Abstraction in Python

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-dialing, etc., but we don't know how these operations are happening in the background. Let's take another example - When we use the TV remote to increase the volume. We don't know how pressing a key increases the volume of the TV. We only know to press the "+" button to increase the volume.

That is exactly the abstraction that works in the object-oriented concept.

In Python, an abstraction is used to hide the irrelevant data/class in order to reduce the complexity. It also enhances the application efficiency.

Abstract classes in Python:

In Python, abstraction can be achieved by using abstract classes and interfaces.

A class that consists of one or more abstract method is called the abstract class. Abstract methods do not contain their implementation. Abstract class can be inherited by the subclass and abstract method gets its definition in the subclass. Abstraction classes are meant to be the blueprint of the other class. An abstract class can be useful when

we are designing large functions. An abstract class is also helpful to provide the standard interface for different implementations of components. Python provides the **abc** module to use the abstraction in the Python program. Let's see the following syntax.

Syntax:

```
from abc import ABC
class ClassName(ABC):
```

We import the ABC class from the **abc** module.

Abstract Base Classes:

An abstract base class is the common application program of the interface for a set of subclasses. It can be used by the third-party, which will provide the implementations such as with plugins. It is also beneficial when we work with the large code-base hard to remember all the classes.

Working of the Abstract Classes:

Unlike the other high-level language, Python doesn't provide the abstract class itself. We need to import the abc module, which provides the base for defining Abstract Base classes (ABC). The ABC works by decorating methods of the base class as abstract. It registers concrete classes as the implementation of the abstract base. We use the ***@abstractmethod*** decorator to define an abstract method or if we

don't provide the definition to the method, it automatically becomes the abstract method. Let's understand the following example.

For Example:

```
# Python program demonstrate
# abstract base class work
from abc import ABC, abstractmethod


class Car(ABC):
    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 ")

    # Driver code


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.

Let's understand another example.

```python
# Python program to define
# abstract class

from abc import ABC


class Polygon(ABC):

    # abstract method
    def sides(self):
        pass


class Triangle(Polygon):

    def sides(self):
        print("Triangle has 3 sides")


class Pentagon(Polygon):

    def sides(self):
        print("Pentagon has 5 sides")
```

```
class Hexagon(Polygon):

    def sides(self):
        print("Hexagon has 6 sides")


class square(Polygon):

    def sides(self):
        print("I have 4 sides")

    # Driver code


t = Triangle()
t.sides()
s = square()
s.sides()
p = Pentagon()
p.sides()
k = Hexagon()
k.sides()
```

Output:

```
Triangle has 3 sides
I have 4 sides
Pentagon has 5 sides
Hexagon has 6 sides
```

In the above code, we have defined the abstract base class named Polygon and we also defined the abstract method. This base class inherited by the various subclasses. We implemented the abstract method in each subclass. We created the object of the subclasses and invoke the **sides()** method. The hidden implementations for the **sides()**

method inside the each subclass comes into play. The abstract method **sides()** method, defined in the abstract class, is never invoked.

Below are the points which we should remember about the abstract base class in Python.

- An Abstract class can contain the both method normal and abstract method.
- An Abstract cannot be instantiated; we cannot create objects for the abstract class.

Abstraction is essential to hide the core functionality from the users.

Python Operator Overloading:

Python operators work for built-in classes. But the same operator behaves differently with different types. For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.

This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y


p1 = Point(1, 2)
p2 = Point(2, 3)
print(p1+p2)
```

Output:

```
Traceback (most recent call last):
  File "/home/nirajan/PycharmProjects/PythonExamples/testPythonOOPs.py", line 9, in <module>
    print(p1+p2)
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

Here, we can see that a TypeError was raised, since Python didn't know how to add two Point objects together.

However, we can achieve this task in Python through operator overloading. But first, let's get a notion about special functions.

Python Special Functions:

Class functions that begin with double underscore __ are called special functions in Python.

These functions are not the typical functions that we define for a class. The __init__() function we defined above is one of them. It gets called every time we create a new object of that class.

There are numerous other special functions in Python.

Using special functions, we can make our class compatible with built-in functions.

```python
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0}, {1})".format(self.x, self.y)


p1 = Point(2, 3)
print(p1)
```

Output:

```
(2, 3)
```

Turns out, that this same method is invoked when we use the built-in function str() or format().

```
>>> str(p1)
'(2,3)'

>>> format(p1)
'(2,3)'
```

So, when you use str(p1) or format(p1), Python internally calls the p1.__str__() method. Hence the name, special functions.

Overloading the + Operator:

To overload the + operator, we will need to implement __add__() function in the class. With great power comes great responsibility. We can do whatever we like, inside this function. But it is more sensible to return a Point object of the coordinate sum.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)


p1 = Point(1, 2)
p2 = Point(2, 3)

print(p1+p2)
```

Output:

```
(3,5)
```

What actually happens is that, when you use p1 + p2, Python calls p1.__add__(p2) which in turn is Point.__add__(p1,p2). After this, the addition operation is carried out the way we specified.

Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Operator	Expression	Internally
Addition	`p1 + p2`	`p1.__add__(p2)`
Subtraction	`p1 - p2`	`p1.__sub__(p2)`
Multiplication	`p1 * p2`	`p1.__mul__(p2)`
Power	`p1 ** p2`	`p1.__pow__(p2)`
Division	`p1 / p2`	`p1.__truediv__(p2)`
Floor Division	`p1 // p2`	`p1.__floordiv__(p2)`
Remainder (modulo)	`p1 % p2`	`p1.__mod__(p2)`
Bitwise Left Shift	`p1 << p2`	`p1.__lshift__(p2)`
Bitwise Right Shift	`p1 >> p2`	`p1.__rshift__(p2)`
Bitwise AND	`p1 & p2`	`p1.__and__(p2)`
Bitwise OR	`p1 | p2`	`p1.__or__(p2)`
Bitwise XOR	`p1 ^ p2`	`p1.__xor__(p2)`
Bitwise NOT	`~p1`	`p1.__invert__()`

Overloading Comparison Operators:

Suppose we wanted to implement the less than symbol < symbol in our Point class.

Let us compare the magnitude of these points from the origin and return the result for this purpose. It can be implemented as follows.

```python
# overloading the less than operator
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __lt__(self, other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y ** 2)
        return self_mag < other_mag

p1 = Point(1,1)
p2 = Point(-2,-3)
p3 = Point(1,-1)

# use less than
print(p1<p2)
print(p2<p3)
print(p1<p3)
```

Output:

```
True
False
False
```

Similarly, the special functions that we need to implement, to overload other comparison operators are tabulated below.

Operator	Expression	Internally
Less than	`p1 < p2`	`p1.__lt__(p2)`
Less than or equal to	`p1 <= p2`	`p1.__le__(p2)`
Equal to	`p1 == p2`	`p1.__eq__(p2)`
Not equal to	`p1 != p2`	`p1.__ne__(p2)`
Greater than	`p1 > p2`	`p1.__gt__(p2)`
Greater than or equal to	`p1 >= p2`	`p1.__ge__(p2)`