Learn Complete Python In Simple Way

OOP's Part - 4

STUDY MATERIAL

Agenda

1) Abstract Method
2) Abstract class
3) Interface
4) Public,Private and Protected Members
5) __str__() Method
6) Difference between str() and repr() functions
7) Small Banking Application

Abstract Method:

- Sometimes we don't know about implementation, still we can declare a method. Such types of methods are called abstract methods.i.e abstract method has only declaration but not implementation.
- In python we can declare abstract method by using @abstractmethod decorator as follows.

- @abstractmethod
- def m1(self): pass

- @abstractmethod decorator present in abc module. Hence compulsory we should import abc module,otherwise we will get error.
- abc → abstract base class module

```
1) class Test:
2)     @abstractmethod
3)     def m1(self):
4)         pass
```

NameError: name 'abstractmethod' is not defined

Eg:

```
1) from abc import *
2) class Test:
3)     @abstractmethod
4)     def m1(self):
5)         pass
```

Eg:

```
1)  from abc import *
2)  class Fruit:
3)      @abstractmethod
4)      def taste(self):
5)          pass
```

Child classes are responsible to provide implemention for parent class abstract methods.

Abstract class:

Some times implementation of a class is not complete,such type of partially implementation classes are called abstract classes. Every abstract class in Python should be derived from ABC class which is present in abc module.

Case-1:

```
1)  from abc import *
2)  class Test:
3)      pass
4)
5)  t=Test()
```

In the above code we can create object for Test class b'z it is concrete class and it does not conatin any abstract method.

Case-2:

```
1)  from abc import *
2)  class Test(ABC):
3)      pass
4)
5)  t=Test()
```

In the above code we can create object, even it is derived from ABC class,b'z it does not contain any abstract method.

Case-3:

```
1)  from abc import *
2)  class Test(ABC):
3)      @abstractmethod
4)      def m1(self):
```

```
5)         pass
6)  t=Test()
```

TypeError: Can't instantiate abstract class Test with abstract methods m1

Case-4:

```
1)  from abc import *
2)  class Test:
3)      @abstractmethod
4)      def m1(self):
5)          pass
6)
7)  t=Test()
```

We can create object even class contains abstract method b'z we are not extending ABC class.

Case-5:

```
1)  from abc import *
2)  class Test:
3)      @abstractmethod
4)      def m1(self):
5)          print('Hello')
6)
7)  t=Test()
8)  t.m1()
```

Output: Hello

Conclusion: If a class contains atleast one abstract method and if we are extending ABC class then instantiation is not possible.

"abstract class with abstract method instantiation is not possible"

Parent class abstract methods should be implemented in the child classes. Otherwise we cannot instantiate child class.If we are not creating child class object then we won't get any error.

Case-1:

```
1)  from abc import *
2)  class Vehicle(ABC):
3)      @abstractmethod
4)      def noofwheels(self):
5)          pass
6)
7)  class Bus(Vehicle): pass
```

It is valid because we are not creating Child class object.

Case-2:

```
1)  from abc import *
2)  class Vehicle(ABC):
3)      @abstractmethod
4)      def noofwheels(self):
5)          pass
6)
7)  class Bus(Vehicle): pass
8)  b=Bus()
```

TypeError: Can't instantiate abstract class Bus with abstract methods noofwheels

Note: If we are extending abstract class and does not override its abstract method then child class is also abstract and instantiation is not possible.

```
1)  from abc import *
2)  class Vehicle(ABC):
3)      @abstractmethod
4)      def noofwheels(self):
5)          pass
6)
7)  class Bus(Vehicle):
8)      def noofwheels(self):
9)          return 7
10)
11) class Auto(Vehicle):
12)     def noofwheels(self):
13)         return 3
14) b=Bus()
15) print(b.noofwheels())#7
16)
```

```
17) a=Auto()
18) print(a.noofwheels())#3
```

Note: Abstract class can contain both abstract and non-abstract methods also.

Interfaces In Python:

In general if an abstract class contains only abstract methods such type of abstract class is considered as interface.

```
1)  from abc import *
2)  class DBInterface(ABC):
3)      @abstractmethod
4)      def connect(self):pass
5)
6)      @abstractmethod
7)      def disconnect(self):pass
8)
9)  class Oracle(DBInterface):
10)     def connect(self):
11)         print('Connecting to Oracle Database...')
12)     def disconnect(self):
13)         print('Disconnecting to Oracle Database...')
14)
15) class Sybase(DBInterface):
16)     def connect(self):
17)         print('Connecting to Sybase Database...')
18)     def disconnect(self):
19)         print('Disconnecting to Sybase Database...')
20)
21) dbname=input('Enter Database Name:')
22) classname=globals()[dbname]
23) x=classname()
24) x.connect()
25) x.disconnect()
```

```
D:\durga_classes>py test.py
Enter Database Name:Oracle
Connecting to Oracle Database...
Disconnecting to Oracle Database...

D:\durga_classes>py test.py
Enter Database Name:Sybase
```

Connecting to Sybase Database...
Disconnecting to Sybase Database...

Note: The inbuilt function globals()[str] converts the string 'str' into a class name and returns the classname.

Demo Program-2: Reading class name from the file

config.txt

EPSON

test.py

```
1)  from abc import *
2)  class Printer(ABC):
3)      @abstractmethod
4)      def printit(self,text):pass
5)
6)      @abstractmethod
7)      def disconnect(self):pass
8)
9)  class EPSON(Printer):
10)     def printit(self,text):
11)         print('Printing from EPSON Printer...')
12)         print(text)
13)     def disconnect(self):
14)         print('Printing completed on EPSON Printer...')
15)
16) class HP(Printer):
17)     def printit(self,text):
18)         print('Printing from HP Printer...')
19)         print(text)
20)     def disconnect(self):
21)         print('Printing completed on HP Printer...')
22)
23) with open('config.txt','r') as f:
24)     pname=f.readline()
25)
26) classname=globals()[pname]
27) x=classname()
28) x.printit('This data has to print...')
29) x.disconnect()
```

Output:

```
Printing from EPSON Printer...
This data has to print...
Printing completed on EPSON Printer...
```

Concreate class vs Abstract Class vs Inteface:

1) If we dont know anything about implementation just we have requirement specification then we should go for interface.
2) If we are talking about implementation but not completely then we should go for abstract class. (partially implemented class).
3) If we are talking about implementation completely and ready to provide service then we should go for concrete class.

```
1)  from abc import *
2)  class CollegeAutomation(ABC):
3)      @abstractmethod
4)      def m1(self): pass
5)      @abstractmethod
6)      def m2(self): pass
7)      @abstractmethod
8)      def m3(self): pass
9)  class AbsCls(CollegeAutomation):
10)     def m1(self):
11)         print('m1 method implementation')
12)     def m2(self):
13)         print('m2 method implementation')
14)
15) class ConcreteCls(AbsCls):
16)     def m3(self):
17)         print('m3 method implemnentation')
18)
19) c=ConcreteCls()
20) c.m1()
21) c.m2()
22) c.m3()
```

Public, Protected and Private Attributes:

By default every attribute is public. We can access from anywhere either within the class or from outside of the class.

Eg: name = 'durga'

Protected attributes can be accessed within the class anywhere but from outside of the class only in child classes. We can specify an attribute as protected by prefexing with _ symbol.

Syntax: _variablename = value

Eg: _name='durga'

But is is just convention and in reality does not exists protected attributes.

private attributes can be accessed only within the class.i.e from outside of the class we cannot access. We can declare a variable as private explicitly by prefexing with 2 underscore symbols.

syntax: __variablename=value

Eg: __name='durga'

```
1)  class Test:
2)      x=10
3)      _y=20
4)      __z=30
5)      def m1(self):
6)          print(Test.x)
7)          print(Test._y)
8)          print(Test.__z)
9)
10) t=Test()
11) t.m1()
12) print(Test.x)
13) print(Test._y)
14) print(Test.__z)
```

Output:

```
10
20
30
10
```

```
20
Traceback (most recent call last):
  File "test.py", line 14, in <module>
    print(Test.__z)
AttributeError: type object 'Test' has no attribute '__z'
```

How to Access Private Variables from Outside of the Class:

We cannot access private variables directly from outside of the class.
But we can access indirectly as follows `objectreference._classname__variablename`

```
1)  class Test:
2)      def __init__(self):
3)          self.__x=10
4)
5)  t=Test()
6)  print(t._Test__x)#10
```

__str__() method:

- Whenever we are printing any object reference internally __str__() method will be called which is returns string in the following format
  `<__main__.classname object at 0x022144B0>`

- To return meaningful string representation we have to override __str__() method.

```
1)  class Student:
2)      def __init__(self,name,rollno):
3)          self.name=name
4)          self.rollno=rollno
5)
6)      def __str__(self):
7)          return 'This is Student with Name:{} and Rollno:{}'.format(self.name,self.rollno)
8)
9)  s1=Student('Durga',101)
10) s2=Student('Ravi',102)
11) print(s1)
12) print(s2)
```

Output without Overriding str():

```
<__main__.Student object at 0x022144B0>
<__main__.Student object at 0x022144D0>
```

Output with Overriding str():

```
This is Student with Name: Durga and Rollno: 101
This is Student with Name: Ravi and Rollno: 102
```

Difference between str() and repr()
OR
Difference between __str__() and __repr__()

- str() internally calls __str__() function and hence functionality of both is same.
- Similarly,repr() internally calls __repr__() function and hence functionality of both is same.
- str() returns a string containing a nicely printable representation object.
- The main purpose of str() is for readability.It may not possible to convert result string to original object.

```
1) import datetime
2) today=datetime.datetime.now()
3) s=str(today)#converting datetime object to str
4) print(s)
5) d=eval(s)#converting str object to datetime
```

```
D:\durgaclasses>py test.py
2018-05-18 22:48:19.890888
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    d=eval(s)#converting str object to datetime
  File "<string>", line 1
    2018-05-18 22:48:19.890888
          ^
SyntaxError: invalid token
```

But repr() returns a string containing a printable representation of object.
The main goal of repr() is unambigouous. We can convert result string to original object by using eval() function,which may not possible in str() function.

```
1) import datetime
2) today=datetime.datetime.now()
3) s=repr(today)#converting datetime object to str
4) print(s)
5) d=eval(s)#converting str object to datetime
6) print(d)
```

Output:

```
datetime.datetime(2018, 5, 18, 22, 51, 10, 875838)
2018-05-18 22:51:10.875838
```

Note: It is recommended to use repr() instead of str()

Mini Project: Banking Application

```
1)  class Account:
2)      def __init__(self,name,balance,min_balance):
3)          self.name=name
4)          self.balance=balance
5)          self.min_balance=min_balance
6)
7)      def deposit(self,amount):
8)          self.balance +=amount
9)
10)     def withdraw(self,amount):
11)         if self.balance-amount >= self.min_balance:
12)             self.balance -=amount
13)         else:
14)             print("Sorry, Insufficient Funds")
15)
16)     def printStatement(self):
17)         print("Account Balance:",self.balance)
18)
19) class Current(Account):
20)     def __init__(self,name,balance):
21)         super().__init__(name,balance,min_balance=-1000)
22)     def __str__(self):
23)         return "{}'s Current Account with Balance :{}".format(self.name,self.balance)
24)
25) class Savings(Account):
26)     def __init__(self,name,balance):
27)         super().__init__(name,balance,min_balance=0)
28)     def __str__(self):
29)         return "{}'s Savings Account with Balance :{}".format(self.name,self.balance)
30)
31) c=Savings("Durga",10000)
32) print(c)
33) c.deposit(5000)
34) c.printStatement()
35) c.withdraw(16000)
```

```
36) c.withdraw(15000)
37) print(c)
38)
39) c2=Current('Ravi',20000)
40) c2.deposit(6000)
41) print(c2)
42) c2.withdraw(27000)
43) print(c2)
```

Output:

```
D:\durgaclasses>py test.py
Durga's Savings Account with Balance :10000
Account Balance: 15000
Sorry, Insufficient Funds
Durga's Savings Account with Balance :0
Ravi's Current Account with Balance :26000
Ravi's Current Account with Balance :-1000
```