Abstract Classes and Interfaces

- Need of an Abstract Class
- Abstract Method and Abstract Class
- Interfaces
- Abstract Classes vs Interfaces

Why we need Abstract Class?

Class is a model or Blueprint for objects.

A class contains attributes and actions which are applicable to all its instances(objects).

A small example of a class 'MyClass' that contains method calculate ( ) to calculate square of a given number.

The copy of the method is available for every object of 'MyClass'

Program 1: A Python program to understand that each object has copy of instance method

```
# A Class with method
class MyClass:
  def calculate(self, x):
    print('Square Value:', x * x)

# All object share same calculate() method
obj1 = MyClass()
obj1.calculate(2)

obj2 = MyClass()
obj2.calculate(3)

obj3 = MyClass()
obj3.calculate(4)
```

```
Square Value: 4
Square Value: 9
Square Value: 16
```

In the above code, **the requirement** of all the above objects are **same**.

But if the requirement of each object has different like:

- object1 needs square of the number

- object2 needs square root of the number
- object3 wants to calculate cube of the number

Possible Implementation issues are:

> 1. We can't write all three functionality in same function calculate( )
>
> 2. If we write three different methods in same class then these methods will be available for all the three objects which is not desirable.

To serve each object with the one and only required method, We can follow steps:

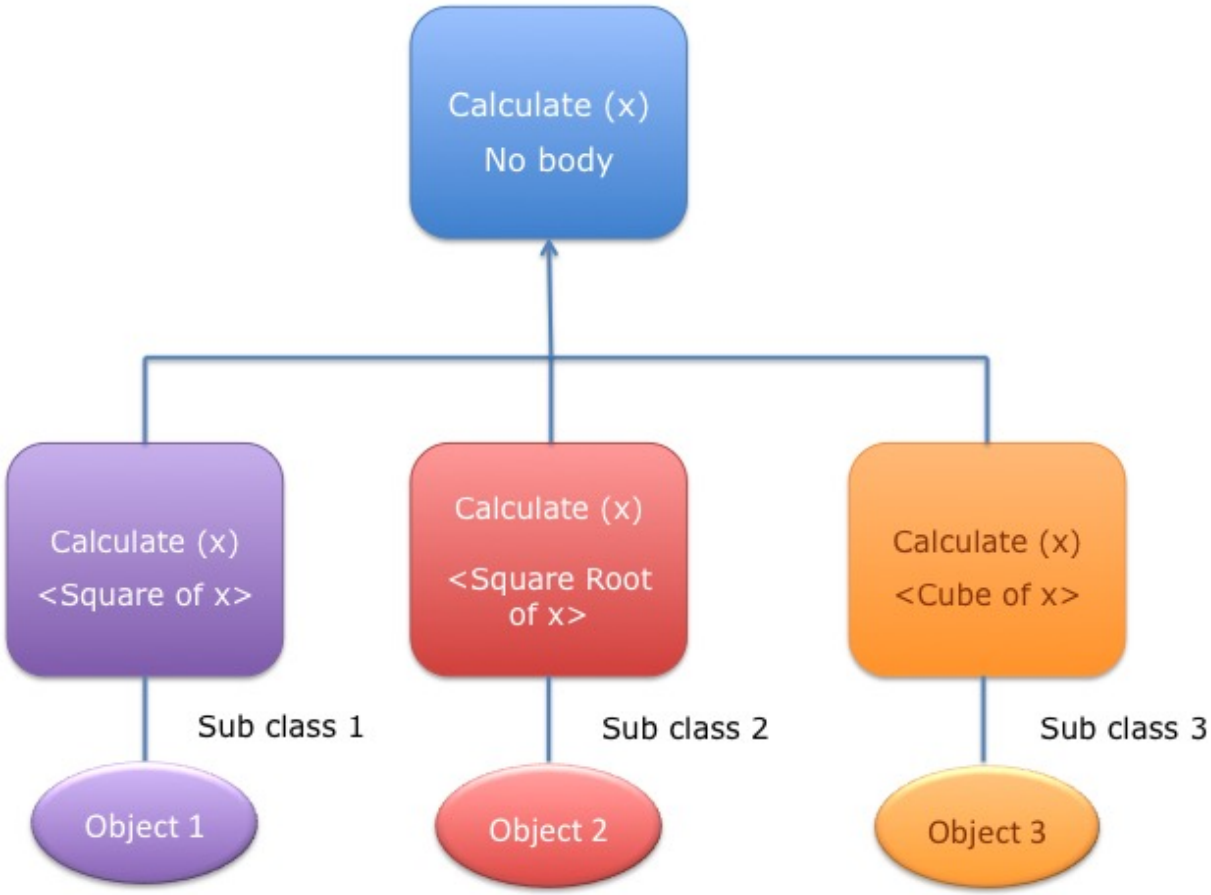
1. First, write a calculate( ) method in MyClass. It means every object wants to calculate something.
2. **Don't write body** for calculate( ) method otherwise the code will be available for all the objects.

> Such method without code is called **Abstract Method**.
>
> The class with the abstract method is called **Abstract Class**

3. Derive three sub classes from base class as

> a. Derive Subclass1 from MyClass and provide body to calculate square of the number
>
> b. Derive Subclass2 from MyClass and provide body to calculate square root of the number
>
> c. Derive Subclass3 from MyClass and provide body to calculate cube of the number

4. It is possible to create objects for the subclasses. Thus requirement is fulfilled



Abstract Method and Abstract Class

Abstract Method:

> It is a method whose action is redefined in the sub classes as per the requirement of the objects.

> Generally, they are written without body.

> It can be written with body as well which will be redefined in sub classes

> Use a decorator @abstractmethod to mark the method as abstract

Abstract Class:

> It is a class that contains abstract method.

> Python Virtual Machine (PVM) can't create the object of an abstract class as implementation is going to be defined in sub classes.

> Whenever abstract class is written, sub classes must be derived and all abstract methods should be implemented

> The abstract class is created by **deriving** it from a **meta class ABC** that belongs to **module abc**

> Syntax can be written as

```
class AbstractClass(ABC):
```

Program 2: Program to create abstract class and sub classes which implement the abstract method of the abstract class

```
# Abstract Class Example
from abc import *
from math import *

class MyClass(ABC):
  @abstractmethod
  def calculate(self, x):
    pass                    # Empty body, No code

# First sub class of MyClass
class Sub1(MyClass):
  def calculate(self, x):
    print("Square of the number:", x * x)

# Second sub class of MyClass
class Sub2(MyClass):
  def calculate(self, x):
    print("Square root of the number:", sqrt(x))

# Third sub class of MyClass
class Sub3(MyClass):
  def calculate(self, x):
    print("Cube of the number:", x ** 3)

# Create Sub1 class object and call calculate method
obj1 = Sub1()
obj1.calculate(16)

# Create Sub2 class object and call calculate method
obj2 = Sub2()
obj2.calculate(16)

# Create Sub3 class object and call calculate method
```

```
obj3 = Sub3()
obj3.calculate(16)
```

```
    Square of the number: 256
    Square root of the number: 4.0
    Cube of the number: 4096
```

Example 2 for Abstract Class

There are many cars on the street. These are all objects of class Car.

Example: Maruti, Benz, Santro, Honda City, etc..

They have common characteristics. Let us take some of them for the implementation.

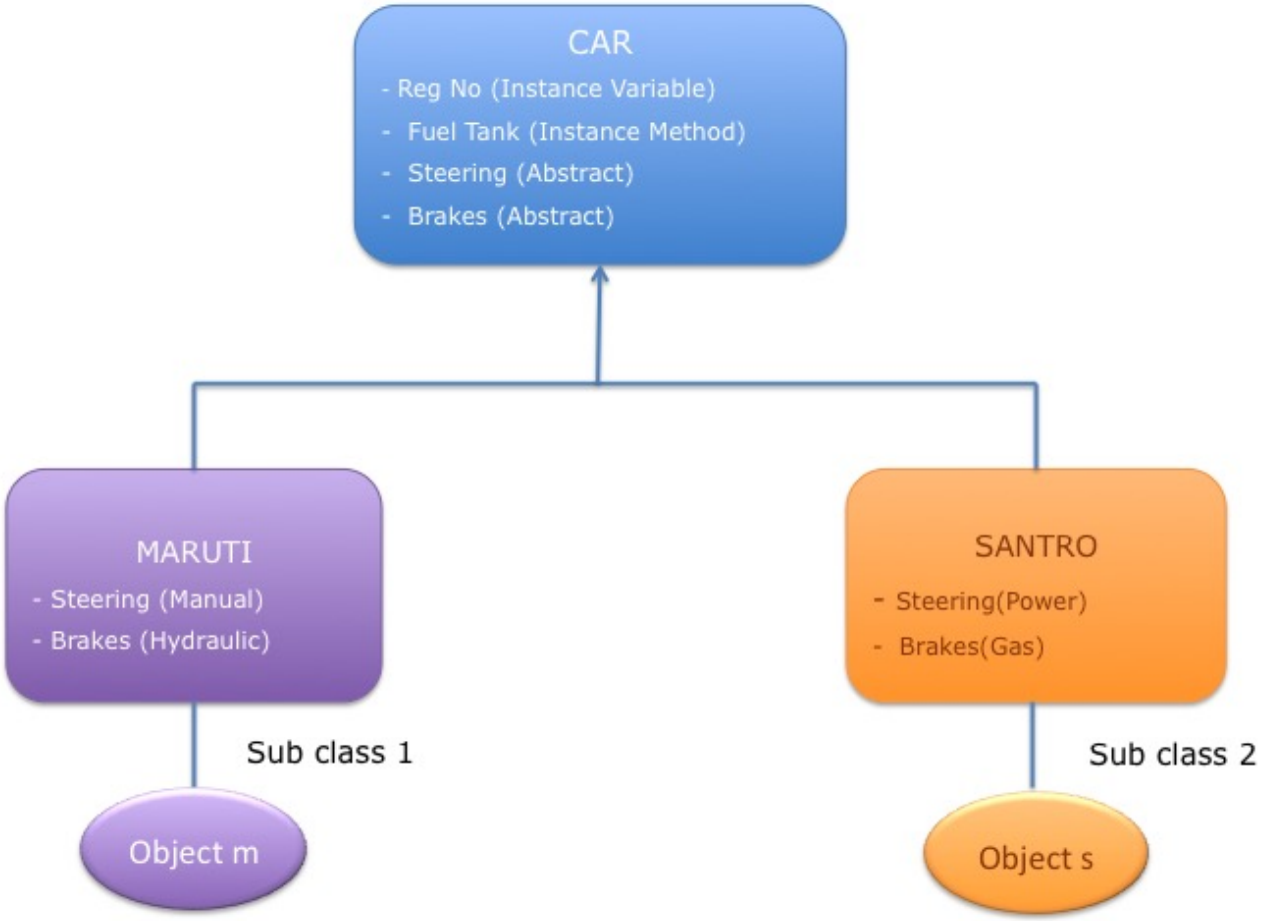
1. Registration Number
2. Fuel Tank
3. Steering
4. Brakes

> **Registration Number** is unique for every car and it is applicable to each one. Therefore it can be considered as **instance variable**.

> Every car has Fuel Tank and let us assume that filling the tank is same mechanism for all the cars. Hence **openTank ()** method can be treated as **instance method** of class Car

> Steering is an action. It can be implemented in different ways for different models. Let us assume that Maruti has manual steering and Santro has power steering. Therefore **steering( )** method must be **abstract method**.

> Similarly, braking is also an action. It can be implemented in different ways for different models. Let us assume that Maruti has hydraulic brakes and Santro has gas brakes. Therefore **braking( )** method must be **abstract method**.



Program 3: Program to create abstract class Car that contains an instance variable, a concrete method and two abstract methods

```
# Creating an abstract class
from abc import *

class Car(ABC):
  def __init__(self, regno):
    self.regno = regno

  def openTank(self):
    print('Fill the fuel into the tank')
    print('For the car with registration No:', self.regno)

  @abstractmethod
  def steering(self):
    pass

  @abstractmethod
  def braking(self):
```

```
  def braking(self):
    pass
```

Program 4: Program in which Maruti sub class is implements the abstract method of the Super Class Car

```
# Sub class for abstract class Car

class Maruti(Car):
  def steering(self):
    print('Maruti uses manual steering')
    print('Drive the car')

  def braking(self):
    print('Maruti uses Hydraulic Brakes')
    print('Apply the brake and stop the car')


# Create objects of class Maruti
m = Maruti(1001)
m.openTank()
m.steering()
m.braking()
```

```
 Fill the fuel into the tank
 For the car with registration No: 1001
 Maruti uses manual steering
 Drive the car
 Maruti uses Hydraulic Brakes
 Apply the brake and stop the car
```

Program 5: Program in which Santro sub class is implements the abstract method of the Super Class Car

```
# Sub class for abstract class Car

class Santro(Car):
  def steering(self):
    print('Santro uses power steering')
    print('Drive the car')

  def braking(self):
    print('Santro uses Gas Brakes')
    print('Apply the brake and stop the car')


# Create objects of class Maruti
s = Santro(5734)
```

```
s.openTank()
s.steering()
s.braking()
```

```
Fill the fuel into the tank
For the car with registration No: 5734
Santro uses power steering
Drive the car
Santro uses Gas Brakes
Apply the brake and stop the car
```

Interfaces

> Abstract class can have some abstract methods and some concrete methods

> When class contains **only abstract methods**, it becomes **an interface**. Only method headers are written in the interface

> Interfaces can be defined as specification of method headers

> Interface concept is NOT EXPLICITLY available in Python. Interface is created as abstract class only.

> It is not possible to create the objects of an interface. Sub classes can be created using interfaces which implements abstract methods.

> **Flexibility:** Each sub class can provide its own implementation for abstract methods

> An interface is useful in comparison with class as it is more flexible to provide necessary implementation needed by the object

---

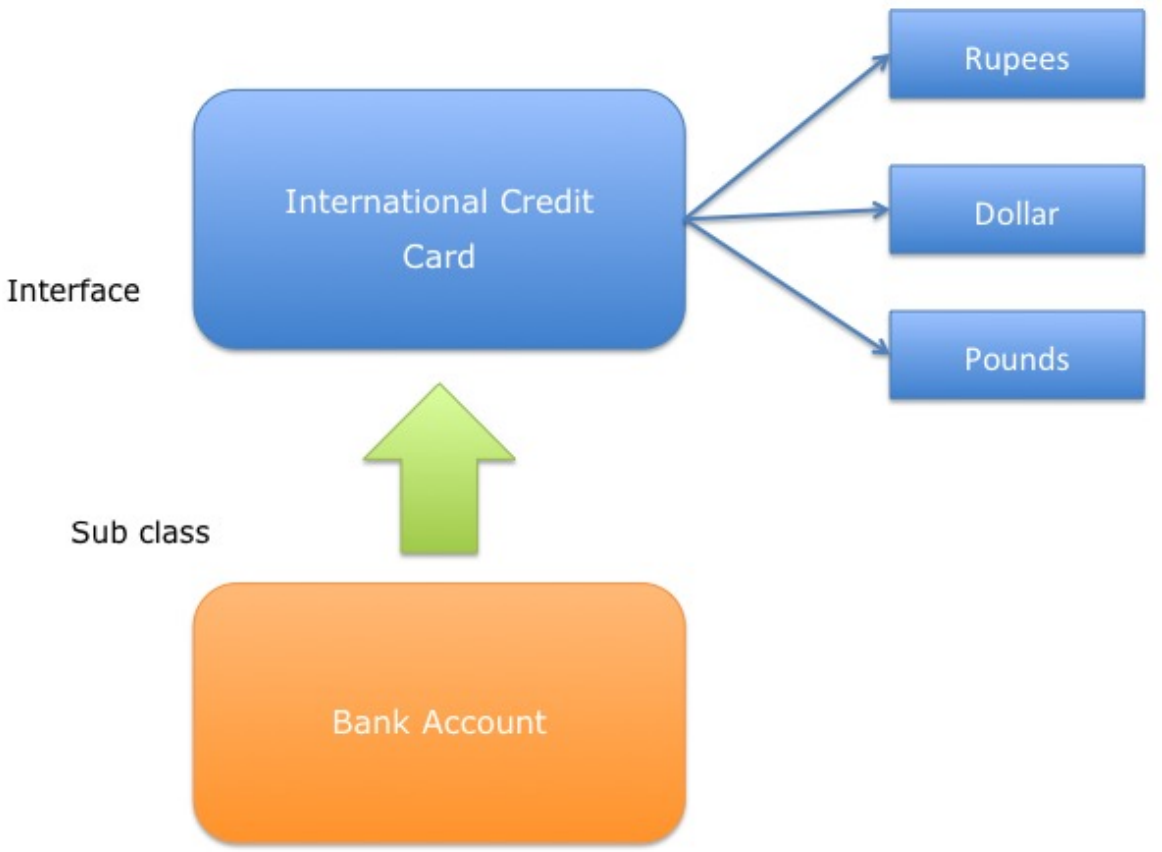
Example of International Credit Card:

> We can pay using credit card in rupees as well as in dollars or any other valid currency.

> Here credit card is similar to interface which performs several task.

> In fact, it is a plastic card and does not hold money physically. It holds only the name of the holder, bank name and account number.

> Behind credit card, we have bank account which acts like sub class to perform necessary transaction.



Example of Interface communicating with different Database:

> When an interface is created it is not necessary for the programmar to provide sub classes for the interface.

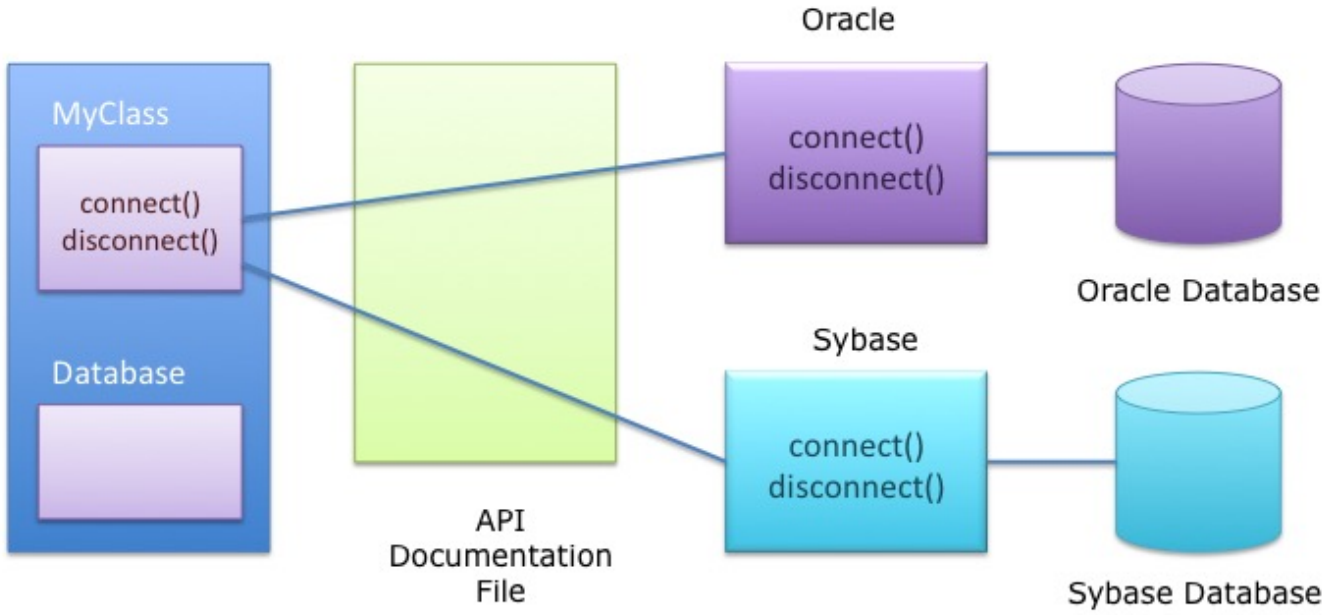
> Any third party can provide the sub class

> The client or user purchases interfaces and sub classes depending on his requirement. If client wants to connect to Oracle then he will purchase Oracle Sub class.

How third party vendors know which methods they should implement?

> API (Application Programming Interface) is a text file or HTML file which contains description of all the features of software, language or product.It has description of classes, methods and attributes.

> This file is reffered by third party vendor to know about the interfaces and methods.
> Then they write same method in the sub classes.



Program 6: Program to develop an interface that connects to any database

```
# Abstract class works like an interface

from abc import *

class DBConnect(ABC):
  def connect(self):
    pass
  def disconnect(self):
    pass


# Sub class for Oracle Database
class Oracle(DBConnect):
  def connect(self):
    print("Connecting to Oracle Database....")
```

```
    print("Connecting to Oracle Database....")

  def disconnect(self):
    print("Disconnected from Oracle..")


# Sub class for Sybase Database
class Sybase(DBConnect):
  def connect(self):
    print("Connecting to Sybase Database....")

  def disconnect(self):
    print("Disconnected from Sybase..")

class Database:
  # Accept database name as a string
  st = input('Enter name of the database:')

  # Convert the string into class name
  classname = globals() [st]

  # Create an object to the class
  x = classname()

  # Call the connect and disconnect method
  x.connect()
  x.disconnect()
```

```
Enter name of the database:Oracle
Connecting to Oracle Database....
Disconnected from Oracle..
```

Program 7: Program which contains a Printer interface and its sub classes to send text to any printer

```
# An interface to send text to any printer
from abc import *

# create an interface
class Printer(ABC):

  def printit(self, text):
    pass

  def disconnect(self):
    pass


# This is sub class for IBM printer
class IBM(Printer):
```

```
  def printit(self, text):
    print(text)

  def disconnect(self):
    print("Printing completed on IBM printer")


# This is sub class for EPSON printer
class Epson(Printer):

  def printit(self, text):
    print(text)

  def disconnect(self):
    print("Printing completed on EPSON printer")


class UsePrinter:
  # Read printer name from user
  str = input("Enter name of the printer:")

  # Convert the string into class name
  classname = globals() [str]

  # Create an object to the class
  x = classname()

  x.printit('This text is sent to printer')
  x.disconnect()
```

```
Enter name of the printer:Epson
This text is sent to printer
Printing completed on EPSON printer
```

Abstract Classes vs. Interfaces

> Python does not provide interface concept explicitly

> It provides abstract classes which can be used as either either abstract classes or interfaces

> Programmar decides when to go for abstract class and when to go for interfaces

1. Criteria for Implementation

> Generally, abstract class is written when some of the features are common among all the objects.

> If there are no common features among the objects then Interfaces are written

2. Responsibility of Sub Classes:

> There is a responsibilty of programmar to provide sub classes whenever abstract class is written

> If interface is written, then any third party vendor will take responsibility of providing sub classes

3. Speed of Execution:

> Interfaces are slow when compared to abstract classes.

> In case of Interfaces, whenever method is called, PVM need to search for implementation which is installed elsewhere in the system.

> For abstract classes common methods are defined in base class and generally sub classes are written at same place. Hence searching time is less.