OOP:

```
- Creating Basic ATM machine using class, object, constructor & method
- Creating Basic ATM using all above OOP ways plus while: True
- Why we need constructor and type of constructor
- Creating own data types (Fraction class)
- Encapsulation
- Public and Private variable
- memory level storage of private variable __dict__ (to see)
- name mangling
- What is reference variable
- Merging list of object with loop
- @property decorators
- @static method
```

In [1]:

```python
class Atm:                  #class name in camel case

    def __init__(self):       #Constructor and instance variable
        self.pin=" "
        self.balance=0


    def display(self):                    #Creating methods
        print("1. Create Pin")
        print("2. Check Balance")
        print("3. Deposit Amount")
        print("4. Withdrwal Amount")
        print("5. Exit")

        enter=input("Enter what you want to do\n")

        if enter=="1":
            self.createpin()
        elif enter=="2":
            self.checkbalance()
        elif enter=="3":
            self.depositamount()
        elif enter=="4":
            self.withdrawl()
        else:
            self.exit()

    def createpin(self):
        self.pin=input("Create pin, enter number you want as pin: ")
        print("Congrats pin created")

    def checkbalance(self):
        self.enterpin=input("Enter pin to check balance: ")
        if self.enterpin==self.pin:
            print("Your balance is",self.balance)
        else:
            print("Wrong pin, try again")
            self.enterpin=("Enter pin to check balance: ")

    def depositamount(self):
        self.enterpin=input("Enter pin to deposit amount: ")
        if self.enterpin==self.pin:
            print("Valid pin, now deposit amount you want")
            self.deposit=float(input("Deposit amount\n"))
            self.balance=self.balance+self.deposit
        else:
            print("Wrong pin, try again")
            self.enterpin=("Enter pin to check balance: ")

    def withdrawl(self):
        self.enterpin=input("Enter pin to withdraw amount: ")
        if self.enterpin==self.pin:
            print("Valid pin, now withdraw amount you want")
            self.withdraw=float(input("withdraw amount\n"))
            self.balance=self.balance-self.withdraw
        else:
            print("Wrong pin, try again")
            self.enterpin=("Enter pin to check balance: ")

    def exit(self):
        print("Thank you for using ATM")
```

In [10]:

```python
#lets create object

useofatm=Atm()
```

In [11]:
```
useofatm.display()
```
```
1. Create Pin
2. Check Balance
3. Deposit Amount
4. Withdrwal Amount
5. Exit
Enter what you want to do
1
Create pin, enter number you want as pin: 1234
Congrats pin created
```
In [ ]:
```
#But its not running for other option, what we have to do?
while True:
    useofatm.display()
```
```
1. Create Pin
2. Check Balance
3. Deposit Amount
4. Withdrwal Amount
5. Exit
Enter what you want to do
1
Create pin, enter number you want as pin: 5678
Congrats pin created
1. Create Pin
2. Check Balance
3. Deposit Amount
4. Withdrwal Amount
5. Exit
Enter what you want to do
3
Enter pin to deposit amount: 7896
Wrong pin, try again
1. Create Pin
2. Check Balance
3. Deposit Amount
4. Withdrwal Amount
5. Exit
Enter what you want to do
3
Enter pin to deposit amount: 5678
Valid pin, now deposit amount you want
Deposit amount
10000
1. Create Pin
2. Check Balance
3. Deposit Amount
4. Withdrwal Amount
5. Exit
Enter what you want to do
4
Enter pin to withdraw amount: 5678
Valid pin, now withdraw amount you want
withdraw amount
500
1. Create Pin
2. Check Balance
3. Deposit Amount
4. Withdrwal Amount
5. Exit
```
In [ ]:
```
#its working, you forgot to print actual balance after deposit and withdrawl, we will add it too.
```
In [3]:
```
class Atm:                  #class name in camel case

    def __init__(self):       #Constructor and instance variable
        self.pin=" "
        self.balance=0
        self.display()


    def display(self):                   #Creating methods
        print("1. Create Pin")
        print("2. Check Balance")
        print("3. Deposit Amount")
        print("4. Withdrwal Amount")
        print("5. Exit")

        enter=input("Enter what you want to do\n")

        if enter=="1":
            self.createpin()
        elif enter=="2":
            self.checkbalance()
        elif enter=="3":
            self.depositamount()
        elif enter=="4":
            self.withdrawl()
        else:
            self.exit()

    def createpin(self):
        self.pin=input("Create pin, enter number you want as pin: ")
        print("Congrats pin created")

    def checkbalance(self):
        self.enterpin=input("Enter pin to check balance: ")
        if self.enterpin==self.pin:
            print("Your balance is",self.balance)
        else:
            print("Wrong pin, try again")
            self.enterpin=("Enter pin to check balance: ")

    def depositamount(self):
        self.enterpin=input("Enter pin to deposit amount: ")
        if self.enterpin==self.pin:
            print("Valid pin, now deposit amount you want")
            self.deposit=float(input("Deposit amount\n"))
            self.balance=self.balance+self.deposit
            print("your balance is",self.balance)
        else:
            print("Wrong pin, try again")
            self.enterpin=("Enter pin to check balance: ")

    def withdrawl(self):
        self.enterpin=input("Enter pin to withdraw amount: ")
        if self.enterpin==self.pin:
            print("Valid pin, now withdraw amount you want")
            self.withdraw=float(input("withdraw amount\n"))
            self.balance=self.balance-self.withdraw
            print("your balance is",self.balance)
        else:
            print("Wrong pin, try again")
            self.enterpin=("Enter pin to check balance: ")

    def exit(self):
        print("Thank you for using ATM")
```

In [4]:

```
useofatm2=Atm()
```

```
1. Create Pin
2. Check Balance
3. Deposit Amount
4. Withdrwal Amount
5. Exit
Enter what you want to do
1
Create pin, enter number you want as pin: 1234
Congrats pin created
```

In [5]:

```
#see whats happening now? we do not want user to check everything and need to call, once user provide object
#program start running with display because now we call display at constructor because, we do not give control
#he/she want to see display or not: The stuff like connection to internet, provide GPS in uber, its automated m
#shuld start automatically after program runs. This type of code should be written inside constructor.
```

In [3]:

```
class Atm:                  #class name in camel case

    def __init__(self):       #Constructor and instance variable

        self.pin=" "
        self.balance=0
        self.display()


    def display(self):                   #Creating methods
        while True:
            print("1. Create Pin")
            print("2. Check Balance")
            print("3. Deposit Amount")
            print("4. Withdrwal Amount")
            print("5. Exit")

            enter=input("Enter what you want to do\n")

            if enter=="1":
                self.createpin()
            elif enter=="2":
                self.checkbalance()
            elif enter=="3":
                self.depositamount()
            elif enter=="4":
                self.withdrawl()
            else:
```

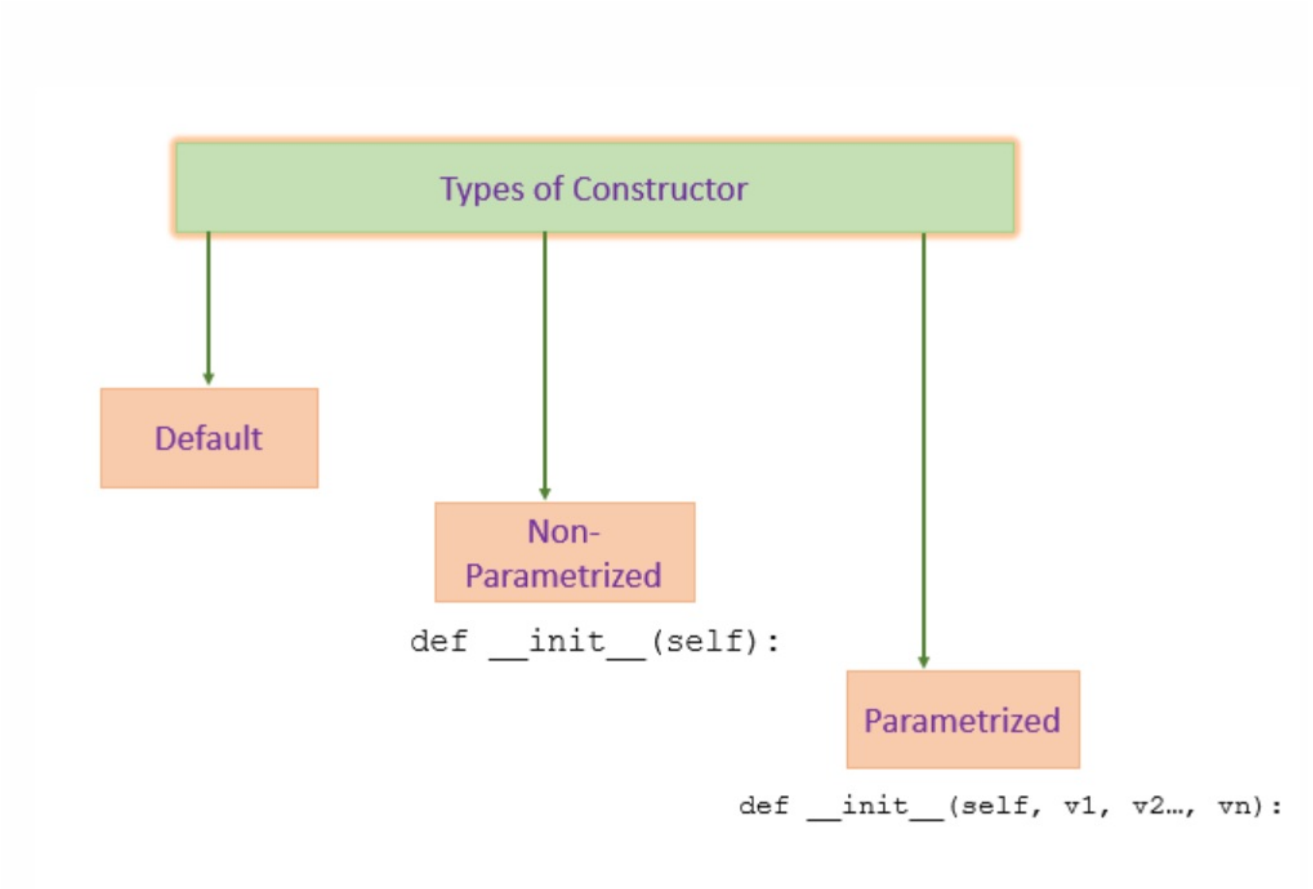
```python
                self.exit()
                break

    def createpin(self):
        self.pin=input("Create pin, enter number you want as pin: ")
        print("Congrats pin created")

    def checkbalance(self):
        self.enterpin=input("Enter pin to check balance: ")
        if self.enterpin==self.pin:
            print("Your balance is",self.balance)
        else:
            print("Wrong pin, try again")
            self.enterpin=("Enter pin to check balance: ")

    def depositamount(self):
        self.enterpin=input("Enter pin to deposit amount: ")
        if self.enterpin==self.pin:
            print("Valid pin, now deposit amount you want")
            self.deposit=float(input("Deposit amount\n"))
            self.balance=self.balance+self.deposit
            print("your balance is",self.balance)
        else:
            print("Wrong pin, try again")
            self.enterpin=("Enter pin to check balance: ")

    def withdrawl(self):
        self.enterpin=input("Enter pin to withdraw amount: ")
        if self.enterpin==self.pin:
            print("Valid pin, now withdraw amount you want")
            self.withdraw=float(input("withdraw amount\n"))
            self.balance=self.balance-self.withdraw
            print("your balance is",self.balance)
        else:
            print("Wrong pin, try again")
            self.enterpin=("Enter pin to check balance: ")

    def exit(self):
        print("Thank you for using ATM")
```

In [4]:
```python
useofatm2=Atm()
```

```
1. Create Pin
2. Check Balance
3. Deposit Amount
4. Withdrwal Amount
5. Exit
Enter what you want to do
1
Create pin, enter number you want as pin: 123
Congrats pin created
1. Create Pin
2. Check Balance
3. Deposit Amount
4. Withdrwal Amount
5. Exit
Enter what you want to do
3
Enter pin to deposit amount: 1000
Wrong pin, try again
1. Create Pin
2. Check Balance
3. Deposit Amount
4. Withdrwal Amount
5. Exit
Enter what you want to do
3
Enter pin to deposit amount: 123
Valid pin, now deposit amount you want
Deposit amount
1000
your balance is 1000.0
1. Create Pin
2. Check Balance
3. Deposit Amount
4. Withdrwal Amount
5. Exit
Enter what you want to do
2
Enter pin to check balance: 123
Your balance is 1000.0
1. Create Pin
2. Check Balance
3. Deposit Amount
4. Withdrwal Amount
5. Exit
Enter what you want to do
5
Thank you for using ATM
```

In [1]:
```
#Now it worked.
```

Constructor in Python:



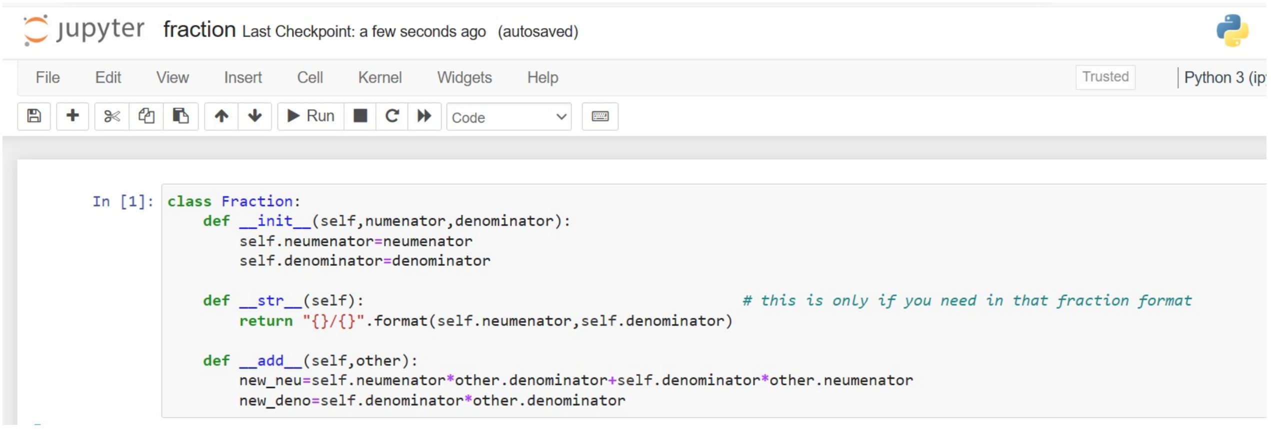
Types of constructor

```
In [2]: #Creates own data type in Python
```

But first lets learn how to create module on our own module.

Steps & screenshot:

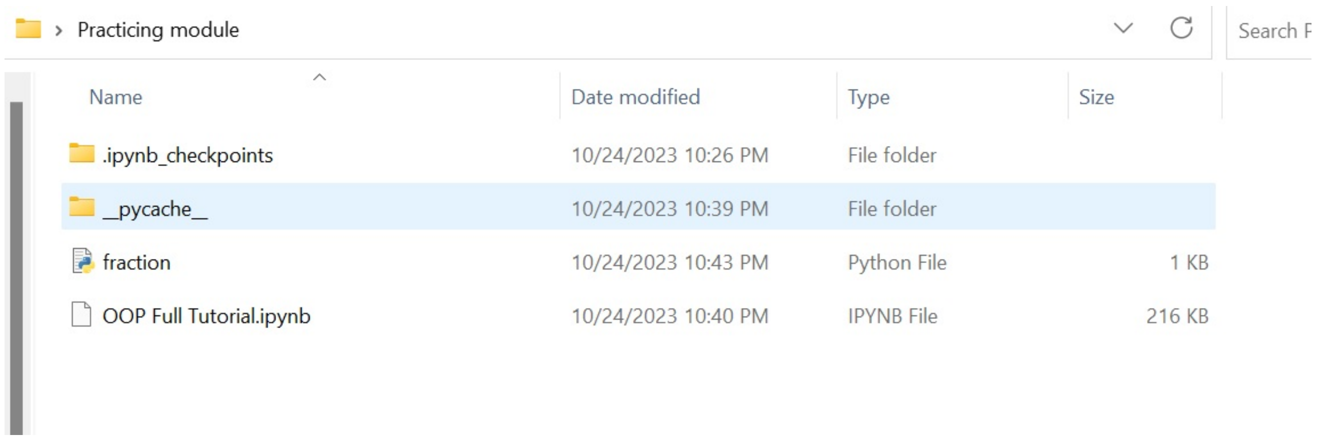
- open new notebook and write some function in it.


jupyter fraction Last Checkpoint: a few seconds ago (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ip
Run
Code

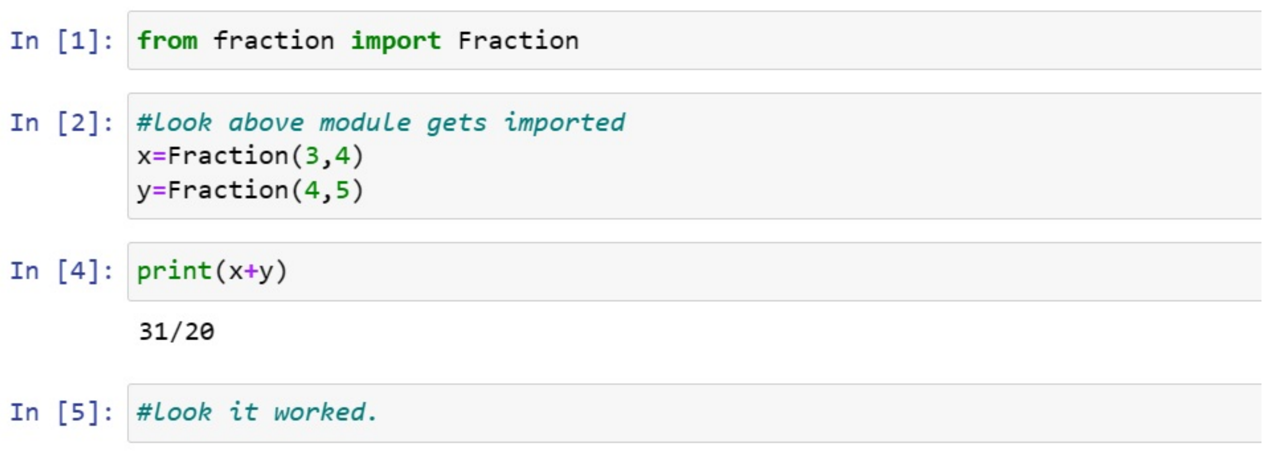
```
In [1]: class Fraction:
            def __init__(self,numenator,denominator):
                self.neumenator=neumenator
                self.denominator=denominator

            def __str__(self):                              # this is only if you need in that fraction format
                return "{}/{}".format(self.neumenator,self.denominator)

            def __add__(self,other):
                new_neu=self.neumenator*other.denominator+self.denominator*other.neumenator
                new_deno=self.denominator*other.denominator
```

- Convert in in py while downloading
- Now save py file in same folder with other file where we want to import

Like I saved it in dekstop



```
Name                      Date modified          Type           Size
.ipynb_checkpoints        10/24/2023 10:26 PM    File folder
__pycache__               10/24/2023 10:39 PM    File folder
fraction                  10/24/2023 10:43 PM    Python File    1 KB
OOP Full Tutorial.ipynb   10/24/2023 10:40 PM    IPYNB File     216 KB
```



```
In [1]: from fraction import Fraction

In [2]: #Look above module gets imported
        x=Fraction(3,4)
        y=Fraction(4,5)

In [4]: print(x+y)

        31/20

In [5]: #Look it worked.
```

```
In [2]: #As well as while creating Fraction class (above we can see in screenshot), we can use format as a return too a
```

Encapsulation in Python

```
In [3]: class Car:
            def __init__(self,amount:int,color:str,model:str):
                self.amount=amount
                self.color=color
                self.model=model
```

```
In [4]: #so lets create object now:
        car1=Car(2000,"red","honda")
```

```
In [5]: print(car1.amount)
```

```
2000
```

```
In [9]: #lets create second object
        car2=Car(3000,"blue","honda")
```

```
In [10]: print(car2.amount)
```

```
3000
```

```
In [21]: #Now lets try to hide attribute.
         class Car:
             def __init__(self,amount:int,color:str,model:str):
                 self.amount=amount
                 self.color=color
                 self.__model=model

             def display(self):
                 print("The color of car is", self.color)
                 print("The model of car is",self.model)
```

```
In [23]: car2=Car(5000,"yellow","Ferrari")
```

```
In [24]: print(car2.amount)
```

```
5000
```

```
In [25]: print(car2.color)
```

```
yellow
```

```
In [26]: print(car2.model)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[26], line 1
----> 1 print(car2.model)

AttributeError: 'Car' object has no attribute 'model'
```

In [19]:
```
#see because we have hide it.
```

In [27]:
```
car2.display()
```

```
The color of car is yellow
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[27], line 1
----> 1 car2.display()

Cell In[21], line 10, in Car.display(self)
      8 def display(self):
      9     print("The color of car is", self.color)
---> 10     print("The model of car is",self.model)

AttributeError: 'Car' object has no attribute 'model'
```

In [28]:
```
#see it show color but not model? why beacsue we have use encapsulation to hide it. But, in project, if the per
#want to access it? how is that possible?
```

In [29]:
```
print(car2._Car__model)         #in this way
```

```
Ferrari
```

This process is known as name mangling.

Getter and Setter Method

In [1]:
```
#first of all again lets understand encapsulation, private and public variable concept clearly
```

In [4]:
```
class Finance:
    def __init__(self):
        self.expenses=1000
        self.revenue=5000

    def display(self):
        print(f'Revenue of company according to finance department is {self.revenue}')

class HR:
    def __init__(self):
        self.num_of_employee=num_of_employee
```

In [5]:
```
#lets create object

f1=Finance()
```

In [6]:
```
f1.display()
```

```
Revenue of company according to finance department is 5000
```

In [7]:
```
print(f1.revenue)
```

```
5000
```

In [12]:
```
#Now this is creation of normally 2 class with its attributes and attributes/variable are public so that HR dep
#can use finance department variable and may modified it mistakely.

class Finance:
    def __init__(self):
        self.expenses=1000
        self.revenue=5000

    def display(self):
        print(f'Revenue of company according to finance department is {self.revenue}')


class HR:
    def __init__(self):
        self.numofemployee=33
        print(f1.revenue)  #we are directly printing and using it
```

In [13]:
```
#create HR class
h1=HR()
```

```
5000
```

In [14]:
```
#see we are using f1 object inside HR class and we are seeing value of revenue from Finance class, which are di
```

In [15]:
```
#so to secure it, we have to hide the data, we have to make private.
```

In [26]:
```
class Finance:
    def __init__(self):
        self.expenses=1000
        self.__revenue=5000

    def display(self):
        print(f'Revenue of company according to finance department is {self.revenue}')
        print(f'Total expenses is {self.expenses}')


class HR:
    def __init__(self):
        self.numofemployee=33
        print(f1.revenue)  #we are directly printing and using it
```

In [27]:
```
f1=Finance()
```

In [28]:
```
print(f1.expenses)
```

```
1000
```

In [29]:
```
f1.display()   #lets see because one is private and one is public
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[29], line 1
----> 1 f1.display()

Cell In[26], line 7, in Finance.display(self)
      6 def display(self):
----> 7     print(f'Revenue of company according to finance department is {self.revenue}')
      8     print(f'Total expenses is {self.expenses}')

AttributeError: 'Finance' object has no attribute 'revenue'
```

In [36]:
```
#But, know? we can use private inside __init__ beacuse it is inside only that merhod where it is created. lets
class Finance:
    def __init__(self):
        self.expenses=1000
        self.__revenue=5000
        print(f'Revenue of company according to finance department is {self.__revenue}')

    def display(self):
        print(f'Total expenses is {self.expenses}')
        print(f'Revenue of company according to finance department is {self.__revenue}')

class HR:
    def __init__(self):
        self.numofemployee=33
        print(f1.__revenue)  #we are directly printing and using it
```

In [37]:
```
f2=Finance()
```

```
Revenue of company according to finance department is 5000
```

In [38]:
```
#But, can we use it in method,
f2.display()
```

```
Total expenses is 1000
Revenue of company according to finance department is 5000
```

In [39]:
```
#can we use it in HR class? outside Finance class?
h1=HR()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[39], line 2
      1 #can we use it in HR class? outside Finance class?
----> 2 h1=HR()

Cell In[36], line 15, in HR.__init__(self)
     13 def __init__(self):
     14     self.numofemployee=33
---> 15     print(f1.revenue)

AttributeError: 'Finance' object has no attribute 'revenue'
```

In [40]:
```
#No we cannot use, but can we use in the same way in Finance? lets see again
class Finance:
    def __init__(self):
        self.expenses=1000
        self.__revenue=5000


    def display(self):
        print(f'Total expenses is {self.expenses}')
        print(f1.__revenue)

class HR:
    def __init__(self):
        self.numofemployee=33
        print(f1.__revenue)  #we are directly printing and using it
```

In [41]:
```
f3=Finance()
```

In [42]:
```
f3.display()
```

```
Total expenses is 1000
5000
```

In [47]:
```
#lets see How this private variable are stored in memory level
print(f3.__dict__)
```

```
{'expenses': 1000, '_Finance__revenue': 5000}
```

In [45]:
```
#yes we can use, because display is inside class Finance.
#But if we want to use it outside class, we have to use name mangled (see above __dict__, we will use same)

class HR:
    def __init__(self):
        self.numofemployee=33
        print(f1._Finance__revenue) #see memory level store for reference
```

In [46]:
```
h4=HR()
```

```
5000
```

What is Reference variable?

- During object creating, we store object in variable, that variable is reference variable technically.

ex:

if ATM is class, suppose atm1 is variable to create object:

atm1=ATM()

then atm1 is reference variable.

Merging Loops + Object (group of object) - in list, tuples and dictionary if needed

In [1]:
```
class Student:
    def __init__(self, name, age):
        self.name=name
        self.age=age

C1=Student("Ram",24)
C2=Student("Hari",34)
C3=Student("Shyam",44)

L=[C1,C2,C3]

for i in L:
    print(i.name,i.age)

```

```
Ram 24
Hari 34
Shyam 44
```

In [2]:
```
#see we can merge object groups in list and we can run loop, we can use method too like:
class Student:
    def __init__(self, name, age):
        self.name=name
        self.age=age

    def display(self):
        print("My name is",self.name,"and age is",self.age)

C1=Student("Ram",24)
C2=Student("Hari",34)
C3=Student("Shyam",44)

L=[C1,C2,C3]

for i in L:
    i.display()
```

```
My name is Ram and age is 24
My name is Hari and age is 34
My name is Shyam and age is 44
```

Python Decorators

- @Property
- @classmethod
- @static

In [7]:
```
#Property

class Student:
    def __init__(self, name, grade):
        self.name=name
        self.grade=grade
        self.message=self.name +  " got grade :" + self.grade

stud1=Student("Ram","C")
```

In [8]:
```
print(stud1.name)
print(stud1.grade)
print(stud1.message)
```

```
Ram
C
Ram got grade :C
```

In [9]:
```
#suppose now if we need to change grade
stud1.grade = "A"
```

In [10]:
```
print(stud1.name)
print(stud1.grade)
print(stud1.message)
```

```
Ram
A
Ram got grade :C
```

In [11]:
```
#see grade got changed but not message, why? because, the derived attribute will not change when we change orig
```

In [13]:
```
#But, if we change attribute to method? It will work
class Student:
    def __init__(self, name, grade):
        self.name=name
        self.grade=grade

    def display(self):
        return self.name +  " got grade :" + self.grade

stud1=Student("Ram","C")
```

In [14]:
```
print(stud1.name)
print(stud1.grade)
print(stud1.display())
```

```
Ram
C
Ram got grade :C
```

In [15]:
```
#Here it got changed, but, whats the problem? the user who is using our class has to use this :() parenthesis t
#message
```

Here, came the use of decorators, due to this user/client need not to worry about adding parenthesis in 1000 of lines code in real scenario

In [16]:
```
#first lets see without @property decorators
class Student:
    def __init__(self, name, grade):
        self.name=name
        self.grade=grade

    def display(self):
        return self.name +  " got grade :" + self.grade

stud1=Student("Ram","C")
```

In [17]:
```
print(stud1.name)
print(stud1.grade)
print(stud1.display)
```

```
Ram
C
<bound method Student.display of <__main__.Student object at 0x0000021C36F23F40>>
```

In [18]:
```
#see above,what happened? display does not give any output, it act as object
#now use property
class Student:
    def __init__(self, name, grade):
        self.name=name
        self.grade=grade

    @property
    def display(self):
        return self.name +  " got grade :" + self.grade

stud1=Student("Ram","C")
```

In [19]:
```
print(stud1.name)
print(stud1.grade)
print(stud1.display)
```

```
Ram
C
Ram got grade :C
```

In [20]:
```
#see, this is the use of property decorators.
```

setter method

In [21]:
```
#suppose, client want to change name and grade, he wants shyam with grade A.
#And, client cannot change code or want to add any extra things, he just want to change method and output, so w
```

In [1]:
```
#setter method
class Student:
    def __init__(self, name, grade):
        self.name=name
        self.grade=grade

    @property
    def display(self):
        return self.name +  " got grade :" + self.grade

    @display.setter
    def display(self,display_str):
        part=display_str.split(" ")  #sent is just stored variable, we can use any name
        print(part)  #just to show, how it look after split
        self.name=part[0]
        self.grade=part[-1]

```

In [2]:
```
#above we do from coder side, now what client can do?
#But what he/she say, he/she want shyam instead of Ram and A insted of grade A, they just need to do
```

In [3]:
```
stud1=Student("Ram","C")
stud1.display=("Shyam got grade :A")
print(stud1.name)
print(stud1.grade)
print(stud1.display)
```

```
['Shyam', 'got', 'grade', ':A']
Shyam
:A
Shyam got grade ::A
```

Lets see another example of property decorators

In [10]:
```
class Marks:
    def __init__(self, totalreceivedmarks):
        self.totalreceivedmarks=totalreceivedmarks
        self.per=(totalreceivedmarks/600)*100

    def percentage(self):
        return self.per
```

In [11]:
```
#lets create object
student1=Marks(400)
print(student1.totalreceivedmarks)
#we have use return in method so have to use print below
print(student1.percentage())
```

```
400
66.66666666666666
```

In [19]:
```
#But, now, if client want to change total received marks?
student1.totalreceivedmarks=550
```

In [20]:
```
print(student1.percentage())
```

```
66.66666666666666
```

In [21]:
```
print(student1.totalreceivedmarks)
```

```
550
```

In [22]:
```
#see marks got changed but not percentage, what we have learned in previous example, derived attribute will sti
```

```
#but also, client and user are changing orginal marks so here we can use encapsulation by setting marks attribu
#as well as @property decorators so that if needed client can change marks too.
```

What I am saying that: In python nothing is actually a private, we just use encapsuation to protect from unwanted accident changes.

In [40]:
```
#Firstly lets use encapsulation
class Marks:
    def __init__(self, totalreceivedmarks, name):
        self.__totalreceivedmarks=totalreceivedmarks
        self.name=name
        self.per=(self.__totalreceivedmarks/600)*100

    def percentage(self):
        return self.per

    def display(self):
        print("His name is", self.name)
        print("his marks is",self.__totalreceivedmarks)
```

In [44]:
```
student2=Marks(400,"Krishna")
```

In [36]:
```
print(student2.name)
```

```
Krishna
```

In [45]:
```
print(student2.totalreceivedmarks)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[45], line 1
----> 1 print(student2.totalreceivedmarks)

AttributeError: 'Marks' object has no attribute 'totalreceivedmarks'
```

In [27]:
```
#its error because we have make it private attribute with the concept of encapuslation
```

In [42]:
```
student2.display()
```

```
His name is Krishna
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[42], line 1
----> 1 student2.display()

Cell In[34], line 13, in Marks.display(self)
     11 def display(self):
     12     print("His name is", self.name)
---> 13     print("his marks is",self.totalreceivedmarks)

AttributeError: 'Marks' object has no attribute 'totalreceivedmarks'
```

In [39]:
```
#see we got name but not marks
#now another method?
print(student2.percentage())
```

```
66.66666666666666
```

Why, we got percentage?

The reason print(student2.percentage()) works even though self.**totalreceivedmarks is intended to be private is that in Python, name mangling is used for attributes with double underscores (e.g., self.**totalreceivedmarks). Name mangling changes the name of the attribute in a way that makes it harder to accidentally override in subclasses, but it does not make the attribute completely private.

When you create an instance of the class and access student2.percentage(), Python still recognizes self.**totalreceivedmarks as _Marks**totalreceivedmarks behind the scenes. This modified name is used to access the attribute. So, student2.percentage() can access self.**totalreceivedmarks through the modified name _Marks**totalreceivedmarks.

This behavior allows for a limited form of encapsulation

In [48]:
```
#But, if we want to see display, we know that we can use name mangling to print it like
print(student2.__dict__)
```

```
{'_Marks__totalreceivedmarks': 400, 'name': 'Krishna', 'per': 66.66666666666666}
```

In [49]:
```
print(student2._Marks__totalreceivedmarks)
```

```
400
```

In [50]:
```
#But this will be hard for client/user to use name mangling so we can use decorators here. - Property decorator
```

In [2]:
```
class Marks:
    def __init__(self, totalreceivedmarks, name):
        self.__totalreceivedmarks=totalreceivedmarks
        self.name=name
        self.per=(self.__totalreceivedmarks/600)*100

    def percentage(self):
        return self.per

    def display(self):
        print("His name is", self.name)
        print("his marks is",self.__totalreceivedmarks)

    @property
    def totalreceivedmarks(self):
        return self.__totalreceivedmarks

    @totalreceivedmarks.setter
    def totalreceivedmarks(self,value):
        self.__totalreceivedmarks=value
```

In [4]:
```
#now how client can use?
student = Marks(450, "John")
print("Original marks:", student.totalreceivedmarks)
student.totalreceivedmarks = 500
print("Updated marks:", student.totalreceivedmarks)
```

```
Original marks: 450
Updated marks: 500
```

In [5]:
```
#see above instead of private attributes, we can access and update using property decorators.
```

Static Method

In [2]:
```
#It does not need self and cls

class Person:
    def __init__(self,name:str,salary:int):
        self.name=name
        self.salary=salary

    def display(self):
        print("Person name is", self.name, "and salary is",self.salary)

    @staticmethod
    def check_age(age):
        if age>16:
            print("Person is eligible for voting")
        else:
            print("Person is not eligible for voting")
```

In [3]:
```
#creating object
person1=Person("Ram",20000)
person2=Person("Hari",300000)
```

In [4]:
```
print(person1.salary)
```

```
20000
```

In [5]:
```
person2.display()
```

```
Person name is Hari and salary is 300000
```

In [6]:
```
person1.check_age(19)
```

```
Person is eligible for voting
```

In [7]:
```
person2.check_age(14)
```

```
Person is not eligible for voting
```

In [ ]:
```
#Now, we will look OOP concepts more in one small project- Next file...Continue.
```