Python has a garbage collector that handles memory management automatically. The \_ \_ del \_ \_ ( ) method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

**Syntax of destructor declaration:**

def del (self):

# body of destructor

**Note:** A reference to objects is also deleted when the object goes out of reference or when the program ends.

**Example 1:** Here is the simple example of destructor. By using del keyword we deleted the all references of object ‘obj’, therefore destructor invoked automatically.

# Python program to illustrate destructor class Employee:

# Initializing

def init (self):

print('Employee created.')

# Deleting (Calling destructor) def del (self):

print('Destructor called, Employee deleted.')

obj = Employee() del obj

# Output:

Employee created

Destructor called, Employee deleted

# Inheritance:

One of the major advantages of Object Oriented Programming is reusability. Inheritance is one of the mechanisms to achieve the reusability. Inheritance is used to implement is-a relationship.

Definition: A technique of creating a new class from an existing class is called inheritance. The old or existing class is called base class or super class and a new class is called sub class or derived class or child class.

The derived class inherits all the variable and methods of the base class and adds their own variables and methods. In this process of inheritance base class remains unchanged.

Syntax to inherit a class:

Class MySubClass(object):

Pass(Body-of-the-derived-class) Example :

class Pet:

def init (self,name,age): self.name=name self.age=age

class Dog(Pet):

def sound(self):

print("I am {} and My age is {} and I sounds Like".format(self.name,self.age)) print("Bow Bow..")

class Cat(Pet):

def sound(self):

print("I am {} and My age is {} and I sounds Like".format(self.name,self.age)) print("Meow Meow..")

class Parrot(Pet):

def sound(self):

print("Hello I am {} and My age is {} ".format(self.name,self.age)) p1=Dog("Dozer",4)

p2=Cat("Edward",3) p3=Parrot("Jango",6) p1.sound() p2.sound() p3.sound()

Example 2:

class Person:

def init (self,name,age): self.name=name self.age=age

def display(self):

print("name=",self.name)

print("age=",self.age) class Teacher(Person):

def init (self,name,age,exp,r\_area): Person. init (self,name,age) self.exp=exp

self.r\_area=r\_area def displayData(self):

Person.display(self) print("Experience=",self.exp) print("Research area=",self.r\_area)

class Student(Person):

def init (self,name,age,course,marks): Person. init (self,name,age) self.course=course self.marks=marks

def displayData(self):

Person.display(self) print("course=",self.course) print("marks=",self.marks)

print("\*\*\*\*\*\*\*\*TEACHER\*\*\*\*\*\*\*\*\*\*\*") t=Teacher("jai",55,13,"cloud computing") t.displayData() print("\*\*\*\*\*\*\*\*STUDENT\*\*\*\*\*\*\*\*\*\*\*") s=Student("hari",21,"B.Tech",99) s.displayData()

# Types of inheritance:

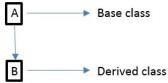
Python supports the following types of inheritan:

1. Single inheritance
2. Multiple Inheritance
3. Multi-level Inheritance
4. Multi path Inheritance
5. Hierarchical Inherttance

# Single Inheritance:

When a derived class inherits features from only one base class, it is called Single inheritance.

Syntax:

class Baseclass:

<body of base class> class Derivedclass(Baseclass):

<body of the derived class> Example:

class A:

i=10

class B(A):

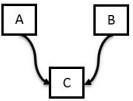
j=20

obj=B()

print("member of class A is",obj.i) print("member of class B is",obj.j)

# Multiple Inheritance:

When derived class inherits features from more than one base class then it is called Multiple Inheritance.

Syntax:

class Baseclass1:

<body of base class1> class Baseclass2:

<body of base class2>

class Derivedclass(Baseclass1,Baseclass2):

<body of the derived class>

e.g. class A:

i=10

class B:

j=20

class C(A,B):

k=30

obj=C()

print("member of class A is",obj.i) print("member of class B is",obj.j) print("member of class C is",obj.k)

# Multi-Level Inheritance:

When derived class inherits features from other derived classes then it is called Multi-level inheritance.

Syntax:

class Baseclass:

<body of base class> class Derivedclass1(Baseclass):

<body of derived class 1>

class Derivedclass2(Derivedclass1):

<body of the derived class2>

e.g.

class A:

i=10

class B(A):

j=20

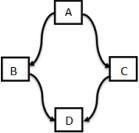
class C(B):

k=30

obj=C()

print("member of class A is",obj.i) print("member of class B is",obj.j) print("member of class C is",obj.k)

# Multi Path Inheritance:

Syntax:

class Baseclass:

<body of the base class> class Derived1(Baseclass):

<body of the derived1> class Derived2(Baseclass):

<body of the derived2>

class Derived3 (Derived1,Derived2) :

<body of derived3>

e.g.

class A:

i=10

class B(A):

j=20

class C(A):

k=30

class D(B,C):

ijk=40 obj=D()

print("member of class A is",obj.i) print("member of class B is",obj.j) print("member of class C is",obj.k) print("member of class Cis",obj.ijk)

**Hierarchical inheritance** More than one derived classes are created from a single base.

class Person:

def init (self,name,age): self.name=name self.age=age

def display(self):

print("name=",self.name)

print("age=",self.age) class Teacher(Person):

def init (self,name,age,exp,r\_area): Person. init (self,name,age) self.exp=exp

self.r\_area=r\_area def displayData(self):

Person.display(self) print("Experience=",self.exp) print("Research area=",self.r\_area)

class Student(Person):

def init (self,name,age,course,marks): Person. init (self,name,age) self.course=course self.marks=marks

def displayData(self):

Person.display(self) print("course=",self.course) print("marks=",self.marks)

print("\*\*\*\*\*\*\*\*TEACHER\*\*\*\*\*\*\*\*\*\*\*") t=Teacher("jai",55,13,"cloud computing") t.displayData() print("\*\*\*\*\*\*\*\*STUDENT\*\*\*\*\*\*\*\*\*\*\*") s=Student("hari",21,"B.Tech",99) s.displayData()

# Polymorphism:

The word polymorphism means having many forms. In python we can find the same operator or function taking multiple forms. That helps in re using a lot of code and decreases code complexity.

# Polymorphism in operators

* The + operator can take two inputs and give us the result depending on what the inputs are.
* In the below examples we can see how the integer inputs yield an integer and if one of the input is float then the result becomes a float. Also for strings, they simply get concatenated.

# Example:

a = 23

b = 11

c = 9.5

s1 = "Hello" s2 = "There!"

print(a + b) print(type(a + b)) print(b + c) print(type (b + c)) print(s1 + s2) print(type(s1 + s2))

# Polymorphism in built-in functions

We can also see that different python functions can take inputs of different types and then process them differently. When we supply a string value to len() it counts every letter in it. But if we give tuple or a dictionary as an input, it processes them differently.

Example:

str = 'Hi There !'

tup = ('Mon','Tue','wed','Thu','Fri')

lst = ['Jan','Feb','Mar','Apr']

dict = {'1D':'Line','2D':'Triangle','3D':'Sphere'} print(len(str))

print(len(tup)) print(len(lst)) print(len(dict))

# Polymorphism in inheritance:

**Method Overriding:**

It is nothing but same method name in parent and child class with different functionalities. In inheritance only we can achieve method overriding. If super and sub classes have the same method name and if we call the overridden method then the method of corresponding class (by using which object we are calling the method) will be executed.

e.g.

class A:

i=10

def display(self):

print("I am class A and I have data",self.i)

class B(A):

j=20

def display(self):

print("I am class B and I have data",self.j)

obj=B() obj.display()

OUTPUT :

I am class B and I have data 20

Note: In above program the method of class B will execute. If we want to execute method of class A by using Class B object we use super() concept.

# Super():

In method overriding , If we want to access super class member by using sub class object we use super()

e.g class A:

i=10

def display(self):

print("I am class A and I hava data",self.i)

class B(A):

j=20

def display(self):

super().display()

print("I am class B and I hava data",self.j)

obj=B() obj.display() OUTPUT:

I am class A and I have data 10 I am class B and I have data 20

Note: In above example both the functions (display () in class A and display () in class B) will execute

Note: Name mangling is the encoding of function and variable names into unique names so that linkers can separate common names in the language.

**overloading operators**

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example, operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because ‘+’ operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called Operator Overloading.

# Python program to show use of + and \* operator for different purposes.

print(1 + 2)

# concatenate two strings print("Learn"+"For")

# Product two numbers print(3 \* 4)

# Repeat the String

print("Learn"\*4) Output:

3

LearnFor 12

LearnLearnLearnLearn

**Example 2:**

Changing the behavior of operator is as simple as changing the behavior of method or function. You define methods in your class and operators work according to that behavior defined in methods. When we use + operator, the magic method add is automatically invoked in which the operation for + operator is defined.

class A:

def init (self, a):

self.a = a

def add (self, o): # adding two objects return self.a + o.a

ob1 = A(1) ob2 = A(2)

Ob1.a=1 Ob2.a=2

Ob3.a=”sai” Ob4.a=”kumar” Ob5.a=[ 2,5,6,2 ]

Ob6.a=[ 34.6,12 ]

ob3 = A("sai") ob4 = A("kumar") ob5=A([2,5,6,2])

ob6=A([34.6,12])

print(ob1 + ob2) print(ob3 + ob4) print(ob5 + ob6)

# OUTPUT:

>>> 3

saikumar

[2, 5, 6, 2, 34.6, 12]

# DATA MODELING EXAMPLES:

# In the Python programming language, each entity is treated as an object. Moreover, unlike other programming languages like C or Java, Python does not work with primitive data or non-primitive data types. Everything whether it be integer, float, string, function, or list in python is represented by objects or relationships between objects. ****Python Data Model**** is composed of these objects and it is one of the main reasons make Python a great and widely used programming language.

## What is Data Modelingin Python?

The process of creating Data Models using the syntax and environment of the Python programming language is called Data Modelingin Python. A Data Model is a data abstraction model that organizes different elements of data and standardizes the way they relate to one another and to the properties of real-world entities. In simple words, Data Modelingin Python is the general process by which this programming language organizes everything internally and it treats and processes data.

The Data Model is the building block of Python. Internally Data Model has its design and code blocks for its implementation. It is composed of entities, and entities are none but the objects. It is said that everything is an object in Python. Each entity has its own attributes (properties) and methods(actions) or behaviour associated with it. Each object has three attributes: an identity, a type, and a value. Let’s learn them in detail.

## Identity of an object

Every object, either for Data Modelingin Python or any other activity involving Python’s environment, has an identity which can never change once it is created. Think of identity as an object’s address in the memory.

**id()** is the function that gives us the identity of the object or we can say it returns the virtual memory address of the object. The memory address will be an integer value.

Python Code:

>>> a='hevodata'

>>> id(a)

1397871458544

>>> b=101

>>> id(b)

1623965024

>>> c='hevodata'

>>> id(c)

1397871458544

From the above code of Data Modelingin Python, we can see variables ‘a’ and ‘b’ have different memory addresses but ‘a’ and ‘c’ have the same memory address as they have the same value.

We can check if two objects are identified using the **‘is’**operator. The ‘is’ operator basically compares the identity of two objects. If the identity of two variables is the same then it returns ‘True’ otherwise ‘False’.

Python Code:

>>> a is b

False

>>> a is c

True

From the above code and output, it is clear that variables ‘a’ and ‘c’ have the same identity while ‘b’ has a different identity than ‘a’ and ‘c’. This is how the identity of an object is decided during Data Modelingin Python.

## Type of an Object

During Data Modelingin Python, the type of an object means the name of the class to which the object belongs. Function **type()** tells the type of the object. By knowing the type of an object, it is easy for user’s to specify two things.

1. The operation allowed on that object
2. The set of values the object can hold.

The type of an object cannot change but under certain controlled conditions, it is possible to change the type of an object. Although it is not a good idea and not advisable as well.

**Python Code:**

>>> a='hevodata'

>>> x=type(a)

>>> print("Type of variable 'a' is: ", x)

Type of variable 'a' is: <class 'str'>

>>> b=101

>>> y= type(b)

>>> print("Type of variable 'b' is: ", y)

Type of variable 'b' is: <class 'int'>

>>> fruits = ('apple', 'banana', 'grapes', 'orange')

>>> t = type(fruits)

>>> print("Type of variable 'fruits' is: ", t)

Type of variable 'fruits' is: <class 'tuple'>

From the above python code and its output, you can see types of the different objects as different classes like ‘str’, ‘int’, and ‘tuple’. There are more classes as well in Python

## Value of an Object

An object’s value during Data Modelingin Python is the data that is stored for that object. The value object can hold is decided on the basis of the type of object.

**Python Code:**

>>> var='article'

>>> print("Value of variable 'var' is: ", var)

Value of variable 'var' is: article

In the above code, ‘var’ is the variable and ‘article’ is the value of ‘var’.

Object values are changeable and it depends on their type. Python supports the following 2 types of objects based on their values:

* [Mutable Objects](https://hevodata.com/learn/data-modelling-in-python/#O1)
* [Immutable Objects](https://hevodata.com/learn/data-modelling-in-python/#O2)

There is some type for which the value of an object cannot change those are called immutable objects and whose value can be changed are called mutable objects.

### 1) Mutable Objects

The mutability of objects is decided on the basis of their type. Lists, Dictionaries are mutable objects. Those objects whose values can be changed are called Mutable objects.

The following Python code is useful for creating a list for Data Modeling in Python:

#Let's create a list

>>> a = [11, 22, 33]

>>> print("List values: ",a)

>>> print("Identity of a: ",id(a))

List values: [11, 22, 33]

Identity of a: 1397871407048

>>> a[0] = 1 #Change first value of list

>>> print("Changed List values: ",a)

>>> print("Identity of a: ",id(a))

Changed List values: [1, 22, 33]

Identity of a: 1397871407048

From the above code, you can see the identity of the list remains the same but the value of the list changed. So, the list is mutable.

2) Immutable Objects

During Data Modelingin Python, Immutable Objects are the objects that stored data but their values cannot be modified. Numbers, Strings, Tuples, and Sets are immutable.

The following Python code is useful for creating a variable with string value during Data Modelingin Python:

#Let's create a varible with string value

s = "Hevo"

print("Variable value: ",s)

print("Identity of s: ",id(s))

Variable value: Hevo

Identity of s: 1397871732528

s = "Data" #Change value of varibale 's'

print("Variable value: ",s)

print("Identity of s: ",id(s))

Variable value: Data

Identity of s: 1397836021296

From the above code, if you change the value of a string variable, its identity changes. It means the value of the object at id ‘1397871732528’ does not change but a new variable with the same name but a different value is created at memory address ‘1397836021296’. So, you can conclude strings are immutable.

**Special Methods for Data Modelingin Python:**

Understanding special methods for Data Modelingin Python are very important for you as a data professional. Special methods describe the internal functioning of the basic object operations. The special methods name starts and trails with two underscores. Another name for special methods is the dunder methods or magic methods.

The special method names allow your objects to implement, support, and interact with basic language constructs such as Iteration, object creation, object destruction, collections, attribute access, etc.

Below are some of the examples of special methods for Data Modeling in Python which help to

understand how these built-ins work in Python.

The  \_\_init\_\_() method is for initialization and is called by the python interpreter itself when an instance of an object is created.

The  len(x) method is for the length calculation of an object, internally the python interpreter calls x.\_\_len()

Call x[2] to get an item at location 2, internally the python interpreter calls x.\_\_getitem\_\_(2)

When str(x) is called, internally the python interpreter, calls x.\_\_str\_\_()

Operators are also magic methods, add operator x + y actually turns to x.\_\_add\_\_(y)

You can also write your own class with your own special methods for Data Modelingin Python. The below example shows a code to get a particular item on a list and find the length of the list and better understanding.

class MyList:

def \_\_init\_\_(self, \*args):

self.\_data = list(args)

def \_\_getitem\_\_(self, index):

out = self.\_data[index]

return (out)

def \_\_len\_\_(self):

return len(self.\_data)

x = MyList(11, 22, 33, 4, 5) #List initialization

# Get length of list

print("Length of list: ",len(x))

# Get an item of list

print("Item of list: ", x[2])

**Output:**

Length of list: 5

Item of list: 33

In the above example, we have seen the implementation of three special methods, initialization, length, and getting an item. The above code is just an example. Programmers can implement many more functions like these magic methods and excel at Data Modelingin Python.

# Case Study An ATM:

class ATM:

def init (self):

self.balance=0

print("new account created") def deposit(self):

amount=int(input("enter amount to deposit")) self.balance=self.balance+amount

print("new balance is:",self.balance) def withdraw(self):

amount=int(input("enter amount to withdraw")) if self.balance<amount:

print("Insufficient Balance")

else:

def enquiry(self):

self.balance=self.balance-amount print("new balance is:",self.balance)

print("Balance is:",self.balance)

a=ATM()

a.deposit() a.withdraw() a.enquiry() **OUTPUT:**

>>>

new account created

enter amount to deposit15000 new balance is: 15000

enter amount to withdraw5648 new balance is: 9352

Balance is: 9352

# Adding and retrieving dynamic attributes of classes:

Dynamic attributes in Python are terminologies for attributes that are defined at runtime, after creating the objects or instances.

# Example:

class EMP:

employee = True e1 = EMP()

e2 = EMP()

e1.employee = False

e2.name = "SAI KUMAR" #DYNAMIC ATTRIBUTE

print(e1.employee)

print(e2.employee) print(e2.name)

print(e1.name) # this will raise an error as name is a dynamic attribute created only for #the e2 object

# Data Hiding

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then are not be directly visible to outsiders.

Example

class JustCounter:

secretCount = 0 def count(self):

self. secretCount += 1 print self. secretCount

counter = JustCounter() counter.count() counter.count()

print counter. secretCount

When the above code is executed, it produces the following result − 1

2

Traceback (most recent call last):

File "test.py", line 12, in <module> print counter. secretCount

AttributeError: JustCounter instance has no attribute ' secretCount'

Python protects those members by internally changing the name to include the class name. You can access such attributes as *object.\_className attrName*. If you would replace your last line as following, then it works for you −

.........................

print counter.\_JustCounter secretCount

When the above code is executed, it produces the following result − 1

2

2