Abstract classes and abstract methods (abc module)
This *module* provides the infrastructure for defining abstract base classes (ABCs) in *Python. It is a default module which comes with python software.*

Abstract method
A method without implementation is called abstract method (OR) empty method is called abstract method.
Abstract method defines a specification or rule which has to be followed by implementation class or inherited class.
When more than one class having same role with different implementation, then that method is declared as abstract.

Syntax:

```
@abstractmethod
def <method-name>(self,arg1,arg2,arg3,…):
        pass
```

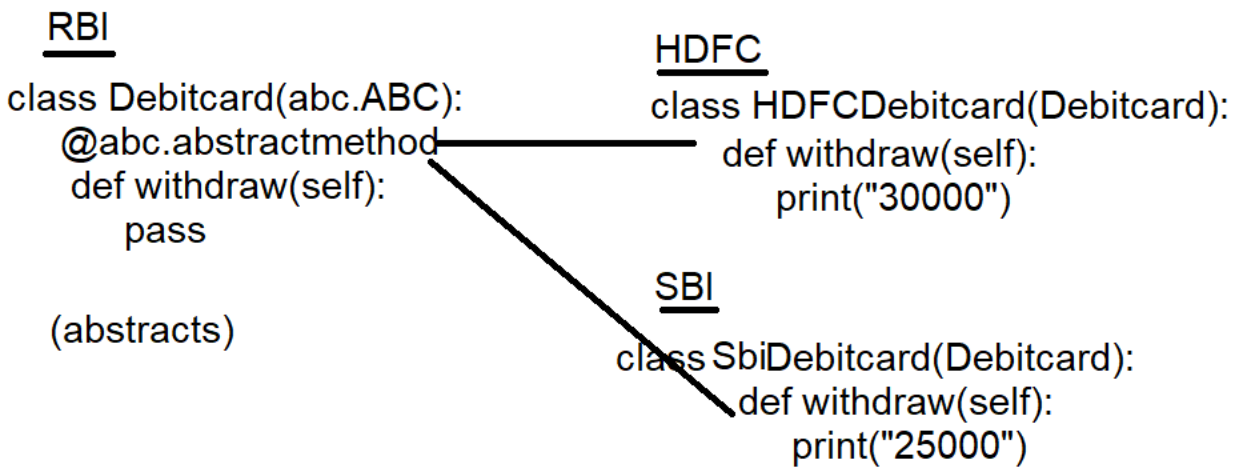
Abstract method must override by inherited class.
Abstract method is defined inside abstract class.

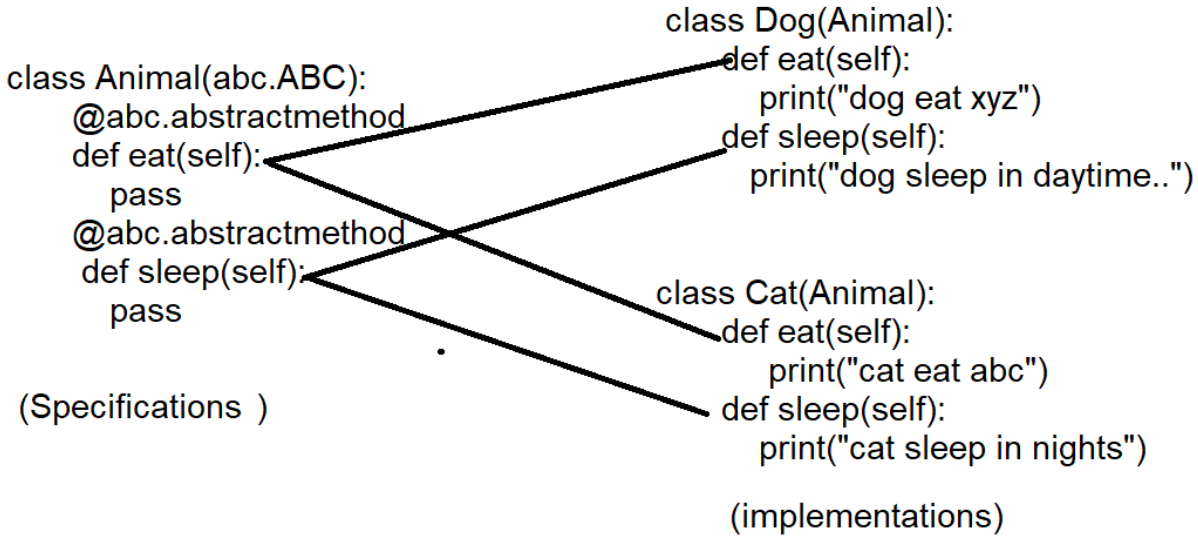
Abstract class
Abstract class is a collection of abstract methods and non abstract methods.

Syntax:

```
class <class-name>(abc.ABC):
        abstract methods
        non abstract methods
```

abstract class is inherited but it is not used for creating object.





Example:

```
import abc
class A(abc.ABC):
    @abc.abstractmethod
    def m1(self):
        pass
class B(A):
    def m1(self):
        print("m1 of B class")
```

```
class C(A):
    def m2(self):
        print("m2 of B class")


objb=B()
objb.m1()
objc=C()
```

Output:

```
Traceback (most recent call last):
  File "C:\Users\nit\PycharmProjects\pythonProject1\ooptest44.py", line 17,
in <module>
    objc=C()
         ^^^
TypeError: Can't instantiate abstract class C with abstract method m1
m1 of B class

Process finished
```

Example:

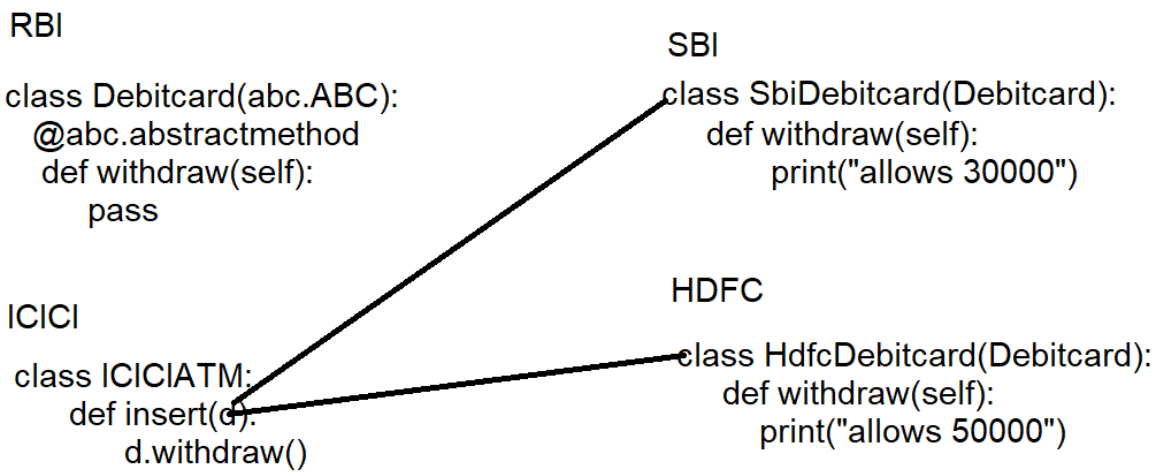
```
import abc
class Shape(abc.ABC):
    def __init__(self):
        self.dim1=None
        self.dim2=None
    def readDim(self):
        self.dim1=float(input("Enter Dim1 "))
        self.dim2=float(input("Enter Dim2 "))
    @abc.abstractmethod
    def findArea(self):
        pass

class Triangle(Shape):
    def __init__(self):
        super().__init__()
    def findArea(self):
        a=0.5*self.dim1*self.dim2
```

```
        return a


class Rectangle(Shape):
    def __init__(self):
        super().__init__()
    def findArea(self):
        a=self.dim1*self.dim2
        return a


t1=Triangle()
r1=Rectangle()
t1.readDim()
area1=t1.findArea()
print(f'Area of triangle1 is {area1:.2f}')
r1.readDim()
area2=r1.findArea()
print(f'Area of rectangle is {area2:.2f}')
```

Output:

```
Enter Dim1 1.2
Enter Dim2 1.5
Area of triangle1 is 0.90
Enter Dim1 1.5
Enter Dim2 2.5
Area of rectangle is 3.75
```

Using abstract classes and abstract method a program can achieve runtime polymorphism.
An ability of a reference variable change its behavior based on the type of object assigned is called runtime polymorphism or duck typing.



Example:

```
import abc
class Sim(abc.ABC):
    @abc.abstractmethod
    def connect(self):
        pass


class JioSim(Sim):
    def connect(self):
        print("connect to jio network")

class AirtelSim(Sim):
    def connect(self):
        print("connect to airtel network")

class Mobile:
    def insert(self,s):
        s.connect()


iphone=Mobile()
jiosim1=JioSim()
airtelsim1=AirtelSim()
iphone.insert(jiosim1)
```

iphone.insert(airtelsim1)

Output:
connect to jio network
connect to airtel network

Inner classes or nested classes
A class within class is called inner class or nested class.
Inner classes are two types

1. Member class
2. Local class

If a class is defined as a member of class, it is called as member class.
If a class is defined inside method or block is called local class.

```
class <outer-class>:
      class <member-class>:
            variables
            methods
       def method-name(self):
           class <local class>:
              variables
              methods
```

Member class is used anywhere within outer class
Local class is used within declared method.