25.OOPs

June 9, 2020

OOPs

[10]:
```
class Person:
    def __new__(cls, *args):
        """
            new is used to create a new object of type cls
        """
        print(f"creating a new object of class {cls}")
        return object.__new__(cls)
    def __init__(self, name, country):
        """
            self is object space, init is used initlize default values at␣
 ↪object creation time
        """
        print("Initlizing Data to newely created object")
        self.name = name
        self.country = country
    def __str__(self):
        return self.name.upper()
```

[11]:
```
p1 = Person('sachin yadav', 'india')
p2 = Person('rajat goyal', 'india')
```

```
creating a new object of class <class '__main__.Person'>
Initlizing Data to newely created object
creating a new object of class <class '__main__.Person'>
Initlizing Data to newely created object
```

[12]:
```
print(p1)
```

```
SACHIN YADAV
```

[13]:
```
print(p2)
```

```
RAJAT GOYAL
```

[14]:
```
p1.name
```

[14]:
```
'sachin yadav'
```

[15]:
```
p1.country
```

[15]:
```
'india'
```

[16]:
```
p2.name
```

[16]:
```
'rajat goyal'
```

[17]:
```
p2.country
```

[17]:
```
'india'
```

[18]:
```
p1.language = 'english' # dynamic binding
```

[19]:
```
p1.language
```

[19]:
```
'english'
```

0.0.1 Data Hiding

by this we can hide some sensitive information inside class or object that can only be accesse
by methods not directly

[20]:
```
class A:
    def __init__(self, msg):
        self.secret = msg
    def get_msg(self):
        """
            getter method to access data of object
        """
        print(self.secret)
    def set_msg(self, new_msg):
        """
            setter method to manipluate data at run time
        """
        self.secret = new_msg
```

[21]:
```
a = A('some information')
```

[22]:
```
a.get_msg()
```

```
some information
```

[23]:
```
a.set_msg('go corona go') # dynamic binding
```

[24]:
```
a.get_msg()
```

```
go corona go
```

[26]:
```
a.secret # public variable
```

[26]:
```
'go corona go'
```

[27]:
```
a.secret = 'i just hacked you' #
```

[28]:
```
a.get_msg()
```

```
i just hacked you
```

__variable is a hidden variable or you can say private variable in python that can only be accessed inside class space

[29]:
```
class A:
    def __init__(self, normal, secret):
        self.msg = normal
        self.__secret = secret
    def get_msg(self):
        print(self.msg)
    def get_secret(self):
        print(self.__secret)
    def set_msg(self, new_msg):
        self.msg = new_msg
```

[31]:
```
a = A('visible outside', 'visible inside')
```

[32]:
```
a.get_msg()
```

```
visible outside
```

[33]:
```
a.msg
```

[33]:
```
'visible outside'
```

[34]:
```
a.get_secret()
```

```
visible inside
```

[35]:
```
a.__secret # data hiding
```

```
␣
↪---------------------------------------------------------------------------
```

```
AttributeError                            Traceback (most recent call last)

<ipython-input-35-1274b3ebe29d> in <module>
----> 1 a.__secret


AttributeError: 'A' object has no attribute '__secret'
```

Name mangling

there is secret variables in python

if you decleare variable with double underscore like this __variable than internally python automatically will change it's name _classname__variable

every object stores data in it, and we can view object data by using object.__dict__ or vars

[52]:
```
class A:
    "hello i am docstring"
    def __init__(self, name):
        self.name = name
    def hello(self):
        print("Hello world")
    def __str__(self):
        return self.name.upper()
```

[53]:
```
vars(A)
```

[53]:
```
mappingproxy({'__module__': '__main__',
              '__doc__': 'hello i am docstring',
              '__init__': <function __main__.A.__init__(self, name)>,
              'hello': <function __main__.A.hello(self)>,
              '__str__': <function __main__.A.__str__(self)>,
              '__dict__': <attribute '__dict__' of 'A' objects>,
              '__weakref__': <attribute '__weakref__' of 'A' objects>})
```

[54]:
```
A.__dict__
```

[54]:
```
mappingproxy({'__module__': '__main__',
              '__doc__': 'hello i am docstring',
              '__init__': <function __main__.A.__init__(self, name)>,
              'hello': <function __main__.A.hello(self)>,
              '__str__': <function __main__.A.__str__(self)>,
              '__dict__': <attribute '__dict__' of 'A' objects>,
              '__weakref__': <attribute '__weakref__' of 'A' objects>})
```

[56]:
```
a = A('sachin')
```

[57]:
```
a.__dict__ # dictionary representation of object
```

[57]:
```
{'name': 'sachin'}
```

[58]:
```
vars(a)
```

[58]:
```
{'name': 'sachin'}
```

[59]:
```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.__salary = salary
    def __str__(self):
        return self.name.upper()
    def get_salary(self):
        return self.__salary
```

[60]:
```
e1 = Employee('sachin', 20000)
```

[61]:
```
print(e1)
```

```
SACHIN
```

[62]:
```
e1.get_salary()
```

[62]:
```
20000
```

[63]:
```
e1.name # ?
```

[63]:
```
'sachin'
```

[66]:
```
e1.__salary #? private
```

```
␣
↪---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call␣
↪last)

<ipython-input-66-6e88f11c5f42> in <module>
----> 1 e1.__salary #? private

AttributeError: 'Employee' object has no attribute '__salary'
```

[67]:
```
vars(e1)
```

[67]:
```
{'name': 'sachin', '_Employee__salary': 20000}
```

Name Mangling - __var --> _classname__var

[68]:
```
e1.__salary
```

```
␣
↪---------------------------------------------------------------------------

       AttributeError                            Traceback (most recent call␣
↪last)

       <ipython-input-68-f90054552aba> in <module>
   ----> 1 e1.__salary


       AttributeError: 'Employee' object has no attribute '__salary'
```

[69]:
```
e1._Employee__salary
```

[69]:
```
20000
```

[70]:
```
e1._Employee__salary = 13000
```

[71]:
```
e1.get_salary()
```

[71]:
```
13000
```

[72]:
```
print(e1)
```

```
SACHIN
```

[74]:
```
e1.__salary = 50000 #  no name, dyanmic binding
```

[75]:
```
e1.get_salary()
```

[75]:
```
13000
```

[76]:
```
e1.__salary
```

[76]:
```
50000
```

[77]:
```
vars(e1)
```

[77]:
```
{'name': 'sachin', '_Employee__salary': 13000, '__salary': 50000}
```

Abstraction

abstraction = data hiding + encapsulation

0.0.2 single level inheritance

[78]:
```
class Parent:
    def car(self):
        print("I have marutii 800")
    def bike(self):
        print("Basic Splendra bike")
```

[79]:
```
class Child(Parent):
    pass
```

[81]:
```
c = Child()
```

[82]:
```
c.car()
```

```
I have marutii 800
```

[83]:
```
c.bike()
```

```
Basic Splendra bike
```

Multi-level Inheritance

[84]:
```
class Dada:
    def haveli(self):
        print("very cool very old haveli")

class Parent(Dada):
    def bike(self):
        print("I have a royal enfield")


class Child(Parent):
    def car(self):
        print("I have a car")
```

[85]:
```
c = Child()
```

[86]:
```
c.haveli()
```

```
very cool very old haveli
```

[87]:
```
c.bike()
```

```
I have a royal enfield
```

[88]:
```
c.car()
```

```
I have a car
```

Over-riding

when parent and child share same methods or data we always use latest information such that onl
childs method and data is accesible by chlid object

self --> access specifer (object)

this --> c++

a.hi # self.hi
. is also access specifier

[89]:
```
class A:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return self.name.upper()
```

[90]:
```
class B(A):
    pass
```

[92]:
```
b = B('sachin')
```

[93]:
```
print(b)
```

```
SACHIN
```

[94]:
```
class A:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return self.name.upper()
```

[95]:
```
class B(A):
    def __init__(self): # over-riding
        self.name = 'ha ha ha over-riding parents __init__ method'
```

[96]:
```
b = B()
```

[97]:
```
print(b)
```

```
HA HA HA OVER-RIDING PARENTS __INIT__ METHOD
```

[103]:
```
class A: # BASE CLASS or Parent Class
    def bike(self):
        print("normal spelendra bike")
    def car(self):
        print("marutii 800")
    def some_fun(self):
        print("some working that should not be used in chlid")
```

[104]:
```
class B(A): # extending Base A class / Derived Class / Child Class
    def bike(self):
        print("Bullet")
    def some_fun(self):
        pass
```

[105]:
```
b = B()
```

[106]:
```
b.car()
```

```
marutii 800
```

[107]:
```
b.bike()
```

```
Bullet
```

[ ]:
```

```

[108]:
```
b.some_fun()
```

[110]:
```
class A:
    def __init__(self, name):
        self.name = name.title().strip()
    def __str__(self):
        return self.name
```

[115]:
```
class B(A):
    def __init__(self, name, country):
        super().__init__(name)
        self.country = country
        # A.__init__(self, name)
```

[116]:
```
b = B('sachin yadav', 'india')
```

[117]:
```
print(b)
```

```
Sachin Yadav
```

[119]:
```
b.country
```

[119]:
```
'india'
```

[120]:
```
class A: # BASE CLASS or Parent Class
    def bike(self):
        print("normal spelendra bike")
    def car(self):
        print("marutii 800")
    def some_fun(self):
        print("some working that should not be used in chlid")
class B(A): # extending Base A class / Derived Class / Child Class
    def bike(self):
        super().bike()
        print("Bullet")
    def some_fun(self):
        pass
```

[121]:
```
b = B()
```

[122]:
```
b.bike()
```

```
normal spelendra bike
Bullet
```

0.1 Hierarchical

[123]:
```
class Parent:
    def bike(self):
        print("some bike")
```

[126]:
```
class Child1(Parent):
    def laptop(self):
        print("surface book 2")
```

[127]:
```
class Child2(Parent):
    def mobile(self):
        print("one plus 8 pro")
```

[128]:
```
c1 = Child1()
```

[129]:
```
c2 = Child2()
```

[130]:
```
c1.bike()
```

```
some bike
```

[133]:
```
c2.bike()
```

```
some bike
```

[134]:
```
c1.laptop()
```

```
surface book 2
```

[135]:
```
c2.mobile()
```

```
one plus 8 pro
```

Multiple Inheritance

[148]:
```
class Papa:
    def change_channel(self):
        print("Switch Channel to news channel")
    def pocket_money(self):
        print("Source of our Income")

class Mummy:
    def change_channel(self):
        print("Switch to star plus i want to watch saas bhi kabhi bahu thi")
    def pyar(self):
        print("Source of infinte love")
```

[149]:
```
class Child(Mummy, Papa):
    pass
```

[150]:
```
c = Child()
```

[151]:
```
c.change_channel() # ?
```

```
Switch to star plus i want to watch saas bhi kabhi bahu thi
```

```
ambiguity -> state of confusion where you can decide which selection is good
```

```
name mangling
```

```
Method Resolution Order
```

[152]:
```
help(Child)
```

```
Help on class Child in module __main__:

class Child(Mummy, Papa)
 |  Method resolution order:
 |      Child
 |      Mummy
 |      Papa
 |      builtins.object
 |
 |  Methods inherited from Mummy:
 |
```

```
 |  change_channel(self)
 |
 |  pyar(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Mummy:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from Papa:
 |
 |  pocket_money(self)
```

Polymorphism

```
* function overloading -> no applicable in python just because over-riding

* Operator Overloading
```

[153]:
```
class A:
    def __add__(self, other):
        return 'bhai bhai'
```

[154]:
```
a1 = A()
```

[155]:
```
a2 = A()
```

[156]:
```
a1 + a2
```

[156]:
```
'bhai bhai'
```

```
+   __add__
-   __sub__
*   __mul__
/   __truediv__
//  __floordiv__
**  __pow__

<   __lt__
<=  __le__
>   __gt__
>=  __ge__
==  __eq__
```

```
!=  __nq__

+= __iadd__
-= __isub__
...


len __len__
```

Message Passing

when object pass messeges to each using shared memory

[158]:
```
class A:
    msg = 'class variable or shared variable'
    def show(self):
        print(self.msg)
    def update(self, msg):
        A.msg = msg
```

[159]:
```
a = A()
b = A()
```

[160]:
```
a.show()
```

```
class variable or shared variable
```

[161]:
```
a.update('hello other budy')
```

[162]:
```
b.show()
```

```
hello other budy
```

[163]:
```
b.update('ha ha ha')
```

[164]:
```
a.show()
```

```
ha ha ha
```

[165]:
```
import time
time.ctime()
```

[165]:
```
'Tue Jun  9 20:45:25 2020'
```

[174]:
```
class Grras:
    notice_board = 'Welcome to Grras Notice Board'
    def __init__(self, name, subject):
        self.name = name
```

```
        self.subject = subject
    def show_info(self):
        print("Name: ", self.name)
        print("Subject: ", self.subject)
    def __str__(self):
        return self.name.title()
    def show_notice_board(self):
        print(Grras.notice_board) # class variable
    def update_notice_board(self, new_msg):
        Grras.notice_board += "\n" + time.ctime() +"\t"+ new_msg + f"--> {self.
↪name.title()}"
```

[175]:
```
sachin = Grras('sachin yadav', 'data science')
rajat = Grras('rajat goyal', 'cloud computing')
```

[176]:
```
sachin.show_notice_board()
```

```
Welcome to Grras Notice Board
```

[177]:
```
sachin.update_notice_board("Today, I am on Leave manage accordingly.")
```

[178]:
```
rajat.show_notice_board()
```

```
Welcome to Grras Notice Board
Tue Jun  9 20:48:04 2020        Today, I am on Leave manage accordingly.-->
Sachin Yadav
```

[179]:
```
rajat.update_notice_board('Okay!! i will handle your batches')
```

[180]:
```
sachin.show_notice_board()
```

```
Welcome to Grras Notice Board
Tue Jun  9 20:48:04 2020        Today, I am on Leave manage accordingly.-->
Sachin Yadav
Tue Jun  9 20:48:26 2020        Okay!! i will handle your batches--> Rajat Goyal
```

[181]:
```
sachin.update_notice_board('thanks')
```

[182]:
```
rajat.show_notice_board()
```

```
Welcome to Grras Notice Board
Tue Jun  9 20:48:04 2020        Today, I am on Leave manage accordingly.-->
Sachin Yadav
Tue Jun  9 20:48:26 2020        Okay!! i will handle your batches--> Rajat Goyal
Tue Jun  9 20:48:52 2020        thanks--> Sachin Yadav
```

Basic OOPs in Python

slots

property
class method
static method

meta
abstract

Assignment

create a vector class and it's operations with operator

[ ]: