Class variable(static variable) and Instance Variables

- class vairbales and static variables are same

In [1]:

```python
class Car:
    # static or class level variable
    #can be updated using class name for all objects
    wheels = 4
    def __init__(self):
    # non static or object/instance level variable
    # can be updated using object name for a particular object
        self.mileage = 10
        self.company = "BMW"
```

In [2]:

```python
c1  = Car()
c2 = Car()
```

In [6]:

```python
# since wheel is a class level variable can be accessed using class name
print(Car.wheels)
print(c1.wheels)

print(c1.mileage)
Car.mileage

```

```
4
4
10

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[6], line 6
      3 print(c1.wheels)
      5 print(c1.mileage)
----> 6 Car.mileage

AttributeError: type object 'Car' has no attribute 'mileage'
```

In [7]:

```python
# since wheel is a class level variable, can be accesses using class name
c1.wheels
```

Out[7]:

```
4
```

In [8]:

```python
# class level variable is updeated using class, will be pdated for every o
Car.wheels = 5
```

In [9]:

```python
Car.wheels, c1.wheels,c2.wheels
```

Out[9]:

```
(5, 5, 5)
```

In [10]:

```
# add a new attribute in my object names wheels
c1.wheels = 6
c1.color = 'Black'
```

In [11]:

```
Car.wheels, c1.wheels,c2.wheels
```

Out[11]:

```
(5, 6, 5)
```

In [15]:

```
Car.wheels = 10
c2.wheels = 30
```

In [16]:

```
Car.wheels, c1.wheels,c2.wheels
```

Out[16]:

```
(10, 6, 30)
```

In [ ]:

```
c1.owner = "Abdullah"
```

In [ ]:

```
Car.color = "Red"
```

In [ ]:

```

```

Class methods , Instance methods , static methods

- Class methods and startic methods are not same

In [17]:

```
class Student:
    school = "SSUET"
    def __init__(self, m1,m2,m3):
        self.m1 = m1
        self.m2 = m2
        self.m3 = m3

    # instance methods
    def avg(self):
        return (self.m1+self.m2+self.m3)/3

    def information(self):
        return self.school

```

In [18]:

```python
 1  s1 = Student(89,98,90)
 2  s2 = Student(80,90,70)
 3  # instance method called using instance  s1
 4  print(s1.avg())
 5  
 6  # instance method called using instance  s2
 7  print(s2.avg())
 8  
 9  # instance method called using class name but instance is passed a argumen
10  print(Student.information(s1))
11  print(Student.information(s2))
12  print(Student.school)
13  print(s1.school)
14  
15  # instance method called using class name>>error
16  print(Student.information())
17  print(Student.avg())
18  
```

```
92.33333333333333
80.0
SSUET
SSUET
SSUET
SSUET

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[18], line 16
     13 print(s1.school)
     15 # instance method called using class name>>error
---> 16 print(Student.information())
     17 print(Student.avg())

TypeError: Student.information() missing 1 required positional argument: 'sel
f'
```

avg is an instance method it required a instance/object to be called.
information is a also a instance method which requires instance/object to be called.

any instance method can not be called using a class name.

we need a class method to be called by using a class name

class methods can be called using class name as well as instance name
to make a methods class method we use a decorator @classmethod
this way information method can be called using an object as well as a class

In [19]:

```python
class Student:
    school = "SSUET"
    def __init__(self, m1,m2,m3):
        self.m1 = m1
        self.m2 = m2
        self.m3 = m3
    def avg(self):
        return (self.m1+self.m2+self.m3)/3

    @classmethod
    def information(cls):
        return cls.school
```

In [20]:

```python
s1 = Student(89,98,90)
s2 = Student(80,90,70)
# instance method called using instance  s1
print(s1.avg())
# instance method called using instance  s2
print(s2.avg())

########Class method can be called using class as well as instance######

# class method is called using instance
print(s1.information())
print(s2.information())

# class method is called using class name
print(Student.information())
print(Student.information())
```

```
92.33333333333333
80.0
SSUET
SSUET
SSUET
SSUET
```

Static Methods

- static methods are methods that donot require instance or class

In [21]:

```
class Student:
    school = "SSUET"
    def __init__(self, m1,m2,m3):
        self.m1 = m1
        self.m2 = m2
        self.m3 = m3
    def avg(self):
        return (self.m1+self.m2+self.m3)/3

    @classmethod
    def information(c):
        return c.school

    @staticmethod
    def hello():
        return "Im a static method"
```

In [22]:

```
s4 = Student(23,24,25)
```

In [23]:

```
s4.hello()
```

Out[23]:

```
'Im a static method'
```

In [24]:

```
Student.hello()
```

Out[24]:

```
'Im a static method'
```

In [26]:

```
class Battery():
    def __init__(self,manuf, cell, weight, amp, watt,price):
        self.manuf =manuf
        self.cell =cell
        self.weight=weight
        self.amp =amp
        self.watt =watt
        self.price =price
```

In [30]:

```
class ElecCar:
    def __init__(self,make,model,year,engine):
        self.make =make
        self.model = model
        self.year = year
        self.engine =engine
        # instance used as attributes
        self.battery = Battery("Osaka",27,60,200,12,60000)
    def carruns(self):
        pass
    def carstop(self):
        pass

```

In [34]:

```python
e = ElecCar('honda',2024,2024,2000)
e.battery.manuf
```

Out[34]:

```
'Osaka'
```

In [ ]:

```python

```

Inner Classes

- Class inside a class is called inner class

In [35]:

```python
class Student:
    def __init__(self, name, rollno):
        self.name =name
        self.rollno=rollno
    def show(self):
        print(self.name, self.rollno)
s1 = Student('Nasir',2)
s2 = Student('Hassan',3)

s1.show()
s2.show()
```

```
Nasir 2
Hassan 3
```

```
Case: A student in IT class must hava laptop
so there is an attribute of laptop for student
```

In [36]:

```python
class Student:
    def __init__(self, name, rollno,laptop):
        self.name =name
        self.rollno=rollno
        self.laptop = laptop
    def show(self):
        print(self.name, self.rollno,self.laptop)
s1 = Student('Nasir',2, "HP")
s2 = Student('Hassan',3,"Lenovo")

s1.show()
s2.show()
```

```
Nasir 2 HP
Hassan 3 Lenovo
```

What if i need to add more detail of my laptop??

Should i send the details as arguments to init??

We can create a separate class for laptop and use its object as attribute to Student

or we can also create an inner class of laptop inside Student

Instance as attribute

A class outside the class, its instance can be used as attribute in another class

In [37]:

```python
class Laptop:
    def __init__(self,brand,cpu,ram):
        self.brand =brand
        self.cpu =cpu
        self.ram =ram

    def show(self):
        print(self.brand,self.cpu,self.ram)
```

In [38]:

```python
class Student:
    def __init__(self, name, rollno):
        self.name =name
        self.rollno=rollno
        self.laptop = Laptop("Hp","Corei7",16)

    def show(self):
        print(self.name, self.rollno)
        self.laptop.show()


s1 = Student('Nasir',2,)
s2 = Student('Hassan',3,)

s1.show()

s2.show()

```

```
Nasir 2
Hp Corei7 16
Hassan 3
Hp Corei7 16
```

Inner Class

- A class clreated inside a class is called inner class

In [39]:

```python
class Student:
    def __init__(self, name, rollno):
        self.name =name
        self.rollno=rollno
        self.laptop = self.Laptop("Hp","i7",'16Gb')

    def show(self):
        print(self.name, self.rollno)
        self.laptop.show()

    class Laptop:
        def __init__(self,brand,cpu,ram):
            self.brand =brand
            self.cpu =cpu
            self.ram =ram
        def show(self):
            print(self.brand,self.cpu,self.ram)

s1 = Student('Nasir',2)
s2 = Student('Hassan',3)

s1.show()

s2.show()

```

```
Nasir 2
Hp i7 16Gb
Hassan 3
Hp i7 16Gb
```

In [6]:

```python
s1.laptop.show()
s2.laptop.show()
```

```
Hp i7 16Gb
Hp i7 16Gb
```

Inheritance

In [40]:

```python
class A:
    def feature1(self):
        print("Feature1 is working")
    def feature2(self):
        print("Feature2 is working")


a1 = A()
a1.feature1()
a1.feature2()
```

```
Feature1 is working
Feature2 is working
```

Single inheritance

In [41]:

```
class B(A):
    def feature3(self):
        print("Feature3 is working")
    def feature4(self):
        print("Feature4 is working")
b1 = B()

b1.feature1()
b1.feature3()
b1.feature4()
b1.feature2()
```

```
Feature1 is working
Feature3 is working
Feature4 is working
Feature2 is working
```

Multilevel Inheritance

In [42]:

```
class C(B):
    def feature5(self):
        print("Feature5 is working")
    def feature6(self):
        print("Feature6 is working")
c1 = C()

c1.feature1()
c1.feature2()
c1.feature3()
c1.feature4()
c1.feature5()
c1.feature6()
```

```
Feature1 is working
Feature2 is working
Feature3 is working
Feature4 is working
Feature5 is working
Feature6 is working
```

Multiple Inheritance

In [43]:

```python
class A:
    def feature1(self):
        print("Feature1 is working")
    def feature2(self):
        print("Feature2 is working")


a1 = A()
a1.feature1()
a1.feature2()
```

```
Feature1 is working
Feature2 is working
```

In [44]:

```python
class B():
    def feature3(self):
        print("Feature3 is working")
    def feature4(self):
        print("Feature4 is working")
b1 = B()
b1.feature3()
b1.feature4()

```

```
Feature3 is working
Feature4 is working
```

In [45]:

```python
class C(A,B):
    def feature5(self):
        print("Feature5 is working")
c1 = C()
c1.feature1()
c1.feature2()
c1.feature3()
c1.feature4()
c1.feature5()
```

```
Feature1 is working
Feature2 is working
Feature3 is working
Feature4 is working
Feature5 is working
```

Contructor(initializer) in Inheritance and Method Resolution Order

In [46]:

```python
class A:
    def __init__(self):
        print("In init of A")

    def feature1(self):
        print("Feature1 is working")
    def feature2(self):
        print("Feature2 is working")

class B(A):

    def feature3(self):
        print("Feature3 is working")

    def feature4(self):
        print("Feature4 is working")
a1=A()
b1=B()

# constructor of A will be called even if we are creating
# object of B since B dont have any contrcutor
```

```
In init of A
In init of A
```

In [47]:

```python
class A:

    def __init__(self):
        print("In init of A")

    def feature1(self):
        print("Feature1 is working")
    def feature2(self):
        print("Feature2 is working")

class B(A):
    def __init__(self):
        print("In init of B")


    def feature3(self):
        print("Feature3 is working")

    def feature4(self):
        print("Feature4 is working")
a1=A()
b1=B()

# constructor of B will be called now as object of B is created
```

```
In init of A
In init of B
```

In [48]:

```python
class A:
    def __init__(self):
        print("In init of A")

    def feature1(self):
        print("Feature1 is working")
    def feature2(self):
        print("Feature2 is working")

class B(A):
    def __init__(self):
        print("In init of B")
        super().__init__()

    def feature3(self):
        print("Feature3 is working")

    def feature4(self):
        print("Feature4 is working")
a1=A()
b1=B()

# if we want to call the init of A when object of B
#is creating we will use super()
```

```
In init of A
In init of B
In init of A
```

In [49]:

```python
class A:
    def __init__(self):
        print("In init of A")

    def feature1(self):
        print("Feature1 is working")

    def feature2(self):
        print("Feature2 is working")
    def show(self):
        print("I am showing class A")
```

In [50]:

```python
class B:
    def __init__(self):
        print("in init of B")


    def feature3(self):
        print("Feature3 is working")

    def feature4(self):
        print("Feature4 is working")

    def show(self):
        print("I am showing class B")
```

In [53]:

```python
class C(A,B):
    def __init__(self):
        print("init of C")
        super().__init__()
        # now we have two parent classes super will call init of???
        # There is a term called MRO
        # Method resolution is from Left to Right
        # init of A will be called


c  = C()
```

```
init of C
In init of A
```

In [54]:

```python
# show() of class A will be called (MRO)
c.show()
```

```
I am showing class A
```

In [55]:

```
class C(B,A):
    def __init__(self):
        print("init of C")
        super().__init__()
        # now we have two parent classes super will call init of???
        # There is a term called MRO
        # Method resolution is from Left to Right
        # init of A will be called
c  = C()
```

```
init of C
in init of B
```

In [56]:

```
# show() of class B will be called (MRO)
c.show()
```

```
I am showing class B
```

In [57]:

```python
class A:
    def __init__(self):
        print("In init of A")

    def feature1(self):
        print("Feature1 is working")

    def feature2(self):
        print("Feature2 is A working")
    def show(self):
        print("I am showing class A")


class B():
    def __init__(self):
        print("in init of B")


    def feature2(self):
        print("Feature2 B is working")

    def feature4(self):
        print("Feature4 is working")

    def show(self):
        print("I am showing class B")


class C(A,B):
    def __init__(self):
        print("init of C")
        super().__init__()
        super().show()
    def feature2(self):
        print("I m in C")

    def feat(self):

        super().feature2()

c  = C()
c.feat()
```

```
init of C
In init of A
I am showing class A
Feature2 is A working
```

Polymorphism

can be implemented by the following techniques:

- Duck typing
- Operator overloading

- Method Overloading
- Method Overriding

Duck Typing

If there a bird which is: - walking like a duck - which is quaking like a duck - which is swimming like a duck then it is a duck

Means its behaviour is just like a duck although it not a duck

In [59]:

```
class Student:
    def useLibrary(self):
        print("Reading Books")
        print("Making Notes")

s1 = Student()
################################
class Teacher:
    def useLibrary(self):
        print("Reading Books")
        print("Making Notes")
        print("Prepare Question Paper")

t1 = Teacher()
#####################################

class Library:
    def welcome(self, obj):
        obj.useLibrary()


lib  =Library()

lib.welcome(s1)
lib.welcome(t1)
```

```
Reading Books
Making Notes
Reading Books
Making Notes
Prepare Question Paper
```

Another Example of Duck Typing

In [24]:

```
class PyCharm:
    def execute(self):
        print("Compiling")
        print("Running")
ide1 = PyCharm()

#################


class VsCode:
    def execute(self):
        print("Grammar Checking")
        print("Spell checking")
        print("Compiling")
        print("Running")
ide2 = VsCode()
#############################

class Laptop:
    def code(self,ide):
        ide.execute()

lap1 = Laptop()
lap1.code(ide1)
lap1.code(ide2)

# it matters not what class is but it must have a method execute
# like if it has a behaviour like a duck than it is a duc
```

```
Compiling
Running
Grammar Checking
Spell checking
Compiling
Running
```

Operator Overloading

In [45]:

```
a = "hello"
class Merainteger:
    pass
mera_int1 = Merainteger()
print(type(mera_int1))
print(a)
print(mera_int1)

```

```
<class '__main__.Merainteger'>
hello
<__main__.Merainteger object at 0x0000019C2CA07490>
```

In [39]:

```
a = 10
b = 20

print(a + b)
# when we use a + operator, in backend it calls add method of int class(th
# because both the supplied operands are of type(class) integers \
# or we can say that both are objects of int class
print(int.__add__(a,b))
```

```
30
30
```

In [41]:

```
a = "10"
b = "20 "

print(a + b)
# when we use a + operator, in backend it calls add method of str class(th
# because both the supplied operands are of type(class) string
# or we can say that both are objects of string class
print(str.__add__(a,b))
```

```
1020
1020
```

In [46]:

```
class Student:
    def __init__(self, m1,m2):
        self.m1 = m1
        self.m2 = m2
s1 = Student(80,90)
s2 = Student(70,90)
```

In [47]:

```
# Can we add objects of Student class ?????
s1 + s2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[47], line 2
      1 # Can we add objects of Student class ?????
----> 2 s1 + s2

TypeError: unsupported operand type(s) for +: 'Student' and 'Student'
```

In [ ]:

```
#Since we have not defined any add function in student class that can add
# We will over load add method in our student class also
```

In [48]:

```
class Student:
    def __init__(self, m1,m2):
        self.m1 = m1
        self.m2 = m2

    def __add__(self,other):
        m1 = self.m1 + other.m1
        m2 = self.m2 + other.m2

        return Student(m1,m2)


    def __gt__(self,other):
        sum_s1 = other.m1 + other.m2
        sum_s2 = self.m1 + self.m2
        if sum_s1 > sum_s2:return True
        else:return False


s1 = Student(80,90)
s2 = Student(70,60)
```

In [54]:

```
s3 = s1 + s2

```

Out[54]:

```
150
```

In [ ]:

```
print(s3)
# it will print the address of the object
# if we want to print the value
# we need to override a function __str__
```

In [55]:

```
s2>s1
```

Out[55]:

```
True
```

In [56]:

```
s1>s2
```

Out[56]:

```
False
```

In [57]:

```
help(str)
```

```
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |  
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |  
 |  Methods defined here:
 |  
 |  __add__(self, value, /)
 |      Return self+value.
 |  
```

In [59]:

```python
class Student:
    def __init__(self, m1,m2):
        self.m1 = m1
        self.m2 = m2

    def __add__(self,other):
        m1 = self.m1 + other.m1
        m2 = self.m2 + other.m2
        newObj = Student(m1,m2)
        return newObj

    def __str__(self):
        return f" Hello I am a student: {self.m1} {self.m2}"


    def __gt__(self,other):
        sum_s1 = other.m1 + other.m2
        sum_s2 = self.m1 + self.m2
        if sum_s1 > sum_s2:return True
        else:return False


s1 = Student(80,90)
s2 = Student(70,60)
```

In [60]:

```python
s4 = s1 + s2
```

In [63]:

```
print(s4)
print(s1)
print(s2)
```

```
 Hello I am a student: 150 150
 Hello I am a student: 80 90
 Hello I am a student: 70 60
```

In [64]:

```
class Student:
    def __init__(self, m):
        self.m = m


    def __add__(self,other):
        new_m = self.m + other.m
        #130       60      70

        newObj = Student(new_m)#130
        return newObj

    def __str__(self):
        return f"{self.m}"


    def __gt__(self,other):
        other = other.m
        self = self.m
        if self > other:return True
        else:return False


s1 = Student(60)
s2 = Student(70)
```

In [65]:

```
print(s1)
print(s2)
s3 = s1+s2
print(se)
```

```
60
70
130
```

In [68]:

```
print(s1>s2)
```

```
False
```

In [69]:

```
print(s2>s1)
```

```
True
```

Abstraction

Abstract Class and Methods in Python

- Python does not support Abstraction
- we will use a module ABC for abstraction
- ABC means Abstract Base Classes

In [1]:

```python
# A normal class and a normal method
class Computer:
    def process(self):
        print('running')

```

In [3]:

```python
# A method that only has declaration but has nothing in it is method
class Computer:
    def process(self):
        pass
# A class having a methods that has no body
```

```
Hiding the implementation details of a method is called abstraction
We can not create an object of abstract classes
```

In [4]:

```python
com1 = Computer()
com1.process()

# There is no error and we are able to create onject and call method
# because its not an abstract class and not an abstract method.
```

In [5]:

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

In [11]:

```python
class Computer(ABC):
    @abstractmethod
    def process(self):
        pass

    @abstractmethod
    def greet(self):
        print("Hello")
#        To make a class abstract
#           - It must inherit the ABC class from abc module
#           - It must have atleast a abstract method (which is defined usi

```

In [12]:

```python
com1 = Computer()

# We can not create the objects of abstract classes
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[12], line 1
----> 1 com1 = Computer()

TypeError: Can't instantiate abstract class Computer with abstract methods gr
eet, process
```

In [15]:

```python
class Laptop(Computer):

    def process(self):
        print("It is running")
#     def greet(self):
#         print("Salam")
lap1 = Laptop()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[15], line 7
      4         print("It is running")
      5 #     def greet(self):
      6 #         print("Salam")
----> 7 lap1 = Laptop()

TypeError: Can't instantiate abstract class Laptop with abstract method greet
```

```
If we want to instantiate a drive class we must have to supply the implememtation of all the abstract methods of the abstract class.
```

What is the use this concept or functionality

```
Through abstraction we can provide a user an interface that only show the behaviour not the implmetation of that behaviour
Like we show a user a computer that runs process and user can only see the name of behavoiur but there is no implementation in it.
Implementation is done in child class
```

In [16]:

```python
from abc import ABC, abstractmethod
class Car(ABC):
    @abstractmethod
    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()
```

```
The mileage is 30kmph
The mileage is 27kmph
The mileage is 25kmph
The mileage is 24kmph
```

In [17]:

```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()
```

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

Python OOPs Public, Protected and Private

```
Public private and protected functionalities are highly restricted (strongly typed) in most of the typed languages
But in python you will not be restricted to access public private and protected variables, they can be overridden.
```

In [61]:

```
# All class ariables are public by defualt
# All instance variables are public by default
class Car():
    # public class variable can be accessed from any where
    wheels = 4
    def __init__(self,windows, doors, enginetype):

        #Public instance Variable can be acceesed from anywhere
        self.windows =windows
        self.doors =doors
        self.enginetype =enginetype
```

In [62]:

```
audi = Car(4,5,"Diesel")

# you can view the dir of audi object to check the aceessible item to it
# you will notice all three instance vairbbles are present in the list
dir(audi)
```

Out[62]:

```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'doors',
 'enginetype',
 'wheels',
 'windows']
```

In [63]:

```
class Suzuki(Car):
    def __init__(self,windows,doors,enginetype, hp):
        super().__init__(windows,doors,enginetype)
        self.hp = hp
suz = Suzuki(4,4,"Petrol","1600")

# you will notice public variables are all accessible to child class also
dir(suz)
```

Out[63]:

```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'doors',
 'enginetype',
 'hp',
 'wheels',
 'windows']
```

Proof of concept

In [64]:

```
#public variable can be accessed

print(audi.windows)

# public variable can be modified
audi.windows= 6
# accessing modified value
print(audi.windows)
```

```
4
6
```

In [65]:

```python
# to make a variable protected use a single underscore before a variable n
class Car():
    def __init__(self,windows, doors, enginetype):

        #Protected variables: should be acceesed from a sub class only by
        # but python dont restrict actually
        self._windows =windows
        self._doors =doors
        self._enginetype =enginetype
        self.hello = "heoooooooo"


audi = Car(4,5,"Diesel")
dir(audi)

class Suzuki(Car):
    def __init__(self,windows,doors,enginetype, hp):
        super().__init__(windows,doors,enginetype)
        self.hp = hp
suz = Suzuki(4,4,"Petrol","1600")

dir(suz)
```

Out[65]:

```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 '_doors',
 '_enginetype',
 '_windows',
 'hello',
 'hp']
```

In [ ]:

```
audi.hello
```

In [ ]:

```
audi._windows = 10
```

In [ ]:

```
audi._windows
```

In [66]:

```
class Car():
    def __init__(self,windows, doors, enginetype):

        #Private Variable can not be acceesed from anywhere
        self.__windows =windows
        self.__doors =doors
        self.__enginetype =enginetype
#           self.name = "Nasir"
audi = Car(4,5,"Diesel")
```

In [ ]:

In [ ]:

In [ ]: