Abstract classes and abstract methods

“abc” module of python provides classes and methods for working with abstract classes and methods.

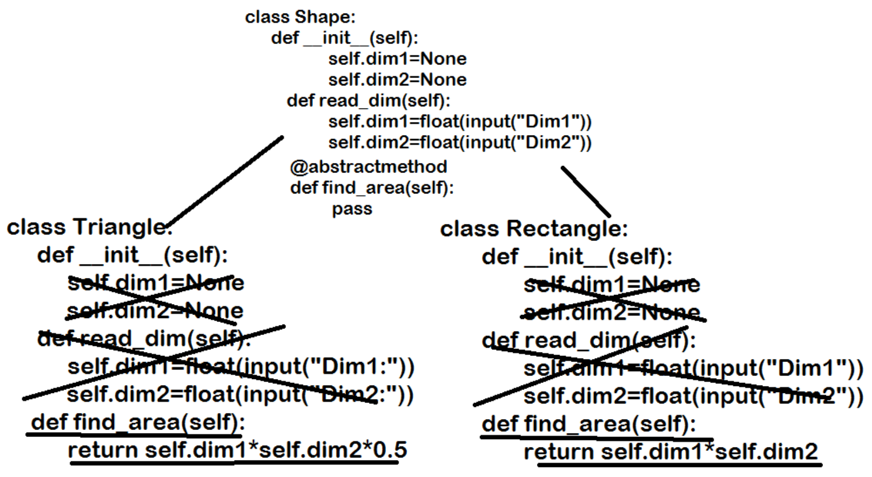
abstract method
A method without implementation is called abstract method
Abstract method defines set of rules and regulations which has to be followed by every derived class.
Abstract method defines a protocol, which has to be followed by every derived class or sub class.
We can define abstract method only within abstract class.
@abstractmethod decorator is used to define method as abstract

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

abstract method must be override inside derived class.

Abstract class
abstract class is collection of abstract methods and non abstract methods.
Abstract class define set of rules and regulations used for developing a class.
Abstract class is an abstract data type(ADT).
A data type which allows to build similar data types is called abstract data type.
Abstract class is inherited but we cannot use abstract class for creating object.

Syntax:
```
class <class-name>(abc.ABC):
        variables
        methods
                non abstract methods (object level methods,class level, static
methods)
        abstract methods
```



```
class Shape:
    def __init__(self):
        self.dim1=None
        self.dim2=None
    def read_dim(self):
        self.dim1=float(input("Dim1"))
        self.dim2=float(input("Dim2"))
    @abstractmethod
    def find_area(self):
        pass
```

```
class Triangle:
    def __init__(self):
        ~~self.dim1=None~~
        ~~self.dim2=None~~
    ~~def read_dim(self):~~
        ~~self.dim1=float(input("Dim1:"))~~
        ~~self.dim2=float(input("Dim2:"))~~
    def find_area(self):
        return self.dim1*self.dim2*0.5
```

```
class Rectangle:
    def __init__(self):
        ~~self.dim1=None~~
        ~~self.dim2=None~~
    ~~def read_dim(self):~~
        ~~self.dim1=float(input("Dim1"))~~
        ~~self.dim2=float(input("Dim2"))~~
    def find_area(self):
        return self.dim1*self.dim2
```

Example:

```
import abc
class A(abc.ABC):
    @abc.abstractmethod
    def m1(self):
        pass
    def m2(self):
        print("m2 of A")

class B(A):
    def m1(self):
        print("overriding method")

def main():
    objb=B()
    objb.m1()
    objb.m2()
main()
```

Output:

```
overriding method
m2 of A
```

```
>>>
```

Example:

```
import abc
class Shape(abc.ABC):
    def __init__(self):
        self.dim1=None
        self.dim2=None
    def read_dim(self):
        self.dim1=float(input("Dim1:"))
        self.dim2=float(input("Dim2:"))
    @abc.abstractmethod
    def find_area(self):
        pass
class Triangle(Shape):
    def __init__(self):
        super().__init__()
    def find_area(self):
        return self.dim1*self.dim2*0.5
class Rectangle(Shape):
    def __init__(self):
        super().__init__()
    def find_area(self):
        return self.dim1*self.dim2
def main():
    t1=Triangle()
    t1.read_dim()
    area1=t1.find_area()
    r1=Rectangle()
    r1.read_dim()
    area2=r1.find_area()
    print(area1)
    print(area2)
main()
```

Output:

```
Dim1:1.2
Dim2:1.3
Dim1:1.1
Dim2:1.5
```

```
0.78
1.6500000000000001
>>>
```

What is concrete method?
A method with implementation or body is called
Concrete method.

A method without implementation is called abstract method.

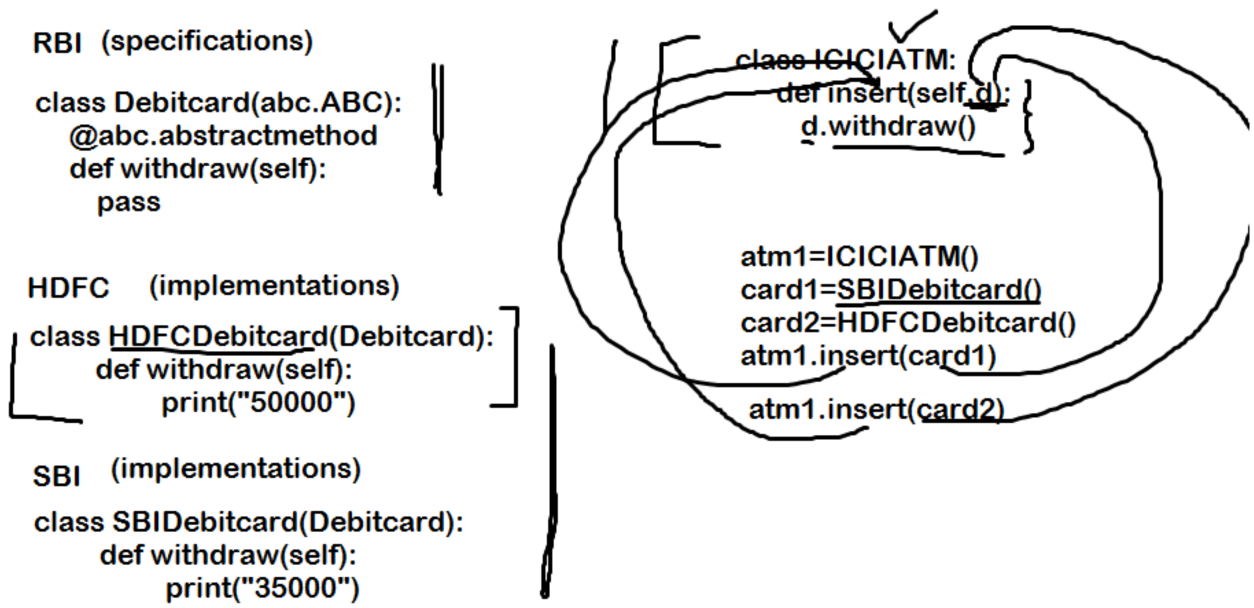
Runtime Polymorphism
What is polymorphism?
“poly” means “many” and “morphism” is forms
Defining one thing in many forms is called polymorphism

What is runtime polymorphism?

An ability of a reference variable change its behavior based on the type of object assigned is called runtime polymorphism
This allows to develop loosely coupled code, the code which work with any type is called loosely coupled code

duck-typing
A programming style which does not look at an object’s type to determine if it has the right interface; instead, the method or attribute is simply called or used (“If it looks like a duck and quacks like a duck, it must be a duck.”)

1. Abstract developer
2. Abstract implementer
3. Abstract caller



```
RBI  (specifications)

class Debitcard(abc.ABC):
   @abc.abstractmethod
   def withdraw(self):
      pass

HDFC    (implementations)
class HDFCDebitcard(Debitcard):
      def withdraw(self):
            print("50000")

SBI   (implementations)
class SBIDebitcard(Debitcard):
      def withdraw(self):
            print("35000")
```

```
class ICICIATM:
    def insert(self,d):
       d.withdraw()

atm1=ICICIATM()
card1=SBIDebitcard()
card2=HDFCDebitcard()
atm1.insert(card1)

atm1.insert(card2)
```

Example:

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

class BSNLSim(Sim):
   def connect(self):
      print("connect to BSNL network")
class AirtelSim(Sim):
   def connect(self):
      print("Connect to Airtel network")
class JioSim(Sim):
   def connect(self):
      print("Connect to Jio network")

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

def main():
   s1=AirtelSim()
```

```
    s2=JioSim()
    s3=BSNLSim()
    oneplus=Mobile()
    oneplus.insert(s1)
    oneplus.insert(s2)
    oneplus.insert(s3)
main()
```

Output:

```
Connect to Airtel network
Connect to Jio network
connect to BSNL network
>>>
```