# (7.7) Abstract Method and Class / Model Class, Interfaces

## Abstract Methods

Sometimes we don't know about implementation,still we can declare a method. Such type of methods are called abstract methods.i.e abstract method has only declaration but not implementation.

In python we can declare abstract method by using @abstractmethod decorator as follows.

```
@abstractmethod
def m1(self): pass
```

@abstractmethod decorator present in abc module. Hence compulsory we should import abc module,otherwise we will get error.

**abc : abstract base class module**

abstract method, which ment to be over-rided in derived class

In [4]:
```
from abc import *

class Test:
    @abstractmethod
    def m1(self):
        pass
```

## Abstract Class

Some times implementation of a class is not complete,such type of partially implementation classes are called abstract classes.

a class which can not instantiate

we can not create object of instance class

Abstract Classes are ment to be inherited

Every abstract class in Python should be derived from ABC class which is present in abc module.

### i) Creation Of Abstract Class

#### Case-1

In [6]:
```
from abc import *
class Test:
    pass
```

```
t=Test()
```

In the above code we can create object for Test class b'z it is concrete class and it does not conatin any abstract method.

## Case-2

In [7]:
```
from abc import *
class Test(ABC):
    pass

t=Test()
```

In the above code we can create object,even it is derived from ABC class,b'z it does not contain any abstract method.

## Case-3

In [9]:
```
from abc import *
class Test:
    @abstractmethod
    def m1(self):
        pass

t=Test()
```

We can create object even class contains abstract method b'z we are not extending ABC class.

## Case-4

In [8]:
```
from abc import *
class Test(ABC):
    @abstractmethod
    def m1(self):
        pass

t=Test()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_1004/930484879.py in <module>
      6         pass
      7
----> 8 t=Test()

TypeError: Can't instantiate abstract class Test with abstract methods m1
```

In [20]:
```
from abc import *
class Test(ABC):
    @abstractmethod
    def m1(self):
        pass
    def m2(self):
        print('Hello')

t=Test()
t.m1()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_1004/1327908807.py in <module>
      7         print('Hello')
      8
----> 9 t=Test()
     10 t.m1()

TypeError: Can't instantiate abstract class Test with abstract methods m1
```

If a class contains atleast one abstract method and if we are extending ABC class then instantiation is not possible.

# ii) Inheritation of Abstract class

Parent class abstract methods should be implemented in the child classes. otherwise we cannot instantiate child class.

In [21]:
```
from abc import *
class Vehicle(ABC):
    @abstractmethod
    def noofwheels(self):
        pass

class Bus(Vehicle): pass
```

Note: If we are extending abstract class and does not override its abstract method then child class is also abstract and instantiation is not possible.

In [22]:
```
from abc import *
class Vehicle(ABC):
    @abstractmethod
    def noofwheels(self):
        pass

class Bus(Vehicle): pass
b = Bus()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_1004/2863302957.py in <module>
      6
      7 class Bus(Vehicle): pass
----> 8 b = Bus()

TypeError: Can't instantiate abstract class Bus with abstract methods noofwheels
```

Note: Abstract class can contain both abstract and non-abstract methods also

In [23]:
```
from abc import *
class Vehicle(ABC):
    @abstractmethod
    def noofwheels(self):
        pass

class Bus(Vehicle):
    def noofwheels(self):
        return 7

class Auto(Vehicle):
```

```
    def noofwheels(self):
        return 3

b=Bus()
print(b.noofwheels())#7
a=Auto()
print(a.noofwheels())#3
```

```
7
3
```

# Problem On Abstract Class

In [24]:
```
from abc import ABC, abstractmethod

class Car(ABC): # now Car is an abstract class
    #-----------------compulsory method-----------------
    @abstractmethod
    def abs(self):
        """abs must be there with today's norms"""
    @abstractmethod
    def air_bags(self):
        """every should have two air bags"""
    @abstractmethod
    def pollution(self):
        """every engine of car should follow bs6 norms"""
    @abstractmethod
    def child_safety(self):
        """every should have child and speed lock"""
    #---------------------------------------------------
    def ac(self):
        """you can have or can not have ac in car"""
    def hill_control(self):
        """you can give hill control system"""
```

In [25]:
```
c = Car()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_1004/96233051.py in <module>
----> 1 c = Car()

TypeError: Can't instantiate abstract class Car with abstract methods abs, air_bags, child
_safety, pollution
```

In [26]:
```
class Nano(Car):
    def __init__(self, modal_name, price):
        self.modal_name = model_name
        self.price = price
    def __str__(self):
        return self.model_name
```

In [27]:
```
a = Nano()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_1004/612694336.py in <module>
----> 1 a = Nano()
```

```
TypeError: Can't instantiate abstract class Nano with abstract methods abs, air_bags, chil
d_safety, pollution
```

In [28]:
```
class Nano(Car):
    def __init__(self, model_name, price):
        self.model_name = model_name
        self.price = price
    def __str__(self):
        return self.model_name
    def abs(self):
        print("Nano has 300 mm disk at rear with ABS system")
    def air_bags(self):
        print("Nano has 2 air bags on front seats for basic safety")
    def pollution(self):
        print("Nano has passed all bs6 pollution emission norms ")
    def child_safety(self):
        print("Auto lock at speed of 60 kmph")
```

In [33]:
```
a = Nano("Nano Base Model 2021", 150000)
print(a)
a.air_bags()
a.pollution()
a.abs()
```

```
Nano Base Model 2021
Nano has 2 air bags on front seats for basic safety
Nano has passed all bs6 pollution emission norms
Nano has 300 mm disk at rear with ABS system
```

In [32]:
```
a.hill_control()
a.ac()
```

# Interfaces

In general if an abstract class contains only abstract methods such type of abstract class is considered as interface.

In [37]:
```
from abc import *
class DBInterface(ABC):
    @abstractmethod
    def connect(self):pass
    @abstractmethod
    def disconnect(self):pass

class Oracle(DBInterface):
    def connect(self):
        print('Connecting to Oracle Database...')
    def disconnect(self):
        print('Disconnecting to Oracle Database...')

class Sybase(DBInterface):
    def connect(self):
        print('Connecting to Sybase Database...')
    def disconnect(self):
        print('Disconnecting to Sybase Database...')

dbname=input('Enter Database Name:')
classname=globals()[dbname]
x=classname()
```

```
x.connect()
x.disconnect()
```

```
Enter Database Name:Oracle
Connecting to Oracle Database...
Disconnecting to Oracle Database...
```

# Note: Concreate class vs Abstract Class vs Inteface

1. If we dont know anything about implementation just we have requirement specification then we should go for interface.

2. If we are talking about implementation but not completely then we should go for abstract class.(partially implemented class)

3. If we are talking about implementation completely and ready to provide service then we should go for concrete class.

In [ ]: