Python Abstract Concept:

- Today here, we are going to discuss the concept of Abstraction in Python for the Object-Oriented Programming approach.
- Basically, Abstraction focuses on hiding the internal implementations of a process or method from the user.
- In this way, the user knows what he is doing but not how the work is being done.
- Let us dig a bit deeper into the topic to find its importance in real life and programming.

What is Abstraction in Python?

- In Object Oriented Programming, Inheritance, Polymorphism and Encapsulation go hand in hand. But Abstraction is also an essential element of OOP.
- For example, people do not think of a car as a set of thousands of individual parts.
- Instead they see it as a well-defined object with its own unique behavior.
- This abstraction allows people to use a car to drive without knowing the complexity of the parts that form the car.
- They can ignore the details of how the engine transmission, and braking systems work.
- Instead, they are free to utilize the object as a whole.
- A powerful way to manage abstraction is through the use of hierarchical classification.
- This allows us to layer the semantics of complex systems, breaking them into more manageable pieces.
- From the outside, a car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, etc.
- In turn, each of these subsystems is made up of smaller units.
- The point is that we manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions.
- This can also be applied to computer programs using OOP concepts. This is the essence of object-oriented programming.

Steps to Create Abstract Classes and Methods in Python:

- To declare/create an Abstract class, we firstly need to import the "abc" module.
- This "abc" module contains "ABC" class (i.e, Abstract Base Class) and "abstractmethod" decorator.
- We can create userdefined abstract class by using predefined abstract class.
- Create required abstract methods by using "abstractmethod" decorator with @ symbol on top of methods.
- Any classes which contains abstract methods then called as abstract classes.
- We can not instanciate/create objects for abstract classes.
- If we are triying to create objects for abstract classes then it throws exception like
  TypeError: Can't instantiate abstract class Sample with abstract methods task
- Any method which is just define/declared structure with out body implimentation statements and contains @abstractmethod decorator on top of it then called as abstract method.
- Any class which contains all abstract methods implimentations properly then that class is called as "Concrete" class.
- We can create objects for Concrete class and we can access members of class.

Let us look at an example.

Example:

```
from abc import ABC, abstractmethod
class abs_class(ABC):
    #abstractmethod
    def m1(self):
      =====
```

- Here, abs_class is the abstract class inside which abstract methods or any other sort of methods can be defined.
- As a property, abstract classes can have any number of abstract methods and any number of other methods.

For example we can see below:

```
from abc import ABC, abstractmethod
```

```
class abs_class(ABC):
    #normal method
    def method(self):
        #method definition

    @abstractmethod
    def Abs_method(self):
        #Abs_method definition
```

--->> Here, method() is normal method whereas Abs_method() is an abstract method implementing @abstractmethod from the abc module.

Python Abstraction Example:

Now that we know about abstract classes and methods, let's take a look at an example which explains Abstraction in Python.

Code:

```
from abc import ABC, abstractmethod
class Absclass(ABC):
    def print(self,x):
        print("Passed value: ", x)
    @abstractmethod
    def task(self):
        print("We are inside Absclass task")

class test_class(Absclass):
    def task(self):
        print("We are inside test_class task")

class example_class(Absclass):
    def task(self):
        print("We are inside example_class task")

#object of test_class created
test_obj = test_class()
```

```
test_obj.task()
test_obj.print(100)

#object of example_class created
example_obj = example_class()
example_obj.task()
example_obj.print(200)

print("test_obj is instance of Absclass? ", isinstance(test_obj, Absclass))
print("example_obj is instance of Absclass? ", isinstance(example_obj, Absclass))
```

Output:

```
We are inside test_class task
Passed value : 100
We are inside example_class task
Passed value : 200
test_obj is instance of Absclass? True
example_obj is instance of Absclass? True
```

Exaplantion:

- Here, Absclass is the abstract class that inherits from the ABC class from the abc module.
- It contains an abstract method task() and a normal print() method which are visible by the user.
- Two other classes inheriting from this abstract class are test_class and example_class.
- Both of them have their own task() method (extension of the abstract method).
- After the user creates objects from both the test_class and example_class classes and invoke the task() method for both of them, the hidden definitions for task() methods inside both the classes come into play.
- These definitions are hidden from the user. The abstract method task() from the abstract class Absclass is actually never invoked.
- But when the print() method is called for both the test_obj and example_obj, the Absclass's print() method is invoked since it is not an abstract method.

Note: We cannot create instances of an abstract class. It raises an Error.

Example:

Q. Accessing abstract class in multiple concrete classes and implementing its methods.?

```
from abc import ABC,abstractmethod
class A(ABC):
    def __init__(self,value):
        self.value = value

    @abstractmethod
    def add(self):
        pass

    @abstractmethod
    def sub(self):
        pass

    # normal method
    def mul(self):
        print("The Multiplication is :",self.value * 10)

class B(A):
    def add(self):
        print("The Addition is :",self.value + 10)

class C(B):
    def sub(self):
        print("The Subtraction is :",self.value - 10)

class D(A):
    def add(self):
        print('The Addition is:', self.value + 20)
    def sub(self):
        print('The Subtraction is:', self.value - 20)
```

```
#b = B()
cobj = C(100)
cobj.add()
cobj.sub()
cobj.mul()

d = D(200)
d.add()
d.sub()
d.mul()
```

Output:

```
The Addition is : 110
The Subtraction is : 90
The Multiplication is : 1000
The Addition is: 220
The Subtraction is: 180
The Multiplication is : 2000
```

Example:

Q. Create abstract class constructor and access it in child classes

```
from abc import ABC,abstractmethod
class Cal(ABC):
    def __init__(self,value):
        self.value = value

    @abstractmethod
    def add(self):
        pass

    @abstractmethod
    def sub(self):
        pass

class C(Cal):
    def add(self):
```

```
        print(self.value + 10)

    def sub(self):
        print(self.value - 10)

# cobj = Cal()
cobj = C(100)
cobj.add()
cobj.sub()
```

Output:

```
110
90
```

Example:

```
import abc
class Shape(metaclass=abc.ABCMeta):
  @abc.abstractmethod
  def area(self):
    pass

class Rectangle(Shape):
  def __init__(self, x,y):
    self.l = x
    self.b=y
  def area(self):
    return self.l * self.b

r = Rectangle(10,20)
print ('area: ',r.area())
```

Output:

```
area:  200
```