ABSTRACTION

Abstraction is used to hide the internal functionality of the function from the users. The users only interact with the basic implementation of the function, but inner working is hidden. User is familiar with that "what function does" but they don't know "how it does."

In simple words, we all use the smartphone and very much familiar with its functions such as camera, voice-recorder, call-dialling, etc., but we don't know how these operations are happening in the background.

In Python, we can achieve abstraction using ABC (abstraction class) and abstractmethod decorator. A class that consists of one or more abstract method is called the abstract class. An abstract method is a method that has a declaration but does not have an implementation. Python provides the abc module to use the abstraction in the Python program.

Syntax:

```
from abc import ABC,abstractmethod
class class_name(ABC):
        @abstractmethod
        def methodName(self):
                pass
```

We cannot create object of an abstract class. So, to access its contents we have to create child of it. But in child class we need to implement all the abstract methods that are defined in abstract class.

for e.g.

```
from abc import ABC, abstractmethod
class Car(ABC):
   @abstractmethod
   def mileage(self):
      pass

class Tesla(Car):
   def mileage(self):
      print("The mileage is 30kmph")
```

```
class Suzuki(Car):
    def mileage(self):
        print("The mileage is 25kmph ")
class Duster(Car):
     def mileage(self):
          print("The mileage is 24kmph ")

class Renault(Car):
    def mileage(self):
           print("The mileage is 27kmph ")

t= Tesla ()
t.mileage()

r = Renault()
r.mileage()

s = Suzuki()
s.mileage()
d = Duster()
d.mileage()
```

OUTPUT:

```
The mileage is 30kmph
The mileage is 27kmph
The mileage is 25kmph
The mileage is 24kmph
```

In the above code, we have imported the abc module to create the abstract base class. We created the Car class that inherited the ABC class and defined an abstract method named mileage(). We have then inherited the base class from the three different subclasses and implemented the abstract method differently. We created the objects to call the abstract method.

Creating common interface for abstract methods:

```
from abc import ABC,abstractmethod

class Connection(ABC):
    @abstractmethod
    def commit(self):
        pass

    @abstractmethod
    def rollback(self):
        pass

class Oracle(Connection):
    def commit(self):
        print('commit of Oracle')

    def rollback(self):
        print('rollback of Oracle')

class MySQL(Connection):
    def commit(self):
        print('commit of MySQL')

    def rollback(self):
        print('rollback of MySQL')

def common(con):
    con.commit()
    con.rollback()

common(Oracle())
common(MySQL())
```

OUTPUT:

```
commit of Oracle
rollback of Oracle
commit of MySQL
rollback of MySQL
```

Indirect subclass:

We can create indirect subclass of an abstract class by registering it through our abstract class. The need of indirect subclass is when we do not want to implement all the abstract methods of an abstract class. In indirect subclass, it is not mandatory to implement all the abstract methods of abstract class.

```
from abc import ABC,abstractmethod

class A(ABC):
    @abstractmethod
    def m1(self):
        pass

    @abstractmethod
    def m2(self):
        pass

class B:
    def m1(self):
        print("m1--B")
        #super().m1() #this will generate error as B is not a direct subclass of A
A.register(B)
b = B()
b.m1()
```

Behaviour of indirect subclass with issubclass function:

```
from abc import ABC,abstractmethod
class A(ABC):
    @abstractmethod
    def m1(self):
        pass

    @abstractmethod
    def m2(self):
        pass

class B:
    def m1(self):
        print("m1--B")
```

```
class C(A):
    def m1(self):
        print("m1--C")

    def m2(self):
        print("m2--C")

print('Before registering class B with A')
print(issubclass(B,A))
A.register(B)
print('After registering class B with A')
print(issubclass(B,A))

print('Printing the subclasses of A --> ')
for subclass in A.__subclasses__():
    print(subclass.__name__)
```

OUTPUT:

```
Before registering class B with A
False
After registering class B with A
True
Printing the subclasses of A -->
C
```

In the above output, subclasses method is use to get all the immediate direct subclass of class A. It will display only class C not B as B is indirect subclass of class A.