OOPs Definition

Definition

Python supports both procedure-oriented and object-oriented features.
Object-Oriented Programming (OOPs) is a programming paradigm that uses classes and objects to look similar to real life. OOPs has four central concepts: Encapsulation, Abstraction, Inheritance, and Polymorphism. OOPs mainly works upon two keys is Attributes and Functionality.

Classes

Classes consist of Data and Functions. The Data in class is known as Members and functions of the class are known as methods.

Syntax:

```
Class className:
    Statement1_Class
    Statement2_Class
    Statement3_Class
    Statementn_Class
```

Example:

```
Class student:
    No = 100
    Sub = “Maths”
Print(student.No, student.Sub)
```

Objects

Using Python Programming, we can create objects in two ways that are as follows:

- Attribute Referencing
- Instantiation

Attribute Referencing

In this method, class properties can be accessed using the class name itself.

Syntax:

className:propertyName

Example:

```
Class student;
        No_of_admission = []
student.No_of_admission = 100
print(student.No_of_admission)
```

Instantiation

While Instantiation, an empty object is created, and this object is made, and this object is assigned with the given object name.

Syntax:

Object_name:class_name()

After creating the class instance, the properties of the class are accessed using following

Object_name:propertyname

Example:

```
Class student;
        No_of_admission = []
```

```
Calling.student();

Calling.No_of_admission = 100

Print(Calling.No_of_admission)
```

Encapsulation

Encapsulation means wrapping data and the functions in a single place.

Data Abstraction

The data abstraction is the process by which only a few necessary features are highlighted while the internal details are suppressed. For example, while using mobile, the user uses all the features without knowing the internal hardware and software. Thus Data Abstraction reduces the complexity associated with data encapsulation.

Polymorphism

Using polymorphism, a single entity can act in multiple forms. For example, a shape can be both a circle and a triangle.

Inheritance

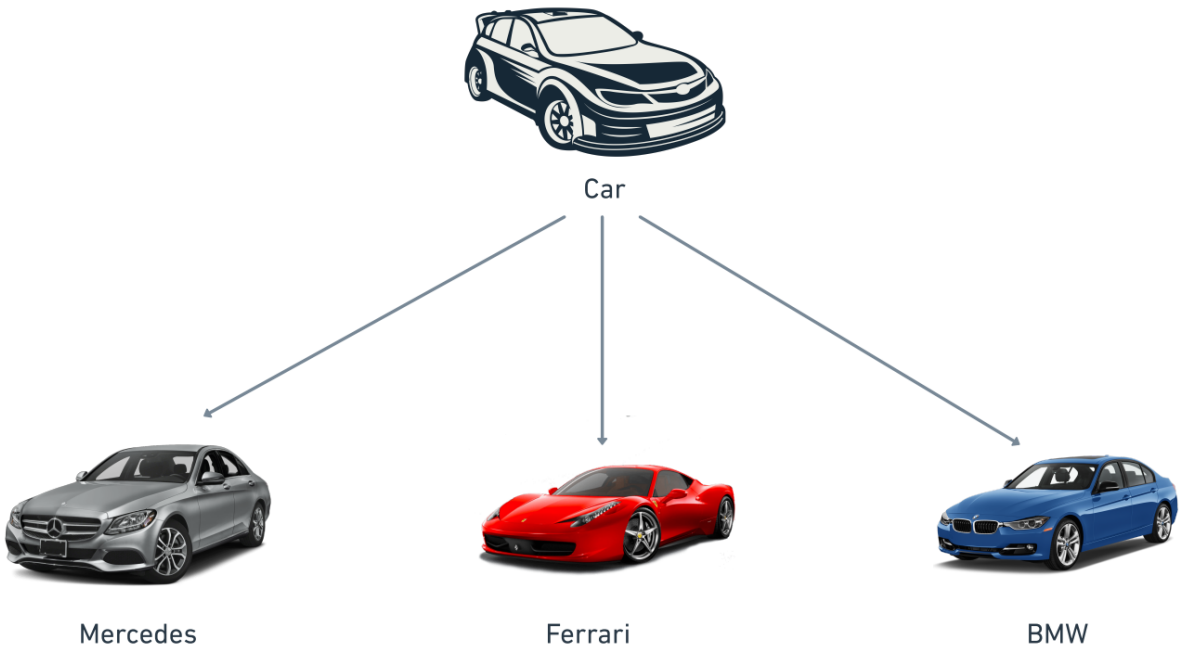
Inheritance allows a class to derive the properties of another class. For example class1 has data1 & function 1 where class 2 has data1 & function 1 of class 1 and class3 has data1 and functions 1 of class 1.

Example of OOPs in the Industry

Example of OOPs using Cars:-

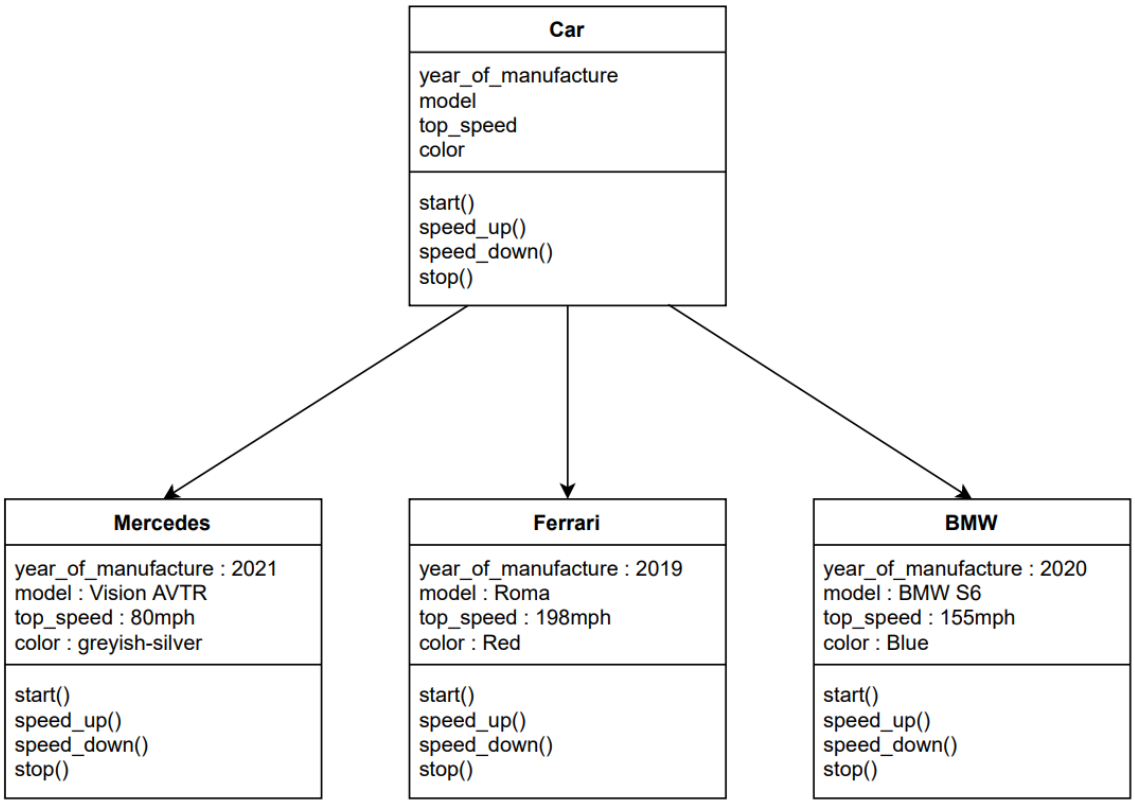
Suppose you have details of many cars. All cars have the same properties like a year of manufacture, Model, Top speed, Color, etc., and the same functions like start, speed up, speed down, stop, etc. so we can say that all types of cars have the same information so we can create a common structure for store the information. This structure is called class and can have different objects. Each object shows a different type of car with different information.

Here we will make Car class, and it will work as a basic template for other objects. We will make car class objects (Ferrari, BMW, and Mercedes). Each Car Object will have its own, Year of Manufacture, model, Top Speed, color, Engine Power, efficiency, etc.



The car class would allow the programmer to store similar information unique to each car (different models, colors, top speeds, etc.) and associate the appropriate information.

Understanding example using flowchart:-



Real World Class Modeling

Object-oriented programming (OOPs) is a programming paradigm based on objects. Objects that are real-world entities like cars, mobile, trees, etc.

In OOPs, there are generally four pillars that are as follows

1. Polymorphism
2. Inheritance
3. Encapsulation
4. Abstraction

In the following part, you can see how OOPs connect with real-world examples.

Polymorphism

For Polymorphism, let us consider a girl. The girl can be many things. She can be Mother, Sister, and Student, etc. The same person can have different roles.

As per the Polymorphism definition in Python, Polymorphism lets us define methods in the child class with the same name as the methods in the parent class.

Inheritance

For Inheritance, let us consider dogs. The dogs can have the same color, name, size, etc., but they are not the same dog.

As per the Inheritance definition in Python, it is a concept in OOPs where a new class can modify existing classes.

Encapsulation

For Encapsulation, let us consider a company. A company can have several departments like the Production Department, HR Department, Marketing Department, etc. All these departments are necessary to make up a company.

As per the Encapsulation definition in Python, It is wrapping up variables and methods into a single entity.

Abstraction

For Abstraction, Let us consider a Mobile Phone. We can do many things on a mobile phone like making a call, taking pictures and playing games, etc. It doesn't show the inside process of how it is doing things.

As per the Abstraction definition in Python, It is a process of hiding the actual implementation of the method by only showing a method signature.

Why Real-World Modelling is Needed

- To make the development and maintenance of projects more effortless.
- To provide the feature of data hiding that is good for security concerns.
- We can solve real-world problems if we are using object-oriented programming.
- It ensures code reusability.
- It lets us write generic code: which will work with a range of data, so we don't have to register basic stuff over and over again.

Classes

In python, data encapsulation is supported with “class”. Class consists of both data and functions. The data in a class is called a member while functions in class are called as methods.

Definition of class

The definition of class is given below.

```
class class_name:
        Statement1_class
        Statement2_class
        Statement3_class
        .....
        Statementn_class
```

The statement in the class consists of either data or function.

```
Class class_name:
        Statements_classbody
        def  function_name(argument_list):
                Statement_1
                Statement_2
                .....
                Statement_n
        statements_classbody
```

Example

An example code for a class with members is given below.

```
1. class ex:
        x=100
        y=”abcdefgh”
   print(“Member value of x=”, ex.x)
   print(“Member value of y=”, ex.y)
```

2. class ex:
    def f1():
        print(“this is class method”)
    ex.f1()

Objects

In python, objects for classes can be created in two different ways. They are

- Attribute referencing
- instantiation.

Attribute Referencing

In attribute referencing, Python class properties can be accessed by using the class name itself. The syntax of the attribute referencing is

class_name.propertyname

Example

```
class ex:
    x=[]
ex.x=input("Enter the value of member x: ")
print("The value of ex.x is = ", ex.x)
```

Instantiation

During the instantiation process, an empty object is created and this object is assigned with the given object name. The syntax for the class instantiation is

object_name = classs_name()

After creating the class instantiation, the properties of the class are accessed with the following syntax

object_name.propertyname

Example

```
class ex:
    x=[]
a=ex()
a.x=input("Enter the value of member x: ")
print("The value of a.x is = ", a.x)
```

Constructors

In python programming, the __init__() method will be executed during object creation. Hence, this method is used as a constructor for the class. The __init__() method or the constructor is used to initialize the class variables. The constructor method should always take the self pointer as the first argument. The remaining arguments will be passed during the object creation. An example for the constructor is given below.

Example:

```
class ex:
        def __init__(self, arg):
                self.a=arg
obj=ex(300)
print("Value defined by the constructor=", obj.a)
```

Output:

```
Value defined by the constructor= 300
```

Destructors

In python programming, the __del__() method will be executed during objects get destroyed. When a program is finished execution, all the objects are managed by memory management automatically. The __del__() method is called when all references to the object have been deleted.

Example:

```
class ex:
        def __init__(self, arg):
                self.a=arg
        def __del__(self):
                print("Object is deleted.")
obj=ex(300)
print("Value defined by the constructor=", obj.a)
del obj
```

Output:

```
Value defined by the constructor= 300
Object is deleted.
```

Access Specifiers

Like other programming languages (Which supports OOPs concept) for example C++, Java; Python also uses Access specifiers which are used to restrict access to variables and methods of the class. Python has three access specifiers, which are Public, Protected and Private. Python uses underscore (“_”) symbol to determine the access specifiers for a specific data member or a member function of a class. Access specifiers have an important role to play in securing data from unauthorised access.

Public Access Specifiers

All data members which have declared initially without any prevention are public access specified. This type of data members are open to use within the class method or outside in any non member function. This type of data members have no underscore in the beginning of their name.

Example

```
class Student:
    schoolName = 'XYZ School' # class attribute

    def __init__(self, name, age):
        self.name=name # instance attribute
        self.age=age # instance attribute
```

All data members and member functions are public by default.

Protected Access Specifiers

Those data members which are declared as Protected are only accessible into the present class and derived class. In non member functions, Protected data members are not accessible. We use a single underscore in the beginning of their name to declare them as private.

Example

```
class Student:
    _schoolName = 'XYZ School' # protected class attribute
```

```
    def __init__(self, name, age):
        self._name=name  # protected instance attribute
        self._age=age # protected instance attribute
```

Private Access Specifiers

The members of a class which are declared as Private, are only accessible within the class only. This is the most secure access specifier. We use double underscore in the beginning of their name to declare them as private.

Example

```
class Student:
    __schoolName = 'XYZ School' # private class attribute

    def __init__(self, name, age):
        self.__name=name  # private instance attribute
        self.__salary=age # private instance attribute
    def __display(self):  # private method
         print('This is private method.')
```

Encapsulation

Encapsulation is one of the most fundamental concept of object-oriented programming. In OOPs, we need to wrap more than one data type and method together. This type of wrapping is called encapsulation. Encapsulation puts some restrictions on data variables and methods to access directly and can prevent accidental change. For this, we use access specifiers which we have read in the previous lecture.
A class is an example of encapsulation in which we wrap some data types and methods together.
Consider a real-life example of encapsulation, let assume there is a student who has a name, roll no., father's name, mobile number, address, etc.

Example:

```
class student:
        def __init__(self):
                self.name='Raj'
                self.rollno=123
                self.fathername='kailash'
                self.mobile=9237428321
```

Inheritance

Inheritance is used to derive the properties of one class to another class. In general, the properties from the base class can be derived into subclasses. So, the subclass contains the properties of the base class as well as its own properties.
There are three types of inheritance in Python. They are

Single inheritance
Multiple inheritance
Multilevel inheritance

Single inheritance

In single inheritance, the subclass can derive the properties of only one base class. The syntax for single inheritance is given below.

Syntax:

```
class base_class:
        #body of base class
class subclass_name(base_class):
        #body of subclass
```

Single inheritance can only inherit the public properties of the base class.

Example:

```
class a:
        x=200
class b(a):
        y=300
        def f1(self):
                print("Properties of sub class:", self.x, self.y)
obj1=b()
obj.f1()
```

Output:

```
Properties of sub class: 200 300
```

The hidden properties of the base class cannot be derived into subclass. If an attempt is made to use them, it will raise an error.

Multiple inheritance

In multiple inheritance, the subclass derives the properties of more than one base class. Multiple base classes are separated with ",". The syntax of multiple inheritance is given below.

Syntax:

```
class base_class_1:
        # body of base class 1
class base_class_2:
        # body of base class 2
class subclass_name(base_class_1, base_class_2):
        # body of the sub class
```

Multiple inheritance can only inherit the public properties of all base classes.

Example:

```
class a:
        x=200
class b:
        y=100
class c(a, b):
        z=300
        def f1(self):
                print("Properties of sub class:", self.x, self.y, self.z)
obj1=c()
obj.f1()
```

Output:

```
Properties of sub class: 200 100 300
```

Multilevel Inheritance

In multilevel inheritance, the properties of base class1 are derived into subclass1. The properties of subclass1 are derived into subclass2. The syntax of multilevel inheritance is given below.

Syntax:

```
class base_class_1:
```

```
        # body of base class 1
class sub_class_1(base_class_1):
        # body of sub class 1
class sub_class_2(sub_class_1):
        # body of the sub class 2
```

Multilevel inheritance can only inherit the public properties of all base classes and sub-class.

Example:

```
class a:
        x=200
        def disp(self):
                print("Properties of base class 1:", self.x)
class b(a):
        y=100
        def f1(self):
                self.disp()
                print("Properties of sub class 1:", self.y)
class c(a, b):
        z=300
        def f2(self):
                self.f2()
                print("Properties of sub class 2:", self.z)
obj1=c()
obj.f2()
```

Output:

```
Properties of base class 1: 200
Properties of sub class 1: 100
Properties of sub class 2: 300
```

Polymorphism

The word polymorphism means having many forms. In programming, polymorphism means the same function name but different structures being used for different types.

Polymorphism with class methods:

The following example shows the best way to explain polymorphism in class method.

Example:

```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()
    country.type()
```

Output:

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

In the above example, both classes India and USA have the same method name but print different messages. This is the best example of polymorphism with the class method.

Polymorphism with Inheritance:

In inheritance, polymorphism means child class and parent class have the same name method. In inheritance, the child class inherits the method from the parent class. If there is a method that is inherited from the parent class and they are not working as child class then we re-implement the method in the child class. This type of re-implementing a method in the child class is known as Method Overriding.

Example:

```
class Bird:
    def intro(self):
        print("There are many types of birds.")

    def flight(self):
        print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
    def flight(self):
        print("Sparrows can fly.")

class ostrich(Bird):
    def flight(self):
        print("Ostriches cannot fly.")

obj_bird = Bird()
```

```
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

Output:

```
There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.
```