Start of OOPs in Python

In [1]:
```
class Car:
    pass

# initialize an object
car1 = Car()
car1
```

Out[1]:
```
<__main__.Car at 0x7f9bcc5be760>
```

In [2]:
```
car1.windows=5
car1.doors=5
```

In [3]:
```
car2 = Car()
car2.windows=3
car2.doors=2

print(car2.windows)
```

```
3
```

In [4]:
```
car2.enginetype="petrol"
print(car2.enginetype)
```

```
petrol
```

so far we have seen that there is no limit on the number of attributes. It is a bad approach. To overcome this, there is __init__() in built function. It is initialization constructor, used to initialize the number of properties that should be initialized inside of that particular class.

even if no 'Pass' is mentioned inside the class definition, instances can create its own attributes. These attributes is limited to that instance only, and is not creted by default for any other instances. These attributes can be easily used in class defined functions, if that method's definitionhas mention of that attribute created, like "self.\<out_attr>"

In [5]:
```
# prog to add two complex numbers using custom funvtion

class Complex:
    def __init__(self, real, img):
        self.real=real
        self.img=img

    def add(self, number):
        real=self.real+number.real
        img=self.img+number.img
        result=Complex(real, img)
        return result

n1=Complex(4,5)
n2=Complex(3,2)

result=n1.add(n2)
print(result.real, result.img)
```

```
7 7
```

In [6]:
```
dir(car1)
```

Out[6]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
```

```
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'doors',
 'windows']
```

In [7]:
```
class Samp():
    def __init__(self, atr1, atr2, atr3):
        self.atr1=atr1
        self._atr2=atr2
        self.__atr3=atr3
```

In [8]:
```
obj1=Samp(2,3,6)
```

In [9]:
```
dir(obj1)
```

Out[9]:
```
['_Samp__atr3',
 '__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
```

```
'__reduce__',
'__reduce_ex__',
'__repr__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'__weakref__',
'_atr2',
'atr1']
```

In [10]:
```
class Caar:
    def __init__(self, window, door, enginetype):
        # creating/initializing attributes/variables(windows, doors, enginetypes) inside this class
        self.windows=window    # no underscore : public access specifier
        self._doors=door    # 1 underscore : protected access specifier
        self._enginetypes=enginetype    # 2 underscores : private access specifier

    def self_driving(self):
        return "This is a {} engine car".format(self._enginetypes)
```

In [11]:
```
caar1=Caar(4, 5, "petrol")
# as soon as Caaris called, by defalult __init__() constructor is called, and it initializes the variables/attribute
```

the self is referencing the object, and for that object all the variables/attributs are created

In [12]:
```
caar2=Caar(3,4,"diesel")
```

In [13]:
```
print(caar1.windows)
print(caar2._doors)
print(caar2._enginetypes)
```

```
4
4
diesel
```

In [14]:
```
caar1.self_driving()
```

Out[14]:
```
'This is a petrol engine car'
```

INHERITANCE

In [15]:
```
class Caaar:
    def __init__(self, window, door, enginetype):
        self.windows=window
        self.doors=door
        self.enginetypes=enginetype

    def drives(self):
        print("The person drives the car")
```

making child class

In [16]:
```
class audi(Caaar):
    def __init__(self, windows, doors, enginetypes, enableai):
        super().__init__(windows, doors, enginetypes)
        self.enableai=enableai
    def selfdriving(self):
        print("Audi supports self-driving")
```

In [17]:
```
audiQ7=audi(5,5,"diesel", True)
```

In [18]:
```
dir(audiQ7)
```

Out[18]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
```

```
'__gt__',
'__hash__',
'__init__',
'__init_subclass__',
'__le__',
'__lt__',
'__module__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'__weakref__',
'doors',
'drives',
'enableai',
'enginetypes',
'selfdriving',
```

MULTIPLE INHERITANCE

In [19]:
```
class A:
    def method1(self):
        print("A class method is called here.")

    def meth_a(self):
        print("this is met A!")
```

In [20]:
```
class B(A):
    def method1(self):
        print("B class method is called here.")

    def meth_b(self):
        print("this is met B!")
```

In [21]:
```
class C(A):
    def method1(self):
        print("C class method is called here.")

    def meth_c(self):
        print("this is met C!")
```

In [22]:
```
class D(B,C):
    def method1(self):
        print("D class method is called here.")

    def meth_d(self):
        print("this is met D!")
```

Making instance of class D

In [23]:
```
d=D()
```

In [24]:
```
d.meth_d()
```

```
this is met D!
```

In [25]:
```
d.method1()
```

```
D class method is called here.
```

In [26]:
```
# calling method1 of parent class with the instance of child class. (two methods)
B.method1(d)    # M1
B.meth_b(d)    # M1

d.meth_b()    # M2
d.meth_a()
```

```
B class method is called here.
this is met B!
this is met B!
```

In [27]:
```
C.method1(d)
C.meth_c(d)
```

```
C class method is called here.
this is met C!
```

In [28]:
```
# since A is parent of parent of D, so relevant attributes of A will also be inherited to D
A.method1(d)
A.meth_a(d)
```

```
A class method is called here.
this is met A!
```

Calling methods from the parent class as soon as class method is called

In [29]:
```
class E(D):
    def method1(self):
        print("E class method is called here.")
        A.method1(self)    # calling parent class method using class. requires self keyword
        B.method1(self)
        C.method1(self)
        D.method1(self)
        # super() return temporary object of the super class of the subclass. Simply, its like an object of an paren
        super().method1()    # this doesn't need self keyword, and is orthodox method to call parent class method
        # super().super().method1()  # bakaiti nahi chalta idhar
```

In [30]:
```
e=E()
```

In [31]:
```
e.method1()
```

```
E class method is called here.
A class method is called here.
B class method is called here.
C class method is called here.
D class method is called here.
D class method is called here.
```

OOPs magic methods in Class

Some magic happens in the background that helps to create the class's object.

In [32]:
```
c=Caaar(4,5,"diesel")
```

In [33]:
```
c
```

Out[33]:
```
<__main__.Caaar at 0x7f9bcc55cdc0>
```

In [34]:
```
dir(c)    # all the double underscore enclodes methods listed below are called magic methods
```

Out[34]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
```

```
'__weakref__',
'doors',
'drives',
'enginetypes',
```

In [35]:
```
class Caaar:
    def __init__(self, window, door, enginetype):
        self.windows=window
        self.doors=door
        self.enginetypes=enginetype

    def drives(self):
        print("The person drives the car")

    def __str__(self):
        return "The object has been initialized"
```

In [36]:
```
c=Caaar(4, 5, "Diesel")
```

We can override magic methods. print(\<objects>) calls __str()__. Here we'll override it, with our custom function definition.

In [37]:
```
print(c)
```

```
The object has been initialized
```

In [38]:
```
c
# it'll be the same as it was giving in print(c). But now we've overridden __str__() which is called by print(<obj>)
```

Out[38]:
```
<__main__.Caaar at 0x7f9bcc594370>
```

In [39]:
```
c.__sizeof__()
```

Out[39]:
```
32
```

In [40]:
```
c.__str__()
```

Out[40]: 'The object has been initialized'

In [41]:
```
class Caaar:
    def __new__(self, window, door, enginetype):
        print("New is being called even before init. It is called during class\'s object creation")
    def __init__(self, window, door, enginetype):
        self.windows=window
        self.doors=door
        self.enginetypes=enginetype

    def drives(self):
        print("The person drives the car")

    def __str__(self):
        return "The object has been initialized"
```

In [42]:
```
c=Caaar(6, 7, "Petrolllllll")
```

```
New is being called even before init. It is called during class's object creation
```

Class methods and Class Variables

In [43]:
```
class Bike:
    base_price = 100000    # class var. Any obj of this class will have common base price at any instance. Value may

    def __init__(self, window, door, power):
        self.windows = window
        self.doors = door
        self.power = power

    def what_base_price(self):
        print("The base price : {}".format(self.base_price))

    @classmethod    # decorator
    def revise_base_price(cls, inflation):
        cls.base_price = cls.base_price + cls.base_price * inflation
```

ways to access base price:

In [44]:
```
bike1 = Bike(4, 5, 2000)
print(bike1.base_price)    # M1
bike1.what_base_price()    # M2

print(Bike.base_price)    # M3
```

```
100000
The base price : 100000
100000
```

In [45]:
```
# updating inflated rate for the first time
Bike.revise_base_price(0.10)
```

In [46]:
```
print(bike1.base_price)    # M1
bike1.what_base_price()    # M2

print(Bike.base_price)    # M3
```

```
110000.0
The base price : 110000.0
110000.0
```

Here we are seeing that the class variable is also getting updated from the object. But this not allowed in real world practice. So avoid/restrict updation of class variable using its instance.

In [47]:
```
bike1.revise_base_price(0.10)
```

In [48]:
```
print(bike1.base_price)    # M1
bike1.what_base_price()    # M2

print(Bike.base_price)    # M3
```

```
121000.0
The base price : 121000.0
121000.0
```

In [49]:
```
bike2 = Bike(9, 12, 2000)
```

In [50]:
```
# for object bike2
print(bike2.base_price)     # M1
bike2.what_base_price()     # M2

print(Bike.base_price)     # M3
```

```
121000.0
The base price : 121000.0
121000.0
```

Static Methods in Python

As soon as the class gets loaded, the first thing that gets initialized is STATIC METHOD. Its lifetime is until when application is running. Unlike the other way, it doesn't get initialized everytime, when the instance being made. Initialize class for just once, and it will live forever. It can be called with classname, instancename, etc. It is fast.

In [51]:
```
class Bike:
    base_price = 100000    # class var. Any obj of this class will have common base price at any instance. Value may

    def __init__(self, window, door, power):
        self.windows = window
        self.doors = door
        self.power = power

    def what_base_price(self):
        print("The base price : {}".format(self.base_price))

    @classmethod    # decorator
    def revise_base_price(cls, inflation):
        cls.base_price = cls.base_price + cls.base_price * inflation

    @staticmethod
    def check_year(year):    # we don't provide here neither INSTANCE nor CLASS parameter (or dont provide any param
        if year==2021:
            return True
        else:
            return False
```

In [52]:
```
Bike.check_year(2021)
```

Out[52]: True

In [53]:
```
bike1 = Bike(4, 5, 2000)
```

In [54]:
```
bike1.check_year(2034)
```

Out[54]: False

Use Case: If check_year gives True in statement *if(bike1.check_year()*, then *Pass*. Else, call *Bike.revise_base_price* and update the base_price.