Object Oriented Programming System

For introducing real word entities, in our programming world we need object oriented concept. In object oriented concept, we are having so many important terminologies like,

1. class
2. object

class : class may be define as a blueprint of an object, in which we are defining object properties and action/behaviors. Hear, properties can be represented by variables and action or behavior can be represented by methods.(Class may be define as a blue-print that contains attributes like variables and methods).

Syntax for defining any class:

Class class_name:
"doc string"
Contractors-
Variables-
Instance variable
Static variable
Local variable
Methods-
Instance method
Static method
Class method

Object : instance of a class is known as object.

Properties of oops concept:

1. abstraction (Data-security)
2. encapsulations (Data-security)
3. inheritance (Code-Reusability)
4. polymorphism (Code-Reusability)

What are Constructors :-- In any programming language, a constructor is a method that is automatically invoked whenever an instance (object) of a class is created. There is no need to explicitly call it. Typically, the constructor is used to perform any necessary initializations when the object is being created. In Python, the constructor is a method named __init__. The first parameter of this method should be self, which refers to the instance or object of the current class.

Syntax:
def __init__(self):
body of the constructor

In Python, Constructor is mandatory or not:--- No, it is not mandatory for a class to have a constructor. Whether a class includes a constructor depends entirely on the requirements. If any initialization is needed during object creation, then a constructor should be used. Otherwise, it is not necessary. A Python program remains valid even without a constructor.

```
class Test:
    def __init__(self):
        print("Constructor executed....!!!!!!!")
t = Test()

O/P:--
Constructor executed....!!!!!!!
```

Can constructor called explicitly? :---

Yes, we can call constructor explicitly with object name. But since the constructor gets executed automatically at the time of object creation, it is not recommended to call it explicitly.

```
class Student:
    def __init__(self):
        print("Constructor called.....")

obj = Student() # Constructor called implecitilly or automatically when we are creating object...
obj.__init__() # we are calling explicitally constructor method
obj.__init__()              # we are calling explicitally constructor method
obj.__init__()              # we are calling explicitally constructor method

O/P:--
Constructor called.....
Constructor called.....
```

```
Constructor called.....
Constructor called.....
```

Note: Including a constructor is not mandatory. If we do not include a constructor, Python will internally provide an empty constructor. This can be verified using the dir(class_name) built-in method.

```
# Constructor is not mandatory for any class, it is optional on the bases of our requirement.
Class Test:
    def m1(self):
        print("Instence method executed….!!!!!!")
t = Test()
t.m1()
print(dir(Test))

O/P:--
Instence method executed….!!!!!!
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'm1']
```

How many parameters we passed in constructor:---
Constructor can accept n number of parameters. It totally depends on our requirements. All values that need to be initialized during object creation should be passed to the constructor. The first parameter of the constructor should always refer to the current instance, which is typically denoted as self.

Without parameter (except self):

```
class Student:
    def __init__(self):
        print("Constructor called.....")
        print(self) #
```

```
stu = Student()

O/P:--
Constructor called.....
<__main__.Student object at 0x00000245668B3400>
```

Hear, self contains the current object address.

With parameters:

```
class Student:
    ''' This class is develop by Neeraj for demo'''
    def __init__(self,name,roll,marks):
        self.name=name
        self.roll=roll
        self.marks = marks
    def display(self):
        print("my name is", self.name)
        print("my roll no is", self.roll)
        print("my marks is", self.marks)

# help(Student)
obj1= Student("Neeraj",101,84)
print(obj1.name)
print(obj1.roll)
print(obj1.marks)
print(Student.__doc__)
obj1.display()

O/P:--
Neeraj
101
84
 This class is develop by Neeraj for demo
my name is Neeraj
my roll no is 101
my marks is 84
```

Multiple constructors in class:

We can define multiple constructors (__init__()) methods in a class but always last one is executed.

```
class Student:
    ''' This class is develop by Neeraj for demo'''
    def __init__(self,name,roll,marks):
        self.name=name
        self.roll=roll
        self.marks = marks
    def __init__(self,name,roll,marks,city):
        self.name=name
        self.roll=roll
        self.marks = marks
        self.city = city

    def display(self):
        print("my name is", self.name)
        print("my roll no is", self.roll)
        print("my marks is", self.marks)
        print("my city is", self.city)

# help(Student)
obj1= Student("Neeraj",101,84)
obj1= Student("Neeraj",101,84,"Bhopal")
print(obj1.name)
print(obj1.roll)
print(obj1.marks)
print(Student.__doc__)
obj1.display()
O/P:---

obj1= Student("Neeraj",101,84)
TypeError: Student.__init__() missing 1 required positional argument: 'city'


class Student:
    ''' This class is develop by Neeraj for demo'''
    def __init__(self,name,roll,marks):
        self.name=name
        self.roll=roll
```

```
        self.marks = marks
    def __init__(self,name,roll,marks,city):
        self.name=name
        self.roll=roll
        self.marks = marks
        self.city = city
    def display(self):
        print("my name is", self.name)
        print("my roll no is", self.roll)
        print("my marks is", self.marks)
        print("my city is", self.city)

# obj1= Student("Neeraj",101,84)
obj1= Student("Neeraj",101,84,"Bhopal")
print(obj1.name)
print(obj1.roll)
print(obj1.marks)
print(obj1.city)
obj1.display()

O/P:--
Neeraj
101
84
Bhopal
my name is Neeraj
my roll no is 101
my marks is 84
my city is Bhopal
```

Types of Variables in a Class in Python:---

Inside a class, we can have three types of variables. They are:

1. **Instance variables (object level variables)**
2. **Static variables (class level variables)**
3. **Local variables**

1.Instance Variables in Python:

If the value of a variable is changing from object to object then such variables are called as instance variables.

```python
# Instence Variable.....
class Student:
    ''' This class is develop by Neeraj for demo'''
    def __init__(self,name,roll,marks,city):
        self.name=name
        self.roll=roll
        self.marks = marks
        self.city = city
    def display(self):
        print("my name is", self.name)
        print("my roll no is", self.roll)
        print("my marks is", self.marks)
        print("my city is", self.city)

stu1 = Student("Neeraj",101,"90","Bhopal")
stu2 = Student("Rahul",102,"92","Indore")
print(stu1.name)
print(stu2.name)
stu1.display()
stu2.display()
print(stu1.__dict__)
print(stu2.__dict__)


O/P:--
Neeraj
Rahul
my name is Neeraj
my roll no is 101
my marks is 90
my city is Bhopal
my name is Rahul
my roll no is 102
my marks is 92
my city is Indore
{'name': 'Neeraj', 'roll': 101, 'marks': '90', 'city': 'Bhopal'}
{'name': 'Rahul', 'roll': 102, 'marks': '92', 'city': 'Indore'}
```

```python
# Instence Variable.....(By using instence method)
class Student:
    ''' This class is develop by Neeraj for demo'''
```

```
    def display(self,name,roll,marks,city):
        self.name=name
        self.roll=roll
        self.marks = marks
        self.city = city
        print("my name is", self.name)
        print("my roll no is", self.roll)
        print("my marks is", self.marks)
        print("my city is", self.city)
stu = Student()
stu.display("Neeraj",101,"90","Bhopal")
print(stu.name)
stu.display("Rahul",102,"92","Indore")
print(stu.name)
print(stu.__dict__)

O/P:--
my name is Neeraj
my roll no is 101
my marks is 90
my city is Bhopal
Neeraj
my name is Rahul
my roll no is 102
my marks is 92
my city is Indore
Rahul
{'name': 'Rahul', 'roll': 102, 'marks': '92', 'city': 'Indore'}
# Instence Variable.....(By using object)
class Student:
    def __init__(self):
        print("This is constructor")
    def m1(self):
        print("This is instance method")
t=Student()
t.m1()
t.a=10
t.b=20
t.c=55
print(t.a)
```

```
print(t.b)
print(t.c)
print(t.__dict__)

O/P:--
This is constructor
This is instance method
10
20
55
{'a': 10, 'b': 20, 'c': 55}
```

Accessing instance variables

The instance variable can be accessed in two ways:

1. By using self variable
2. By using object name

By using self variable:--- We can access instance variables within the class by using self variable.

```
# Access instence variable.....(by using self reference variable)
class Student:
  def __init__(self):
     self.a=10
     self.b=20
  def display(self):
     print(self.a)
     print(self.b)
s= Student()
s.display()
O/P:---
10
20
```

By using object name :--- We can access instance variables outside of the class by using object name.

```
# Access instence variable.....(by using object name)
class Student:
   def __init__(self):
      self.a=10
      self.b=20
```

```
s= Student()
print(s.a)
print(s.b)

O/p:--
10
20

# instance variable:----------
class Student:
    def __init__(self,name,age,city):
        # n,a,c instance variables....
        # instance variable declare inside the constructor
        self.n = name
        self.a = age
        self.c = city
    def show_details(self,num):
        print(self.n) # access instance variable inside instance methods..
        print(self.a) # access instance variable inside instance methods..
        print(self.c) # access instance variable inside instance methods..
        print(self.x) # access instance variable inside instance methods..
        self.n1 = num # instance variable declare inside instance method

obj1 = Student("Neeraj",37,"Bhopal")
obj1.x = 100  # instance variable declare outside of the class
obj1.show_details(151)
print(obj1.n1) # access instance variable outside of the class..
print(obj1.n) # access instance variable outside of the class..
print(obj1.a) # access instance variable outside of the class..
print(obj1.c) # access instance variable outside of the class..
```

2. Static Variables in Python

If the value of a variable is not changing from object to object, such types of variables are called static variables or class level variables. We can access static variables either by class name or by object name. Accessing static variables with class names is highly recommended than object names.

```
# Static variable.....
class Student:
    ''' This class is develop by Neeraj for demo'''
    School_name="SHSC"
    def __init__(self,name,roll,marks,city):
        self.name=name
        self.roll=roll
        self.marks = marks
        self.city = city
    def display(self):
        print("my name is", self.name)
        print("my roll no is", self.roll)
        print("my marks is", self.marks)
        print("my city is", self.city)

stu = Student("Neeraj",101,"90","Bhopal")
stu.display()
```

Declaring static variables in Python:

We can declare static variable in the following ways,

1. **Inside class and outside of the method**
2. **Inside constructor**
3. **Inside instance method**
4. **Inside class method (@classmethod):**
   We can declare and initialize static variable inside class method in two ways, one is using class name, other is using cls pre-defined variable.
5. **Inside static method (@staticmethod)**

```
# 1.  Declaring static variable  Inside class and outside of the method.....
class Demo:
   a=20
   def m(self):
       print("this is method")
       print(Demo.a)
       print(obj.a)
obj = Demo()
print(Demo.a)
print(obj.a)
obj.m()
```

```
O/P:--
20
20
this is method
20
20

# 2.  Declaring static variable Inside constructor
class Demo:
    def __init__(self):
        print("this is constructor")
        Demo.a=20
    def m(self):
         print("This is method")
         print(Demo.a)
         print(obj.a)
print(Demo.__dict__)
obj = Demo()
print(Demo.a)
print(obj.a)
obj.m()

O/P:--
this is constructor
20
20
This is method
20
20
# 3. Declaring static variable inside instance method
class Demo:
    def m(self):
         Demo.a=20
         print("This is method")
         print(Demo.a)
         print(obj.a)
obj = Demo()
obj.m()
print(obj.a)
print(Demo.a)
```

```
O/P:-
This is method
20
20
20
20

# 4(1). Declaring static variable inside class method
class Demo:
    @classmethod
    def m(cls):
        Demo.a=20
        print("This is class-method")
        print(Demo.a)
        print(obj.a)
obj = Demo()
obj.m()
print(obj.a)
print(Demo.a)

O/P:--
This is class-method
20
20
20
20

# 4(2). Declaring static variable inside class method
class Demo:
    @classmethod
    def m(cls):
        cls.a=20
        print("This is class-method")
        print(Demo.a)
        print(obj.a)
obj = Demo()
obj.m()
print(obj.a)
print(Demo.a)

O/P:--
```

```
This is class-method
20
20
20
20

# 5. Declaring static variable inside static method
class Demo:
    @staticmethod
    def m():
        Demo.a=20
        print("This is class-method")
        print(Demo.a)
        print(obj.a)
obj = Demo()
obj.m()
print(Demo.__dict__)
print(obj.a)
print(Demo.a)

O/P:--
This is class-method
20
20
20
20
```

Accessing static variables:-

The **static** variable can be accessed in two ways:

1. By using class name (highly recommended).
2. By using object name.

Accessing static variables either outside or inside of the class by using either class name or object name.

```
class Student:
    ''' This class is develop by Neeraj for demo'''
    School_name="SHSC"
    def __init__(self,name,roll,marks,city):
        self.name=name
```

```
        self.roll=roll
        self.marks = marks
        self.city = city
    def display(self):
        print("my name is", self.name)
        print("my roll no is", self.roll)
        print("my marks is", self.marks)
        print("my city is", self.city)
        print(Student.School_name) # Access static variable by using class name
        print(stu.School_name) # Access static variable by using object name

stu = Student("Neeraj",101,"90","Bhopal")
stu.display()
print(stu.School_name)  # Access static variable by using object name
print(Student.School_name) # Access static variable by using class name

O/P:--
my name is Neeraj
my roll no is 101
my marks is 90
my city is Bhopal
SHSC
SHSC
SHSC
SHSC
```

3.Local Variables in Python:

The variable which we declare inside of the method is called a local variable. Generally, for temporary usage we create local variables to use within the methods. The scope of these variables is limited to the method in which they are declared. They are not accessible outside of the methods.

```
# Local variable.....
class Demo:
    def m(self):
        a=10 #Local Variable
        print(a)
    def n(self):
        print(a) #'a' is local variable of m() so it raise error
```

```
d=Demo()
d.m()
d.n()

O/P:--
print(a) #'a' is local variable of m()
NameError: name 'a' is not defined
```

If you want to access local variable outside of the block then use global keyword.

```
class Demo:
  def m(self):
     global a
     a=10 #Local Variable
     print("instence methos m()")
     print(a)
  def n(self):
     print("instence methos n()")
     print(a) #'a' is local variable of m()
d=Demo()
d.m()
d.n()
print("access local variable outside of the class")
print(a)

O/P:--
instence methos m()
10
instence methos n()
10
access local variable outside of the class
10
```

Types of Methods in a Class

In python, we can classify the methods into three types from the perspective of object oriented programming.

1. Instance Methods
2. Class Methods
3. Static Methods

Instance Methods in Python:

Instance methods are methods which act upon the instance variables of the class. They are bound with instances or objects, that"s why called as instance methods. The first parameter for instance methods should be self variable which refers to instance. Along with the self variable it can contain other variables as well.

```
# instence method
class Demo:
  def __init__(self, a):
     self.a=a
  def m(self):
     print(self.a)
d=Demo(10)
d.m()

O/P:-
10
```

Class Methods in Python:

Class methods are methods which act upon the class variables or static variables of the class. We can go for class methods when we are using only class variables (static variables) within the method.

1. Class methods should be declared with @classmethod.
2. Just as instance methods have 'self' as the default first variable, class method should have 'cls' as the first variable. Along with the cls variable it can contain other variables as well.
3. We can access class methods by using class name or object reference.

```
# class method
class Test:
   x=200
   @classmethod
   def get_radius(cls):
      return cls.x
obj=Test()
print("class methos access by using class name")
print(Test.get_radius())
print("class methos access by using object name")
print(obj.get_radius())
```

```
O/P:--
class methos access by using class name
200
class methos access by using object name
200
```

Static Methods in Python: The static methods, in general, utility methods. Inside these methods we won't use any instance or class variables. No arguments like cls or self are required at the time of declaration.

1. We can declare static method explicitly by using @staticmethod decorator.
2. We can access static methods by using class name or object reference.

```
# static method
class Demo:
  @staticmethod
  def sum(x, y):
     print(x+y)
  @staticmethod
  def multiply(x, y):
     print(x*y)

obj = Demo()
print("static methos access by using class name")
Demo.sum(2, 3)
Demo.multiply(2,4)
print("static methos access by using class name")
obj.sum(2, 3)
obj.multiply(2,4)

O/P:--
static methos access by using class name
5
8
static methos access by using class name
5
8
```

FEATURES OF OOPS:

1. Inheritance (Code reusability)
2. Polymorphism (Code reusability)
3. Encapsulation (Data security)
4. Abstraction (Data security)

Inheritance:-----

Inheritance is the passing of properties to someone. In programming languages, the concept of inheritance comes with classes.

1. Creating new classes from already existing classes is called inheritance.
2. The existing class is called a super class or base class or parent class.
3. The new class is called a subclass or derived class or child class.
4. Inheritance allows sub classes to inherit the variables, methods and constructors of their super class.

Advantages of Inheritance:

1. The main advantage of inheritance is code re-usability.
2. Time taken for application development will be less.
3. Redundancy (repetition) of the code can be reduced.

Inheritance in Python allows a class (child class) to inherit the properties and methods of another class (parent class). This promotes code reusability and establishes relationships between different classes.

Types of Inheritance in Python:

1. Single Inheritance – A child class inherits from a single parent class.
2. Multiple Inheritance – A child class inherits from more than one parent class.
3. Multilevel Inheritance – A child class inherits from another child class.
4. Hierarchical Inheritance – Multiple child classes inherit from a single parent class.
5. Hybrid Inheritance – A combination of different types of inheritance.

```
# Single Inheritance:-----
class Parent:
    def display(self):
        print("This is the Parent class")

class Child(Parent):
```

```
    def show(self):
        print("This is the Child class")

obj = Child()
obj.display()
obj.show()

# Multiple Inheritance:----
class Parent1:
    def method1(self):
        print("This is Parent1")

class Parent2:
    def method2(self):
        print("This is Parent2")

class Child(Parent1, Parent2):
    def method3(self):
        print("This is Child")

obj = Child()
obj.method1()
obj.method2()
obj.method3()

# Multilevel Inheritance:---
class Grandparent:
    def grandparent_method(self):
        print("This is the Grandparent class")

class Parent(Grandparent):
    def parent_method(self):
        print("This is the Parent class")

class Child(Parent):
    def child_method(self):
        print("This is the Child class")

obj = Child()
obj.grandparent_method()
```

```
obj.parent_method()
obj.child_method()

# Hierarchical Inheritance:----
class Parent:
    def parent_method(self):
        print("This is the Parent class")

class Child1(Parent):
    def child1_method(self):
        print("This is Child1")

class Child2(Parent):
    def child2_method(self):
        print("This is Child2")

obj1 = Child1()
obj1.parent_method()
obj1.child1_method()

obj2 = Child2()
obj2.parent_method()
obj2.child2_method()

# Hybrid inheritance:---
class A:
    def show(self):
        print("Class A")

class B(A):
    def display(self):
        print("Class B")

class C(A):
    def output(self):
        print("Class C")

class D(B, C):  # Inheriting from B and C (Multiple Inheritance)
    def final(self):
        print("Class D")
```

```
obj = D()
obj.show()    # Calls method from A
obj.display() # Calls method from B
obj.output()  # Calls method from C
obj.final()   # Calls method from D

# Checking Method Resolution Order (MRO)
print(D.__mro__)

# Method Overriding in Inheritance
class Parent:
    def show(self):
        print("This is the Parent class")

class Child(Parent):
    def show(self):  # Overriding the parent method
        print("This is the Child class")

obj = Child()
obj.show()


# Using super() to Call Parent Class Methods
class Parent:
    def show(self):
        print("This is the Parent class")

class Child(Parent):
    def show(self):
        super().show()  # Calls the parent class method
        print("This is the Child class")

obj = Child()
obj.show()
```

Method Resolution Order (MRO):

In situations involving multiple inheritance, a particular attribute or method is initially searched in within the current class. If it is not located in the current class, the search proceeds to the parent classes following a depth-first, left-to-right fashion. This sequence of searching is referred to as the Method Resolution Order (MRO).

```
# Method Resolution Order (MRO).
class A:
    def m1(self):
        print("Method m1() is called from class A")

class B:
    def m1(self):
        print("Method m2() is called from class B")

class C(B,A):
    def m3(self):
        print("Method m3() is called")

obj = C()
obj.m1()
obj.m3()

O/P:--
Method m2() is called from class B
Method m3() is called
# Method Resolution Order (MRO).
class A:
    def m1(self):
        print("Method m1() is called from class A")
class B:
    def m1(self):
        print("Method m2() is called from class B")
class C(A,B):
    def m3(self):
        print("Method m3() is called")
obj = C()
obj.m1()
obj.m3()

O/P :--
```

```
Method m1() is called from class A
Method m3() is called
```

```
# Method Resolution Order (MRO).
class A:
  def m1(self):
    print("m1 from A")
class B(A):
  def m1(self):
    print("m1 from B")
class C(A):
  def m1(self):
    print("m1 from C")
class D(B, C):
  def m1(self):
    print("m1 from D")

print(A.mro())
print(B.mro())
print(C.mro())
print(D.mro())

O/P:--
[<class '__main__.A'>, <class 'object'>]
[<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

Polymorphism:-----

The word 'Poly' means many and 'Morphs' means forms. The process of representing "one form in many forms" is called a polymorphism.

1. Duck Typing Philosophy of Python:--

"If it walks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."

```
class Duck:
   def talk(self):
      print("Quack.. Quack")
class Dog:
   def talk(self):
      print("Bow...Bow")
class Cat:
   def talk(self):
      print("Moew...Moew ")

def my_func(obj):
   obj.talk()

duck = Duck()
my_func(duck)

cat = Cat()
my_func(cat)

O/P:--
Quack.. Quack
Moew...Moew
```

In the above program, the function ‘m’ takes an object and calls for the talk() method of it. With duck typing, the function is not worried about what object type of object it is. The only thing that matters is whether the object has a method with name ‘talk()’ supported or not.

2. Overloading:-

1. **Operator Overloading**
2. **Method Overloading**
3. **Constructor Overloading**

Operator Overloading:

```
print(10+20)
print("Neeraj"+" "+"Kumar")

print(10*5)
print("neeraj"*5)
```

```
O/P:-
30
Neeraj Kumar
50
neerajneerajneerajneerajneeraj
```

```
class Book:
   def __init__(self, pages):
       self.pages=pages
obj1=Book(100)
obj2=Book(200)
print(type(obj1))
print(type(obj2))
print(type(obj1.pages))
print(type(obj2.pages))
print(obj1.pages + obj2.pages)
print((obj1.pages).__add__(obj2.pages))

O/P:--
<class '__main__.Book'>
<class '__main__.Book'>
<class 'int'>
<class 'int'>
300
300
class A:
    def __init__(self,x):
        self.x = x
    def __add__(self,other):
        return self.x+other.x
class B:
    def __init__(self,x):
        self.x = x

a = A(10)
b = B(20)
print(a+b) # a.__add__(10+20)

O/P:-
```

```
30

class A:
    def __init__(self,x):
        self.x = x
    def __add__(self,other):
        return self.x+other.x
class B:
    def __init__(self,x):
        self.x = x

a = A(10)
b = B(20)
print(b+a) # b.__add__(10+20) not define __add__ method in B class so it gives error.

O/P:-
TypeError: unsupported operand type(s) for +: 'B' and 'A'
```

Method Overloading

If 2 methods have the same name but different types of arguments, then those methods are said to be overloaded methods. If we are trying to declare multiple methods with the same name and different number of arguments, then Python will always consider only the method which was last declared. **But in Python Method overloading is not possible**. If we are trying to declare multiple methods with the same name and different number of arguments, then Python will always consider only the last method.

Overriding:-

1. **Method overriding**
2. **Constructor overloading**

Overriding

All the members available in the parent class, those are by-default available to the child class through inheritance. If the child class is not satisfied with parent class implementation, then child class is allowed to redefine that method in the child class based on its requirement. This concept is called overriding.Overriding concept applicable for both methods and constructors.

1. **Method Overriding**
2. **Constructor Overriding**

Method Overriding:-

```
class A:
  def display(self):
    print('Display fron class A')
  def show(self):
    print('Show fron class A')
class B(A):
  def display1(self):
    print("Display fron class B")
  def show(self):
    print('Show fron class B')

c=B()
c.display1()
c.show()
c.display()
```

Constructor Overriding

1. If child class does not have constructor, then parent class constructor will be executed at the time of child class object creation.

2. If child class has a constructor, then child class constructor will be executed at the time of child class object creation.

3. From child class constructor we can call parent class constructor by using super() method

```
class Person:
  def __init__(self, name, age):
    self.name=name
    self.age=age
class Employee(Person):
  def __init__(self, name, age, eno, esal):
    super().__init__(name, age)
    self.eno=eno
    self.esal=esal
  def display(self):
    print('Employee Name:', self.name)
```

```
        print('Employee Age:', self.age)
        print('Employee Number:', self.eno)
        print('Employee Salary:', self.esal)
obj1=Employee('Neeraj', 36, 101, 26000)
obj1.display()
obj2=Employee('Rahul',37,102,36000)
obj2.display()

O/P:--
Employee Name: Neeraj
Employee Age: 36
Employee Number: 101
Employee Salary: 26000
Employee Name: Rahul
Employee Age: 37
Employee Number: 102
Employee                                Salary:                                 36000
```

Encapsulation

Encapsulation is the concept of wrapping data and methods that work with data in one unit. Encapsulation is achieved through the use of access modifiers such as 'public', 'private', and 'protected'.

```
class parent:
    def __init__(self):
        self._p = 78

class child(parent):
    def __init__(self):
        parent.__init__(self)
        print ("We will call the protected member of base class: ", self._p)
        self._p = 433
        print ("we will call the modified protected member outside the class: ",self._p)


obj_1 = parent()
obj_2 = child()
print ("Access the protected member of obj_1: ", obj_1._p)
print ("Access the protected member of obj_2: ", obj_2._p)
```

```
O/P:-
We will call the protected member of base class:  78
we will call the modified protected member outside the class:  433
Access the protected member of obj_1:  78
Access the protected member of obj_2:  433
```

Abstraction :---

Abstraction is used to hide the internal functionality of the function from the users. The users only interact with the basic implementation of the function, but inner working is hidden. In Python, an abstraction is used to hide the irrelevant data/class in order to reduce the complexity. It also enhances the application efficiency.
Abstraction in python is defined as a process of handling complexity by hiding unnecessary information from the user. This is one of the core concepts of object-oriented programming (OOP).
Important terminologies:--

1. **Abstract class**
2. **Abstract methods**
3. **Concrete methods**

Abstract class:-- An abstract class is the class which contains one or more abstract methods. An abstract method is the one which is just defined but not implemented.

1. Every abstract class in Python should be derived from the ABC class which is present in the abc module. Abstract class can contain Constructors, Variables, abstract methods, non-abstract methods, and Subclass.
2. Abstract methods should be implemented in the subclass or child class of the abstract class.
3. If in subclass the implementation of the abstract method is not provided, then that subclass, automatically, will become an abstract class.
4. Then, if any class is inheriting this subclass, then that subclass should provide the implementation for abstract methods.
5. Object creation is not possible for abstract class.
6. We can create objects for child classes of abstract classes to access implemented methods.

Abstract methods:-- A method which has only method name and no method body, that method is called an unimplemented method. They are also called as non-concrete or abstract methods.

1. By using @abstractmethod decorator we can declare a method as an abstract method.
2. @abstractmethod decorator presents in abc module. We should import the abc module in order to use the decorator.
3. Since abstract method is an unimplemented method, we need to put a pass statement, else it will result in error.
4. Class which contains abstract methods is called an abstract class.
5. For abstract methods, implementation must be provided in the subclass of abstract class

Here, I am changing the name of abc lib to my_abc and I am doing some changes in that particular library.

my_abc:-

(Link for abc libreary:-- https://github.com/python/cpython/blob/3.12/Lib/abc.py)

```
from my_abc import ABC , abstractmethod

class Fruit(ABC):

    @abstractmethod
    def fruit_shape(self):
        pass

class Mango(Fruit):
    description = "King of fruits"
    def __init__(self, x,y,z):
        self.desc=Mango.description
        self.shape = x
        self.taste = y
        self.color = z

    def fruit_shape(self):
        print(self.desc)
        print(self.shape)


    def fruit_color(self):
        print(self.color)

    def fruit_taste(self):
        print(self.taste)

# obj = Fruit() # You can not create object for any abstract class.
obj1 = Mango("Oval","yellow","sweet")
```

```
obj1.fruit_shape()
obj1.fruit_color()
obj1.fruit_taste()

O/P:--
Welcome to home screen
King of fruits
Oval
sweet
yellow
```

Concrete methods:- A method which has a both method name and method body, that method is called an implemented method. They are also called concrete methods or non-abstract methods.