

# Constructor Overriding

In [ ] : Constructor Overriding --> When the object **is** created, it will call child **class** constructor.  
If we want to call the base **class** constructor **and** initialize the variable instvar.  
we use **super()** function.  
--> It **is** possible **in** Python

## Demostration of Constructor Overriding with super() method

```
In [1]: class Parent:
def __init__(self):
    print("Parent class constructor will be executed")
class Child(Parent):
def __init__(self):
    print("Child Class constructor")
c=Child()
q=Parent()
```

Child Class constructor  
Parent class constructor will be executed

```
In [2]: class Parent:
def __init__(self):
    print("Parent class constructor will be executed")
class Child(Parent):
    pass
c=Child()
q=Parent()
```

Parent class constructor will be executed  
Parent class constructor will be executed

```
In [3]: class Parent:
def __init__(self):
    print("Parent class constructor will be executed")
class Child(Parent):
def __init__(self):
    super().__init__()
    print("Child Class constructor")
c=Child()
```

Parent class constructor will be executed  
Child Class constructor

```
In [4]: class Grandfather:
a=20
b=300
def __init__(self):
    print("GrandFather class Constructor")
    print(self.b)
class Father(Grandfather):
a=30
def __init__(self):
    super().__init__()
    print("Father Constructor")
    print(Father.a)
    print(super().a)

class child(Father):
a=10
def __init__(self):
    super().__init__()
    print("Child class constructor") #child class constructor
    print(super().a) #30
    print(super().a)
    print(child.a) #10
c=child()
```

GrandFather class Constructor  
300  
Father Constructor  
30  
20  
Child class constructor  
30  
30  
10

```
In [5]: class Traditional_Phone:
x=200
y=300
def __init__(self):
    print("Traditional_Phone Constructor")
    print(self.x)
def m1(self):
    print("traditional Phone m1")
class Phone(Traditional_Phone):
def __init__(self):
    print("Phone Constructor")
    super().__init__()
    super().m1()

def m2(self):
    print("phone m2 method")
    super().m1()

x = Phone()
x.m2()
```

Phone Construtor  
Traditional\_Phone Constructor  
200  
traditional Phone m1  
phone m2 method  
traditional Phone m1

```
In [15]: class Traditional_Phone:
x=200
y=300
def __init__(self):
    print("Traditional_Phone Constructor")
    print(self.x)
def m1(self):
    print("traditional Phone m1")

def m2(self):
    print("phone m2 method")
class Phone(Traditional_Phone):
def __init__(self):
    print("Phone Constructor")

x = Phone()
x.m2()
```

Phone Constructor  
phone m2 method

```
In [6]: class Test:
def m1(self):
    print("M1 Method")
x=Test()
x.m1()
```

M1 Method

## Note

In [ ] : Method Overloading --> Not possible (Variable length argument)  
Operator Overloading --> Possible  
Constructor overloading --> **not** possible(variable length argument)  
Method Overriding --> Possible (Parent **is** having one method child want to redefine that method)  
Constructor overriding --> Possible (Parent **class** and child **class**)

## Abstraction

In [ ] : Abstraction --> Hiding the irrelevant thing **from** the user **is** the concept of Abstraction  
--> 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) languages.  
--> Hiding of irrelevant data **from** the user. such that user can only access the properties **and** behaviour of that functionality without knowing internal implementation of that functionality

Example:  
Television  
ATM Machine  
Laptop  
Mobile  
Fan  
Web Applications  
Android Application

## Meaning of Abstract

In [ ] : Abstract --> Abstract **is** something which does **not** talk about completeness . It **is** just partial implementation of anything.

## Abstract Method

In [ ] : Abstract method --> Sometimes we don't know the implementation of a method still we need to declare a method such type of method are known **as** abstract method. (abstrct method have only declaration **not** implementation)  
--> In python **if** you want to declare abstract method then you need use @abstractmethod decorator.  
--> @abstractmethod decorator **is** present **in** abc module. **for** declaring any method **as** abstract you need to **import** abc module(Abstract base class)

Example:  
@abstractmethod  
def getnoofwheel():  
 pass

## Example

```
In [ ] : from abc import *
class test:
    @abstractmethod
    def m1(self):
        pass
```

Note: Child **class** is responsible to implement abstract method of parent class.

## Abstract Class

In [ ] : Abstract Class --> Sometimes we don't know the complete implementation of a class still we need to declare or define a class such type of classes are known **as** abstract classes.  
--> Every abstract **class** is a child **class** of **ABC class** which **is** present **in** abc module.  
--> In abstract **class** it **is** mandatory that atleast one method should be abstract  
--> We cannot create the object of abstract class.

## Implementation of Abstract Class and Abstract Method

```
In [8]: from abc import *
class Vechile(ABC):
    @abstractmethod
    def get_no_wheels(self):
        pass
    def engine(self):
        return "230CC"
class Bus(Vechile):
    def get_no_wheels(self):
        return 8

class Auto(Vechile):
    def get_no_wheels(self):
        return 4
class Bike(Vechile):
    def get_no_wheels(self):
        return 2

c=Bus()
c.get_no_wheels()
```

Out[8]: 8

## Important Conditions

In [ ] : **for** abstract **class** below two conditions are **True**. **and if** both the conditions are true then we cannot create an object of that **class** as that **class** is an abstract class:

1. Class must be a child **class** of ABC
2. Class must having atleast one abstract method

## Important Cases

```
In [ ] : Case1:
from abc import *
class test:
    pass
```

Note --> It **is not** an abstract class. We can create the object of the above **class**

```
In [ ] : Case 2:
from abc import *
class Test(ABC):
    pass
```

Note --> In the above **class** we can create the object even it **is** derived **from** abc **class** because it doesnot contain any abstract method

```
In [ ] : Case 3:
from abc import *
class test(ABC):
    @abstractmethod
    def m1(self):
        pass
```

Note --> We cannot create the object of it the reason **is** it **is** a child **class** of ABC **and** also having atleast one abstract method

```
In [ ] : Case 4:
from abc import *
class test(ABC):
    @abstractmethod
    def m1(self):
        pass
class child(test):
    @abstractmethod
    def m1(self):
        pass
```

c=child()  
Note --> Child **is** responsible to implement abstract method **if** child **is not** implementing it then the child **class** is also an abstract **class** so we cannot create the object of child **as well as** parent

```
In [ ] : Case 5:
from abc import *
class test(ABC):
    @abstractmethod
    def m1(self):
        pass
class child(test):
    pass
```

Note --> **If** we are **not** creating the object of above code then it **is** valid because syntactically it **is** correct but **if** you create an object then it **is not** possible.

## Interfaces

In [ ] : Interface --> Interface concept **is not in** Python. But In java we have a syntax **for** interfaces.  
whereas **in** python we don't have something like this.  
--> **If** any abstract **class** is having all methods **as** abstract such type of abstract **class** are known **as** interfaces.  
--> In python there **is** no any syntax **for** interface but we can achieve interface **with** the help of abstract **class** and method

## Implementation of Interfaces with the help of Abstract Classes.

```
In [27]: from abc import *
class DBInterface(ABC):
    @abstractmethod
    def connect(self):
        pass
    @abstractmethod
    def disconnect(self):
        pass
class Oracle(DBInterface):
    def connect(self):
        print("Connected Successfully")
    def disconnect(self):
        print("Disconnected Successfully")
x=Oracle()
x.connect()
x.disconnect()
print(type(x))
```

Connected Successfully  
Disconnected Successfully  
<class '\_\_main\_\_.Oracle'>